BEAST - an Instrumentation Tool for Stripped Win32 Binaries and Its applications

A Thesis Presented

by

SANTOSH SONAWANE

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

STONY BROOK UNIVERSITY

DECEMBER 2009

Stony Brook University

The Graduate School

Santosh Sonawane

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

Professor Tzi-cker Chiueh - Thesis Advisor
Department of Computer Science

Professor Scott Stoller – Chairperson of Defense
Department of Computer Science

Professor Radu Sion
Department of Computer Science

This thesis is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Thesis

BEAST - an Instrumentation Tool for Stripped Win32 Binaries and Its applications

by

Santosh Sonawane

Master of Science

in

COMPUTER SCIENCE
STONY BROOK UNIVERSITY
2009

Many software security techniques require instrumentation of programs either to add run-time checks or to perform dynamic analysis. Unfortunately, commercially distributed application binaries on the Win32 platform are often stripped of their symbol table, and therefore cannot be easily disassembled, let alone correctly instrumented. BIRD is an instrumentation tool that applies an IA-32 disassembler both statically and dynamically, and successfully guarantees that no instruction in an input binary can be executed without being examined first. Unfortunately, the first version of BIRD has several correctness and performance problems. This report describes our experiences of optimizing the first BIRD prototype to remove these problems. In particular, we develop a speculative disassembly technique that reaps most of the performance benefits of static disassembly while ensuring the same disassembly correctness as dynamic disassembly, a bitmap-based situation check algorithm that reduces the performance overhead associated with situation checking to the minimum, and finally a comprehensive in-place instrumentation technique that relies mostly on instruction substitution and drastically cuts down the number of invocations to debug exceptions (*int 3*). Together these performance optimizations reduce the total instrumentation overhead of a set of Win32 programs from 10-14% to 4-5% on average.
To show usefulness of a tool like BEAST, we developed it further to extract a call graph and a system call site control flow graph (scsfg) from stripped Win32 binaries. A call graph gives a fair overview of dependencies between functions and tells which functions

are linked to each other. A control flow graph adds certain level of determinism to the call graph by enlisting the order in which functions are called. In this report we present design, implementation and evaluation of these applications.

Table of Contents

List of Tables

List of Figures

Acknowledgements

First of all, I wish to sincerely thank Prof. Tzi-cker Chiueh for his guidance and support all through the project. I would also like to thank all the present and past colleagues at Rether Networks Inc (RNI). In particular, Dr. Lap-chung Lam for his insightful advice and help throughout my thesis work and Sheng-I Doong, President of Rether Networks, for her kind support.

Chapter 1

1 Introduction

A significant percentage of cyber attacks that take place on the Internet today exploit vulnerabilities in user-level applications or operating systems. A wide variety of techniques have been proposed to protect vulnerable programs from being exploited, such as address space layout randomization[16], control flow integrity verification[5], system call pattern check[17], etc. Most of these research efforts rely on instrumentation of the protected applications and thus require access to their source code. This requirement is unrealistic in practice, because end users rarely have access to the source code of most commercial applications they use. Therefore, for these protection techniques to work, they have to be applicable to executable binaries directly. Therefore, the key enabling technology for such security-enhancing transformations is correct and efficient binary instrumentation.

To instrument a binary, one needs to be able to examine every instruction before it is executed. Ideally, binary instrumentation should be done statically so that its run-time performance overhead is reduced to the minimum. In practice, however, it is difficult to statically analyze and instrument executable binaries, especially for stripped Win32/X86 binaries, i.e. those that don't come with a symbol table, because even state-of-the-art commercial disassemblers such as IDAPro cannot disassemble these binaries with 100% accuracy, which, unfortunately, is what security-enhancing instrumentation requires. One way to solve this problem is to first execute a binary to be instrumented in an emulator[9], [12] and then run the already encountered instructions in native mode. Another approach, as used in *BIRD*[14], is to disassemble as much as possible an input binary to be instrumented statically, and dynamically invoke the disassembler on the binary's areas that cannot be statically disassembled but executed dynamically. *BIRD*'s approach is simpler and more efficient because it eliminates the run-time emulation overhead and produces more efficient instrumentation code.

There are two major components of *BIRD*’s architecture:

a. Static component
b. Dynamic component

BIRD[14] statically disassembles a binary from its entry point, recursively traversing over direct jumps and calls till all the reachable code is disassembled. In this scan, if *BIRD* finds any indirect control transfer instructions, it instruments those instructions with a call to runtime *check()* function which is present in *BIRD*’s dynamic component, a DLL.

BIRD’s dynamic engine is a DLL which has *init()* and *check()* functions. *Init()* function initializes the runtime data structures using the statically gathered information. *Check()* function verifies targets of indirect control transfer instructions. If target is already in a known region then *BIRD* continues normal execution of the program. If target lies in completely unknown region, then *BIRD* starts dynamic disassembly from the target address and marks the region as known area.

This report describes the design, implementation and evaluation of an optimized binary instrumentation tool for stripped Win32/X86 binaries called *BEAST*. *BEAST* is built on top of *BIRD*, and solves three performance problems in *BIRD*. First, the static disassembler of *BIRD* relies on a pre-defined threshold to determine if a disassembled region in the input binary is code or not. If this threshold is set to a higher value, a smaller percentage of the input binary is successfully disassembled, the disassembly coverage is lower but the disassembly accuracy may be higher. If this threshold is set to a lower value, a higher percentage of the input binary is successfully disassembled, the disassembly coverage is higher but the disassembly accuracy may be lower. Because it is difficult to devise a good threshold value that can be universally applied to all Win32 binaries, for safety *BIRD* has no choice but to set the threshold to a conservatively high value by default, and thus sacrifices the disassembly coverage and run-time performance. *BEAST* successfully overcomes this dilemma with a *speculative disassembly* mechanism that optimistically disassembles the input binary, and uses the optimistically disassembled results only when they are confirmed at run time. As a result, speculative disassembly is able to achieve the best of both worlds: high disassembly accuracy and coverage and high run-time performance.

The second problem with *BIRD* is its reliance on *int 3* instructions for in-place instrumentation. Given an instrumentation point *P* and an instrument code *C*, *BIRD* replaces the 5-byte region starting at *P* with a jump instruction to *C* whenever possible, and replaces the 1-byte region starting at *P* with an *int 3* instruction otherwise. Because an *int 3* instruction is much more expensive than a jump instruction, the fact that *BIRD* did not employ *int 3* instructions efficiently causes serious performance penalty for certain Win32 applications. To address this problem, *BEAST* performs a series of optimizations that explores all possible ways of using jump instructions and eventually successfully reduces the static and dynamic *int 3* instruction counts to the minimum.

The third problem with *BIRD* is the time it spends to perform the expensive lookups at runtime for finding the appropriate entry in the hash tables. *BEAST* performs faster searches as compared to *BIRD* by maintaining an in memory comprehensive 2bit bitmap of the entire code region. This bitmap acts as a O(1) hash since there are no collisions.

Chapter 2

2 Related Work

Accurate disassembly is the foundation for binary instrumentation. There are many excellent commercial disassemblers such as IDAPro[1] and OllyDbg[2] that can achieve high disassembly coverage and accuracy for Win32/X86 binaries. However, they cannot guarantee zero disassembly error and therefore cannot serve as the basis for binary instrumentation.

Moreover, because many commercial packers embed anti-debugger and anti-disassembler techniques inside their unpackers, none of these disassemblers can disassemble packed binaries without occasional crashing, let alone instruments them.

In contrast to commercial disassemblers, *BEAST* chooses to avoid disassembling the unpackers embedded in packed binaries. It does this by incorporating a heuristic technique to precisely identify the point at which the original binary is fully unpacked, and then invokes the disassembler only on the unpacked original binary. This design decision not only greatly simplifies *BEAST's* design and implementation complexity, but also improves its overall robustness significantly.

There are several unpacker programs on the web such as PEiD[3], ASPackdie[7], etc. They use a signature-based approach to identify the packer behind a packed binary, and apply a packer-specific unpacker to unpack the binary. Because of the reliance of packer-specific unpackers, they are typically ineffective against unknown packers or new variants of existing packers.

In contrast, *BEAST* is designed to unpack packed binaries in a packer-independent way. Bintropy[4] computes a binary's entropy to determine whether the binary is packed/encrypted or not. *BEAST* applies a similar idea to determine if a binary is packed or not statically, and if the original binary has been fully unpacked at run time.

There are also several Win32 binary instrumentation tools. However, none of them currently are capable of handling packed binaries as *BEAST*.

Dyninst[11] applies static disassembly to Win32/X86 binary rewriting and optimization. It requires full debugging information to guarantee the safety of instrumentation. *BEAST* does not have this requirement.

Dynamo[9] is another binary interpretation and optimization system that uses a software-based architectural emulator to detect so-called hot traces, i.e. sequences of frequently executed instructions, and optimize them dynamically so that they can run faster. Despite the binary interpretation overhead, Dynamo is able to achieve a non-trivial speedup of 15%-22% for some binaries when compared with their native execution time.

