Anywhere, Any-Time Binary Instrumentation

Andrew R. Bernat, Barton P. Miller
Computer Sciences Department
University of Wisconsin
Madison, WI 53706
{bernat,bart}@cs.wisc.edu

ABSTRACT

The Dyninst binary instrumentation and analysis framework distinguishes itself from other binary instrumentation tools through its abstract, machine independent interface; its emphasis on *anywhere, any-time* binary instrumentation; and its low overhead that is *proportional* to the number of instrumented locations. Dyninst represents the program in terms of familiar control flow structures such as functions, loops, and basic blocks, and users manipulate these representations to insert instrumentation *anywhere* in the binary. We use graph transformation techniques to insure that this instrumentation executes when desired even when instrumenting highly optimized (or malicious) code that other instrumenters cannot correctly instrument. Unlike other binary instrumenters, Dyninst can instrument at *any time* in the execution continuum, from static instrumentation (binary rewriting) to instrumenting actively executing code (dynamic instrumentation). Furthermore, we allow users to modify or remove instrumentation at any time, with such modifications taking immediate effect. Our analysis techniques allow us to insert new code without modifying uninstrumented code; as a result, all uninstrumented code executes at native speed. We demonstrate that our techniques provide this collection of capabilities while imposing similar or lower overhead than other widely used instrumenters.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Binary instrumentation*

General Terms

Experimentation, Performance

Keywords

Binary instrumentation; Dynamic instrumentation; Binary rewriting

PASTE'11 September 5, 2011, Szeged, Hungary.

1. INTRODUCTION

Binary instrumentation is a technique that modifies a binary program, either pre-execution or during execution. This technique can be used to insert monitoring code, such as for performance analysis [8, 17], attack detection [15], cyberforensics [12], or behavior monitoring [5]; to modify program data, such as for dynamic transactional memory [11]; or to perform code replacement, such as for performance steering. For the past several years, we have been developing the Dyninst binary analysis and instrumentation system. By combining these capabilities we have made binary instrumentation more efficient and precise, and enabled a class of tools that combine analysis with instrumentation [8, 16]. Dyninst distinguishes itself from other instrumenters through its *abstract* interface; its emphasis on *anywhere, any-time* (AWAT) instrumentation; and its low overhead that is *proportional* to the number of instrumented locations. We describe recent work that allows Dyninst to provide AWAT instrumentation with proportional cost.

Dyninst allows users to instrument *anywhere* in the binary. Other binary instrumentation approaches allow users to insert instrumentation at instructions [2, 9, 10, 14] or specific types of control flow edges [9, 10]. Surprisingly, instrumenting instructions or edges is not sufficient to capture program behavior based on structural characteristics such as functions or loops. Consider the case of instrumenting the entry of a function; such instrumentation should execute only once per function call. Other binary instrumenters do not directly support instrumenting function entries; instead, users instead instrument the first instruction in the function (assuming sufficient information, such as a symbol table, is available to correctly identify that instruction). However, the first instruction of functions with no preamble code may be executed multiple times per function invocation. Therefore, such instrumentation would execute multiple times per invocation. This problem also holds for instrumenting other control flow features of a program, such as function exits and loop entries, iterations, and exits.

We address this problem by allowing users to specify instrumentation locations in terms of the control flow graph (CFG) in addition to at the instruction level. We define classes of *instpoints* to describe interesting locations in the CFG (e.g., function entries and exits), and users insert instrumentation by annotating these instpoints with instrumentation code. Section 2 describes the techniques we use for instrumenting anywhere in the binary.

Dyninst allows users to insert, remove, or modify instrumentation at *any time* in the execution continuum: pre-

execution (binary rewriting), before code has executed for the first time, or while the instrumented code is currently executing (dynamic instrumentation). Binary rewriting offers several benefits over dynamic instrumentation, such as amortizing the cost of instrumenting a binary over multiple executions or eliminating the need for having the instrumenter present during execution. In contrast, dynamic instrumentation allows users to instrument as the program executes. Other binary instrumentation approaches support either static instrumentation [4,9] or dynamic instrumentation [2,10,14], but not both. Furthermore, other dynamic instrumenters cannot guarantee that instrumenting currently executing code will take immediate effect.

Our approach allows users to use the same tool to instrument both statically and dynamically. Furthermore, during dynamic instrumentation any changes are guaranteed to take immediate effect. During binary rewriting we present the user with a CFG that was derived using static analysis; this CFG supports both analysis and instrumentation. If we are performing dynamic instrumentation we also report runtime events, such as changes to the CFG or system events (e.g., thread creation or process exit). For example, if we are instrumenting a binary that is statically obfuscated or generates code at runtime, we will discover the new code before it executes and report that new code to the user [16]. Section 3 describes these techniques.

Dyninst imposes *proportional cost* by imposing overhead only when instrumented code is executed; furthermore, removing instrumentation also removes its cost. This can significantly reduce overhead if the number of instrumented locations is small (e.g., a subset of functions in the program) or if instrumentation can be removed as the program runs. Binary instrumenters rely on a technique we call *relocation* to insert both instrumentation and any additional infrastructure code the instrumenter requires (e.g., dynamic analysis code). Relocation moves code to a new location where it can be expanded to include new code (instrumentation and analysis code) and transforms it to preserve its original behavior; as a result of this transformation the relocated code frequently executes slower than the original. Other approaches use dynamic analysis to discover all executed code, and thus must relocate all executed code whether or not it is instrumented. As a result, they impose overhead even on uninstrumented code [2, 10, 14].

