Stony Brook University

A study of Binary Instrumentation techniques

Soumyakant Priyadarshan

August 30, 2019

Abstract

Low-level vulnerabilities have remained an important source of compromise in computer systems. Despite the deployment of various protection mechanisms at the OS level/hardware level, attackers have been able to exploit memory corruption vulnerabilities to compromise a program execution. Many compiler or source code based solutions have been proposed to check memory corruption, control flow diversion, etc. However, the unavailability of source code limits the large scale deployment of such solutions. Binary instrumentation can play an important role in enforcing low-level security policies such as CFI (Control flow integrity), SFI (Software fault isolation) and code randomization. Binary instrumentation is the process of introducing new code into a program without changing its overall behavior. Binary instrumentation can be done either at the runtime (Dynamic binary instrumentation) or offline (Static binary instrumentation). Static binary instrumentation (SBI) results in efficient instrumented binaries with less performance overhead. However, SBI is challenging because of data embedded within code and indirect branches. To enforce DEP (Data execution prevention) fully, modern compilers have started to separate data from code by assigning different sections to code and data. Also, to employ ASLR (Address space layout randomization), x86-64 bit programs are being compiled as position independent executable. All the commonly used binaries on most of the x86-64 LINUX distributions are PIE. PIE executables have relocation information which can be exploited to recover indirect branch targets. Exploiting these factors can help SBI become robust, complete and accurate.

This report presents a survey of various static and dynamic binary instrumentation techniques and security policies such as CFI and code randomization that are enforced using binary instrumentation. At the end of this report, we introduce our fine-grained code randomization approach for x86-64 PIE binaries. We exploit the relocation information of the PIE binaries to achieve complete and correct disassembly. This helps us in achieving fine-grained code randomization at the basic block level, without using any symbol or debugging information.

Contents

1 Introduction

Program instrumentation is the process of adding new instructions into a program or modifying existing code of a program. Program instrumentation is primarily used for *security policy enforcement*: CCFIR [28], BinCFI [29] use static binary instrumentation to enforce CFI (control flow integrity) on COTS binaries. ILR [11] uses dynamic binary instrumentation to perform code randomization during the runtime. PIP [22] uses program instrumentation to intercept system calls and enforce policies in order to protect benign applications from getting affected by untrusted applications.A second major application of program instrumentation is *monitoring and debugging*. For instance, Valgrind [15] uses dynamic instrumentation to track dangerous uses of undefined values. Other applications of program instrumentation include program optimization [6], exploit detection/prevention,etc.

Program instrumentation can be achieved either by modifying the source code and rebuilding the program or by changing the binary executable itself. Applying instrumentation or fixing vulnerabilities at the source code level is less complicated and can be highly efficient and accurate because at the program. Instrumenting binaries, on the other hand, can be challenging and less accurate in stripped (COTS) binaries. One major challenge is the content classification problem. To achieve accurate instrumentation correct and complete code discovery is important but distinguishing code from data in a binary can be hard in the absence of symbolic information. However, binary instrumentation can be advantageous because:

- *No need for source code:* Commercial of the shelf (COTS) programs are available in the form of binary only. Even when application's source code is available, it may use external third-party libraries for which source code may not be available. In contrast, binary executables are readily available, making binary based instrumentation techniques more widely applicable.
- *Completeness:* Operating on binary makes it possible to apply instrumentation throughout the program and across all the modules (including third-party shared libraries). This is essential for enforcing security policies such as CFI.
- *Language and compiler independent:* Source-code based tools are specific to each programming language. Moreover, they are often limited to a single compiler. In contrast, binary instrumentation can be applied to binaries compiled from any source language by any compiler.

Apart from the above mentioned advantages, binary based techniques empower end-users to analyze and/or mitigate vulnerabilities, instead of always having to wait for new features or updates from vendors

Binary instrumentation can be achieved in two ways.

- *Dynamic binary instrumentation (DBI)*: Instrument and execute code blocks during the runtime, just before a code block is executed.
- *Static binary instrumentation (SBI)*: Instrument the binary offline or statically.

Both approaches have their pros and cons. Runtime disassembly and instrumentation help in avoiding misinterpretation of data as code, allowing DBI to be more robust [14]. However, runtime code translation and instrumentation results in significant performance overhead. On the other hand, data between code and indirect branches make static binary instrumentation error prone. However, static binary instrumentation can be very efficient as it introduces no extra performance overhead.

With the evolution of compilers, binaries are becoming more static binary instrumentation friendly. For example, to enforce DEP (Data execution prevention) fully, compilers have started creating separate sections for code and data. Similarly, to support ASLR, x86-64 bit programs are being compiled as position independent executables (PIE). The relocation information is an inherent part of PIE executables. Exploiting relocation information can help recover the indirect branch targets. We have exploited the relocation information in x86-64 bit PIE executables to achieve a complete and correct disassembly and apply fine-grained code randomization at basic block level. In this report, we survey few existing binary instrumentation techniques for stripped binaries, examine their shortcomings and propose an effective binary instrumentation technique that exploits the relocation information to overcome the shortcomings. The remainder of this report is organized as follows. In section 2 we discuss static binary instrumentation, challenges in SBI, code discovery techniques and various SBI tools such as PEBIL [13] and Secondwrite [20]. In section 3 we discuss Dynamic binary instrumentation techniques and in section 4 we discuss our proposed fine-grained code randomization technique for x86-64 position independent executables.

2 Static Binary Instrumentation

Static binary instrumentation involves offline disassembly, instrumentation and regeneration of a program binary. SBI results in efficient binaries. The only factor that adds to the performance overhead is the execution of the instrumentation code. However, static instrumentation of stripped binaries without any supplemental information is highly error prone. The following subsections discuss the challenges in static binary instrumentation and survey a few existing disassembly and static binary instrumentation techniques.

2.1 Accurate disassembly

A good instrumentation tool should be able to instrument all code that is to be executed. 100% code coverage is essential for effective implementation of security policies. Undiscovered code may contain vulnerabilities that will be left out from being rectified or protected by the security policy. At the same time, incorrect disassembly will lead to incorrect instrumentation which can result in crashes or malfunction. There are two types of disassembly techniques:

- *Linear disassembly:* The simplest way to disassemble a binary is to linearly disassemble the code section instruction by instruction, starting from the first byte of the code

Original Code	**Objdump output**
.L1:	
jmp .L2	
.byte 0x0f	0: eb 01 jmp 3 <C-0xd>
	2: 0f 48 c7 cmovs %edi,%eax
.L2:	5: c0 10 00 rclb $0x0,(%rax)
movq $0x10, %rax	8: 00 00 add %al,(%rax)
push %rax	a: 50 push %rax
call C	b: e8 00 00 00 00 callq 10 <C>
C:	

Table 2.1: Linear disassembly example

section. This is called linear disassembly and is used by the GNU utility Objdump [10].

- *Recursive disassembly:* Recursive disassembly approaches follow the control flow of the program while disassembling the instructions.

2.1.1 Linear disassembly

Linearly disassembling the code section provides maximum code coverage. However, embedded data and alignment padding can be a problem. Because of the dense encoding of x86 and x86-64 CISC architecture, there is a very high probability that any data byte sequence will get translated to valid instruction. Furthermore, because of the variable-length instruction set of x86 and x86-64 architecture, subsequent valid instructions can get incorrectly translated. As shown in table 2.1, the original code contains a data byte *0x0f* in the middle of the code. Objdump treats it as a valid instruction opcode(*cmovs*) and treats the subsequent code bytes as its operands and hence misinterprets the next instruction as well.

This can create a serious problem as data being interpreted as valid code can end up being instrumented/relocated and can lead to malfunction.

2.1.2 Recursive disassembly

One way to avoid errors due to embedded data is to follow the control flow path of a program. Under ideal conditions, recursive disassembly can provide 100% accurate code recovery. Direct control-flow transfer instructions whose target is embedded in the instruction are easy to decipher. But, the indirect control-flow transfer instructions whose target is determined at the runtime pose a problem. A good binary instrumentation tool needs to have an effective mechanism that can determine all possible indirect jump targets. Indirect branch targets can be classified into following three categories:

- **Code pointer constants**: These are constants used as indirect branch targets. They

.1665746: leaq .1825984(%rip),%rbp .1665753: movslq 0x0(%rbp,%rax,4),%rax .1665758: addq %rax,%rbp .1665761: jmp *%rbp ————Jump table————————- .1825984: 0xfffd9160 .1825988: 0xfffd90b8 .1825992: 0xfffd9008 .1825996: 0xfffd8e5d ————————————————	.1665746: RBP = 1825984/0x1bdcc0, RAX = index of jump table. Assuming RAX = 0 .1665753: RAX = 0xfffd9160, RBP = 0x1bdcc0 .1665758: RBP = RAX + RBP = 0x196e20 .1665761: Jump to 0x196e20

Table 2.2: An example of Jump table obtained from glibc

are generated during compile time and may be present as constants stored in data sections or as immediate operands. Function pointers of high-level languages are compiled into stored code pointers. A simple approach to obtain stored pointers would be to scan the entire binary, byte-by-byte to find constant values that fall within the range of code section. However, data can be misinterpreted as an indirect branch target which may result in disassembly and instrumentation of something that is not valid code.

- **Computed code pointers**: These are constants on which arbitrary computations may be performed before they are used as targets. Arbitrary computation makes it hard to accurately determine the targets statically. Known cases of computed code pointers are:
  - *Jump tables:* Switch-case blocks of high-level C/C++ programs are encoded into jump tables. The entries of the jump tables undergo computation at runtime, typically of the form **(C1 + ind) + C2*. Where *C1* and *C2* are constants and *ind* is the index of the jump table. Table 2.2 shows an example of ICF target computation using jump table.
  - *Exception handler addresses:* In ELF binaries, try/catch blocks for C++ exceptions are present in the eh_frame, eh_frame_hdr and gcc_except_table sections. They are encoded in a specific format (usually as an offset from the start of a function) and need to be decoded during the runtime. Addresses of all try/catch blocks can be obtained by statically parsing the eh sections (eh_frame, eh_frame_hdr and gcc_except_table).
- **Runtime generated pointers:**
  - *Return Addresses*: Return addresses are addresses immediately following a call instruction and are pushed on to stack by the call instruction. They can be computed easily after disassembly is complete.
  - *Instruction pointer relative addresses:* In PIE (Position independent executable) binaries where the instruction addresses are not known until runtime, pointers

PIE executables	**non-PIE executables**
lea 0x24a(%rip), %r8 ... jmp *(%r8)	mov $0x400b10, %r8 ... jmp *(%r8)

Table 2.3: Code pointers as operands

can be loaded as values relative to current instruction pointer value (Table 2.3).

Unstripped programs have symbol tables that contain information regarding function start and range. Exploiting symbol tables can help in recovering all indirectly called functions. PEBIL [13] is one such static binary instrumentation tool that exploits symbol table. However, symbol table will not help in recovering computed code pointers such as jump table targets. The following section describes the recursive disassembly approach of PEBIL.

2.1.3 PEBIL'S DISASSEMBLY

PEBIL [13] works under the assumption that an executable has a symbol table and uses it to obtain function entry points. It then follows a *control-driven disassembly* starting from each function entry. As, most of indirect branch targets found in binaries compiled from high level languages are function pointers, leveraging on symbol table and export table helps PEBIL in recovering most of the indirect branch targets. The only indirect branch targets left are the intra-function indirect branches, e.g. jump table targets. PEBIL employs a peephole examination to determine such indirect branch targets.

Compilers use simple calculations that usually follow a fixed pattern of adding an offset to a given address to access a value related to an indirect jump target. If PEBIL's peephole examination of preceding instructions reveals a fixed memory address, this address is treated as the first entry of a jump table. PEBIL makes an iterative pass over the table to determine all possible targets, stopping when it finds a target larger than the current function range. Authors claim that PEBIL is able to disassemble 99% of code bytes in SPEC CPU2000 Integer benchmarks.

However, symbol table is not available with stripped COTS binaries. A second solution to recover indirect branch targets can be the application of heuristics such as function prologue signature matching to recover possible function entries. All functions usually have a similar code at the entry point, that prepares stacks and registers for use within function. This can be exploited to recover function entry points. Angr's [19] CFGFast algorithm employs this approach.

2.1.4 ANGR'S DISASSEMBLY

Angr's [19] CFGFast approach tries to achieve high code coverage and detect all the functions and their contents. It uses heuristics to identify functions. The disassembly steps of

CFGFast are summarized below:

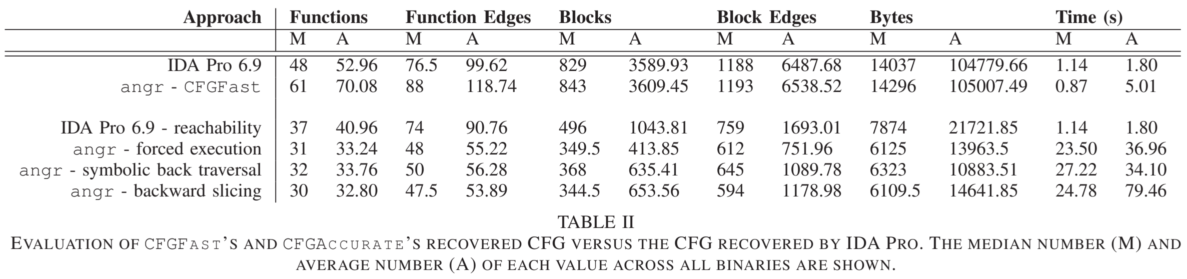
- *Function identification:* Hard-coded function prologue signatures are used to identify function entry points.
- A *Recursive disassembly* is performed starting from the entry point as well as identified function entry points to recover all the directly reached basic blocks.
- *Indirect branch resolution:* Lightweight *alias analysis* and *data-flow tracking* is used to resolve intra-function indirect control flow transfers. *alias analysis* is the technique of identifying program variables that may refer to same memory location. *data-flow tracking* is the technique of determining a set of possible values of a variable at various points in a computer program.