Unfortunately, the Win32/x86 version of Dynamo[10] runs much slower and incurs a performance overhead of between 30% to 40%. The reasons cited for this slowdown are lack of documentation for Win32 API and additional implementation complexities that do not exist on the UNIX platform.

PIN[12] takes a similar dynamic binary emulation and instrumentation approach as Dynamo. It also performs instrumentation entirely at run time. PIN provides a very good Instrumentation interface. It is generic in the sense it separates architecture dependant complexities from the user by providing a well designed instrumentation interface. Since PIN allows instrumentation, it causes runtime overhead. PIN uses optimizations like liveness analysis on this instrumentation code so as to save runtime overhead. *BEAST* is not providing a custom instrumentation interface.

PIN uses a complete dynamic instrumentation approach. It uses ptrace to capture the application context and attaches to it. Its complete dynamic instrumentation approach is somewhat costlier than *BEAST's* speculative approach which minimizes runtime costs, however, *BEAST* introduces considerable amount of debug exceptions (*int 3s*) at runtime owing to its approach of in-place instrumentation. PIN completely rewrites code to other location by caching code and then creates links to these locations. This approach is costlier in terms of memory usage but it obviates the need of using debug exceptions and thus avoids time spent on expensive user to kernel switches due to *int 3*.

PIN creates code cache called traces. They are sequence of instructions to be executed serially. PIN tries to link these traces directly to avoid the transfer of control to VM for indirect branch resolution. For direct branch it can directly link to a single trace, however, for indirect branch it performs branch target prediction. It uses the lea/jecxz approach similar to DynamoRIO to find if the predicted branch is correct. This approach is similar to *BEAST's* approach of pushing target and looking it up in the O(1) cache. If PIN is not able to find the target then it does indirect target resolution by using VM. Here it creates a trace and start its execution. It is similar to *BEAST's* approach of finding if a target address is known/unknown; if unknown then invoke dynamic disassembler. However, *BEAST* introduces *int 3s* during dynamic disassembly if it discovers a short indirect branch. PIN does not have to do this due to code rewriting. Though the number of dynamic *int 3s* introduced are considerably less, they can prove costlier if *int 3s* are executed in a loop.

PIN uses lookups in its hash table to find traces linked with an indirect branch. *BEAST* also performs such lookups, however, our 2 byte-bitmap hashing mechanism, though a little costlier in terms of memory is otherwise quite efficient. It performs O(1) search to find the data structure corresponding to an indirect branch.

PIN handles function returns. To optimize returns they use code cloning, where they create copy of a function per calling context. *BEAST* does not handle return sites. It assumes a behavior where functions return to the caller. Since caller was already disassembled, we assume return site will be disassembled. However, this may not be the case in codes where the return addresses are changed at runtime by modifying EIP. Thus we need to handle return in a similar way as we handle indirect control transfer instruction.

PIN has used register reallocation so as to ensure that it does not modify application registers. *BEAST* simply pushes the registers on stack and pops them out when it is done with handling.

PIN uses a spilling area where it stores the registers content when it reuses them for reallocation. This spilling area needs to be thread specific. *BEAST* does not need a spilling area, but at run time *BEAST* calculates relocated address of a target. If it finds that the target is relocated, it needs to redirect control to this region. This computation of relocated target and actual transfer to the target address needs to be done in a thread specific manner. We exploit the thread specific stack for this. If *BEAST's check()* routine - runtime handler function, is handling an indirect call and it realizes that the target address is relocated then it changes the return address of *check* routine to the relocated target address and pushes the actual return address of check routine below it. Thus after execution of the *check* function, instead of returning to the caller's site, *BEAST* jumps directly to the callee. After callee reaches return, it returns to callers return site.

HDTrans[13] is a light-weight IA32-IA32 binary translation system that combines a set of simple but effective translation techniques with well known trace linearization and code caching optimization techniques. Performance experiments from HDTrans show that static pre-translation is effective only when expensive instrumentation or optimization is required.

QEMU[19] is a processor emulator that uses a dynamic translator to convert instructions in an emulated program into the host's instruction set. Such conversion can be considered as one form of binary disassembly and instrumentation.

Similar to *BEAST*, QEMU also uses page protection to detect SMC and invalidates previously translated code. The slowdown of QEMU compared with native execution is typically a factor of 5 or more.

BIRD [14] is a general binary instrumentation infrastructure that is specially designed to facilitate the development of software security for Windows/X86 binaries. Given a binary program, *BIRD* first statically disassembles it to uncover as many instructions as possible, rewrites it to allow run-time interception at all indirect jumps and calls, and then dynamically disassembles those areas that are explored during execution. Through a combination of static and dynamic disassembly, *BIRD* achieves both high accuracy and high coverage. However, for packed binaries, *BIRD* 's static disassembler can easily fail on the unpacker code. Therefore it cannot do much for packed binary instrumentation.

Chapter 3

3 Overview

3.1 Overview of BEAST

BEAST is derived from *BIRD*, which has two limitations.

First, despite a significant amount of efforts spent on fine-tuning its disassembly algorithm, *BIRD* could not guarantee the static disassembly error rate is zero in all cases when it uses a disassembly algorithm that is more aggressive than recursive traversal[18]. To do so, *BIRD* needs to drastically decrease the coverage of its static disassembly, and thus defers most of the disassembly work until run time. How to achieve 100% static disassembly accuracy while maintaining a high disassembly coverage for Win32/X86 binaries remains an elusive goal for *BIRD*.

Second, like most other existing binary instrumentation systems, *BIRD* cannot handle any form of self-modifying code: Once a binary's code region is disassembled, *BIRD* assumes it never changes and thus never bothers to invalidate the code region's disassembled result if it indeed gets updated.

To overcome the first limitation, *BEAST* uses a speculative disassembly mechanism that logically applies a conservative and an aggressive static disassembly algorithm on the input binary, and produces a conservative disassembly result (high accuracy but low coverage) and a speculative disassembly result (low accuracy but high coverage), respectively. Whenever applicable, the conservative disassembly result overrides the speculative result. Initially *BEAST*'s run-time component only relies on the conservative disassembly result to determine if a certain part of the input binary has been successfully disassembled. However, whenever *BEAST* needs to dynamically disassemble a newly discovered region, it first checks if it can directly leverage the speculative disassembly result before invoking the disassembler. Because most of the speculative disassembly result is correct, i.e. its disassembly error is non-zero but low, *BEAST* can effectively avoid most of the dynamic disassembly work. As a result, this speculative disassembly mechanism can simultaneously achieve the best of both worlds: 100% disassembly accuracy with high disassembly coverage.

Assume a packed binary B contains a compressed or encrypted form of an original binary O. To instrument packed binaries, *BEAST* aims to identify the point during the execution of B at which O is completely recovered. After the unpacker in B completes the unpacking process, it transfers the program's control to the entry point of O. Therefore, the transition from B's unpacker to O corresponds to a jump whose target is a page that is written at run time. *BEAST* uses a page status tracking mechanism to detect such transitions.

BEAST consists of a static component, which performs static disassembling and instrumentation and is implemented as a stand-alone executable, and a dynamic component, which performs dynamic disassembling and page status tracking and is implemented as a DLL that is injected into an instrumented binary when it is executed.

Given an input binary to be instrumented, the static component of *BEAST* first checks if the input binary is packed or not. If the input binary is not packed, *BEAST* statically disassembles the binary to produce a conservative and a speculative disassembly result, and instruments indirect branches and other instrumentation points of interest to the user. If the input binary is packed, *BEAST* will not disassemble or instrument it at all.

In both cases, *BEAST* appends to the end of the input binary useful information that could facilitate dynamic disassembly and instrumentation such as base loading address, module size, code region offset, and packed/non-packed flag.

BEAST's dynamic component contains a *page status tracking* mechanism to detect the end of unpacking during the execution of a packed binary, and invokes a dynamic disassembler and instrumentor when the original binary is completely unpacked.

3.2 General Architecture of BIRD/BEAST

BEAST, in order to provide complete coverage, intercepts all indirect branches and instruments them with a call to runtime *check()* function. Thus, there are two major components of *BEAST*'s architecture:

a. Static component
b. Dynamic component

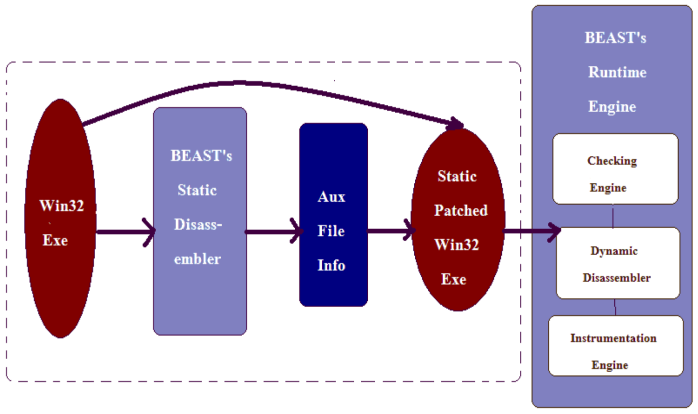

Figure 1 BEAST's architecture

3.2.1 Static component

BEAST statically disassembles a binary from its entry point, recursively traversing over direct jumps and calls till all the reachable code is disassembled. In this scan, if *BEAST* finds any indirect control transfer instructions, it instruments those instructions with a call to runtime "check" function which is present in *BEAST*'s dynamic component, a DLL. Later, *BEAST* uses speculative disassembly to uncover as much of the Unknown Area. It marks these instructions as speculative and dumps related information at the end of the binary. Finally, *BEAST* dumps the indirect branch and unknown area related information at the end of the original binary.