By using static analysis to derive the CFG rather than dynamic analysis we can greatly reduce the amount of uninstrumented code that must be relocated. In conventional binaries, we only relocate instrumented code. This relocation may be performed on a basic block or function basis, depending on the density of instrumentation. As a result, any uninstrumented code executes natively with no overhead; this is also true for code whose instrumentation is removed. For binaries where the CFG is incomplete (e.g., JIT compilers or self-unpacking malware) we rely on relocation to enable dynamic analysis, but unlike other approaches we only relocate the specific locations (e.g., indirect jumps or calls) that we know are incompletely analyzed. Section 3 describes these techniques.

As a result of our analysis and instrumentation techniques, Dyninst provides greater precision of instrumentation without incurring additional cost. For example, in experiments performed on the SPEC benchmark suite we incurred an average 71% execution overhead when instrumenting every basic block, as opposed to the 128% imposed by PIN. This is a worst-case example of our techniques, since it does not leverage the proportional cost of our approach, and we show that the cost we impose decreases as fewer locations are instrumented while other instrumenters impose cost even when executing uninstrumented code. We can also instrument malware that executes incorrectly or crashes when instrumented with other instrumenters, although the overhead we impose is higher than our overhead on conventional binaries due to the tamper-resistance features used by these binaries.

2. ANYWHERE INSTRUMENTATION

The control flow graph (CFG) is a familiar representation of the structural elements in a program (e.g., functions, loops, and basic blocks) and the relationships between them. The CFG is also a useful representation for specifying binary instrumentation in terms of these same structural elements. In this section, we describe Dyninst's CFG-based instrumentation approach. This approach has two challenges. First, we must map reasonable program abstractions onto code that may have been highly transformed due to optimization or obfuscation. Second, we must regenerate instrumented code from the instrumented CFG that both preserves the behavior of the original code and executes instrumentation when the user specifies. We begin by defining the basic CFG abstractions used by Dyninst, including functions, loops, basic blocks, and the *instrumentation points* that are used to specify where instrumentation is inserted. We then describe the complications that result from highly transformed code (e.g., overlapping functions) and how we map our abstractions on to such code. Finally, we describe how we generate new code from the instrumented CFG.

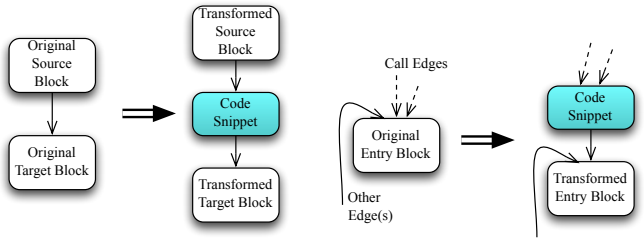
2.1 Control Flow Abstractions

Our CFG is based on five abstractions: the *interprocedural control flow graph* (consisting of *basic blocks* and *edges*), *functions*, and *loops*. Our goal is to provide familiar structural abstractions while hiding the complexities that may occur in the binary. We also define *instrumentation points* (or *instpoints*) for each of these abstractions, which identify locations where instrumentation can be inserted, and *code snippets*, which represent the inserted instrumentation code. As with our control flow abstractions, the goal of these additional two abstractions is to hide the complexity that arises when you insert additional code in the binary. Intuitively, an instpoint is a simple abstraction; when the point is reached during execution, the instrumentation at that point executes. For example, a user may instrument the entry point of a function; such instrumentation is guaranteed to execute once per function invocation, even if the entry block executes multiple times (e.g., as part of a loop). We formalize this concept below.

A CFG is a directed graph $G = (V, E, V_e, V_x, \tau)$:

- The set $V = B \cup \{v_\perp\}$ of vertices corresponding to basic blocks B and a *sink node* $v_\perp$;
- The set $E \subseteq V \times V$ corresponds to control flow edges between blocks;
- The sets $V_e \subseteq V$ of entry and $V_x \subseteq V$ of exit nodes;
- The labeling function $\tau : E \rightarrow \mathcal{T}$ that associates a particular edge in the graph with a type.

We define basic blocks in the conventional way as a linear sequence of instructions $b_i = \langle i_m, \ldots, i_n \rangle$ with a single entry



(a) Edge Rule (b) Function Entry Rule

Figure 1: Two example transformation rules; Figure (a) shows the edge instrumentation rule and Figure (b) shows the function entry rule.

point i_m and single exit point i_n; an instruction may belong to only one block. Unknown control flow is represented by an edge to a unique *sink node* $v_\perp$ that contains no instructions.

Edges are labeled with an *edge type*. We define the following types: *direct, fallthrough, conditional taken, conditional not taken, indirect, call, call fallthrough*, and *return*, where call fallthrough edges link blocks ending with calls to their intraprocedural successors. An edge is *interprocedural* if it leaves a function and *intraprocedural* otherwise.