Relying on function prologue matching allows CFGFast to have high code coverage without any extra effort for discovering indirect branch targets. However, the generated CFG lacks much of the control flow. Angr employs CFGAccurate algorithm that tries to recover, where possible, all jump targets of each indirect branch. CFGAccurate iteratively recovers the control flow graph, starting from the basic block at the entry point. Three steps as mentioned below are followed in order to accurately recover as much code as possible.

- *Forced execution:* Forced execution ensures that both the branches of a conditional branch are executed at a branch point. Starting from the entry-point, this step executes the direct branches and adds the recovered basic blocks to the CFG. Forced execution can be inaccurate in recovering indirect branch targets, as the basic blocks are executed in an unexpected order that differs from the actual program execution. For example, forced execution may result in a state where a jump target is being read from an uninitialized memory location. Hence, all the indirect branches are skipped for later analysis. This step acts as a fast pass to recover all the directly reached basic blocks.
- *symbolic execution:* For every indirect branch identified by forced execution, the CFG is traversed back until the first merge-point (multiple paths converging on the way to indirect branch) is encountered or a threshold number of blocks have passed (Angr uses 8 basic blocks as threshold). Then a symbolic execution followed by a constraint solver is applied to recover the branch targets.

  CFGAccurate considers a jump to be successfully resolved if the number of resolved targets is less than a threshold, i.e. 256. This step is repeated until no new indirect branches are recovered.
- *Backward slicing:* The symbolic execution analysis will fail to resolve some indirect branches because of the lack of context. For example, if an indirect branch target is passed as an argument to another function, then symbolic execution will be unable to resolve it. Hence, a backward slice is computed starting from the indirect branch and extending up to the previous call context. Then a symbolic execution similar to the previous step is performed to recover the targets.

CFGAccurate can still fail in recovering targets for some indirect branches. Hence, employing CFGAccurate alone will result in incomplete disassembly. CFGFast and CFGAccurate was evaluated on CGC (DARPA Cyber grand challenge) binaries. In the absence of ground truth regarding the control flow graph, the results were compared to that of IDA pro. While CFGFast had a slightly better code coverage than than IDA pro, CFGAccurate lagged in terms of code coverage but was better in terms of reachability (Number of basic blocks reachable from program entry point) (Figure 2.1). A separate evaluation performed on CFGFast, shows 70-80% code coverage for SPEC CPU2000 benchmarks [3].



Approach	Functions		Function Edges		Blocks		Block Edges		Bytes		Time (s)	
	M	A	M	A	M	A	M	A	M	A	M	A
IDA Pro 6.9	48	52.96	76.5	99.62	829	3589.93	1188	6487.68	14037	104779.66	1.14	1.80
angr - CFGFast	61	70.08	88	118.74	843	3609.45	1193	6538.52	14296	105007.49	0.87	5.01
IDA Pro 6.9 - reachability	37	40.96	74	90.76	496	1043.81	759	1693.01	7874	21721.85	1.14	1.80
angr - forced execution	31	33.24	48	55.22	349.5	413.85	612	751.96	6125	13963.5	23.50	36.96
angr - symbolic back traversal	32	33.76	50	56.28	368	635.41	645	1089.78	6323	10883.51	27.22	34.10
angr - backward slicing	30	32.80	47.5	53.89	344.5	653.56	594	1178.98	6109.5	14641.85	24.78	79.46

TABLE II
EVALUATION OF CFGFAST'S AND CFGACCURATE'S RECOVERED CFG VERSUS THE CFG RECOVERED BY IDA PRO. THE MEDIAN NUMBER (M) AND AVERAGE NUMBER (A) OF EACH VALUE ACROSS ALL BINARIES ARE SHOWN.

Figure 2.1: Angr's CFGFast and CFGAccurate evaluation. Figure referred from [19]

Binary instrumentation tools such as BinCFI [29] and Secondwrite [20] try to incorporate the properties of both linear and recursive disassembly to achieve complete and correct disassembly.

2.1.5 BinCFI disassembly approach

BinCFI [29] eagerly disassembles the entire binary using a linear disassembly. It then follows error detection and error correction process to ensure correct disassembly. The error correction process depends on the indirect branch target recovery. BinCFI employs following techniques to recover all possible indirect branch targets.

INDIRECT BRANCH TARGET DETECTION :

- *Stored code pointers* or *Code pointer constants* are obtained by performing a byte-by-byte search of the code and data section of the program. This process will generate an over-estimation of the actual set of stored code pointers as it is hard to differentiate between a code pointer and a data constant in a binary without any access to the source code.
- *Computed code pointers*: The authors have pointed out that although it is theoretically possible for a program to perform an arbitrary calculation on a value before using it as a target. However, such pointer arithmetic in a high-level language is not meaningful and hence is very less probable to appear in binaries compiled from high-level languages like C/C++. Such computations are only found in the case of switch-case blocks that are compiled into jump tables. The jump table entries mostly undergo a fixed format of calculation like **(C1 + ind) + C2*. BinCFI identifies the jump ta-

ble locations by performing static analysis on a window of 50 instructions ending with the computed jump instruction. The analysis is performed on all the paths within the function boundary that reach the ICF instruction. Function boundary information is obtained from the export table.

- Other computed jump targets include the *exception handlers* and the *return addresses*. The exception handlers are obtained by exploiting the eh_frame, eh_frame_hdr and gcc_except_table section of the binary. The return addresses are obtained by examining all the call instructions.

The above described techniques are conservative and will always result in an over-estimation of indirect branch target set.

ERROR DETECTION :

- *Invalid opcode*: Due to the dense encoding of x86/x86-64 CISC architecture, the probability of getting an invalid opcode is extremely small. However, if found, it certainly indicates incorrect code discovery.
- *Jump outside the current module*: Any direct control transfer outside the current module that doesn't go through the program-linkage table is considered as incorrect code discovery.
- *Jump into the middle of an instruction*: Any jump into the middle of an instruction indicates either the target is disassembled incorrectly or the jump itself is disassembled incorrectly. BinCFI considers both the possibilities.

ERROR CORRECTION : A detected error indicates the presence of data or alignment bytes that has been misinterpreted as code. BinCFI identifies the extent of these gap in application code and marks it. It considers the last unconditional jump found prior to the error bytes as the beginning and the smallest indirect control flow target that is larger than the address of the error bytes as the end of the gap respectively. BinCFI then disassembles the binary again, avoiding the marked gaps. If more errors are encountered then, the process is repeated again.

The conservative disassembly approach of BinCFI makes sure that all code locations are discovered and disassembled. However, there is a chance of some data being misinterpreted as code.

2.1.6 SECONDWRITE DISASSEMBLY APPROACH

Secondwrite [20] binary instrumentation tool employs recursive disassembly to discover all code locations reachable by direct control transfer instructions. It then speculatively treats the remaining unreachable program bytes as a possible target of ICF instructions and disassembles them. While performing the speculative disassembly it checks for the encountered errors like invalid opcode or jump into the middle of a valid instruction and discards any speculatively disassembled code sequence containing such errors. The speculative approach helps to locate all the valid code locations conservatively and may misinterpret any embedded data as code reachable by an ICF instruction. The speculative approach creates a large target set for ICF instructions and may prohibit the implementation to scale to large binaries. Hence the authors suggest below methods that can be used to further reduce the speculated target set.

- *Binary characterization*: The underlying assumption is that every ICF instruction must have an address operand that must appear within the code/data segment. Secondwrite scans the binary for stored constants. Any stored constant value in the range of the code section is treated as a valid ICF target. The speculated targets that are not found to be stored in the binary are discarded. Binary characterization cannot conclusively predict if a stored constant is an address, but it can conclusively predict if a stored constant is not an address. Hence it predicts an over-estimated set of speculated targets which is smaller than the earlier set of all unreachable code locations. As mentioned in the Secondwrite [20] paper, this helps in eliminating 99% of the speculative target.

  This method doesn't apply to scenarios where the address is computed at the runtime. For example, in PIE programs, an address may be computed as an offset from the current instruction pointer value or a return address stored on the stack. The authors mention that Position independent code is out of scope for the solution discussed in the paper and will be a part of their future work.

  After reducing the speculated target set using Binary characterization, the target set can be further reduced using *constant propagation* and/or *alias analysis*. Authors point out the constant propagation is not helpful in case of x86 architecture. So, Secondwrite employs alias analysis to further reduce the target set.
- *Alias analysis*: Alias analysis can predict if a speculated ICF target is not being loaded into the operand of an ICF instruction even if the operand is being passed through via global memory or as a function argument. Secondwrite validates each ICF instruction operands against each member of the reduced target set provided by the *Binary characterization* process. It keeps all the values that may alias an operand of an ICF instruction and discards the rest.

JUMP TABLE HANDLING : Secondwrite assumes that the jump tables are present as an array of stored code pointers and Secondwrite's Binary characterization process can identify them as stored code pointers. This assumption can fail in case of PIE executables where jump table entries are not absolute addresses and are offsets from a base address.

The speculative approach of Secondwrite can misinterpret some embedded data as code. Hence, to protect embedded data, Secondwrite keeps the original code and data unchanged and creates a new section for instrumented code.

2.2 Preserving control flow branches

Instrumentation results in introduction of new code and relocation of the original code. As a result, the control flow branches (Both direct and indirect) need to be fixed up.

2.2.1 Fixing direct control flow branches

Direct control flow branch instructions have their target embedded within the instruction. Hence, retrieving the direct branch targets is trivial. Direct branch targets can be fixed up in the following ways.

- *Patching the branch instructions:* The branch instructions can be patched to point to the new location of relocated target. However, this can be inefficient in cases where the original jump is a short jump and needs to be converted to a 5-byte long jump to account for the relocated target. This will require the instructions around the jump to be moved to make space for the 5-byte jump.
- *Symbolizing all targets and re-assembling:* A simpler way to deal with direct branches can be to associate a symbol with the direct branch targets. The direct branch instructions can be modified to point to these symbols and the code can be re-assembled. By doing so, the assembler will automatically fix up the direct branches. This approach is used by static binary instrumentation tools such as BinCFI [29].

2.2.2 Fixing indirect control flow branches

Pre-translating the indirect branch targets would be the best solution to deal with indirect branches. However, pre-translation requires accurate identification of all indirect branch targets with zero false positives. CCFIR [28] exploits the relocation information available with the Windows PE binaries to locate and translate the pointers statically. However, the availability of relocation information is not a certainty in non-PIE binaries of Unix systems do not have any relocation information. Ramblr [24] employs a best effort static analysis approach to classify pointers and data that doesn't guarentee the recovery of all indirect branches. Tools such as BinCFI [29] and Secondwrite [20] generate an over-estimation of indirect branch targets. In such a case, pre-translating the identified targets can be a program. Because, changing something that is not a code pointer can lead to malfunction. One solution can be to employ usage point address translation (i.e., to translate an indirect branch target just before the indirect branch is executed).

usage point address translation: Tools such as BinCFI [29], Secondwrite [20] and Binary-stirring [26] instrument every indirect branch instruction to do a runtime address lookup. BinCFI and Secondwrite use a look-up table that contains a mapping for all the pre-identified indirect branch target, to their corresponding new address. On the other hand, Binary-stirring [26] uses the original code section as a look-up table. The old address in the original code section is patched with the new address and the indirect branches are instrumented to look-up the new address just before branching. This approach introduces comparatively less performance overhead as compared to the table look-up approach of BinCFI and Secondwrite. Evaluation performed on SPEC CPU 2000 benchmarks shows that the overhead due to runtime address translation is 2.12% and is higher in case of C++ programs where it reaches to a value close to 20%. BinCFI's address translation introduces an overhead of 8.54%.