To clarify the concepts of

1. Indirect call to a known area and
2. Indirect call to an unknown area consider the following example.

Original code segment:

```
10000000      Push ebp
10000001      Mov ebp, esp
.
10000005      call 0x100000F0
.
1000000A      Mov eax, [0x100000F0]
1000000F      call eax
.
.
1000001A      mov eax, [0x100001F0]
```

```
1000001F        call eax
10000021        add ebx, 0x0A
.
```

Assume that *BEAST* reaches 0x10000000 during static disassembly. Instruction at 0x10000005 is a direct call to 0x100000F0, thus 0x100000F0 becomes a known region. Therefore, indirect call at location 0x1000000F turns out to be an indirect call to known region. Let us assume that 0x100001F0 cannot be statically reached. In this situation the indirect call at 0x1000001F becomes an indirect call to an unknown region.

An indirect control transfer instruction is patched with a trampoline code which transfers control to a stub. For example, the instruction at 0x1000001F, call eax which is a 2 byte instruction is patched with a 5 byte jump to a stub. The region from 0x10000021 to 0x10000023 is the region belonging to next instruction and thus is getting overwritten by the 5 bytes jump. We term this region as *extended region.*

```
1000001F        jmp 10100010
.
.
10100010        push eax
10100011        call check
10100017        call eax
10100019        add ebx, 0A
1010001C        jmp 10000024
```

Above code snippet shows how the location at 1000001F is patched with a jump to a stub at 10100010. This stub pushes the target address on the per thread stack, which can be retrieved in the check function as a parameter. *Check()* performs the runtime handling and returns back. The rest of the code in the extended region is copied as it is to ensure correct execution.

3.2.2 Dynamic component

BEAST's dynamic engine is a DLL which has *init()* and *check()* functions. Init function reads in the dumped information and populates the corresponding data structures. Check function handles the targets of indirect branches. It checks if the target is already known. If yes then it returns. If the target lies in a speculative region and starts that speculative region, then *check()* merges the speculative information about that function and does runtime instrumentation of indirect branches lying in that speculative region. If the target lies in totally unknown region, then *BEAST* dynamically disassembles the region till it hits a known region or it is unable to disassemble further. Thus, the unknown area goes on reducing as *BEAST* uncovers more unknown regions.

Chapter 4

4 Speculative Disassembly

Because binary instrumentation tools modify input binaries, they require 100% disassembly accuracy to preserve the input binaries' original execution semantics. The experiences with *BIRD* suggested that zero disassembly error comes with an expensive cost in terms of disassembly coverage. Fortunately, for legitimate Win32/X86 binaries, disassembly errors occur infrequently. The key idea of *BEAST*'s *speculative disassembly* mechanism is to leverage as much as possible static disassembly results by exploiting run-time information to confirm those that are guaranteed to be free of disassembly errors.

As shown in Chapter 6, speculative disassembly significantly reduces the amount of code that needs to be dynamically disassembled and instrumented, thus effectively performs most disassembly and instrumentation statically while achieving both high disassembly accuracy and coverage.

4.1 Static Component

4.1.1 Baseline Disassembly Algorithm

BEAST's static disassembly algorithm is derived from *BIRD's* static disassembly algorithm, which consists of two passes. In the first pass, the disassembler applies recursive traversal to traverse the input binary's control flow graph starting from its main entry point, and discovers all instructions that are reachable through direct branch instructions, i.e., branch instructions whose target address is known statically. All the other bytes in the input binary that are not reachable in the first pass are called *unreachable* bytes. In the second pass, the disassembler chooses some unreachable bytes as *starting* instructions, and then performs the same control flow graph traversal from these instructions. During the second-pass traversal, the disassembler accumulates a confidence score on the probability that each unreachable byte block correspond to an instruction sequence. At the end of the second pass, a byte block is considered as instructions if and only if its score exceeds a certain threshold and the address of its first byte is evidenced to be a starting instruction.

BIRD uses several heuristics to select the starting instructions, such as function entry point identification, jump table identification, etc. None of them is 100% reliable. Through the confidence score mechanism, the disassembler attempts to capture the essential difference between data and instructions bytes: it is unlikely that data bytes can accumulate multiple evidences that indicate that they are instructions.

At the end, this disassembly algorithm partitions an input binary into a *known area*, which it knows for sure corresponds to instructions or data, and an *unknown area*, whose nature it cannot be absolutely sure of. In addition, all the indirect branch instructions in the known area are instrumented so that when they jump to the unknown area at run time, the dynamical disassembler is invoked on the targets.

4.1.2 Speculative Disassembly Algorithm

In order to ensure that the known area it produces statically does not contain any disassembly error, *BEAST* needs to set the threshold of the confidence score to a very high value. However, this turns a large portion of the input binary into the unknown area, which may lead to significant run-time overhead. Instead of leaving the unknown area completely to the dynamic disassembly, *BEAST* disassembles this area as much as it can using a speculative disassembly algorithm, and records the disassembled results and their corresponding instrumentations. However, *BEAST* does not actually apply these instrumentations to the unknown area.

To allow *BEAST*'s run-time component to leverage static disassembly results, the static disassembler organizes its results into basic units called *code thunks*.

Operationally, *BEAST* builds a code thunk by traversing forward from a starting point using the following rules:

1. Upon a conditional branch instruction, follow both of its arms.
2. Upon a direct jump instruction, jump to the target address.
3. Upon a direct call instruction to a callee that always returns, continue with the next instruction following this call instruction without descending into the callee's function body.
4. Upon an indirect jump or call instruction whose target address can be determined by static analysis, treat this instruction as its direct counterpart and apply Rule 2 or 3.
5. Upon a direct call whose callee does not return, a return instruction, an indirect branch instruction whose target address cannot be determined statically, end the current code chunk.

Each code thunk thus constructed corresponds to a set of instructions that are guaranteed to be reachable by applying recursive traversal from its starting point.

To increase the coverage of the speculative disassembly, *BEAST* searches the entire input binary, including its data areas, for possible starting points for growing code thunks.

From a binary's PE header, *BEAST* first identifies the start and end of the code sections. If the target of a direct call instruction falls within a code section, it is a starting point candidate. If a sequence of bytes looks like a function prolog, the base of this byte sequence is also a starting point candidate. If a data pointer points to a code section, the address contained in the pointer is a starting point candidate. If a 4-byte aligned word contains a value that falls within the address range of a code section, this value is also taken as a possible starting point. For areas in the input binary that do not show any signs of being instructions, *BEAST* disassembles them anyway if there are no disassembly errors. Each code thunk extracted from a binary is represented as a tree, and code thunks are connected with each other because one code thunk transfers control to another code thunk through a direct call or an indirect call whose target address can be statically determined. If a code thunk is valid, the code thunks to which it is connected are also valid. Two code thunks may overlap with each other with or without conflicts. A conflict occurs when the two code thunks cover a common byte range and they have different interpretations for the common bytes. *BEAST* resolves a conflict between two code thunks removing the code thunk with a significantly lower confidence score. In case their confidence scores are similar, *BEAST* removes both code thunks. If two code thunks overlap without conflict, *BEAST* keeps both code thunks.

4.2 Dynamic Component

Whenever *BEAST's* run-time engine intercepts an indirect branch that transfers the control to an unknown area, it first checks whether the target address goes into any code thunk in the speculative disassembly result. If the target address confirms any speculatively disassembled code thunk, *BEAST* directly includes this code thunk into the known area and applies its pre-computed instrumentations to this code thunk.

Intuitively, a target address confirms a code thunk if the code thunk also thinks the address starts an instruction. Because it is expensive to record the start address of every instruction in every speculatively disassembled code thunk, *BEAST* only compares the target address of every intercepted indirect branch with the starting point addresses of code thunks, and confirms a code thunk if it’s starting point address matches an indirect branch's target address.

We can fine-tune this approach to instruction and basic block level. A target address may confirm a code thunk if the code thunk also thinks that this address either starts an instruction or starts a basic block. This level of granularity will be possibly only with additional storage of information, especially we need to store starting address of each and every instruction or starting address of a basic block respectively. Moreover, it is easy to merge function level information like the number of callees, addresses of indirect control transfers in this function etc. So we resort to the function level granularity approach.