We define a *function* as the blocks reachable from an *entry block* traversing only intraprocedural edges. Formally, functions are subgraphs of the CFG $f_i = (V_i, E_i, v_i, X_i, \tau')$ where $V_i \subseteq V$, $E_i \subseteq E$, $v_i \in V_i$ is the *entry block* of the function, $X_i \subseteq V_i$ are the *exit blocks*, and τ' assigns only intraprocedural edge types.

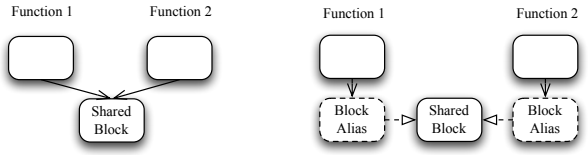
We use natural loops for our loop definition. Loops are defined similarly to functions, as subgraphs of the CFG: $l_j = (V_j, E_j, v_j, X_j, \tau')$ where $V_j \subseteq V$, $e_j \subseteq E$, v_j is the unique entry node, X_j are exit nodes, and τ' assigns only intraprocedural edge types (our loops may not be interprocedural). An edge $e = (v, v_j)$ is a *backedge* if $v \in V_j$. Loops may be nested and share entry blocks; however, the combination of entry and exit blocks uniquely defines a loop.

While these abstractions provide a familiar program representation for binary analysis, the instpoint abstraction supports instrumentation by capturing a certain aspect of program behavior, such as entering a function or traversing an edge between blocks, and allowing users to instrument their program based on these behaviors rather than just in terms of instructions and edges. More formally, an instpoint is an annotation of a subgraph of the CFG. We define two *classes* of instpoints; *augmentation* and *transformation* instpoints. An augmentation instpoint, such as pre-instruction or block entry, adds additional code to an existing basic block rather than transforming the CFG. These instpoints are similar to the instrumentation primitives used by other binary instrumenters. We define the following augmentation instpoints: block entry, pre-instruction, and post-instruction.

Transformation instpoints, (e.g., function entry or edge instpoints) insert instrumentation by adding new blocks and edges to the CFG; for example, edge instrumentation adds a new block to the CFG along a control flow edge. More formally, a transformation instpoint is represented by a particular *subgraph template* and is associated with a graph transformation *rule* that we use to add instrumentation to the CFG. Briefly, a rule $p : L \rightarrow R$ replaces an instance of the subgraph L in a target graph G with the graph R; we represent these rules graphically, as is typical in graph transformation [7]. We define the following transformation

Address	Instruction		
61a901	`cmpl`	`$0x0,`	`%gs:0xc`
61a909	`je`	`61a90c`	
61a90b	`lock cmpxchg`	`%ecx,`	`0x31f4(%ebx)`
61a913	`jne`	`...`	

Figure 2: Example of overlapping blocks from GNU libc on IA-32 compiled with GNU GCC 4.x. The first conditional branch skips past a locking prefix on the compare and exchange instruction; this results in two instructions that overlap. Execution converges at the second branch.



(a) Interprocedural CFG (b) Block Aliases

Figure 3: An example of overlapping blocks in the CFG. Figure (a) shows the interprocedural CFG, with two functions sharing the same exit block. Figure (b) shows the corresponding function representations, with the return block represented by two aliases.

instpoints: function entry and exit; loop entry, exit, beginning of iteration, and end of iteration; block exit; and edge. We show some example transformation rules in Figure 1.

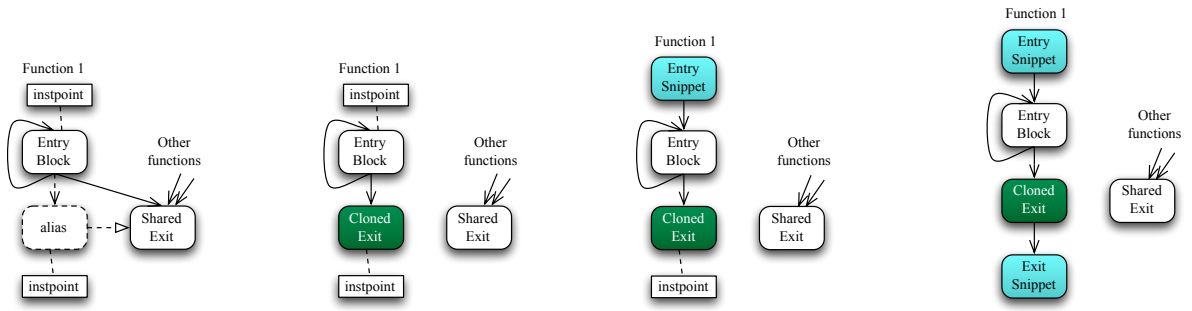
Finally, we define a *code snippet* to represent a sequence of inserted instrumentation. A code snippet is a sequence of code that has a single entry point and exit point, although it may have internal branching. We also assume that code snippets have no side-effects on the execution of the original code; this is a common assumption for instrumentation. As a result, we may represent a sequence of code snippets as a single *snippet block*. In Dyninst, users specify code snippets by either specifying them in terms of an abstract syntax tree or writing them in the C-like DynC language.

2.2 Complicating Cases

Several code structures commonly seen in binaries complicate the mapping of our control flow abstractions onto the binary code. Our goal with such structures is to hide their complexity from the user wherever possible. We describe how we handle two such cases: *overlapping basic blocks* and *overlapping functions*. In this section, we describe these structures and how our abstractions map onto them; we describe how we instrument such structures in Section 2.3.