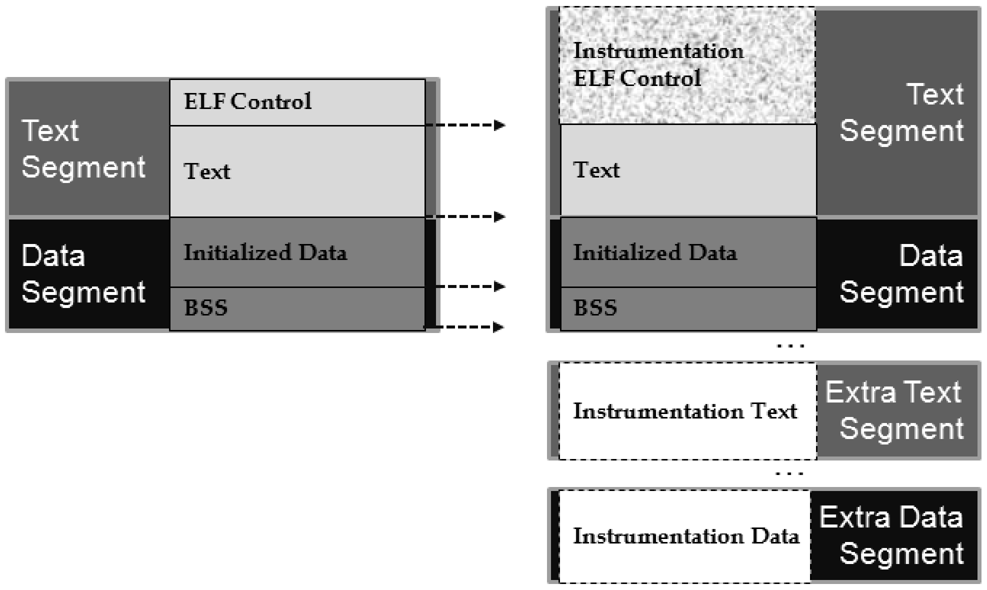
In place patching: One way to avoid runtime address translation is to make sure that the code locations are not changed by instrumentation process. A common technique used on fixed-length platforms to transfer control to the instrumentation code is to replace a single instruction at the instrumentation point with an unconditional jump and moving the replaced instruction to the instrumentation code site. Advantages are:

- Doesn't cause any change to the code and data location and sizes and thereby nullifying the necessity of changing any direct or indirect code and data references.
- Unlike other approaches that need runtime address translation, this approach is more efficient and causes less performance overhead as the only performance overhead that will be introduced is due to the execution of the instrumentation code and the extra jump instruction.

However, it is not so straight forward in platforms with variable-length instruction sets, such as x86 and x86-64 architectures. An unconditional branch instruction that uses a 32-bit offset requires 5 bytes of memory. As instructions can be as small as 1 byte, instrumentation points may not have enough space to put a 5-byte jump instruction. A viable solution can be to put a short jump (2 bytes long) to a nearby location that has a long jump into the instrumentation code. However, there is still a chance that we may not have 2 bytes of space at all instrumentation points. Furthermore, there is a chance that we may not have an additional space of 5 bytes at a nearby location to put the required long jump. Another option is to put a 1-byte interrupt (int3) instruction. This sounds perfect, but can be less efficient. Frequent invocation of heavyweight system call conventions can add to performance overhead [13].

PEBIL [13] is one such tool that avoids the use of runtime address translation. PEBIL relocates and reorganizes code at function level. PEBIL's instrumentation steps (Figure 2.3) can be summarized as below:

- New segments are allocated for instrumentation code and data by statically modifying the ELF control structures (Figure 2.2).
- Functions containing instrumentation points are relocated to the instrumentation code segment and link the original entry point with a jump to the relocated function.
- Relocated function is re-organized to create extra space at instrumentation points to accommodate instrumentation code. Note that the figure 2.3 shows addition of extra jump at instrumentation points. However, PEBIL has an option to inline the instrumentation code.
- All the intra-function direct branches need to be adjusted to account for the code



(a) Layout of an unmodified ELF file.

(b) Layout of a PEBIL-instrumented ELF file.

Figure 2.2: Figure referred from [13]

relocated by step 2. All the intra-function short jumps are first converted to 5-byte long jumps to accommodate the targets that may have been relocated to a place out of range for a short jump. The jumps are then patched to point to new targets.

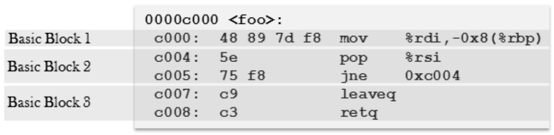
- If the function has any jump table associated with it, then the jump table is re-created accordingly. Note that the jump table identification by PEBIL has been discussed in section 2.1.3.

The extra jump instruction that every call to a relocated function has to go through and the conversion of all the intra-function short jumps to 5-byte jumps can create extra performance overhead. To quantify the overhead, the performance was tested on SPEC CPU2000 Integer Benchmarks with null instrumentation. The average overhead is reported to be 1.2% with a maximum of 4.8%. With basic block counting instrumentation, PEBIL was found to have an overhead of 28%-111% with an average of 62%.

2.2.3 Callbacks

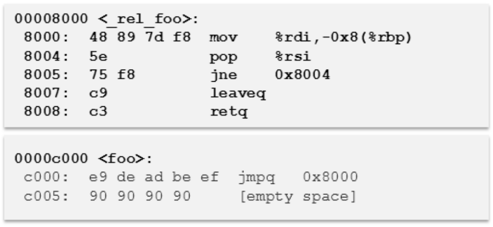
In some cases, a function pointer is passed to an external module which uses the pointer to place a call. One such example is the passing of a pointer to *main* function to the standard library function *_libc_start_main*. Such cases pose a challenge for the static binary instrumentation tools that employ runtime address translation. BinCFI handles such cases by instrumenting all the external libraries used by a program.

Another example where a callback is used is signal handling. In case of signal han-



```
0000c000 <foo>:
 c000:  48 89 7d f8   mov    %rdi,-0x8(%rbp)
 c004:  5e            pop    %rsi
 c005:  75 f8         jne    0xc004
 c007:  c9            leaveq
 c008:  c3            retq
```

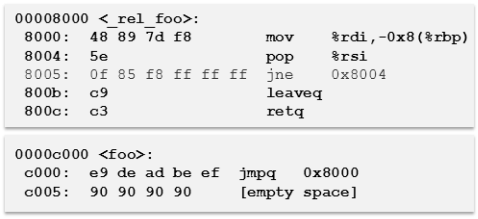
(a) An unmodified application function.



```
00008000 <_rel_foo>:
 8000:  48 89 7d f8  mov    %rdi,-0x8(%rbp)
 8004:  5e           pop    %rsi
 8005:  75 f8        jne    0x8004
 8007:  c9           leaveq
 8008:  c3           retq

0000c000 <foo>:
 c000:  e9 de ad be ef  jmpq   0x8000
 c005:  90 90 90 90     [empty space]
```

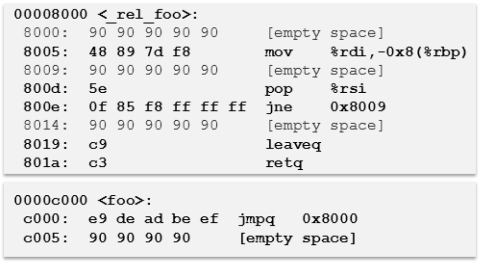
(b) The application function after it has been relocated and the old function entry has been linked to it.



```
00008000 <_rel_foo>:
 8000:  48 89 7d f8         mov    %rdi,-0x8(%rbp)
 8004:  5e                  pop    %rsi
 8005:  0f 85 f8 ff ff ff   jne    0x8004
 800b:  c9                  leaveq
 800c:  c3                  retq

0000c000 <foo>:
 c000:  e9 de ad be ef  jmpq   0x8000
 c005:  90 90 90 90     [empty space]
```

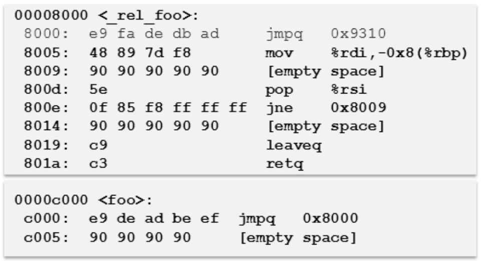
(c) The application function after the branches have been converted to use 32-bit offsets.



```
00008000 <_rel_foo>:
 8000:  90 90 90 90 90      [empty space]
 8005:  48 89 7d f8         mov    %rdi,-0x8(%rbp)
 8009:  90 90 90 90 90      [empty space]
 800d:  5e                  pop    %rsi
 800e:  0f 85 f8 ff ff ff   jne    0x8009
 8014:  90 90 90 90 90      [empty space]
 8019:  c9                  leaveq
 801a:  c3                  retq

0000c000 <foo>:
 c000:  e9 de ad be ef  jmpq   0x8000
 c005:  90 90 90 90     [empty space]
```

(d) The application function after it has been padded with 5 bytes at each instrumentation point.



```
00008000 <_rel_foo>:
 8000:  e9 fa de db ad      jmpq   0x9310
 8005:  48 89 7d f8         mov    %rdi,-0x8(%rbp)
 8009:  90 90 90 90 90      [empty space]
 800d:  5e                  pop    %rsi
 800e:  0f 85 f8 ff ff ff   jne    0x8009
 8014:  90 90 90 90 90      [empty space]
 8019:  c9                  leaveq
 801a:  c3                  retq

0000c000 <foo>:
 c000:  e9 de ad be ef  jmpq   0x8000
 c005:  90 90 90 90     [empty space]
```

(e) The application function after a single basic block (Basic Block 1) has been instrumented.

Figure 2.3: PEBIL-function relocation. Figure referred from [13]

dling, a program registers a signal handler against a particular signal by making the "signal/sigaction" system call. the "signal/sigaction" system call takes a pointer to a handler function as an argument. When a signal is delivered to the program, the control passes to the OS and OS calls the registered handler function. BinCFI intercepts sigaction and signal system calls ang changes the signal handler argument.

Secondwrite's [20] solution for callbacks can be summarized as below.

- *Identifying function pointer arguments:* This approach involves collecting the prototypes of all the functions present in commonly used shared libraries such as the C runtime libraries and detecting calls to all such functions that take a function pointer as an argument and instrumentation code is added to perform a runtime translation .
- *segmentation handler:* For cases where the prototype of a given function is not avail-

able, any callback from that function will land in the original code. The original code is marked as non-executable and hence will result in a segmentation fault. Secondwrite registers a custom segmentation handler which translates the original code address to the instrumented code address using the look-up table.

3 Dynamic Binary Instrumentation

Dynamic binary instrumentation tools perform disassembly, analysis and code transformation while the program executes. Performing instrumentation at the runtime can have the following advantages:

- Unlike static binary instrumentation, DBI tools do not face the content classification problem. Following the program execution and disassembling on the fly helps in getting around the data embedded in code.
- complete coverage of code without any source code or supplemental information.
- No need of altering the program binary.

However, performing disassembly and instrumentation at the runtime can add to the performance overhead.

The standard components of a DBI tool are (i) JIT (Just in time) compiler that disassembles, optimizes, instruments and reassembles the code and (ii) a Code Cache that stores the instrumented code to be rerun when necessary. The DBI tool either injects itself into the client program's memory or loads the client program into its memory. After gaining control of the target program, it starts translating, instrumenting and executing the client's code one block at a time. The definition of code block varies from tool to tool. Depending on the just in time code transformation technique, the DBI tools can be further classified into two categories:

- *Assembly to assembly transformation*: The machine code is translated to assembly form. The original code is kept mostly unchanged except the changes to branch instructions. Each instruction is annotated with a description of its effects. Annotations are used by the tool to guide instrumentation. Instrumentation can be done either by placing a call to the analysis code or by in-lining the analysis code. The instrumented code is re-assembled back to machine code and executed.
- *Disassemble and resynthesize*: Machine code is converted to an IR. This IR is instrumented and then recompiled back to the machine code. The original code is discarded. The effects of every instruction need to be accurately reflected by the IR to ensure correct code generation and accurately guide the instrumentation process.

3.1 Assembly to assembly transformation - a case study of PIN

Pin [14] was designed to provide a robust, portable and easy to use instrumentation tool that can overcome challenges faced by Static instrumentation tools, such as data between code and indirect branch instructions. By deferring code discovery and instrumentation to the runtime, PIN successfully overcomes these challenges.

Pin consists of a virtual machine, a code cache and an instrumentation API used by Pintools. Pintools are instrumentation tools designed PIN users to carry out specific instrumentation tasks such as instrumentation every load/store instructions, etc. Components of VM are:

- *JIT compiler:* Translates and instruments client's code at the runtime.
- *Dispatcher:* Dispatcher launches the translated code stored in the code cache. It takes care of storing and restoring the client application's registers while entering/leaving the VM from/to the code cache.
- *Emulation unit:* Takes care of special cases such as system calls that can not be executed directly from the client's code.