Even with this approximate confirmation approach, *BEAST* is able to eliminate a majority of dynamic disassembling as required by *BIRD* configured with the same confidence score threshold.

As mentioned earlier, once *BEAST’s* run-time engine confirms a code thunk, it leverages the thunk's disassembly and instrumentation results. As a result, not only the amount of dynamic disassembly and instrumentation is greatly reduced, more importantly the quality of instrumentation is significantly improved. In *BIRD*, instrumenting indirect branches in the dynamically disassembled results is always done through the expensive *int 3* instruction because it cannot afford to perform advanced control flow analysis required to safely instrument these branches using simple jump instructions. Because the trampoline code for instrumentation is statically prepared, *BEAST* can carry out instrumentation with almost no overhead as it only needs to paste the trampoline code over the target indirect branch instruction or instrumentation.

The disassembly algorithm *BEAST* applies to a dynamically unpacked binary is different from its static disassembly algorithm in the following ways.

First, an unpacked binary may not have a proper PE header, which contains information such as the session table, the import table, etc., because the unpacker typically performs the job of the Windows loader and therefore can remove all PE header information that the loader needs.

Second, because the disassembly time for an unpacked binary is counted towards the total run time, *BEAST* cannot afford to use advanced program analysis to improve disassembly accuracy and/or instrumentation efficiency. Instead, after an unpacked binary is created, *BEAST* scans it once to derive possible starting points and grow code thunks, and leaves whatever areas left to run-time disassembly.

To improve the efficiency of instrumentations for dynamically disassembled code, *BEAST* records the target addresses of all direct and indirect branches it encounters. Every time it encounters an indirect branch destined to an unknown area that does not correspond to any pre-computed code thunk, it instruments this indirect branch by replacing it with a jump instruction if the indirect branch's address does not match any target addresses *BEAST* maintains.

In addition, if later on the target of a newly discovered branch matches that of a jump instrumentation *BEAST* replaces that jump instrumentation with an *int 3* instruction.

Chapter 5

5 Efficient In-place Instrumentation

The design of *BIRD* intercepts indirect branches by replacing them with an unconditional jump instruction that transfers control to a *check()* function, which performs the necessary processing to ensure that every instruction in a binary program be examined before it is executed. However, it is not always possible to replace an indirect branch instruction (indirect jump or indirect call) with a jump instruction because a short indirect branch instruction takes two bytes but an unconditional jump instruction takes 5 bytes. Replacing a short indirect branch instruction with an unconditional jump instruction is feasible if any of the three bytes following the short indirect branch instruction is not a target of any branch instruction in the program. When an indirect branch instruction is replaced by a 5-byte jump instruction, let's call the additional three bytes as the corresponding *extended region*.

If it is infeasible to replace a short indirect branch with a jump, the only other alternative to instrumenting the indirect branch is to replacing it with an *int 3* instruction, which is also 2 bytes. Because an *int 3* instruction triggers an exception, it is much more expensive than a jump instruction. *BIRD* replaces an indirect branch with an *int 3* instruction if

1. There exists in the same function a direct jump whose target falls into the extended region, or
2. There exist other indirect jump instructions in the same function.

Empirically, *int 3* instructions are used rather frequently in *BIRD*-instrumented binaries and thus account for its major performance overhead. In this section, we will describe a series of optimizations that eventually eliminate almost all *int 3* instructions during static instrumentation.

5.1 Comprehensive Branch Target Analysis

In the original design of *BIRD*, the extended region of each short indirect branch is assumed to lie in the same function as the short indirect branch. However, this is not necessarily always the case in practice. To address this problem, *BEAST* statically collects the target addresses of all direct calls and jumps in the known area, and dynamically checks if the target of any indirect call or jump or any dynamically discovered direct call or jump falls into any extended region. If the extended region of a short indirect branch contains the target of a direct call or jump, the short indirect branch can only be instrumented using an *int 3* instruction. If the extended region of a short indirect branch is found to contain the target of an indirect call or jump or a dynamically discovered direct call or jump at run time, the interception of this short indirect branch is changed from a jump instruction to an *int 3* instruction at run time.

Following code example describes how an extended region of an indirect control transfer instruction may contain start of some other function.

```
//Function F1
1000001A    ret
10000020    Mov eax, 0x1000001A
10000023    Jmp eax
//Function F2
10000025    push ebp
10000026    mov ebp, esp
```

The instruction at location 10000023 has an extended region up to 10000027. However, the instruction at 10000025 starts function F2. Note that 10000023 belongs to function F1 and 10000025 belongs to function F2. If we replace 10000023 with a 5 byte jump then we effectively relocate 10000025. Thus, a direct call to 10000025 needs to be relocated. Since we want to reduce this complicated analysis we patch 10000023 with a one byte *int 3* instruction.

When an indirect jump or call instruction is intercepted at run time, its target address is checked against the known area list to determine if a previously unknown code region is discovered. In this optimization, *BEAST* checks this target address against the list of extended regions to determine if any extended region is invalid; in addition it performs the same check for targets of direct calls and jumps that are discovered at run time. When *BEAST* finds an extended region is invalid, it converts the interception of the corresponding short indirect branch to *int 3*.

In summary, with this optimization, *BEAST* replaces an indirect branch with an *int 3* instruction if

1. There exists in a direct jump whose target falls into the extended region, or
2. There exists a direct call whose target falls into the extended region.

Statically, only the targets of direct calls and jumps in the known area are known. *BEAST* takes a speculative approach to replace all short indirect branches that satisfy the above requirements with a 5-byte jump, but reserves the option to fall back to an *int 3* instruction at run time when the target of an indirect call/jump or a dynamically discovered direct call/jump is found to invalidate an extended region.

That is, by piggybacking extended region invalidation with run-time target address check, *BEAST* is able to remove a portion of the *int 3* instructions used in *BIRD*-instrumented binaries.

5.2 Patching Direct Jumps

Even with the branch target analysis optimization described in the previous subsection, because the extended regions of many short indirect branches in real-world Win32 binaries contain targets of direct jump instructions, these short indirect branches can only be intercepted using an *int 3* instruction. *BEAST* eliminates these *int 3* instructions by patching the direct jumps whose target falls into the extended region of an existing short indirect branch. That is, when a direct jump *D* renders invalid the extended region of a short indirect branch *B*, instead of patching *B* using an *int 3* instruction, *BEAST* will patch *D* so that eventually *D* will jump to where it is supposed to go in the original binary. In this case, both *B* and *D* are processed in *check()*, as well as the control transfer between them. More generally, whenever an instruction *I1* is to be patched and its extended region contains the target of a direct jump *I2*, *BEAST* chooses to patch *I1* with a jump instruction, and proceeds recursively with *I2*, which itself now needs to be patched. This recursion continues with *N* steps, where *N* is a configurable parameter, and stops if further patching is required. In that case, the original short indirect branch is patched with an *int 3* instruction.

Following code example explains the above mentioned approach of patching direct jumps through N recursions:

```
10000000      call eax
10000002      add ebx, 0x0A
.
10000010      je 10000004
10000012      add ecx, 0x0A
.
10000020      je 10000014
10000022      add ebx, 0x 0A
.
10000030      je 10000014
.
```

In the above code snippet example, the instruction at 10000000 is call eax which is an indirect call instruction taking 2 bytes. When it is patched with a 5 bytes jump, we relocate 10000002 inside the stub of 10000000. Now, when the instruction at 10000010 tries to jump at location 10000002, it may have to jump to the appropriate relocated address inside the stub. *BIRD* did not handle this situation. It would have replaced call eax at 10000000 with an *int 3* so that the jump at 10000010 can safely jump with any relocation. However, *BEAST* patches 10000010 with a 5 bytes jump and relocates the jump at 10000010 to jump at appropriate location in the stub. By doing so *BEAST* needs to take care of any instructions which jump into the extended region of this newly replaced region. *BEAST* thus finds out that instruction at location 10000020 and 10000030 jump into the newly replaced extended region of 10000010. *BEAST* patches these instructions with a 5 byte region and applies the above procedure recursively till it finds a point where the currently relocated extended region does not contain target of any direct jump.

In contrast, *BIRD* only patches indirect branches because they are the only ones that need to be intercepted. On the surface, it appears that the idea of patching *conflicting* direct jumps, i.e., those whose target falls into some extended region increases the number of instructions that need to be patched, and indirectly increases the number of *int 3* instructions used. In practice, this is not the case for two reasons. First, whenever *BEAST* patches a conflicting direct jump, it must already avoid one *int 3* instruction. Therefore, patching conflicting direct jumps cannot increase the number of statically placed *int 3* instructions. Second, as *BEAST* shifts through the target instructions for patching, it can successfully avoid int 3 if the current instruction is at least 5-byte long or its extended region does not contain any target. Therefore, in some cases, *BEAST* can indeed decrease the number of statically placed *int 3* instructions.