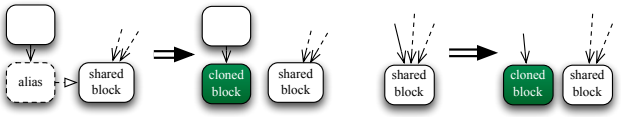
Overlapping basic blocks occur when the same sequence of bytes disassembles to two distinct instruction sequences, both of which may be executed by the program. This situation occurs in variable-length architectures because instructions can overlap; however, features such as delay slots can lead to a similar situation on fixed-length architectures. One example of such code is shown in Figure 2, which uses a conditional branch to optionally skip a locking prefix on a compare and exchange instruction, resulting in two instructions that overlap. This sequence occurs in many GCC-compiled Linux libraries. Overlapping instruction sequences also commonly occur in obfuscated code. We represent each of these code sequences as distinct collections of basic blocks. This representation allows the user to treat the overlapping code sequences as logically disjoint sequences of instructions.

Overlapping functions occur when multiple functions include the same basic block. Optimizing compilers frequently share common code between functions (e.g., register restores



(a) Original CFG (b) Block Cloning (c) Entry Instrumentation (d) Exit Instrumentation

Figure 4: Instrumentation example of two overlapping functions (a). We use block cloning (b) and graph transformations (c, d) to instrument function 1, leaving other functions that overlap with function 1 unmodified.



(a) Function Layer (b) CFG Layer

Figure 5: The rules associated with block cloning. Figure (a) shows the rule for transforming the function CFG. Figure (b) shows the rule for transforming the interprocedural CFG; edges from blocks in other functions are represented with dashed arrows.

or error code) to increase code density. We represent this sharing in our interprocedural CFG (Figure 3a), but hide it at the function layer by using *block aliases* (Figure 3b). A block alias represents a shared block in the context of one particular function, and any analysis or instrumentation performed on the alias is restricted to that function. More formally, let b be a block shared by functions $f_1, \ldots, f_n$. For each such function f_i we create a *block alias*, denoted $a_{(b,f_i)}$. Block aliases replace their corresponding blocks in each function's CFG and instpoints.

A similar case occurs in code produced from languages that support multiple entry points into a function (e.g., Fortran). An alternative representation of such code is as a single function with a set of entry blocks. We instead represent such code as a set of single-entry functions (one per entry point) to keep a consistent function representation. We did this because a single-entry function model can represent multiple-entry functions with a minor cost in efficiency; analysis can be performed on each function individually and then combined, and instrumentation can be copied to each function. However, a multiple-entry model does not support analysis or instrumentation of overlapping single-entry functions, since the multiple-entry representation cannot identify which entry block execution passed through.

2.3 Generating Instrumented Code

The final step required for anywhere instrumentation is to generate new code from the instrumented CFG; an example of this process is shown in Figure 4. The annotations represented by instpoints cannot be directly used to generate code. Instead, we first construct an *augmented* CFG that represents the instrumented code, and this augmented CFG is then used to generate the new instrumented binary code. We generate the augmented CFG in two steps. First, we use *block cloning* to convert all instrumented block aliases to actual blocks, which eliminates sharing of instrumented code. Second, we apply the appropriate graph transformation rule for each instpoint to insert the appropriate code snippets into the graph. The resulting augmented CFG is used to generate code with standard code generation techniques.

We allow users to instrument block aliases as distinct abstractions, and so must ensure the independence of such instrumentation. We do this by *cloning* each such shared block, which creates an identical copy of the block. This copy replaces the instrumented alias in the corresponding function and is added to the interprocedural CFG. We show the block cloning rule in Figure 5, and an example of its application in Figure 4b.

We insert instrumentation into the graph by applying the appropriate graph transformation rule for each instpoint to insert the *code snippets* specified for that instpoint. We show examples of instrumenting function entry points (Figure 4c) and exit points (Figure 4d). The inserted instrumentation is represented by a shaded snippet block.

The result of these two steps is an augmented CFG that contains both original code and instrumentation. We then use this graph to generate the actual instrumented code. This requires three things: compiling instrumentation requests into binary code, updating original and adding additional control flow instructions to ensure the newly generated code matches the CFG, and ensuring that original instructions maintain their original behavior. The first two are performed with standard compiler techniques; the third is handled using the algorithm described in [1].

3. ANY TIME INSTRUMENTATION

The CFG-based instrumentation technique discussed in the previous section generates instrumented code. In this section, we discuss how we insert this instrumented code into the binary at *any time* during execution while imposing *proportional cost*. We define an instrumenter be capable of any time instrumentation if it can insert or modify instrumentation at any point during the execution continuum, including pre-execution (static instrumentation or binary rewriting), while the program is running but has not yet executed code to be instrumented, and while the program is actively executing code to be instrumented. Furthermore, an instrumenter imposes proportional cost if it avoids imposing cost when executing uninstrumented code.

In this section, we describe the techniques Dyninst uses to provide any time instrumentation with proportional cost. To provide context for our work, we first present the techniques used by current binary instrumenters and the *patch-based* instrumentation technique on which our work builds. Current

patch-based instrumenters impose proportional cost, but are not capable of instrumenting at any time because they cannot insert or modify actively executing code. We describe two new techniques that address this lack. The first, *state interception*, directly modifies process state to allow instrumenting executing code. The second, *iterative instrumentation*, provides the ability to modify or remove instrumentation at any point during execution.