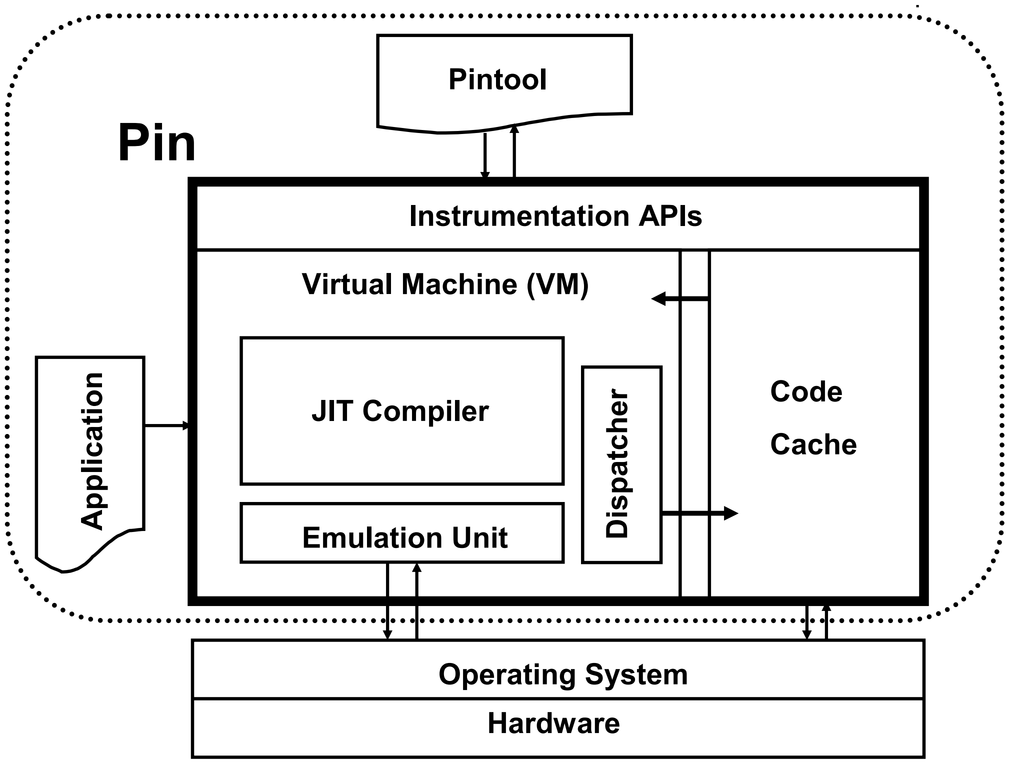


Figure 3.1: PIN- system overview. Figure referred from [14]

PIN'S EXECUTION OVERVIEW: Pin's injector gains control of the client application using UNIX ptrace API and loads Pin into the client's address space. After the loading and initialization of both Pin and pintool, Pin starts the client applications and starts jitting from the entry point. So, during the runtime three applications share the same address space, Pin, pintool and the client. Sharing of the same standard library between the 3 processes can lead to complications. One such example provided by the authors is about non-reentrant functions. If the client enters a non-reentrant function and needs to translate a code block, the control will fall back to the JIT compiler. If the JIT compiler also calls the same function, then it will try to enter the same function while the client is still inside the function, causing an error. Therefore PIN is designed to use three separate copies of the standard library, one each for PIN, pintool and the client.

One viable technique to inject the DBI tool into the client's address space is to use LD_PRELOAD environment variable in LINUX, as done by tools like DynamoRIO [6]. However, this approach will not work for statically linked binaries. Also, injecting an extra library may cause other libraries to be shifted to a higher address. To maintain transparency and keep the application process identical to the original behavior, the Pin's designers avoided using this approach. Also, using LD_PRELOAD will not allow Pin to gain control of the client until after the loader is partially executed.

JIT COMPILATION AND INSTRUMENTATION: Pin compiles from one ISA (Instruction set architecture, e.g. x86) directly into the same ISA without going through an intermediate format. The client code is compiled one *trace* at a time. Pin defines a *trace* as a linear block of code that terminates when (i) an unconditional control transfer instruction is encountered, or (ii) a fixed number of conditional branch instructions is encountered, or (iii) a fixed number of instructions have been fetched. So, a trace can have multiple exits. Each exit point is redirected to a stub that in turn redirects the control to the VM. The VM determines the next target, generates a *trace* if the target has not been translated yet and resumes execution.

Pin's instrumentation API makes it possible to observe the client's state such as the register and memory contents, control flow branches, etc. The user writes procedures called *analysis routine* and writes *instrumentation routine* to determine instrumentation points and place calls accordingly. The JIT compiler calls the *instrumentation routine* after it has generated a *trace*. The *instrumentation routine* traverses the *trace* and places a call to the analysis routines at appropriate instrumentation points. Figure 3.2 shows an example of the code that the user has to write to perform instrumentation. Here, *RecordMemWrite* is the analysis routine and *Instruction* is the instrumentation routine. The JIT compiler calls the *Instruction* function to perform instrumentation. Such a code is compiled to create a *Pintool*. Note that, to prevent the inserted code from overwriting the client's scratch registers, efficient register saves and restores needs to be performed at all the calls to analysis code. Also, to guide the instrumentation (placing calls at instrumentation process) PIN has to annotate every instruction with an appropriate description of their effects such as register reads/writes, stack change, memory read/write, etc.

TRACE LINKING: Trace linking is essential for performance enhancement. It refers to the process of directly branching to the target trace at the end of the execution of a trace bypassing the VM. This is trivial in case of direct branch instructions where the target is statically known and can be replaced with the address of the corresponding translated trace.

```
FILE * trace;

// Print a memory write record
VOID RecordMemWrite(VOID * ip, VOID * addr, UINT32 size) {
    fprintf(trace,"%p: W %p %d\n", ip, addr, size);
}

// Called for every instruction
VOID Instruction(INS ins, VOID *v) {
    // instruments writes using a predicated call,
    // i.e. the call happens iff the store is
    // actually executed
    if (INS_IsMemoryWrite(ins))
        INS_InsertPredicatedCall(
            ins, IPOINT_BEFORE, AFUNPTR(RecordMemWrite),
            IARG_INST_PTR, IARG_MEMORYWRITE_EA,
            IARG_MEMORYWRITE_SIZE, IARG_END);
}

int main(int argc, char *argv[]) {
    PIN_Init(argc, argv);
    trace = fopen("atrace.out", "w");
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_StartProgram(); // Never returns
    return 0;
}
```

Figure 3.2: Example of a Pintool. Figure referred from [14]

For indirect branches, Pin uses a prediction mechanism. It replaces the indirect branch instruction with a *"mov indirect_address, %edx"* and direct jump to a predicted target. At the target site, the %edx register is matched. If the match succeeds the execution continues else control jumps to the next predicted target. If none of the predicted targets match, the control finally jumps to a look-up stub that looks up the target in a hash table. If the target is still not found, then the control falls back to the VM to create a *trace*. This prediction chain is created dynamically and grows as new targets are discovered in the runtime. Example shown in Figure 3.3

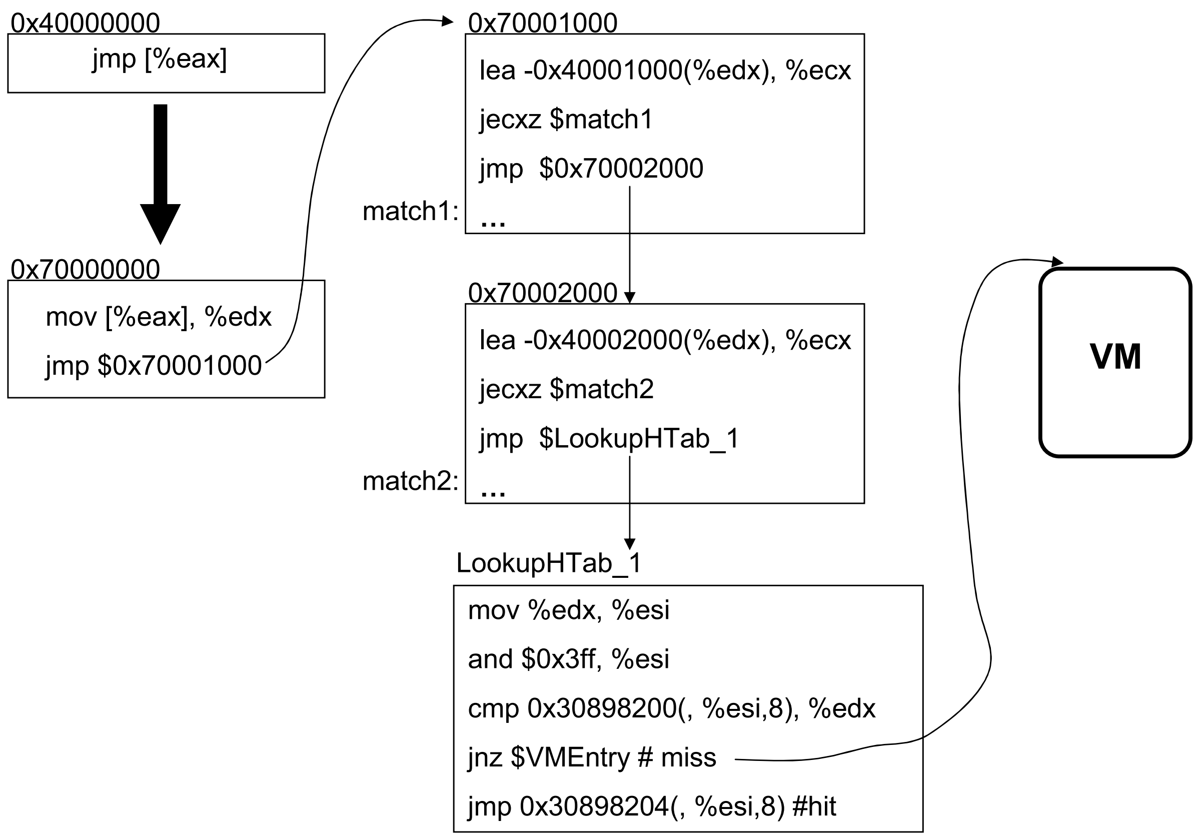


Figure 3.3: Pin's trace linking. Figure referred from [14]

To optimize the target match for *returns*, Pin uses *cloning* (Figure 3.4). A function can be targeted by multiple call-sites. PIN creates a separate clone of the function for every call-site. Hence the prediction chain for every return will essentially have just one target.

REGISTER RE-ALLOCATION: Pin's compilation process needs few registers to be free. As it can be seen in Figure 3.3 the trace linking process requires three free registers (edx, ecx and esi). Also when a call is placed to the analysis code, free registers are needed to pass arguments. Instead of obtaining the registers in an ad-hoc manner, Pin re-allocates registers in both client code as well as analysis code using linear scan register allocation. Because of just in time compilation and no prior knowledge of the program control flow, Pin has to perform an iterative liveness analysis and register re-allocation, one trace at a time. The liveness analysis result of each trace is stored in a hash table. If the target of a trace is statically known, then the liveness information of the target trace is obtained to compute more precise liveness information of the current trace.

Because of trace linking, the reconciliation of register bindings of the current trace and target trace is essential. If the target has not been compiled yet, then the target is recompiled using the parent trace's register binding. If it has already been compiled and the register bindings of the current and target trace differ, then the reconciliation code is

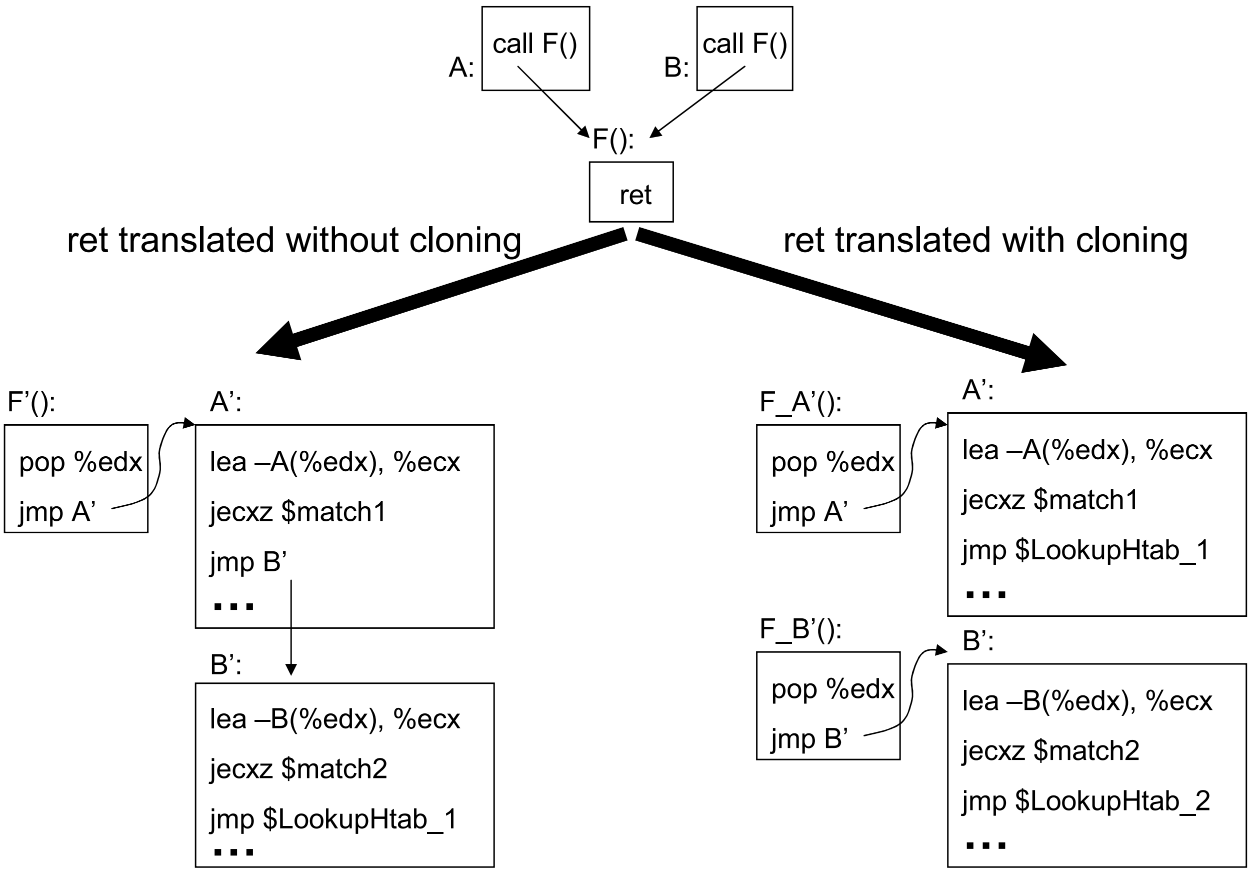


Figure 3.4: Pin's trace linking. Figure referred from [14]

added before the branch. The reconciliation code is a set of *mov* instructions that copy the values from the virtual/re-allocated registers of the current trace to the virtual registers of the target trace.