With the addition of the "patching direct jump" optimization, *BEAST* replaces an indirect branch with an *int 3* instruction if

1. There exists a direct call whose target falls into the branch's extended region.

Empirically, as shown in the Performance Evaluation section, statically placed *int 3* instructions are all but eliminated with this optimization, because the chance of a direct call's target falling into a short indirect branch's extended region is very slim in practice.

To patch a direct unconditional jump, *BEAST* uses the following stub:

```
Push target_delta
Call check
Jmp [location]
```

where *target_delta* is the difference between the original target of the direct jump and the stack location holding it, and is an input parameter to *check(),* and *location* is a memory location that holds the actual target address as calculated by *check()*.

Direct conditional jumps require additional processing because they typically depend on the contents of the EFLAGS register. Accordingly, *BEAST* simulates a direct conditional jump with unconditional jumps using the following stub:

```
Push target_delta
Call check
```

```
Je if_equal
Jmp if_no_equal
if_equal:
Jmp [location]
if_not_equal:
Fall-through instruction
```

where *target_delta* and *location* mean the same as the direct unconditional jump case.

5.3 Dynamic Instrumentation Algorithm

In addition to statically determining whether to patch a short indirect branch with an *int 3* instruction using the criterion described in the previous subsection, *BEAST* also needs to dynamically patch those short indirect branches that are discovered at run time. Because *BEAST* cannot afford much analysis at run time, implementation complexity is the key consideration:

1. A dynamically discovered short indirect branch is always patched with an *int 3* instruction.
2. If a dynamically discovered direct jump conflicts with the extended region of a patched instruction *I*, then find out why *I* needs to be patched. If *I* is not a short indirect branch, it must be because *I*'s target conflicts with the extended region of another instruction. Continue tracing backwards this way until the instruction is a short indirect branch, and convert the patching for this branch to *int 03*. All the intermediate instructions are reverted to their original non-patched form.

Empirically more than 60% of the patched branches were direct branches with their target addresses known statically. Exploiting this to fix their target addresses in the stub removes the need to call *check()* at run time. For example, the stub for patching a direct conditional jump becomes

```
Je if_equal
Jmp if_not_equal
If_equal:
Jmp actual_addr_in_stub
If_no_equal:
Fall-through instructions
```

where *actual_addr_in_stub* is the statically computed target address of the direct branch.

5.4 Implementation Issues

5.4.1 Multithreading

Because most Win32 binaries are multi-threaded, *BEAST* also provides multi-threading support by making *check()* re-entrant, particularly the target address of a patched indirect branch in one thread should be separate from that in another thread.

Moreover, *BEAST* guards the common data structures used in *check()*, e.g. unknown area list, extended region list, function entry point information, etc., with proper fine-grained locks so that the accesses to them are both thread-safe and efficient.

5.4.2 Low-Overhead Exception Checking

BEAST reduces the total count of statically placed *int 3* instructions, however, at the expense of increasing the number of invocations of *check()* at run time. Even though an *int 3* exception takes much longer than a simple function call, the fixed overhead in *check()* to perform target address lookups can still be quite costly and adds up quickly. With the "direct jump patching" optimization, the possibility of a direct jump's target hits an extended region increases, and the size of the extended region list also increases accordingly. We have tried to use a cache to capture recently appearing extended regions, but the cache proves to be of not much use because most accesses to it result in a miss. Moreover, because the target of a patched branch may potentially be relocated into a stub of another patched branch, *BEAST* has to check the target of a patched branch to ascertain whether the target is in the known area or is relocated or not, and incurs the associated expensive lookup overhead. However, most of the times, a patched branch's target is known and not relocated. In these common cases, ideally the look-up time should be constant O(1), and the expensive look-up should be avoided.

To speed up the common-case look-up, *BEAST* constructs a two-bit-per-byte metadata map for each code module. For a given code module, say a DLL, its metadata map indicates whether each byte in it is known and/or relocated. At run time, given the target address of a patched branch, *check()* first consults with the corresponding metadata map with the target address. If the target address byte is known and not relocated, *check()* returns control to its caller immediately. If the target address byte is known and relocated, *check()* proceeds further to calculate the relocated address of the target. If the target address byte is unknown and not relocated, *BEAST* performs another lookup to search for the corresponding speculatively disassembled function, and performs dynamic disassembly starting with the target address if such a speculative function is not found. It is not possible for a target address to be unknown and relocated.

These per-module metadata maps drastically reduce the look-up performance overhead in *check()* at the expense of additional memory usage. For a 10-Mbyte binary program, the additional memory consumption for these metadata maps is 2.5 Mbytes, which seems to be modest.

5.4.3 Relocation in DLLs

As opposed to Executables, Dynamic linked libraries on Windows may not load at the same base address as provided in the header of the DLL. With this in mind, those entries which depend on absolute addresses are added to the relocation table in the DLL. Windows loader fixes the entries from the relocation table by adding a delta which is calculated by finding the difference between the actual loading address and the address mentioned in the header. *BEAST* takes care of all such instructions, whether newly added by *BEAST's* instrumentation code or previous instructions which are relocated to a new stub, by adding them to the relocation table and removing the older entries as needed.

BEAST also stores only the relative virtual address (rva) of required data and fixes them at runtime when *init()* function loads the dynamic data structures using this statically dumped information.

Chapter 6

6 Performance Evaluation

6.1 Methodology

In this section we present the set of programs on which *BEAST* was tested. Further, we list various configurations of *BEAST* which were used to demonstrate gradual improvement in performance. We also specify the technical specifications of the environment on which *BEAST* was tested. Finally, we describe the testing tool set used to make various measurements like elapsed time, startup time, *int 3* counts, and static disassembly coverage improvement.

6.1.1 Set of programs

To evaluate the effectiveness of *BEAST*, we test it on two major categories of programs, batch programs and GUI programs. Table 1 shows the set of batch programs. These programs are comp (comparing two 4MB files), find (finding a given character from a 4MB ASCII file), sort (sorts a 4MB ASCII) and ping (ping ip address with 1024 bytes data). Table 2 shows set of GUI applications on Windows on which *BEAST* was tested. These programs are Yahoo Messenger, Firefox, Acrobat Reader, Movie Maker and Safari.

6.1.2 Set of configurations

BEAST was built in incremental improvements over *BIRD.* As *BEAST* evolved, 5 versions of *BEAST* were produced, each succeeding version an incremental improvement over previous one. Each of the versions significantly differs from each other with respect to the instrumentation principle. Version 1.0 of *BEAST*, already handled the speculative disassembly part. Version 1.1 through version 1.3 attempted to reduce the number of *int 3s* by applying various instrumentation techniques for replacement of an indirect control transfer instruction. Version 2.0 mainly concentrated on reduction of time spent in the *check()* function performing the expensive lookups related to target address and towards multithreading support.

Version 1.1 performs global analysis as opposed to version 1.0's per-function analysis. There are various aspects of Win32 binaries which make disassembly harder. It is tough to decide function boundaries since a function can have multiple return sites. Also, inside a function segment a return statement may not necessarily lie at the end. In fact there are cases where a return is followed by an indirect jump. By performing global analysis version 1.1 finds out targets of direct calls (starting address of a function) lie in the extended region of some indirect branch. In such situation, the indirect branch is replaced with *int 3*. Since version 1.0 did not perform global analysis, it was never sure if any start of a function lies in the extended region of any indirect branch. So it replaced all the short indirect jumps (jump instructions whose length is less than 5 bytes) with *int 3.* Also, if the extended region of an indirect branch consists of a target of any direct jump then it instruments that indirect branch with an *int 3.*

Version 1.2 took a step further by selectively patching direct unconditional jumps whose target lie in the extended region of an indirect branch. Thus the indirect branch could now be instrumented with a 5 byte jump to the stub. However, if targets of conditional jumps lie in the extended region of an indirect branch then the indirect branch is replaced with an *int 3*.

Version 1.3 performed patching of conditional direct jumps as well. It has to simulate a direct conditional jump with a series of unconditional direct jumps as explained in Chapter 5.3. This change brought a drastic reduction in the number of *int 3s* introduced during static time instrumentation. It was owing to the fact that most of the direct jumps are conditional direct jumps since most of the programming constructs like *for, while, do-while, if* are transformed into direct conditional jumps.

Version 2.0 reduced the time spent at runtime on the expensive lookups performed to find the data structure elements corresponding to the given target address. It used the 2bit-per-byte bitmap metadata as explained in Chapter 5.4.2. It also incorporated a better support for multithreading by providing fine grained locks and using per-thread stack as opposed to its predecessors which used a shared data structure with a big lock around.

6.1.3 Environment

BEAST was tested on a Pentium IV 2.8 GHz, 1 GB RAM with Windows XP, service pack 2. *BEAST's* static component was compiled using GCC on cygwin and dynamic component is organized as a DLL which requires Microsoft SDK 2003 and Microsoft VC++ 2005 compiler.