3.1 Instrumentation Overview

Binaries rarely include sufficient space to insert instrumentation without moving original code. Instead, instrumenters create a copy of this code that is instrumented and then execute the copied code in place of the original. This code is copied using a technique we call *relocation*. Relocating a region of code produces a new version that emulates the behavior of the original code, but contains sufficient space to insert instrumentation. This new code may execute slower than the corresponding original code due to this emulation; we have previously described relocation in detail and presented an approach that lowers this overhead by emulating only visible behavior [1]. Once a region of code has been relocated, the instrumenter must ensure it is executed in place of the original code. This is done with one of two methods. The first, *patch-based* instrumentation, overwrites the original code with *interception branches* to the relocated code, as shown in Figure 7 [3, 9]. The second, *JIT* instrumentation, instead relocates all code as it is executed [2,10,14]. This second approach does not modify original code but imposes relocation overhead on all code rather than just instrumented code, and thus does not impose proportional cost.

Patch-based instrumentation operates in three phases: selection, relocation, and patching. The *selection* phase selects the code that will be relocated and patched; we refer to this code as the *selected code*. The selected code may range from a single instruction to several functions. The *relocation* phase creates a copy of the code selected in the previous phase; we then use the techniques described in Section 2 to add instrumentation to this region and copy the resulting code into the binary. The *patching* phase patches the binary with interception branches that will jump from the selected region to the relocated region.

The selection phase identifies the selected code, which may consist of a set of instructions, basic blocks, or functions that do not need to be contiguous. The chosen code must be large enough to contain the interception branches with which it will be patched, as we cannot overwrite either data or non-selected code. We also consider two other factors. To satisfy our goal of proportional cost, we want to select as little uninstrumented code as possible. However, we also want to minimize how many interception branches are executed to minimize the cost involved in executing this extra code. Previous approaches have used single basic blocks or single functions. Instead, we select all instrumented functions as well as functions that overlap with an instrumented function; this allows us to improve performance by eliminating interception branches for control flow within the selected group. We represent this set of functions as F.

The relocation phase creates the instrumented code sequence and copies it into the binary. We create a modified CFG using the techniques described in Section 2.3. We then generate code to match this CFG, using a combination of the relocation techniques described in our previous work [1] to move original code and standard compiler techniques to generate instrumentation code. These relocation techniques determine and preserve only the visible behavior of the moved code, allowing any aspects of behavior that are not visible to the program as a whole to differ; this can significantly reduce the overhead caused by relocation.

```
   input  : A set F of selected functions
   output: A set IB of interception branches
1  IB ← ∅;
2  for each function f ∈ F do
3  |  Let v_f be the entry block of f ;
4  |  if f was instrumented at entry then
5  |  |  Let i_f = FunctionEntryPoint(f) ;
6  |  |  Let sb_f = SnippetBlock(i_f) ;
7  |  |  IB ← IB ∪ {(v_f, sb_f)}
8  |  else if v_f was instrumented at entry then
9  |  |  Let i_v = BlockEntryPoint(v_f) ;
10 |  |  Let sb_v = SnippetBlock(i_v) ;
11 |  |  IB ← IB ∪ {(v_f, sb_v)}
12 |  else if s was cloned then
13 |  |  Let cl be the clone of v_f in f ;
14 |  |  IB ← IB ∪ {(v_f, cl)}
15 |  else
16 |  |  Let v'_f be the relocated copy of v_f ;
17 |  |  IB ← IB ∪ {v_f, v'_f}
```

Figure 6: An algorithm for calculating interception branches for each selected function entry point.

The patching phase overwrites original code with interception branches to the corresponding locations in the newly installed instrumentation code. We represent these branches as a set of tuples $IB = \{(s_1, t_1), \ldots, (s_i, t_n)\}$ where each s is the block that will be patched and t the target block of the branch. We show a simplified version of the algorithm we use to derive IB in Figure 6. For each function, we first identify its entry block. If either the function or block was instrumented at its entry, we want to use the inserted snippet block as the target (lines 4-11). Otherwise, we use the block clone as a target (lines 12-14) or the relocated block itself (lines 15-17), as appropriate. The presented algorithm assumes only function and block level instrumentation for simplicity; our implementation includes cases for loop and instruction instpoints as well.

Once we have derived IB we generate code for each interception branch. We use two forms of interception branches: a direct branch or a calculated branch. Direct branches are preferred due to their lower overhead, but may not have sufficient range. If we cannot use a direct branch we instead use a sequence that first calculates the destination address in a register and then uses an indirect branch to reach that destination. Clearly, this sequence both requires more space than a single direct branch and imposes higher overhead; thus, we use a direct branch whenever possible.

To ensure correct execution, interception branches cannot overlap, overwrite non-selected code, or overwrite data. This requires that the selected code be large enough to contain the corresponding set of interception branches. If this is not the case we have two alternatives. The first is to enlarge the region of selected code and recalculate the set of interception branches; this is our preferred approach. The second is to use trap instructions instead of interception branches. Unfortunately, using traps requires the execution of a signal handler and thus imposes higher overhead than a branch. As a result, we use traps only as a last resort.