Performance evaluation shows that, with null instrumentation, Pin has a 60% overhead for SPEC2000 integer benchmark and 5% overhead for SPEC2000 floating point benchmark. The authors point out that high overhead for integer benchmark is due to the presence of a larger number of indirect branches in the integer benchmark binaries. This shows that the major reason for performance overhead is the runtime address translation.

3.2 DISASSSEMBLE AND RESYNTHESIZE - A CASE STUDY OF VALGRIND

Valgrind [15] comes under a set of tools known as the *Shadow value tools* and is designed to perform heavyweight dynamic binary analysis. Shadow value tools maintain a shadow of every register and memory location. Each shadow value records information about its corresponding value's history. Such shadow value tools can be used to track undefined bit values (uninitialized or derived from undefined values), taint tracking, etc. As pointed out by the authors of Valgrind [15], maintaining shadow values requires some standard steps to be followed. Such as:

- Provide and maintain shadow registers just like the corresponding normal registers.
- Provide and maintain shadow memory for every memory location. Access must be controlled to provide safety in multi-threading environment.
- Instrument every read/write instruction that access register or any memory location.
- Instrument system calls as almost every system call access registers, stack and memory locations.
- Intercept memory allocations done at program start-up.
- Instrument system calls that allocate and de-allocate memory (e.g. brk, mmap).
- Instrument stack allocation and de-allocation. This can be expensive as the stack pointer is updated very frequently in a program.
- Heap allocators present in standard libraries such as Glibc, handout heap blocks from larger chunks and maintain book-keeping information. A shadow value tool must track the heap allocations and de-allocations done at library level and mark book-keeping data as non-active as this book-keeping data shouldn't be accessed by the client program.
- Keep a side channel to output information like less used file descriptors and files, etc.

VALGRIND'S EXECUTION OVERVIEW :

- Instead of injecting itself into the client program's address space, Valgrind launches itself first and then loads the client program into its address space and doesn't rely on the system's dynamic linker and loader. This gives Valgrind better control over the memory layout of the client.
- Valgrind itself runs on the host/real CPU and uses the host registers. While the client program is run on a simulated or guest CPU and it uses the guest/simulated registers. Valgrind assigns a memory block called *Threadstate* to each thread of the client program. The *Threadstate* is used to hold the thread's guest and shadow register values.
- Valgrind translates code blocks on demand. A code block is a linear set of instructions ending when (i) an instruction limit is reached or (ii) a branch to unknown target is hit or (iii) more than 3 unconditional branches to known targets are hit. Valgrind's code translation process consists of 8 steps that translate machine code to IR, optimize the IR, add instrumentation, perform register allocation and assemble back to machine code. Valgrind's IR is architecture neutral.
- Once translated, the translations are stored in a fixed size, linear-probe hash table.
- Once translated a translation can be executed. At the end of the execution of a translated block, all the register values have been written back to the threadstate and the control passes to the *dispatcher*. The dispatcher looks for the next translation in a small direct-mapped cache. If found, the execution continues, else the control passes to the *scheduler*. The scheduler looks for the address of the next translation in the

hash table and adds it to the direct-mapped cache. The control passes back to the dispatcher and this time it succeeds in finding the target translation. Valgrind doesn't support chaining of translated blocks like other DBI tools. This can affect the performance because of frequent visit to the dispatcher. However, authors mention that this hurts the performance less than expected because Valgrind's dispatcher is fast and Valgrind chases across many unconditional jumps during translation.

HANDLING SPECIAL CASES :

- **System calls**: System calls read and write from registers, stack and memory locations. But Valgrind can not trace into the kernel. Hence, to keep track of system calls, the control falls back to the Valgrind's scheduler whenever a system call happens. The scheduler copies the guest registers into host registers and makes the system call for the client. When the system call returns, the scheduler copies the host registers back to guest registers and passes the control to the client.

  Also, the system calls involving resources such as memory, file descriptors, etc needs to be pre-checked as the client shouldn't access the tool's resources. If any such case happens, Valgrind aborts the system call without consulting the kernel.

- **Threads**: Threads pose a challenge as the load/store instructions no longer remain atomic because every load/store to memory is now associated with corresponding load/store to the shadow memory. It is difficult to make sure that the shadow locations are accessed in the same order by the threads as the original memory locations. This is true for both the uni-processor and multi-processor systems. To deal with this problem, Valgrind serializes the thread execution. Only one thread holding a lock will be executed while the rest of the threads will remain in a blocked state. The active running thread drops the thread whenever it makes a blocking system call or it has been running for a while. In a sense, although the kernel chooses which thread to run next, Valgrind dictates the thread switch.

- **Signals**: When a program sets a signal handler, it gives the kernel a callback that is used to deliver the signal to the program. As the call to the handler is made from the kernel, this may let the client execute natively. Worst, if the signal handler doesn't return and does a long jump instead. The tool would lose control completely. Hence Valgrind intercepts all system calls that register signal handlers, keeps a note of the address of the client's handler and registers its own handler. When a signal arrives the Valgrind creates the signal frame and runs the client's handler on the simulated CPU. If the handler is observed to return, Valgrind removes the frame from the client's stack and resumes the client's execution from where it was before the signal was delivered. Valgrind delivers asynchronous signals in between translation block execution so that the load/store instructions remain unaffected.

REASON'S FOR OPTING IR OVER ASSEMBLY: As mentioned earlier, supporting shadow values requires heavyweight instrumentation such as instrumenting all the register accesses, stack accesses, load/store instructions, system calls, memory allocation/deallocation, etc.

An IR representation makes all the side effects of instructions explicit and hence makes it easier to accurately detect and instrument all the required instrumentation points. Whereas it may be difficult to recognize all the side effects of an instruction from its machine code or assembly representation.

The Valgrind's IR representation breaks complex machine instructions that carry out multiple operations into simpler single operation instructions. Hence it exposes all implicit intermediate values such as memory addresses calculated by complex addressing modes. Therefore it makes tracking and updating all the shadow values easier.

All code (instrumentation as well as client program code) are represented using the same IR which means that client code and complex instrumentation code (required to maintain shadow values and perform analysis) can be interleaved in an arbitrary manner and optimized equally well without worrying about analysis code perturbing condition codes of client program or spilling garbage into register values of client program.

Finally, the use of temporaries in IR makes shadow value manipulations easier.

CONS OF DISASSEMBLE AND RESYNTHESIZE APPROACH: Using IR requires additional effort to ensure that the compilation phase generates good code. Whereas using *Copy and annotate* approach ensures that good client code stays good with less effort.

Performing *machine code->assembly->IR->assembly->machine code* translation along with other operations such as code optimization and register allocation during the runtime affects the performance overhead and is not suitable for lightweight instrumentation. Evaluation on SPEC CPU2000 benchmarks showed that Valgrind's null instrumentation slowed down by a factor of 4.3x. Heavyweight instrumentation such as *Memcheck* slowed down by a factor of 22.2x.

4 SECURITY POLICY ENFORCEMENT USING INSTRUMENTATION

Memory corruption vulnerabilities are the most commonly exploited software vulnerabilities. These exploits initially took the form of malicious code that was injected into the stack or the heap area, with control redirected to this code by corrupting a return address on the stack or a function pointer. No-Execute memory protection schemes such as DEP was employed to stop attackers from executing any injected code. This gave rise to new exploitation techniques known as code reuse attacks. Return to libc [21] is one such attack in which the attacker redirects the control flow to an existing library function such as system("/bin/sh"). ROP [7][8] is another form of code-reuse attack in which the attacker chains together short sequences of instructions already present in the code segment. The dense encoding of CISC architecture makes it possible to construct turing complete gadgets using the instruction sequences already present in the code segment. The fixed address of the executable segments made it easier for the attackers to guess the location of such code gadgets. To frustrate the attackers ASLR [4] was introduced. ASLR randomizes the location of every executable segment, thereby making it harder for the attackers to find

Direct control flow transfer	**Indirect control flow transfer**
396d: call 3290	3b4c: jmp *%rax
3962: je 3972	3890: jmp *0x21c75a(%rip)
362b: jmp 3230	14788: ret

Table 4.1: Examples of control flow transfer instructions obtained from objdump output of ls

and jump into the code of choice. However a lot of executable with fixed load addresses are still in use in the real world. Also, attackers can by-pass ASLR by leaking pointer values using information leakage vulnerabilities or by brute force. Defense mechanisms against such code reuse attacks include:

- **bounds checking:** to restrict attackers from exploiting buffer overflow vulnerabilities.
- **Stack canary** and **shadow stack:** to prevent attackers from corrupting the return addresses stored on the stack.
- **CFI:** to prevent redirection of the control flow of the program.
- changing the semantics of code gadgets by employing code transformations.
- employing fine-grained code randomization to break the attacker's assumption about the location of code gadgets, etc.

Some of these techniques, e.g., stack canaries, have already been widely deployed. Among the rest, CFI and code randomization are the two most amenable to binary instrumentation, so we describe them in this section

4.1 Control Flow Integrity

Control flow integrity [29][28] provides a foundation for enforcing low-level security policies on binary code. It has also been used extensively to mitigate control flow hijack attacks such as ROP and JOP. In code reuse attacks the attacker exploits memory corruption vulnerabilities to redirect the control flow to attacker-chosen code gadgets. Code gadgets are a sequence of instruction bytes ending with an indirect control transfer instruction *ret/jmp */call **. These code gadgets can be present anywhere in the code segment. With the knowledge of the executable format, the attacker can statically analyze the code segment and predetermine the sequence of bytes that he can use as gadgets. One way to prevent such attacks would be to prevent the attacker from jumping to any arbitrary location.

control flow transfer instructions in a binary are responsible for changing the execution flow of a program either conditionally or unconditionally. These instructions can be broadly classified into two categories. Direct control flow instructions and indirect control flow instructions (Table 4.1). Since code is non-writable in contemporary operating systems, attackers cannot manipulate the target of direct control flow transfer instructions and hence they must rely on indirect control flow transfer instructions.

Indirect control flow instructions can further be subdivided into two groups, forward edges (*jmp */call **) and backward edges (*ret*). C++ virtual method calls, calls using function pointers and switch case blocks are translated to use indirect calls and jumps. Targets of these calls and jumps are computed at the runtime and are stored in some register or memory location. Targets of return instructions are pushed on to the stack by the call instruction preceding the return in the control flow path. The attacker can take advantage of any memory corruption vulnerability and modify the targets of such indirect branch instructions.

Implementing CFI involves statically computing the control flow graph (CFG) of a program and adding instrumentation to perform run time validations on the computed jump targets such that the program doesn't violate the pre-computed CFG. Effective implementation of CFI depends on:

- *Accurate code discovery*: Every indirect control flow instruction needs to be discovered and instrumented so that their targets can be validated at runtime. At the same time misidentifying embedded data as code and instrumenting it can break the program.
- *Accuracy of control flow graph*: Effectiveness of a CFI technique is determined by the preciseness of the CFG which in turn depends on the indirect branch target recovery. The granularity of a CFI depends on:
  - Size of the possible target set for each indirect branch instruction (the closer it is to actual target set, the better it is).
  - Number of false positives in the target set (the lesser the better).

In short, binary based CFI implementations suffer from problems that inherently come with static binary instrumentation. Many CFI solutions are implemented within compiler tool chain [27] [5][25][2] and rely on semantic information that are not available in binaries. The first CFI technique introduced by Abadi et. al. [1] works on binary and relies on relocation information to recover indirect branch targets. However, it is coarse-grained as its CFG allows every indirect branch (*ret/jmp */call **) to target all address taken locations. CCFIR [28] follows a similar approach and relies on relocation information. However, it is more fine-grained as it generates separate target set for *jmp */call ** and *ret* instructions. **BinCFI**[29] is another CFI technique that operates directly on binary and doesn't rely on relocation or any other supplemental information. In the following section, we will discuss BinCFI and CCFIR.