6.1.4 Testing tools

Binaries instrumented with *BEAST* were tested for total run time, startup time, *int 3* counts and improvement in static disassembly coverage. We wrote a tool which measured the total elapsed time for set of batch programs. We simply exec'd the test application and when the control returned to our program we measured the elapsed time. For GUI programs, we used the Microsoft VC++ StartupIdleTime function, which returns control to our program when the GUI application turns idle. This measurement, though not accurate, gives a proportionate measure of startup time for various configurations of *BEAST*. We calculated the number of *int 3s* by simply using a counter. However, for dynamic *int 3* calculation for GUI programs during startup, we used shared memory across our testing tool and the test application binary. The test application writes the number of *int 3s* to the shared memory. When control is returned to our testing tool, we take the *int 3* count from shared memory. To measure improvement in disassembly coverage and overhead due to speculative disassembly, we compare total elapsed time for the application binaries when they are patched with *BEAST v1.0* against *BEAST v1.0* with speculative disassembly disabled.

Programs	***Original App.***	***BIRD***	***BEAST v1.0***	***BEAST v2.0***
comp	1.42	1.66	1.65	1.56
find	0.73	0.92	0.87	0.74
sort	0.63	0.95	0.91	0.77
ping	3.12	3.18	3.16	3.16

TABLE 1: Total Elapsed Time for batch programs patched using different version of *BEAST* and *BIRD*

6.2 Performance Results

In this section, we discuss performance of various *BEAST* configurations. We present measurements based on time, coverage and *int 3* counts. Further, we tell how much of space overhead is incurred by *BEAST* in order to achieve this performance.

6.2.1 Time based

Programs	***Original App.***	***BIRD***	***BEAST v1.0***	***BEAST v1.3***	***BEAST v2.0***
Yahoo Messenger	1.6	2.7	2.6	2.42	2.31
Firefox	0.45	0.67	0.64	0.58	0.55
Safari	1.2	3.1	2.55	2.34	2.11
Movie Maker	0.12	0.90	0.51	0.44	0.38
Acrobat Reader	0.87	2.1	1.61	1.42	1.33

TABLE 2: Startup time for batch programs patched using different versions of *BEAST* and *BIRD.* Startup times are considerably higher because of the overhead due to the initial loading of data structures from the dump.

Table 1 tabulates total elapsed time for a set of batch programs and Table 2 tabulates startup time for a set of GUI applications. Time measurements are performed for Original Application and application patched by *BIRD*, *BEAST v1.0* and *BEAST v2.0*. For binaries like Acrobat Reader, Movie Maker and Safari, startup penalty is very high due to loading of dumped information. *BEAST v2.0* is faster than *BEAST v1.3* because it reduces time spent on lookups at runtime. *BEAST v1.3* reduces *int 3* exception handling time as compared to *BEAST v1.0,* thus reducing execution time. However, the time taken by init function slightly increases because more amount of information is dumped in higher versions of *BEAST v.1.0 Int 3* count for batch programs is very low, so we did not show performance of *BEAST v1.3* for batch programs. *BEAST v1.0* performs speculative disassembly thus reducing runtime disassembler invocations, so it is faster than *BIRD.*

6.2.2 *Int 3* based

Table 3 compares *BEAST v1.0* and *BEAST v2.0* on the basis of total number of *int 3* dynamically encountered for a set of GUI applications. Measurements are performed for the startup time of GUI applications. Programs like Safari, Acrobat Reader and Yahoo Messenger have considerable amount of dynamic *int 3s* because of the increasing number of dynamic disassembly invocation as compared to Movie maker and Firefox.

Programs	***BEAST v1.0***	***BEAST v2.0***
Yahoo Messenger	8133	642
Firefox	338	7
Safari	24542	6589
Movie Maker	5378	12
Acrobat Reader	25132	2048

TABLE 3: Comparison between *BEAST v1.0* and *BEAST v2.0* on the basis of dynamically encountered *int 3s*

6.2.3 Space overhead

BIRD appends to the application binary a lot of information like stubs, indirect branch information, speculative function information and unknown areas. Further, *BEAST v2.0* dumps a map which stores 2-bits of information for ever byte in the code section. Thus, it increases image size by one fourth the size of code section. Thus the overall size of binary may become one and half times the original size.

6.3 Analysis

In this section, we present analysis of gradual changes in *BEAST* and their corresponding impact on performance.

6.3.1 *BIRD* to *BEAST v1.0*

BEAST introduces speculative disassembly over *BIRD*. If we observe Table 1and Table 2, we notice that *BEAST v1.0* performs better than *BIRD*. *BEAST* uses the speculative disassembly results after verification of the speculative code at runtime. Thus it saves a certain amount of time as compared to *BIRD*. Startup time taken by *BIRD* should have been lesser as it is not meant to dump speculative information. However, while testing of *BIRD* we purposely dump the speculative information and load it during init. This is done so as to have a fair overview of time saved by speculative disassembly on a long run.

Table 4 shows the total number of speculative functions verified over a certain span of time against the total number of invocations to dynamic disassembly. Since the applications are GUIs we recorded these values after a span of 6 seconds. We cannot get an unbiased result for GUI applications because they wait for inputs from user, unlike batch programs.

Programs	***Speculative hits***	***Dynamic disassembly invocations***
Yahoo Messenger	1166	36
Firefox	93	2
Safari	1306	18
Movie Maker	477	29
Acrobat Reader	11686	109

TABLE 4: Number of speculative function hits and dynamic disassmebler invocations

6.3.2 *BEAST v1.0* to *BEAST v1.3*

In comparison with *BEAST v1.0, BEAST v1.3* drastically reduced the count of *int 3s* introduced during static disassembly. Table 5 shows the drastic reduction in *int 3* count, introduced during static disassembly and Table 3 shows reduction in actual number of *int 3s* encountered at runtime.

Programs	***BEAST v1.0***	***BEAST v2.0***
Yahoo Messenger	21121	47
Firefox	126	0
Safari	3513	23
Movie Maker	15385	0
Acrobat Reader	28134	44

TABLE 5: Comparison between *BEAST v1.0* and *BEAST v2.0* on the basis of *int 3s* introduced during static disassembly

6.3.3 *BEAST v1.3* to *BEAST v2.0*

BEAST v2.0 implemented a bitmap which saved unnecessary time wasted on lookups inside runtime *check()* function. Table 2 shows time saved by *BEAST v2.0* over *BEAST v1.3*.

6.4 Qualitative Evaluation and Some Issues

BEAST was tested on some other Windows application, however, due to some issues as listed below we could not evaluate *BEAST* on these softwares.

1. Internet Explorer

BEAST works fine on IE executable, however IE extensively depends on DLLs from the Windows folder and these DLLs cannot be modified as Windows does not allow us to do so. There is a registry entry which specifies that DLLs from Windows/System32 folder cannot be modified.

2. Integrity checks in some DLLs

It was found that some of the DLLs had some integrity checks which did not allow the patched DLLs to run successfully. For example, one of the main DLLs of Safari performed some checks and threw exception when it discovered

Chapter 7

7 Applications of *BEAST*

This section describes design, implementation and evaluation of two applications which we built using *BEAST*. Call graph and system call site control flow graph extractor were built using *BEAST*.

7.1 Call Graph

A call graph is a directed graph that represents calling relationships between subroutines in a computer program [20]. Specifically, each node represents a procedure and each edge (f,g) indicates that procedure f calls procedure g. Thus, a cycle in the graph indicates recursive procedure calls.

7.1.1 Data Structures

BEAST maintains a per function data structure which it populates during disassembly. This data structure includes following important members:

```
struct function{
        uint            flags:8;        //SYSCALL related
        uint            sure_mode:1;    // speculative-0
        char *          name;
        uint            start_addr;
        uint            end_addr;
        varray_type             indirect_jmps;
        varray_type             replaced_jmps;
        varray_type             indirect_calls;
        varray_type             short_brs;
        varray_type             djmps;
        varray_type             direct_calls;
        varray_type             func_segs;
        varray_type     bb_array;       //Basic block array for this function
        varray_type     parents;        //Callers
        varray_type     children;       //Callees
};
```

Function boundaries are indicated by start and end address fields. In case of multiple returns end address is set to 0. Indirect/Direct jumps and calls are stored in an array. BB_array stores the entire control flow. Parent and children arrays store the callers and callees of the function. Children array holds the addresses of call sites whereas callgraph_array holds the callee functions' start addresses. Storing this construct is useful for runtime creation of call graphs from the unknown regions.

7.1.2 Call Graph Construction

A call graph is generated by traversing from the entry point of the binary. During static analysis, till all the reachable code region is discovered, the graph is constructed. This call graph is dumped at the end of the binary which is further used for runtime graph creation.