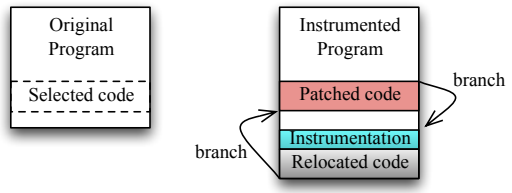


Figure 7: Example of patch-based instrumentation; the original program is shown on the left, and the patched program on the right.

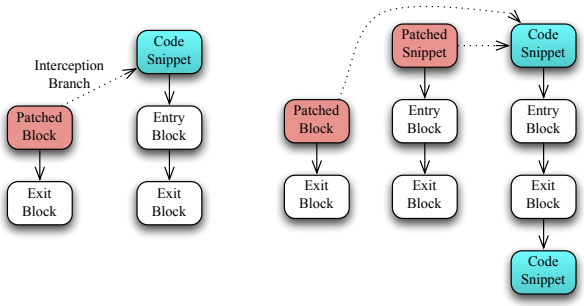
```
   input  : An executing instruction i_j, block
            b = {i_1, ..., i_j, ..., i_n}, and function f
   output: A destination address d
1  if i_j = i_1 ∧ b was instrumented then
2  |  Let p_b = EntryPoint(b) ;
3  |  Let sb_b = SnippetBlock(p_b) ;
4  |  d ← EntryAddr(sb_b)
5  else if i_j was instrumented then
6  |  Let p_i = PreInsnPoint(i_j) ;
7  |  Let sb_i = SnippetBlock(p_i) ;
8  |  d ← EntryAddr(sb_i)
9  else if b was cloned then
10 |  Let cl = {i'_1, ..., i'_j, ..., i_n} be the clone of b in f ;
11 |  d ← Addr(i'_j)
12 else if b was relocated then
13 |  Let b' = {i'_1, ..., i'_j, ..., i_n} be the relocated copy of v_f ;
14 |  d ← Addr(i'_j)
15 else
16 |  d ← Addr(i_j)
```

Figure 8: An algorithm for calculating state interception target addresses.

3.2 State Interception and Iterative Instrumentation

The patch-based algorithm as described above has two weaknesses. First, it does not support instrumenting programs that are executing inside the selected region, since it only inserts interception branches at the boundaries of this region. As a result, the process may continue to execute original code, or execute part of an interception branch (if the branch overwrote multiple instructions). Second, it does not support modifying previously inserted instrumentation if the program is executing inside the instrumented region. We present two techniques, *state interception* and *iterative instrumentation*, that address these problems.

State interception allows us to instrument programs that are executing inside the selected region by directly modifying process state to move from the current execution point to the corresponding destination in the instrumentation code. We do this by directly modifying the execution context of each thread to transfer execution rather than inserting a branch. We determine this destination with the algorithm shown in Figure 8. This algorithm is a generalization of Figure 6 that handles the additional complexity of intercepting control at arbitrary locations in the program instead of just function entries. The inputs to this algorithm are the currently executing instruction i_j, block b, and function f. We identify i_j and b by examining the CFG, since a program counter uniquely identifies an instruction and basic block. If b belongs to a single function we select that function as f; otherwise, we identify the currently executing function by examining the stack. We transfer execution to block entry (lines 1-4) or pre-instruction (lines 5-8) instrumentation if appropriate; unlike interception branches, we do not include function or loop instrumentation because we cannot guarantee execution has not already passed the entry of these constructs. Finally, if the current execution point was not relocated we leave it unmodified (lines 15-16).



(a) Entry Instrumentation (b) Exit Instrumentation

Figure 9: An example of iterative instrumentation. Figure (a) shows the result of instrumenting the function entry point by patching the entry block with an interception branch. Figure (b) shows the result of further instrumenting the exit point; we have patched both the original entry block as well as the entry snippet inserted in the previous step.

Iterative instrumentation allows a user to further instrument previously instrumented code or remove instrumentation at any time. Since our CFG transformation approach essentially inlines instrumentation into the relocated code, further modifying this code is difficult. Instead, we generate a new version of instrumentation taking into account any changes the user has made. We do this in three steps. First, we use the patch-based instrumentation algorithm described above to generate a new version of instrumentation. Second, we patch all previous versions of instrumentation code with interception branches to this new version. Third, we use state interception to immediately transfer any active execution of a previous version to the newest version. We show an example of iterative instrumentation in Figure 9.

Iterative instrumentation allows us to modify instrumentation at any time, but may result in multiple redundant copies of instrumentation. We lessen this cost using two mechanisms. First, if the user has removed all instrumentation we simply restore the original code. Second, we garbage collect old copies of instrumentation when we can determine that a copy can no longer be executed. We determine this with a stack walk over all executing threads; if no address in these walks corresponds with an instrumentation version we conclude that version is dead code and can be collected.