4.1.1 Related works on CFI

- **BinCFI**[29] statically disassembles and instruments the computed jump instructions to validate the computed targets during runtime. It deals with the two challenges described above as follows:
  - *Accurate code discovery*: BinCFI's disassembly process has been discussed in section 2. The linear disassembly followed by the error correction process makes sure that all possible code locations are disassembled. It protects any misinter-

	Returns, Indirect jumps	**PLT Targets, Indirect Calls**
Return Addresses	Y	
Exception Handling Addresses	Y	
Exported Symbols		Y
Code Pointer constants (stored pointers)	Y	Y
Computed Code address (Jump table targets)	Y	Y

Table 4.2: BinCFI model. Figure referred from the BinCFI paper[29].

preted data by keeping the old code section unchanged.

- *Accuracy of control flow graph*:
    - *False positives in indirect branch target set:* As discussed in chapter 1, BinCFI's indirect branch target detection process generates an over-estimation of possible target set. So, it may have some false positives but doesn't have any false negatives. This is essential for preserving functionality of a program.
    - *Size of indirect branch target set*: In complex binaries there can be several exceptions. For example, compiler optimization can result in a function being reached by a *jmp* instead of a *call* instruction, *returns* being used as jumps (signal handling, thread switch, etc), jumps being performed on return addresses (longjmp), etc. Such cases prohibits any general assumption regarding the target set of an indirect branch instruction. Hence, BinCFI's CFG generates a coarse-grained target set (Table 4.2).

BinCFI's runtime address translation technique helps in enforcing the policies mentioned in table 4.2. BinCFI is an effective CFI solution for stripped COTS binaries without any symbolic, debugging or relocation information. It certainly raises the bar for attackers, but the conservative approach leads to a relaxed CFG that leaves some space for the attackers to exploit.

- **CCFIR** [28] on the other hand depends on relocation information present in Windows DLLs to discover the targets of computed jump instructions.
    - *Code discovery*: CCFIR supplements its recursive disassembly with the code pointers recovered from the relocation table. Hence, it can achieve complete disassembly for Windows DLLs. It makes a few assumptions regarding the binary:
        - No instruction pointer relative pointer computation.
        - Jump tables are an array of absolute addresses and can be accessed via relocation.
        - Only data present in the code section are jump tables. Hence, using the

	Indirect Calls/ Jumps	**Returns**	**Returns to sensitive functions**
Exported Functions	Y		
Relocated code pointers	Y		
Return Addresses		Y	
Return Addresses in Sensitive Functions			Y

Table 4.3: CCFIR model

relocation table can help in locating all data within code.

Such assumptions are not applicable everywhere. For example, x86-64 bit binaries often use instruction pointer relative code pointer computation which cannot be resolved simply by exploiting relocation information. Also, the jump table computation in Linux ELF binaries is based on relative offsets from a base address and does not require relocation.

– *Accuracy of control flow graph*:

  ∗ *False positives in indirect branch target set:* Relying on relocation information helps CCFIR achieve zero false positives for Windows DLLs.

  ∗ *Size of indirect branch target set*: CCFIR segregates the indirect branch targets into finer subsets for each type of indirect branch instruction (*jmp */call ** and *ret*). Indirect calls and jumps are allowed to target exported functions and relocated code pointers. Whereas returns are allowed to target only valid return addresses 4.3. CCFIR's strict policies don't leave any room for exceptional cases such as returns being used as jumps and jumps targeting return addresses.

To protect return addresses, CCFIR relocates the call instructions. Doing so affects the transparency of the instrumented program which can break functionalities such as exception handling that depend on code uniformity.

The inability to decode the computed jumps forces the CFI techniques to be conservative to preserve the program behavior. This affects the strictness of CFI leaving some space for the attacker to launch an attack. Typearmor [23] implementation tries to make the computed CFG stricter by further comparing the call-site signature with a callee prototype. However, getting the argument count of the callee and the caller from a binary is difficult. Typearmor uses a conservative static analysis to determine an over-estimation of argument count at the call site and an under-estimation of argument count at callee. It allows call-sites to target the callees with argument count less than or equal to that of the call-site. Additionally, it scrambles the unused registers at call sites to prevent data flow between code gadgets. The prototype matching along with scrambling the unused registers at call sites certainly raises the bar and can prevent COOP [18] and other known forms of code reuse attacks. But the *less than/equal to* comparison can reduce the target set roughly by 50% only and hence, leaves some space for the attacker to launch an attack.

4.2 Code Randomization

Code gadgets are a short sequence of instructions that perform a specific operation such as load value to a register or pop a value from a register, followed by an indirect jump instruction or a return. Code gadgets can be classified into two categories, (i) Intended gadgets and (ii) Unintended gadgets (Figure 4.1). An intended gadget is a part of the execution flow of the program. Whereas the unintended gadgets start at a misaligned address which is not reachable by the normal execution flow of the program. Possible solutions to prevent the attacker from using a gadget can be:

- *Changing gadget semantics:* Changing the semantics of intended gadgets without affecting the behavior of the program is not possible. However, it is possible to break the unintended gadgets by replacing the instructions of which they are a part of, with semantically equivalent instructions. Compiler extensions such as g-free [16] achieves this via a series of instruction transformation such as register reallocation, instruction transformation, and jump offset adjustment. Achieving the same in binaries can difficult because of the content classification problem (code vs data). In the subsequent section we will discuss IPR [17] that performs similar transformations on binary.
- *randomizing code location:* Randomizing individual units of a program such as basic blocks or functions will make it harder for the attacker to guess the location of preferred gadgets or to change the semantics of the gadget. Performing fine-grained randomization on binary comes with the inherent challenges of binary instrumentation. In the subsequent section we will be discussing binary stirring [26] and ILR [11] that perform fine-grained randomization using static and dynamic binary instrumentation respectively.

4.2.1 Related works on code randomization

Binary based randomization techniques must take care of the following points:

- *Accurate disassembly*: Any data misidentified as code and modified by the randomization process can result in malfunction.
- *Indirect branch target translation*: Code randomization can result in relocation of code. Hence, indirect branch targets need to be identified and updated to preserve the program functionality.

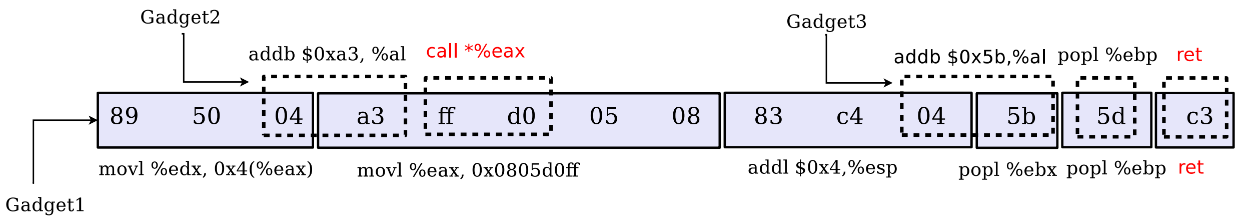


Figure 4.1: Example of gadgets. Figure referred from gfree paper [16]

- *Transparency:* Functionalities like exception handling depend on code uniformity and hence, can be affected by the randomization process.

Dynamic binary instrumentation can help get around disassembly and indirect branch target recovery problem. **Instruction location randomization (ILR)** [11] uses dynamic binary instrumentation to randomize the location of every instruction.

- *code discovery and randomization:* Even though ILR performs a static/offline disassembly and randomization, the instructions are not relocated until runtime. Therefore, any misidentified remains unharmed as it will never be reached by the control flow of the program and hence will not be relocated. This opens a door for ILR to perform an exhaustive disassembly and discover every possible code. After disassembly, ILR generates a set of *rewrite rules*. The rewrite rules specify the new location of every identified instruction. A *fall-through map* is generated to guide the transfer of control between instructions. A fall-through map as shown in figure 4.2 denotes the location of the next instruction.
- *Indirect branch target translation:* To support indirect branches the ILR scans the binary to locate all possible indirect branch targets and creates a fall-through map for each of them. This means that, if the attacker can leak the unrandomized indirect branch targets then he will be able to successfully divert the control flow to these locations.

  By randomizing the location of call instructions, ILR randomizes the return addresses too. However, all return addresses can not be randomized. In PIC code, a return address may be used to access a relative memory location. Also, return addresses are used by exception handling functionality. ILR deals with PIC code that uses return addresses by statically analyzing call sites and checking for any such access. If any such case is found, then that return address is left unrandomized. However, exception handling still remains vulnerable.

ILR uses process level virtual machine (PVM) to fetch instructions as per the rewrite rules, translate and execute in proper order. The rewrite rules scatter the instructions over the address space of 32 bits. However, monitoring and translating every instruction during the runtime adds to the performance overhead.

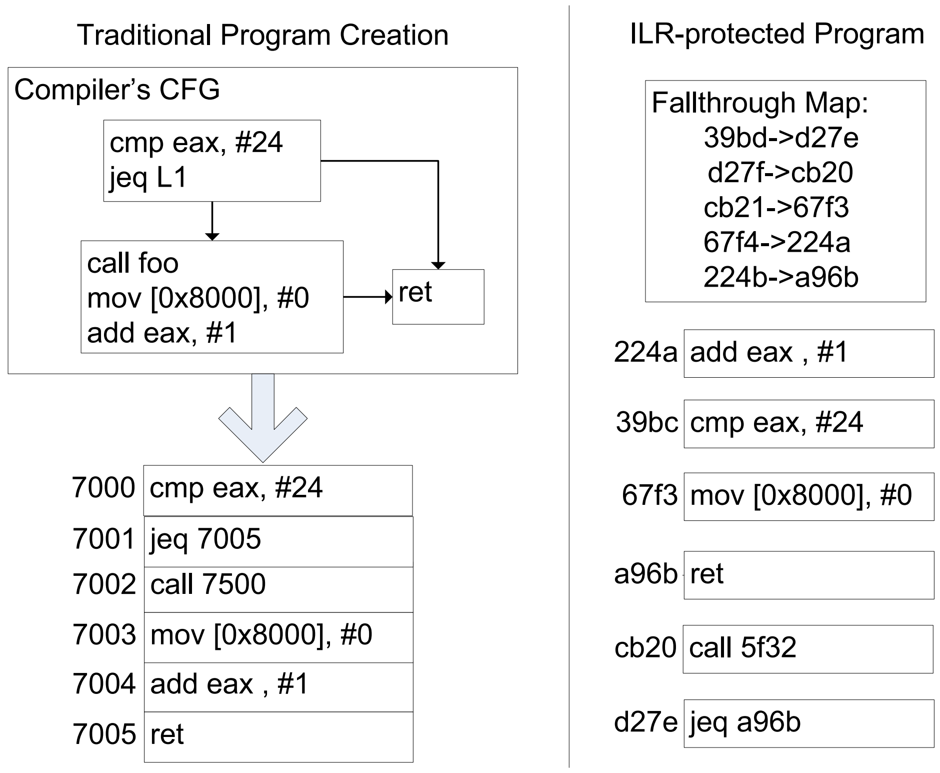


Figure 4.2: Example of ILR rewrite rules. Figure referred from [11]

To avoid runtime overhead, static binary instrumentation is the best approach. **Binary-stirring** (Wartell, Mohan, Hamlen & Lin,2012) [26] statically transforms the binary into a randomizable interpretation, which is used by a load-time re-assembler to reorder the code at the basic block level.

- *Code discovery and randomization:* Binary-stirring uses a recursive disassembly (IDA pro) to discover all possible code locations. Any misinterpreted data is protected keeping the old code section unchanged. After disassembly, the code is broken into basic blocks that are randomized during the load time. Randomization is done by Binary-stirring's initialization code that runs before the application is started.
- *Indirect branch target recovery and translation:* Every location in the code section that matches with prologues for known calling conventions signature is treated as a valid branch target. A runtime address lookup is then used to translate these targets during the runtime.

  Such assumptions that involve prologue matching are inefficient and error-prone. For example, complex binaries that have complex switch-case blocks that are usually translated into jump tables. The targets encoded in these jump tables do not necessarily point to a function beginning and hence may not constitute a function prologue. Also, any embedded data can get matched with function prologue and thereby can get overwritten.

Binary-stirring was tested on SPEC CPU2000 benchmark for both Windows and Linux platform. The average performance overhead was noticed to be 6.6%. The randomization approach of Binary-stirring doesn't preserve transparency. Change in code locations and return addresses on the stack can lead to failure of functionalities like exception handling. The binary-stirring approach doesn't take care of such special cases.

Unlike Binary-stirring, **In place code randomization (IPR)** [17] follows a much more conservative approach to deal with impreciseness of static disassembly and to preserve transparency. IPR proposes a series of in-place code transformation techniques, that change the semantics of unintended code gadgets without affecting the original code location, semantics and size.