Speculative analysis is the phase where *BEAST* predicts some bytes as instructions and performs disassembly starting from those points. This information is verified at runtime when control is transferred to the starting address of any of the speculatively disassembled function. During speculative analysis, per function call graphs are constructed and dumped. During runtime, if a speculative region corresponding to a function F is verified then the entry point of F is added to the caller function's callgraph_children array.

During runtime when an indirect control transfer instruction is encountered, *check()* routine is invoked. This routine performs checks for the target address. If this target lies in a completely unknown region then dynamic disassembly takes place. During dynamic disassembly call graph is generated in the same way as it is during static disassembly.

Inside a function F, if an instruction X is a direct call instruction then we add the callee function to the callgraph_children array of function F. If X is an indirect call then we patch it so as to handle it during runtime. At runtime, when we are handling the indirect call, we know the runtime target address of X and add the callee function corresponding to that address to the callgraph_children array.

7.1.3 Evaluation

Programs	***Total Nodes***	***Static Nodes***	***Speculative Nodes***	***Average No. of Children***
Batch programs	533	104	429	3
Outlook Express	28	10	18	1
MSN Messenger	2082	28	2054	4
Safari	11344	312	11032	6
Movie Maker	13398	412	12986	5

TABLE 6: Call Graph for Win32 executables: Enlisting Total Call Nodes, Nodes found statically, speculatively and Average Number of Children nodes

Table 6 lists count of call nodes discovered during speculative and static analysis for the given set of Win32 binaries. It also shows the average number of children per call node.

Batch programs is a set of batch programs which perform search, sort, find on 4MB data files.

7.2 System Call Site Control Flow Graph

A control flow graph (CFG) in computer science is a representation, using graph notation, of all paths that might be traversed through a program during its execution [21]. A System Call Site Control Flow Graph bothers only to maintain those nodes in a control flow graph which lead to system call nodes and trims all other nodes.

7.2.1 Design

We distinguish following types of nodes in the control flow.

- Call site node: A node corresponding to a call instruction in a binary.
- Return site node: A node corresponding to the physically next instruction of a call instruction.
- Entry node: A node corresponding to the entry point of a function.
- Exit node: A node corresponding to the exit point of a function.

7.2.1.1 Basic Block

A full-fledged basic block control flow data structure is constructed by *BIRD*. We used this already existing control flow data structure to build the system call site control flow graph.

The basic block data structure looks like this:

```
struct bb_struct {
        int     index; //index of BB in function->bb_array
        uint    begin; //start address
        uint    end;    // end address
        struct edge_struct *    in_edges;
        struct edge_struct *    out_edges;
        int     visited; //for BFS
        scsfg_list      call_site; //First call site in this BB
        scsfg_list      last_ret_site;  //Last return node in this BB
};
```

Each basic block of function F is added to the bb_array of function F. index gives the index in this array. begin and end are the start and end addresses of this basic block. call_site holds the first call site in this basic block and last return site holds the last return node in this basic block. In and out edges are of type edge_struct which is defined as follows:

```
struct edge_struct {
        int                 type;    // 0-jmp, 1-call, 2-thru
        struct bb_struct         *src; //source basic block
        struct bb_struct         *dst; //destination basic block
        struct edge_struct *out_next; //next out edge
        struct edge_struct *in_next; //next in edge
};
```

Type of edge can be jump/call/fallthrough. Src and dst are the source and destination basic blocks. Also multiple edges to/from a basic block are stored as a linked list.

BIRD builds basic blocks while traversing the binary. Whenever a control flow transfer instruction like direct jump is encountered, following steps are performed:

1. If the target address does not start a basic block, a new basic block is formed.
2. In case the target address lies in between an already existing basic block, the basic block is split into two parts and the end address of the upper basic block is adjusted accordingly.

7.2.1.2 SCSFG

As mentioned above, four different types of nodes are identified, namely call, return, entry and exit node. Further *SCSFG* maintains only those nodes that are either system call nodes or they lead to system call nodes. This section describes creation of scsfg and trimming of it.

The data structure to store scsfg node looks like this:

```
struct scsfg_node{
  unsigned int     type:4; // type of the node
  unsigned int              syscall_related:2; // whether leads to syscall
  unsigned int              visited:2; // for traversal
  unsigned int      sys_num; // system call number
  unsigned int      address; // the return/function address
  char *                    name; //name if any
  list_entry                cfg_prev; //Previous CFG entry
  list_entry                cfg_next; //Next CFG entry
  scsfg_list                table_next; //Next in table
};
```

The sys_num field indicates a custom number assigned to uniquely identify each node. Address stores the return node address for call/return nodes and function's start address for entry/exit nodes. Cfg_prev/cfg_next holds entries to nodes immediately preceding/succeeding current node. These edges are maintained as linked list as there can be multiple prev/next edges. table_next holds the scsfg node that immediately follows this node in the scsfg table which is maintained globally.

A global array of all the scsfg nodes is maintained. During disassembly, at the start of analysis of every function, an entry/exit scsfg is created for that function. Unique and consecutive sys_nums are assigned to these nodes and they are added to the global array of scsfg nodes. If a function corresponds to the entry point of the binary then its entry scsfg node is marked as the entry node of the entire system. Further this node can be used as a starting point (head node) for traversal of the entire scsfg. When a call instruction is encountered, a call scsfg node is created with unique sys_num and a corresponding return node is created with consecutive sys_num. Call_scsfg node points to the entry scsfg node points back to the return node. At the end of per function analysis, the local scsfg of that function is condensed, inorder to fix the links between scsfgs pertaining to different basic blocks. After this condensing, entry scsfg node points to all the first call sites of the succeeding basic blocks. If the succeeding basic block has no call sites but a return site then we point the entry scsfg node to the exit node. If the succeeding basic block does not have any call site/exit nodes then a Breadth First Search is performed on all the successors of this succeeding basic block and their first call sites are linked to the entry point. Further the last return site of each basic block is linked with the succeeding basic blocks' first call sites. Again if no call sites are present then the last return site of the current basic block is linked to the first call sites of the succeeding basic blocks successors. The process continues till the entire local scsfg is constructed.

7.2.1.3 Non System Call Nodes pruning

A node is considered to be a system call node if it corresponds to any of the functions declared in NTDLL.h. A system call related node is a node which eventually leads to any of the functions declared in NTDLL.h. SCSFG is created by pruning any nodes which are not system call related.

We find all those nodes which are system calls by looking up their name against the entries in NTDLL.h. If it finds that an node F is a system call then it marks all its parent nodes (cfg_prev) as system call related nodes and recursively carries on this marking process till it reaches origin (first node). For static disassembly in sure mode, origin node is the entry node corresponding to the entry point of the binary. For speculative disassembly, it can be any entry scsfg node of any function from which disassembly started.

Once marking of all the nodes is done, we prune all those nodes which are not system call related. We straightaway remove all the edges that lead to non system call related nodes. If we prune all the nodes except for the system call leaf nodes then it becomes a highly trimmed scsfg which contains only the information about the calling sequence of system calls.

7.2.2 Runtime SCSFG

Statically and speculatively constructed scsfgs are dumped at the end of the binary. When this binary is loaded, scsfg is reconstructed from this dumped information. When an indirect call control transfer instruction is encountered, the target address is looked up against the speculative functions' start address. If target address starts at a speculative function's start address then we merge the scsfg associated with the speculative function that corresponds to this target. If the indirect control transfer instruction is a jump instruction then its target is generally not the start of any other function. A jump instruction belonging to function F, will not jump to start address of any other function, however, it may jump at the start address of F which will be already disassembled by then.

If the indirect call control transfer leads to an unknown region, then we start construction of scsfg at runtime. This construction is exactly identical to the static construction of scsfg. The entry point of the newly generated scsfg is linked with the caller's call site node and the exit node is linked to the return node of the caller. If the indirect control transfer instruction is a jump instruction then we do exactly the same steps as above except that at the end we link the entry node of the newly discovered scsfg with the last return site of the basic block containing the indirect jump instruction.

7.2.3 Evaluation

SCSFG extraction was tested on the set of binaries shown in Table 7.

Program	Entry Nodes Count	Call Nodes Count	System call nodes Count
Batch programs	533	1473	46
Outlook Express	28	43	18
MSN Messenger	2082	4475	96
Safari	11344	34567	1231
Movie Maker	13398	38712	996

TABLE 7: It shows the total number of entry nodes, call nodes and system call nodes in the scsfg of some well known Win32 binaries.