4. RESULTS

Our instrumentation algorithm provides anywhere, any time instrumentation of the binary while imposing cost proportional to the number of locations instrumented. We verified these characteristics with the following experiments. First, we compared the overhead imposed by our approach with other current instrumenters when instrumenting every basic block with a simple counter, and show that the overhead of our approach is comparable to other current approaches. Second, we compared the results of instrumenting function entry and exit points with the same counter metric, and show that our approach results in equal counts for entries and exits on binaries that lack the debugging informa-

```
int main(int argc) {
  do {
    argc -= 1;
  } while (argc > 0);
  return argc;
}
```

(a) Code Listing

Address	Instruction		
400450:	sub	$0x1,	%edi
400453:	test	%edi,	%edi
400455:	jg	400450	<main>
400457:	mov	%edi,	%eax
400459:	retq		

(b) Disassembly

Figure 10: Code listing and disassembly for our entry and exit instrumentation experiment. The entry block of `main` executes once per argument given to the program.

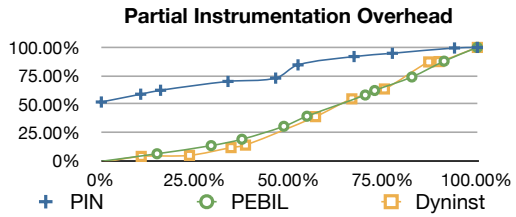


Figure 11: Performance of each instrumenter on partially instrumented programs. The x-axis shows the percentage of total block executions that were of instrumented blocks, and the y-axis is overhead normalized to the overhead of full instrumentation.

tion required by other instrumenters. Third, we compared the overhead of instrumenting a subset of basic blocks to show that our approach imposes proportional cost.

First, we instrumented each program in the SPEC2006 integer suite to count how many basic blocks were executed. We compared our tool with the PIN [10] and DynamoRIO [2] dynamic instrumenters and the PEBIL [9] static rewriter. We did not compare our overhead to that of the popular Valgrind tool [14], as previous research has shown that its overhead is higher than either PIN or DynamoRIO. We measured the time to instrument and execute each benchmark.

The performance results for this experiment are shown in Figure 12; the y-axis is the total time normalized to the uninstrumented run time (100%). The overhead incurred by Dyninst is competitive or better on all benchmarks, with the exception of `bzip` where DynamoRIO incurred lower overhead. The PEBIL rewriter incurs overhead close to Dyninst on many benchmarks; this is unsurprising since both tools use patch-based instrumentation. It performs substantially worse on the `xalan` benchmark due to high instrumentation time, and failed to correctly instrument `gcc` and `omnet`. The DynamoRIO overheads are competitive due to their focus on instrumentation efficiency. Finally, PIN performs worse than other instrumenters. We believe this is due to their more conservative code relocation mechanism; this was particularly harmful on the `h264ref` benchmark.

Second, we compared the accuracy of our function entry instrumentation approach with PIN and PEBIL; to the best of our knowledge DynamoRIO does not provide function-level instrumentation. Our graph transformation approach ensures that entry instrumentation executes once per function invocation. Other instrumenters implement function-level instrumentation in terms of block-level instrumentation, which may cause such instrumentation to execute an incorrect number of times. We constructed a synthetic test program based on a typical optimized code pattern that we have observed in several compilers 10. When compiled, this code results in a loop back-edge to the entry block.

We used each tool to instrument this synthetic benchmark with function entry instrumentation. We considered the tool to have succeeded if it reported a single function entry and failed if it reported multiple function entries. Dyninst successfully reported a single entry, demonstrating that our graph transformation approach ensured that function entry instrumentation executed once even though the entry block executed multiple times. Both PIN and PEBIL reported one function entry for each time the loop executed.

Third, we determined the overhead of partially instrumenting the program. We did this by instrumenting randomly chosen blocks in the `perl` benchmark with a simple counter. We used this counter to determine the percentage of total block executions (as opposed to just percentage of blocks) were of instrumented blocks. We then compared the overhead of each run to the overhead of a fully instrumented benchmark. The data is graphed in Figure 11. The y-axis shows the overhead for each run; 0% indicates zero overhead, and 100% the overhead for a fully instrumented execution. Since Dyninst and PEBIL both use patch-based instrumentation their instrumentation cost decreases to zero as fewer blocks are instrumented; PIN imposes overhead even on uninstrumented code.

5. RELATED WORK

In this section we compare our work to previous work in binary instrumentation, including both dynamic instrumenters and binary rewriters. We compare four characteristics of these tools to our work: the abstractions they use to specify where to insert instrumentation; at what point in the execution continuum they can instrument code; and how much overhead they impose on the binary.

We divide previous work into three categories: *patch-based*, *software dynamic translation* (SDT), and *link-time*. Patch-based instrumenters include PEBIL [9] and Bird [13]. PEBIL provides abstractions for functions and loops, but all instrumentation is performed in terms of instructions and edges. Both tools use techniques similar to those described in Section 3 to insert instrumentation. Bird provides dynamic instrumentation, but cannot iteratively instrument or instrument actively executing code; PEBIL supports only binary rewriting. Since these tools use patch-based instrumentation they provide proportional cost; however, Bird patches single instructions and therefore must rely on traps if the patched instruction is smaller than a branch.