- *Code discovery and transformation:* Uses IDA pro to recursively disassembly to achieve high code coverage while avoiding the use of heuristics such as function prologue matching to prevent any misidentification of data as code. In-place transformations are applied only on the parts that can be confidently identified as code. The transformations are as follows:
  - **Atomic instruction substitution:** Instructions are replaced with semantically equivalent instructions having the same length. Many arithmetic or logical instructions in x86 architecture have such dual equivalent forms. For example, *add r/m32,r32 and add r32,r/m32, test r/m8,r8 and test r8,r/m8,* etc. Changing the instructions will modify certain bytes in the code image and hence, can break the attacker's assumption regarding the semantics of unintended gadgets
  - **Instruction reordering:** Instructions such as registration preservation code at the entry of a function are independent and can be reordered arbitrarily. IPR derives the dependence graph per basic block using simple use-def analysis and reorders the independent instructions within a basic block. Similar to atomic instruction substitution, this transformation can impact the structure of non-intended gadgets.
  - **Register Reassignment**: Changing the register operands of instructions can change the byte sequence of unintended gadgets as well as break the attacker's assumption regarding the registers used in statically identified intended ROP gadgets. The IPR model achieves this by conducting a liveness analysis for every register within correctly identified function bodies.
- *Indirect branch target recovery and translation:* The in-place transformations do not change the code location and size and thereby eliminate the need for indirect branch target translation.

The conservative disassembly and in-place transformation help in avoiding errors caused by impreciseness of static disassembly but leaves some code untransformed and thereby unprotected from code reuse attacks. A gadget may remain intact either because it is not disassembled or it was not affected by the in-place transformations.

Static binary instrumentation based code randomization without any symbolic or debugging information comes with a bargain. The finer is the granularity of the binary randomization approach, the more is the chance of inaccuracy, in terms of misidentifying data as code, misidentifying code pointers, or breaking special functionalities like the exception handling/debugging mechanism, etc. And, the more we try to preserve the functionality and transparency in the behavior of the program, the lesser will be the granularity of randomization achieved.

5 Static Code randomization technique for x86-64 position independent executables

In code reuse attacks, the adversary tries to divert the control flow to code gadgets. Code gadgets are legitimate code already present in the executable segment of a program. In conventional ROP attacks, the adversary has to identify the code gadgets statically. In such a case, simple Randomization at the granularity of functions will be able to break the adversary's assumptions regarding the location of code gadgets. However, new forms of ROP attacks known as JIT-ROP and indirect disclosure based JIT-ROP have surfaced. In JIT-ROP attacks, the adversary is able to read and disassemble the code pages at the runtime to find code gadgets. Such form of ROP attacks can be thwarted by marking code pages as execute-only along with using some form of coarse-grained randomization, such as function permutation. In indirect disclosure based JIT-ROP attacks, the adversary is able to leak stored code pointers from memory or stack (Return address). The adversary can then easily guess the location of code gadgets present around the leaked pointers. Along with execute-only code pages, the code needs to be randomized at much finer granularity to prevent such types of attacks.

Compiler based defense techniques like Readactor [9] try to prevent the disclosure of all stored code pointers by replacing them with trampoline addresses. However, practical application of such compiler based approaches is affected by the availability of source code. Binary based gadget elimination techniques such as IPR [17] apply in-place transformations to the instructions in order to render the gadgets unusable. Such gadget elimination techniques can be helpful against all forms of ROP discussed in the above paragraph. However, such techniques have to deal with the content classification problem (accurate classification of code and data) and therefore follow a conservative approach in which some codes remain untransformed. Binary stirring [26] performs randomization at the granularity of basic blocks. However, use of heuristics such as *function prologue matching* to detect code locations and indirect branch targets can result in impreciseness. Dynamic binary instrumentation based randomization approaches like ILR [11] effectively randomize every instruction, but incur significant runtime overhead. In this section we present our approach of applying randomization at basic block granularity for x86-64 position independent executables (PIE).

To support ASLR, most of the commonly used x86-64 bit binaries are position independent executables (PIE). Which means that the address of code and data is not absolute and is determined during the load time. One important property of the PIE binaries is the re-

Assembly Code	Relocation info		
	Relocation slot	Relocation entry	Relocation type
10: mov 0xf(%rip),%rax			
17: jmpq *(%rax)			
19: mov 0x2(%rip),%rbx			
20: jmpq *(%rbx)	0x26	0x100	R_X86_64_RELATIVE
	0x22	0x500	R_X86_64_RELATIVE
22: 0x500			
26: 0x100			

Table 5.1: Relocation information format for x86-64 ELF binaries

location information. As the code and data address is determined during the runtime, any pointer stored in memory has to be relocated to the new address during the load time. The relocation table (Table 5.1) has a record for all such pointers. Each record in the relocation table has a *relocation slot* that holds the location of a pointer, a *relocation entry* that holds the pointer value and a *relocation type* which indicates how a pointer is going to be updated during the load-time. In this example (Table 5.1), relocation type *R_X86_64_RELATIVE* means the pointer value will be added to the code section base to generate the final value.

We have exploited this property of PIE binaries to develop a robust static binary instrumentation approach. Exploiting relocation information helps us in retrieving all the stored code pointers. This helps us in achieving complete and correct disassembly as well as preserving the code and data references in the instrumented binary.

5.1 Disassembly approach

The major challenge in the static disassembly process is the content classification problem, i.e. distinguishing code and data. To support DEP, modern compilers tend to create a separate section for code and data. In x86-64 bit ELF binaries compiled by gcc, code, read-only data, and writable data are present in three different sections and no embedded data is found within the code section. This fact is supported by the research conducted by Andriesse et al. (2016) [3] where linear disassembly approach is shown to achieve 100% correct and complete code discovery for x86 ELF binaries. However, there is a chance that in large binaries such as *"glibc"* alignment bytes may be present between the functions. Misinterpreting alignment bytes will not create any problem. However, because of the variable-length instruction set architecture of x86, the misinterpretation may trickle down to subsequent valid instructions. To avoid this problem, we start disassembling code from valid code locations. Valid code locations are the program entry point, stored code pointers obtained from the relocation table and the function entry points obtained from the export table. The disassembly approach can be summarized as below:

- **Identifying code pointer constants:** The ELF binary is parsed to obtain the location of code and data sections, the relocated code pointers and the export table entries. A relocation entry may point to a data pointer rather than a code pointer. Similary, export table may contain entries for exported global variables. Therefore we only consider those entries that fall within the code section range.

- The code section is divided into *code blocks* according to the code pointers obtained in step 1.
- Each block is then linearly disassembled using objdump. The linear disassembly helps us to discover the code locations that are only reachable via jump tables.

After the disassembly is complete, control flow graph (CFG) is generated and jump table targets are recovered. The control flow graph generation process is able to identify all basic blocks correctly. The subsequent sections describes the CFG generation process and the jump table recovery process.

5.1.1 Control flow graph

Basic block identification: A basic block is defined as a linear block of code that has only one entry point and only one exit point. Neither code pointers nor direct control instructions can target any location that doesn't mark the beginning of a basic block.

Our basic block detection mechanism is a simple depth first search along the control flow graph. We start scanning instructions from the beginning of each *code block*. We stop when we encounter a branch instruction. The branch is marked as the end of the basic block and the instruction scan continues from the branch target. In case, the branch is conditional, the scanning is done at the fall-through address too. In case, a branch happens to target the middle of a pre-identified basic block, then the basic block is broken into two. There is a chance that we may encounter a call or jump instruction targeting a location outside the range of the current *code block*. Such cases are noted as *inter code block targets* and processed in the second phase. By following the control flow we may miss the basic blocks that are reached via jump tables as the jump table targets have not been computed yet. So, we treat any code location immediately following an unconditional jump *(call/jmp/ret)* as a valid basic block entry point and examine them.

After all the *code blocks* have been divided into basic blocks, all the noted cases of *inter code block targets* are processed. For each such target, we find the pre-identified basic block to which the target belongs. If the target is in the middle of the basic block, then we break the basic block into two.

For each correctly identified basic block, we maintain two forward edges. One pointing to the branch target and one pointing to the fall-through block.

Function identification: Every call instruction target is marked as a valid function entry point. A function body is defined as a set of basic blocks that lay between two function entry points. This approach will cover all the basic blocks including the ones that are only reached via a jump table. This approach will treat a multi-entry function as separate functions. Hence we merge two consecutive function bodies if there exists a branch between the two function bodies. The merging process is done recursively until no merge is possible.

A function may only be reachable via a tail call where a jump instruction is used rather than a call. At this point, we are not handling such tail calls as it is not trivial for the code randomization process. If a function is reachable only via tail call, then it will not be identified as a separate function. Rather the tail called function will be treated as a part of the function body immediately preceding itself.

JUMP TABLE DECODING: The jump tables entries are offsets from a base to the actual target. The base may be the base of the jump table or some address within the code section. At the runtime, the base is loaded to a register and a jump table entry is added to the base to compute the actual target. The computation is typically of the form **(C1 + ind) + C2.* Where *C1* and *C2* are constants and *ind* is the index of the jump table. Table 2.2 shows an example of ICF target computation using a jump table. IN x86-64 bit PIE binaries, the base is usually loaded using *"lea"* instructions as shown in table 2.2.

To detect jump tables we first locate all the indirect jump instructions within a function, that use a register as an operand. For each of the located indirect branches, we try to find a single path from every *"lea"* instruction present in the function to the indirect branch instruction. We perform a static analysis on each of these paths. The static analysis checks for the calculation pattern of the form **(C1 + ind) + C2* and outputs the two constants *C1* and *C2*. *C1* is inferred as the location of the jump table and *C2* is inferred as the base used to compute the target of the jump table. The underlying assumption is that if a load (*lea*) instruction and an indirect branch instruction pair is a part of a jump table computation, then all the paths from the load instruction to the indirect branch must follow the jump table computation pattern. So, performing static analysis on any one of the paths is sufficient to determine the jump table location. The range of the jump table is hard to determine as the index (*ind*) computation may cross the function boundary. Therefore we conservatively consider the location of the next jump table as the end of the current jump table. Finally, we iterate over each of the jump tables to determine the actual targets using the calculation of the form **(C1 + ind) + C2.*

We found one special case in SPEC CPU2017 gcc benchmark where the basic block that loads the jump table base makes an indirect jump to the basic block that performs the final computation and determines the target. The indirect jump made by the first basic block is part of another jump table computation. To overcome such scenarios, we recursively decode the jump tables and keep adding edges to our control flow graph until no more jump table is detected.

5.2 CODE RANDOMIZATION

Our disassembly and control flow graph generation process makes sure that all the basic blocks have been accurately identified. This gives us the ability to perform fine-grained randomization at the basic block level and scatter the basic blocks throughout the code section. However, there are two main challenges that we may face:

- *Preserving functionality:* Functionalities like exception handling and crash reporting, which depend on code uniformity can be affected by the randomization process.
- *Performance* : Randomization can affect the performance in the following ways:
  - Scattering the basic blocks can cause disruption of cache locality.
  - For every fall-through block one extra jump will be executed.
  - Conversion of short jumps into 5-byte long jumps

PRESERVING FUNCTIONALITY: Functionalities like exception handling are sensitive to the code location. Before discussing about how we preserve exception handling, lets discuss how it works. The compiler encodes the exception handling information in the eh_frame, eh_frame_hdr and gcc_except_table of an ELF binary. Any application that has exception handling must have these three sections. The gcc_except_table holds the location of try/catch blocks. The eh_frame section:

- holds the address range of every function.
- if the function has any try/catch block, then it holds a pointer to a record in gcc_except_table.
- holds the encoded rules that need to be followed while unwinding the stack from the current function frame.

The eh_frame_hdr section is a binary search table that stores a pointer to the eh_frame record for every function. The binary search table is arranged according to the function start address. The exception handling process can be subdivided into two steps.

- *eh record retrieval:* An exception throw is usually compiled to a standard library function call. The standard library function retrieves the eh_frame record of the current function by comparing the return address on the stack with the function boundaries encoded in the eh_frame section.
- *stack unwinding:* If the current function doesn't have any try/catch block stack unwinding is done to retrieve the previous caller. This unwinding is done according to the rules encoded in the eh_frame record.

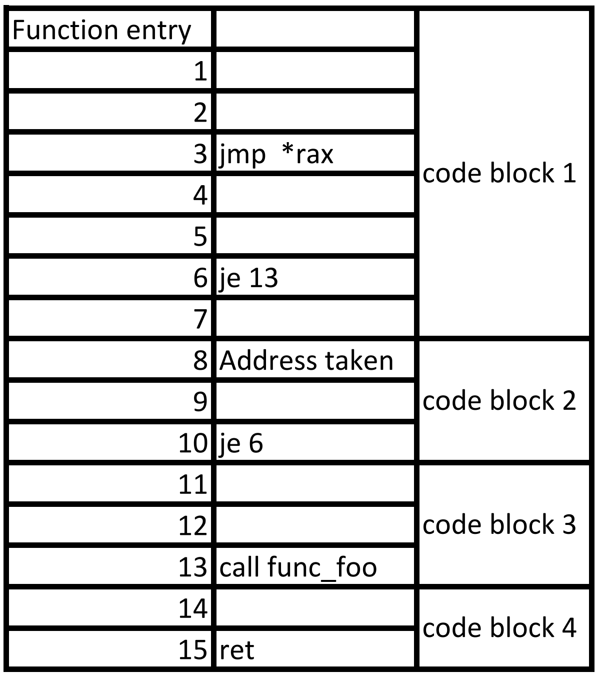
Any randomization process that relocates a call site beyond the function boundary will result in the failure of eh_frame record retrieval. Therefore, our randomization process limits the basic block scattering to the function boundaries. Post randomization and binary re-generation, we re-encode all the exception handling related sections.