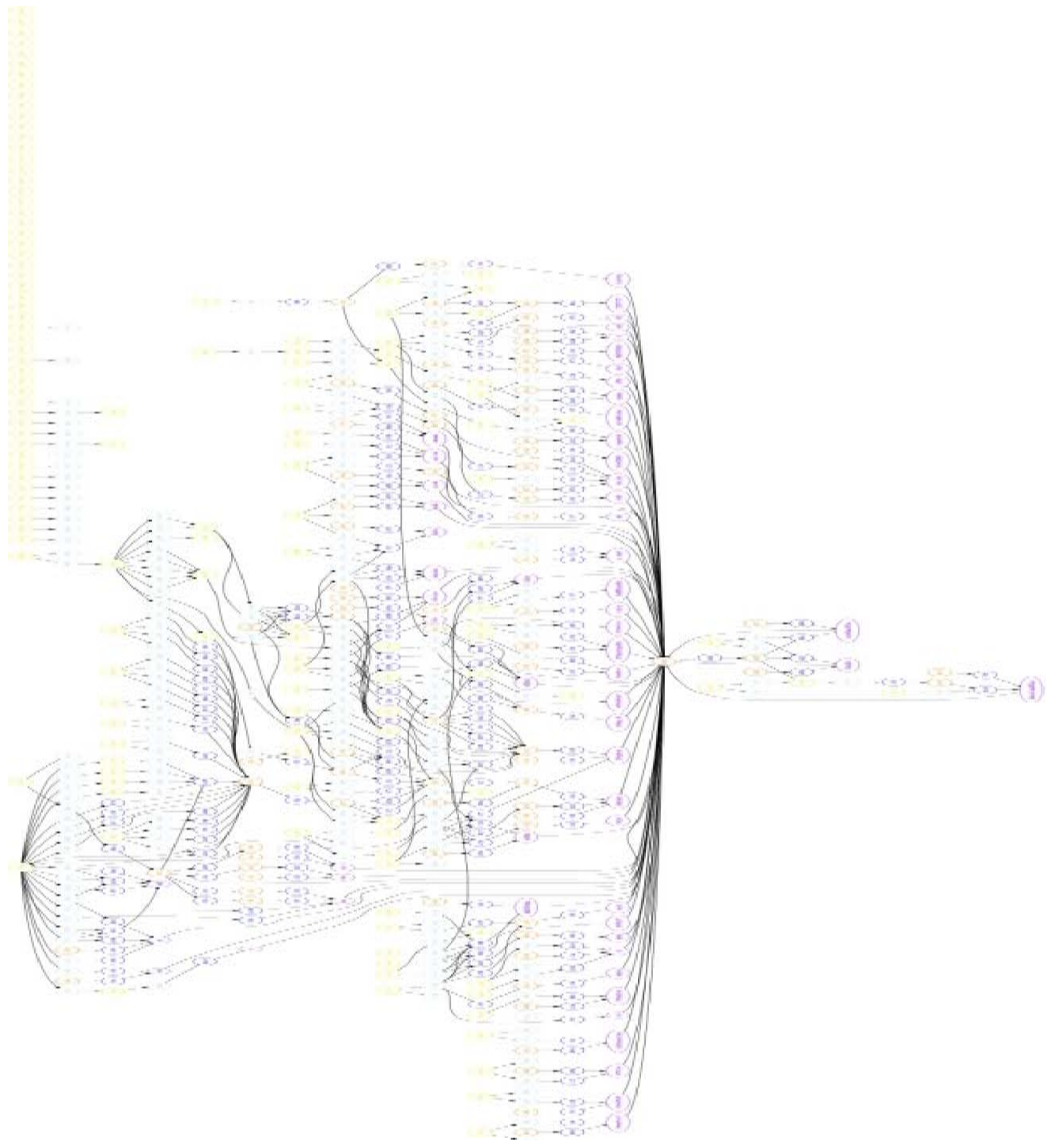

Figure 2: SCSFG graph for BATCH programs.

Chapter 8

8 Conclusion and Future Work

Binary analysis and instrumentation is an enabling technology for enhancing the security strength of commodity binaries through program transformation. This report describes the design, implementation and evaluation of a new Win32/X86 binary instrumentation system called *BEAST*, which improves over existing binary instrumentation tools with two new capabilities: (1) the ability to achieve zero disassembly error while minimizing the run-time performance overhead, and (2) the ability to instrument binaries packed by commercial packers by detecting the end of binary unpacking. More concretely, this work makes the following research contributions:

- A speculative disassembly and instrumentation mechanism that achieves both high disassembly accuracy and disassembly coverage,
- A low-overhead execution tracking mechanism that can detect the end of binary unpacking using virtual memory hardware and entropy/data-flow computation
- A fully working *BEAST* prototype that has been successfully tested against a set of Windows console programs as well as the Acrobat Reader, Safari Browser, Movie Maker and Windows Live Messenger.

Also the application of *BEAST* to extract call graph and system call site control flow graph shows how a disassembly/instrumentation tool can be used to extract complete and accurate call graph and scsfg. Moreover, it reaps all the benefits of *BEAST* thus having a very low runtime overhead.

BEAST source code can be further improved by porting it to an object oriented language like C++. A well designed class hierarchy and template classes can allow *BEAST* to be extended for any underlying architecture. Providing a better interface will help user to develop applications using *BEAST*. Other than above mentioned applications *BEAST* can be further used for a variety of applications:

1. BDBG: A full-fledged binary debugger which can perform data flow analysis and provide users with correct debugging decisions. BDBG can maintain a stack as well as jump trace. It can be further attempt to tell the exact reason for a corrupt instruction. One may think of developing such an application as it will greatly aid programmers who need to debug binaries.
2. PAID: Program semantics aware intrusion detection can perform checks against the extracted scsfg to verify if the invocation to system calls are in an expected fashion. Any violations should be detected and prevented.

There are certain DLLs which perform integrity checks and thus forbid any patches on them. *BEAST* may need to find out a way to bypass these checks. For programs like Internet Explorer and IIS which depend a lot on System DLLs, *BEAST* is not able to get a complete coverage because Windows forbids users from modifying the DLLs in Windows/System32 folder. *BEAST* needs to tackle this problem in order to patch all the DLLs in the set of Internet Explorer binaries.

Even though a *BEAST* is functional for certain packed binaries, it has been experimented only with custom binaries. It needs to be further tested for large binaries. One of the pitfalls of handling packed binaries is to perform the disassembly entirely at runtime. It incurs a very high runtime overhead.

Bibliography

[1] IDAPro. IDA Pro Disassembler.
http://www.datarescue.com/.

[2] OllyDbg.
Oleh Yuschuk.
http://www.ollydbg.de/.

[3] PEiD.
JIBZ, QWERTON, SNAKER, AND XINEOHP.
PEiD. http://peid.has.it/.

[4] Robert Lyda, James Hamrock.
Using Entropy Analysis to Find Encrypted and Packed Malware
IEEE Security and Privacy, vol.5, no.2, pp.40-45, Mar/Apr, 2007.

[5] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, Jay Ligatti.
Control-flow integrity.
Proceedings of the 12th ACM conference on Computer and communications security,
Alexandria, VA, USA. Pages 340-353, 2005.

[6] UPX. the ultimate packer for executables.
http://upx.sourceforge.net/.

[7] ASPack. the advanced Win32 executable file compressor.
http://www.aspack.com/.

[8] PECompact. PE packer.
http://www.bitsum.com/pec2.asp.

[9] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia.
Dynamo: a transparent dynamic optimization system.
ACM SIG-PLAN Notices, 35(5):1--12, 2000.

[10] D. Bruening, E. Duesterwald, and S. Amarasinghe.
Design and implementation of a dynamic optimization framework for windows.
In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization FDDO-4*,
December 2000.

[11] Dyninst.
An application program interface (api) for runtime code generation.
http://www.dyninst.org/.

[12] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood.
Pin: building customized program analysis tools with dynamic instrumentation.
In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190--200, New York, NY, USA, 2005. ACM Press.

[13] Swaroop Sridhar, Jonathan S. Shapiro and Prashanth P. Bungale.
HDTrans: A Low-Overhead Dynamic Translator.
In *Proc 2005 Workshop on Binary Instrumentation and Applications*, 2005.

[14] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi cker Chiueh.
Bird: Binary interpretation using runtime disassembly.
In *Proc of Code Generation and Optimization(CGO) 2006*, pages 358--370, 2006.

[15] DEP.
Data Execution Prevention in Wikipedia.
http://en.wikipedia.org/wiki/Data_Execution_Prevention

[16] Mark Russinovich.
Inside the Windows Vista Kernel: Part 3.
http://www.microsoft.com/technet/technetmag/issues/2007/04/VistaKernel/

[17] Lap Chung Lam and Tzi cker Chiueh.
Automatic extraction of accurate application-specific sandboxing policy.
In *7th International Symposium on Recent Advances in Intrusion Detection*, Sophia Antipolis, France, September 2004.

[18] B. Schwarz, S. K. Debray, and G. R. Andrews.
Disassembly of executable code revisited,
IEEE *Ninth Working Conference on Reverse Engineering*, Richmond, October 2002.

[19] Fabrice Bellard.
QEMU, a Fast and Portable Dynamic Translator.
In *Proc of USENIX 2005 Annual Technical Conference*, *FREENIX Track*, pp 41-46, 2005.

[20] Call Graph
http://en.wikipedia.org/wiki/Call_graph

[21] Control Flow Graph
http://en.wikipedia.org/wiki/Control_flow_graph