SDT instrumenters [2, 10, 14] apply just-in-time compilation (JIT) principles to binary instrumentation by dynamically constructing a *code cache* that contains an instrumented copy of the program. Execution occurs entirely within this code cache and the instrumenter; as a result, no original code is executed. These tools can instrument instructions, basic blocks, and edges, but do not identify loops. PIN [10] supports function-level instrumentation, but relies on the presence of a symbol table to identify function entries and does not guarantee correct identification of function exits; furthermore, they may incorrectly execute entry instrumentation multiple times per function invocation. Since all code is copied to and executed from the cache these tools impose overhead even when executing uninstrumented code.

Link time rewriters insert instrumentation into original code without either patching or constructing a code cache; instead, the original code is moved as necessary to make room for instrumentation [4, 6]. These tools require link-time information to perform this movement correctly, lim-

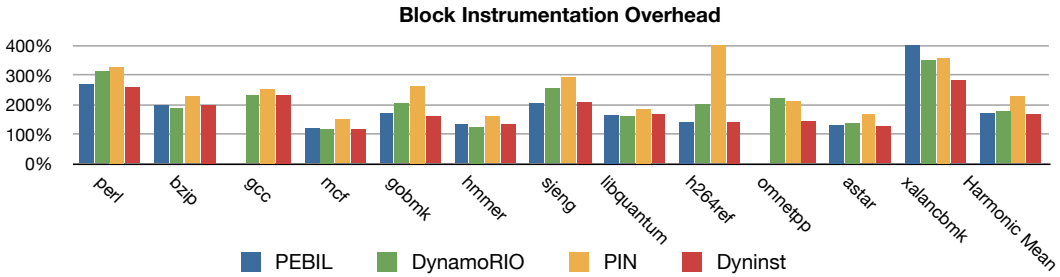


Figure 12: Performance of Dyninst, PEBIL, DynamoRIO, and PIN performing a basic block count. The y-axis is total execution time normalized to the uninstrumented execution time. Missing values indicate the tool did not successfully instrument that benchmark.

iting their application. These instrumenters do not modify the CFG and thus cannot guarantee correct instrumentation of function entries. Since they insert instrumentation directly into original code they do impose proportional cost.

6. CONCLUSION

We have presented techniques for performing anywhere, any-time instrumentation on a binary. By using graph transformations to add instrumentation, we can instrument the binary in terms of instructions, basic blocks, loops, and functions; our experiments show that this approach results in more accurate instrumentation when we instrument functions. By using patch-based instrumentation we can instrument at any time, from binary rewriting to instrumenting currently executing code. Furthermore, we impose overhead proportional to the number of instrumented locations.

7. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments and suggestions. This research funded in part by Department of Homeland Security grant FA8750-10-2-0030 (funded through AFRL), National Science Foundation grants CNS-0716460 and OCI-1032341, and Department of Energy grants DE-SC0004061 and DE-SC0002154.

8. REFERENCES

[1] A. R. Bernat, K. Roundy, and B. P. Miller. Efficient, sensitivity resistant binary instrumentation. In *International Symposium on Software Testing and Analysis (ISSTA)*, Toronto, Canada, July 2011.

[2] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *First Annual International Symposium on Code Generation and Optimization*, San Francisco, CA, USA, March 2003.

[3] B. Buck and J. Hollingsworth. An API for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[4] B. De Bus, B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere. Link-time optimization of ARM binaries. In *ACM conference on Languages, Compilers, and Tools (SIGPLAN/SIGBED)*, Jun 2004.

[5] W. Drewry and T. Ormandy. Flayer: exposing application internals. In *Workshop on Offensive Technologies (WOOT)*, Boston, MA, USA, August 2007.

[6] A. Eustace and A. Srivastava. ATOM: A flexible interface for building high performance program analysis tools. In *Winter 1995 USENIX Conference*, New Orleans, LA, USA, January 1995.

[7] R. Heckel. Graph transformation in a nutshell. In *Electronic Notes in Theoretical Computer Science*, pages 187–198. Elsevier, 2006.

[8] Krell Institute. Open SpeedShop. http://www.openspeedshop.org/wp/, September 2010.

[9] M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely. PEBIL: Efficient static binary instrumentation for linux. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, White Plains, NY, USA, March 2010.

[10] 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 *Programming Language Design and Implementation (PLDI)*, Chicago, IL, USA, June 2005.

[11] M. M. Olszewski, J. Cutler, and J. Steffan. Judostm: A dynamic binary-rewriting approach to software transactional memory. In *16th International Conference on Parallel Architecture and Compilation Techniques*, Brasov, Romania, 2007.

[12] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy (SP)*, Oakland, CA, USA, May 2007.

[13] S. Nanda, W. Li, L.-C. Lam, and T. Chiueh. Bird: binary interpretation using runtime disassembly. In *International Symposium on Code Generation and Optimization (CGO)*, New York, NY, USA, March 2006.

[14] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Programming Language Design and Implementation (PLDI)*, San Diego, CA, USA, June 2007.

[15] J. Newsome, D. Brumley, D. Song, J. Chamcham, and X. Kovah. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *13th Annual Network and Distributed Systems Security Symposium*, San Diego, CA, USA, February 2006.

[16] K. A. Roundy and B. Miller. Hybrid analysis and control of malware binaries. In *Recent Advances in Intrusion Detection (RAID)*, Ottawa, Canada, September 2010.

[17] S. Shende and A. D. Malony. The tau parallel performance system. *International Journal of High*

Performance Computing Applications, 20(2):287–311,
Summer 2006.