The stack unwinding rules defined in the eh_frame specify the stack increment between a range of instructions. If we scatter the basic blocks arbitrarily within a function, these rules will no longer apply and hence, we need to re-encode these rules. We performed our experiments by choosing not to implement the rule encoding at this point. It can be done with some extra engineering effort. By not doing so, the exception handling will still work for the cases where the catch block is present in the same function as the throw statement and no stack unwinding needs to be done. Note that at this point, we do not support crash reporting. Its a part of our future work. Our randomization approach was applied on glibc, libreoffice, gedit, coreutils and SPEC CPU2017 Integer benchmark binaries and we were able to successfully regenerate functional binaries.

PERFORMANCE TRADE-OFF: Code randomization can add to the performance overhead by disturbing the cache locality, adding extra jump instructions and converting short jumps into 5-byte long jumps. To negotiate with performance our approach supports randomization at three different levels of granularity.

- *Function permutation:* To reduce the performance overhead many randomization techniques randomize code at function granularity [12][9]. Applying function permutation only can provide high enough entropy and help to prevent static ROP by breaking the statically inferred knowledge about the location of code gadgets. We randomized the functions by permuting the function bodies retrieved from the eh_frame section and then re-encoding the eh_frame and eh_frame_hdr sections. The resulting performance overhead was 0.06%.
- *Indirect branch target hiding:* Function permutation implemented with execute-only code pages, can prevent JIT-ROP attacks where the attacker reads code pages in the runtime to find gadgets. However, it is prone to indirect disclosure based JIT-ROP attacks where the adversary can leak code pointers (stored in memory and stack) and guess the location of gadgets present around the code pointer. Our second approach randomizes all locations that can be leaked by the adversary. Such code locations include return addresses and code pointers stored in memory. The randomization steps are as follows:
  - A function is broken into blocks as shown in figure 5.1. A linear block of code can only terminate at call instructions. Other than this, any basic block that is a target of an indirect branch (jump tables or stored code pointers) is treated as a separate block to randomize the stored code pointers.
  - The blocks are then shuffled within the function. Additionally, a direct jump instruction targeting the indirectly accessed blocks (e.g., code block 2 and code block 4 in figure 5.1) is added and the pointers to these blocks are replaced with pointers to the respective jump instructions. The addition of extra jump helps in hiding the code pointers.
  - In the final step, the functions are permuted.

  This approach tries to thwart all possible ROP attacks while minimizing the performance overhead. Static ROP attacks in which the attacker statically infers the location of code gadgets will be broken by function permutation only. Randomization of return addresses and stored code pointers can prevent indirect disclosure based JIT-ROP. And implementing execute-only code page along with this transformation will prevent conventional JIT-ROP where the adversary tries to read the code pages to locate gadgets.
- *Basic block randomization:* To prevent all types of ROP attacks (static ROP, JIT-ROP, indirect disclosure based JIT-ROP), there is no need to go beyond the second ap-



Function entry		code block 1
1		
2		
3	jmp *rax	
4		
5		
6	je 13	
7		
8	Address taken	code block 2
9		
10	je 6	
11		code block 3
12		
13	call func_foo	
14		code block 4
15	ret	

Figure 5.1: Indirect branch target hiding

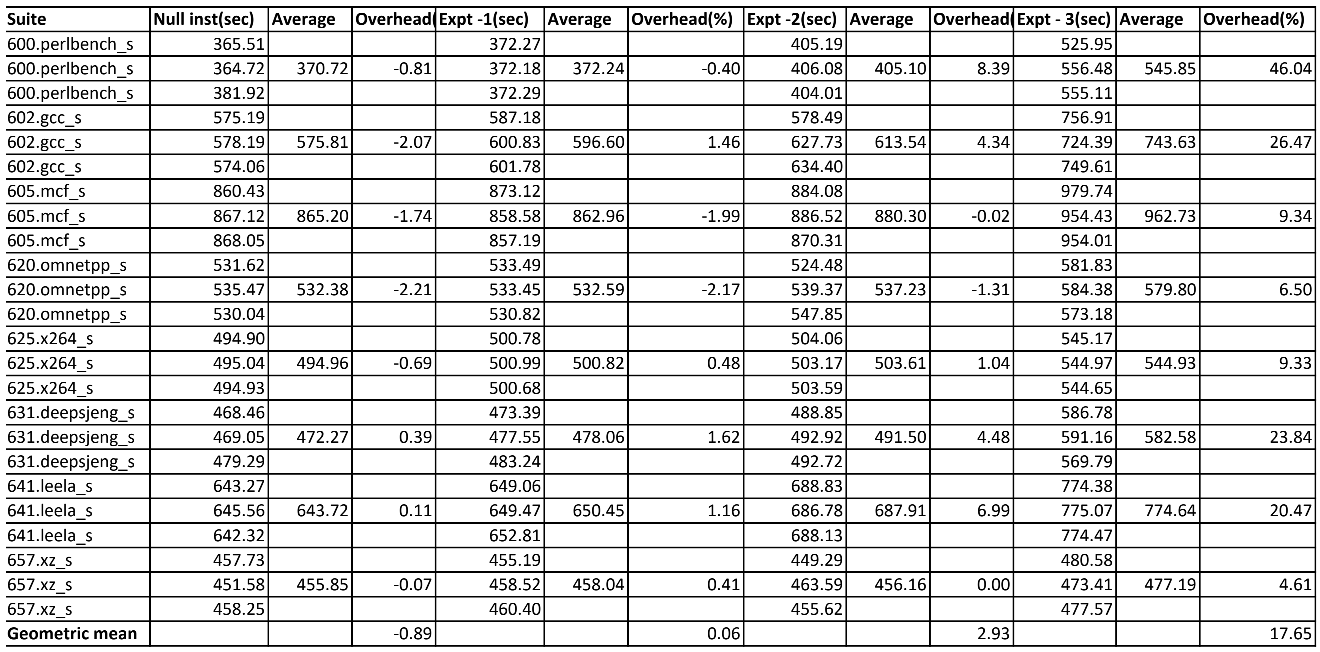
proach. However, randomization at the granularity of basic block can provide very high entropy. We performed randomization at the basic block granularity by arbitrarily scattering the basic blocks within the function ranges obtained from the eh_frame section and then randomizing the functions. This will result in addition of an extra jump for every fall-through block and at the same time can affect the cache locality.

5.3 Experimental evaluation

To get an idea about the overhead caused by our instrumentation process alone, we regenerated the SPEC CPU2017 integer benchmark binaries with null instrumentation (i.e. no randomization) and measured the performance. The results show that no performance overhead (-0.89%) was caused by the instrumentation process (Figure 5.2).

We then performed three different experiments on SPEC CPU2017 integer benchmark binaries to measure the performance overhead of the three randomization approaches discussed above:

- *Experiment 1 (Function permutation)*: Experiment-1 evaluates the performance of our function permutation approach. Function permutation causes minimal performance overhead of 0.06% (Figure 5.2). Permuting the functions can affect the cache locality, but the effect should be minimal. This is evident from the results (0.06% overhead). Preventing addition of any extra jump instruction by leaving the basic block



Suite	Null inst(sec)	Average	Overhead(	Expt -1(sec)	Average	Overhead(%)	Expt -2(sec)	Average	Overhead(	Expt - 3(sec)	Average	Overhead(%)
600.perlbench_s	365.51			372.27			405.19			525.95		
600.perlbench_s	364.72	370.72	-0.81	372.18	372.24	-0.40	406.08	405.10	8.39	556.48	545.85	46.04
600.perlbench_s	381.92			372.29			404.01			555.11		
602.gcc_s	575.19			587.18			578.49			756.91		
602.gcc_s	578.19	575.81	-2.07	600.83	596.60	1.46	627.73	613.54	4.34	724.39	743.63	26.47
602.gcc_s	574.06			601.78			634.40			749.61		
605.mcf_s	860.43			873.12			884.08			979.74		
605.mcf_s	867.12	865.20	-1.74	858.58	862.96	-1.99	886.52	880.30	-0.02	954.43	962.73	9.34
605.mcf_s	868.05			857.19			870.31			954.01		
620.omnetpp_s	531.62			533.49			524.48			581.83		
620.omnetpp_s	535.47	532.38	-2.21	533.45	532.59	-2.17	539.37	537.23	-1.31	584.38	579.80	6.50
620.omnetpp_s	530.04			530.82			547.85			573.18		
625.x264_s	494.90			500.78			504.06			545.17		
625.x264_s	495.04	494.96	-0.69	500.99	500.82	0.48	503.17	503.61	1.04	544.97	544.93	9.33
625.x264_s	494.93			500.68			503.59			544.65		
631.deepsjeng_s	468.46			473.39			488.85			586.78		
631.deepsjeng_s	469.05	472.27	0.39	477.55	478.06	1.62	492.92	491.50	4.48	591.16	582.58	23.84
631.deepsjeng_s	479.29			483.24			492.72			569.79		
641.leela_s	643.27			649.06			688.83			774.38		
641.leela_s	645.56	643.72	0.11	649.47	650.45	1.16	686.78	687.91	6.99	775.07	774.64	20.47
641.leela_s	642.32			652.81			688.13			774.47		
657.xz_s	457.73			455.19			449.29			480.58		
657.xz_s	451.58	455.85	-0.07	458.52	458.04	0.41	463.59	456.16	0.00	473.41	477.19	4.61
657.xz_s	458.25			460.40			455.62			477.57		
Geometric mean			-0.89			0.06			2.93			17.65

Figure 5.2: Spec CPU2017 integer benchmark results

unrandomized also results in low performance overhead.

- *Experiment 2 (indirect branch target hiding):* Experiment-2 evaluates the performance of indirect branch target hiding approach. This approach involves execution of one extra jump for every indirect branch instruction. Also, this approach can affect the cache locality to some extent as it involves some degree of code re-ordering within the function boundary. The performance overhead of this approach is found to be 2.93% (Figure 5.2).
- *Experiment 3 (Basic block randomization):*: This is the most fine-grained randomization our approach can perform while preserving the exception handling functionality. The performance overhead caused is 17.65% (Figure 5.2). The possible reasons for the high overhead can be (i) Extra jump instruction executed for every relocated fall-through block, (ii) disruption of cache locality and (iii) conversion of short jumps into 5-byte long jumps. To determine the actual reason behind performance overhead, we performed an experiment in which the basic blocks are left unrandomized but an additional jump instruction was added for every fall-through block. We observed that the overhead was close (17-18%) to the basic block randomization. Hence, it can be concluded that the extra jumps are the main reason of high performance overhead.

6 Conclusion

The process of binary instrumentation has always been a bargain between performance and accuracy. Despite having a high performance overhead, the dynamic binary instrumentation have always been the preferred option because of its ease of use, ability to instrument all code, ability to handle large and complex binaries. Static binary instrumentation on the other hand offers a more efficient option, but the challenges with SBI still remain unsolved. To deal with the challenges, the SBI tools try to be conservative and lose some accuracy in the process. Scalability has also been an issue with SBI. However, with the evolution of compilers, executables are becoming more and more SBI friendly. For example, to enforce DEP gcc no longer keeps code and data in the same section. This solves the content classification problem. And to enforce ASLR, binaries are being compiled as position independent executable. This means that the binaries will have relocation information as an inherent property of position independent executable. These properties of modern binaries can be exploited to enhance the accuracy and scalability of SBI tools. This has been proved by our static code randomization approach. We are able to apply fine-grained randomization at basic block level by leveraging on the relocation information of PIE executables. These properties of modern x86 PIE binaries have opened a door for the development of accurate and robust SBI tools.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 263–277. IEEE, 2008.

[3] D. Andriesse, X. Chen, V. Van Der Veen, A. Slowinska, and H. Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[4] S. Bhatkar, D. C. DuVarney, and R. Sekar. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, 2005.

[5] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 353–362. ACM, 2011.

[6] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 2003.

[7] 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*. ACM, 2008.

[8] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010.

[9] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy*, pages 763–780. IEEE, 2015.

[10] GNU Texinfo 6.5. GNU Utility Objdump. `https://sourceware.org/binutils/docs/binutils/objdump.html`.

[11] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd my gadgets go? In *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012.

[12] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 339–348. IEEE, 2006.

[13] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 175–183. IEEE, 2010.

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40. ACM, 2005.

[15] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42. ACM, 2007.

[16] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010.

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

[18] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015.

[19] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016.

[20] M. Smithson, K. ElWazeer, K. Anand, A. Kotha, and R. Barua. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 52–61. IEEE, 2013.

[21] Solar Designer. "return-to-libc" attack. *Bugtraq, Aug*, 1997.

[22] W.-K. Sze and R. Sekar. A portable user-level approach for system-wide integrity protection. In *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013.

[23] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016.

[24] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017.

[25] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE Symposium on Security and Privacy*, pages 380–395. IEEE, 2010.

[26] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012.

[27] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, 2009.

[28] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *IEEE Security and Privacy*, 2013.

[29] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *USENIX Security*, 2013.