© 2017 Mika Latimer

TRACE-WEIGHTED BINARY COMPARISON FOR SOFTWARE UPDATE MANAGEMENT

BY

MIKA LATIMER

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Associate Professor Michael Bailey

ABSTRACT

As software systems grow in complexity, they become difficult to manage. This applies to both developers, who must maintain the code, and users, who must decide when to accept updates. A software patch intended to fix one error may introduce a new problem in a more important part of the executable. This can be difficult to predict even when source code is available, which is often not the case. To help simplify this decision, we introduce a technique to estimate the impact of a software patch, based on how the software has been used in the past. We analyze programs for which we have source code to check the results, but our approach is intended to be useful even when there is no source code available. By analyzing a large number of related programs, which tend to share a substantial amount of code, we show that adding execution traces to the static binary analysis creates much more informative results than binary diffing alone.

TABLE OF CONTENTS

LIST OF TABLES v

LIST OF FIGURES vi

LIST OF ABBREVIATIONS vii

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 RELATED WORK 3
- 2.1 PatchAdvisor 3
- 2.2 BMAT 3
- 2.3 DarunGrim2 4
- 2.4 BinSlayer 5
- 2.5 CoP 6
- 2.6 BinHunt 7
- 2.7 MalwareHunt 8
- 2.8 Matching Execution Histories 9

CHAPTER 3 IMPLEMENTATION 10
- 3.1 Overview 10
- 3.2 Execution Tracing 10
- 3.3 Basic Blocks 12
- 3.4 Code Paths 13
- 3.5 Patching and Reconstruction 13
- 3.6 Basic Block Matching 14
- 3.7 Binary Analysis 17

CHAPTER 4 EVALUATION 20
- 4.1 Test Cases 20
- 4.2 Performance 21
- 4.3 Results 23
- 4.4 Use Comparison: Split 31

CHAPTER 5 CONCLUSION 33
- 5.1 Future Work 33

APPENDIX A BRANCH TRACE STORE 35

APPENDIX B BINARY PATCH TYPES 37
B.1 ExamDiff Pro 37
B.2 bsdiff 38
B.3 Courgette 39

APPENDIX C EXECUTABLE FORMAT 40
C.1 Linux ELF 40
C.2 Windows PE 40

APPENDIX D SINGLE VARIABLE CHANGE 44

REFERENCES 48

LIST OF TABLES

3.1 Conditional branch pairs. 16

4.1 Block analysis runtimes. 23
4.2 Description of split options. 30

C.1 List of 64-bit ELF header fields. 41
C.2 List of 64-bit ELF section header fields. 41
C.3 List of COFF header fields. 42
C.4 List of PE section header fields. Some fields are intended for object files and not executables. 42

LIST OF FIGURES

3.1 Architecture of the analysis system. Shaded areas indicate files involving the new version of the binary. 11
3.2 Overview of trace generation. 12
3.3 Two cases demonstrating each possibility for branches in A and C to be equivalent. Blocks A and B are in the old executable and C and D are in the new one. Dashed lines indicate an assigned matching. 16
3.4 Top 20 score:block change ratios in coreutils, 8.22-8.23. 18

4.1 A pseudorandom branch code block. 21
4.2 Average slowdown for selected coreutils programs. 22
4.3 Top 20 scores for coreutils 8.22-8.23. 24
4.4 Blocks for coreutils 8.22-8.23. 24
4.5 Blocks for coreutils 8.23-8.24. 25
4.6 Scores for coreutils 8.23-8.24. 26
4.7 Blocks that changed in version 2.29.1 of binutils. 27
4.8 Fraction of blocks that changed version 2.29.1 of binutils. . . . 27
4.9 Score for changes in version 2.29.1 of binutils. 28
4.10 Percent modified blocks in versions 2.26-2.27 of binutils. . . . 29
4.11 Score for versions 2.26-2.27 of binutils. 30
4.12 Score for split version 8.22-8.23 using different options. 31
4.13 Score for split version 8.23-8.24 using different options. 32

A.1 IA32_DEBUGCTL MSR as used for BTS. Shaded bits are reserved. 36
A.2 DS save area. We pad the BTS buffer to start at a page boundary, but it is only required to start at a doubleword boundary. 36

B.1 A possible patch for the ExamDiff Pro file format. 37
B.2 A possible patch for the bsdiff file format. 38

D.1 Test function. 45
D.2 Graph of the test program. 46
D.3 Basic block reflecting change. 47

LIST OF ABBREVIATIONS

BTS	Branch Trace Store
DS	Debug Store
CFG	Control Flow Graph
ICFG	Inter-Procedural Control Flow Graph
CG	Call Graph
MSR	Model Specific Register
PLT	Procedure Linkage Table
CPU	Central Processing Unit
MHz	Megahertz
GHz	Gigahertz
ID	Identifier
RAM	Random Access Memory
DLL	Dynamic Link Library
kB	Kilobyte(s)
MB	Megabyte(s)
IR	Intermediate Representation
NOP	No Operation
ELF	Executable and Linkable Format
PE	Portable Executable

CHAPTER 1

INTRODUCTION

While software patches are intended to add or improve functionality, or remove vulnerabilities, it is entirely possible for such an update to unintentionally break a feature or even add new vulnerabilities. Therefore, patching as soon as an update becomes available is not always the best decision. On the opposite end of the spectrum, leaving software out of date has its own risks. In addition to missing out on any intended improvements on functionality or performance, once a patch is released, any existing vulnerabilities that are fixed in the patch become public. Even if such a fix is not described such that the security hole is obvious, many reverse engineering tools exist such that the patch can be used to find it. This vulnerability can then be used to exploit any users who are using the old version of the software.

Exhaustively testing every single update for even one piece of software can be an extremely time-consuming task which is not always feasible. It is important to know which updates require more thorough testing and which can be deployed quickly, since patching too early and too late can both have unfortunate consequences. Additionally, different workloads use different parts of programs, so the correct tests must be run so that changes are not overlooked when source code and detailed update descriptions are not available.

We present a method to quantify the expected impact of a software update using the binaries themselves and execution traces of the older version on x64 systems. This method attempts to determine if the patch modifies code paths that are commonly used, which means that more testing should be performed before applying the update. Because this is highly dependent on program inputs, tests can be selected by finding the workloads which score the highest in our analysis; testing unchanged code would be a waste of time. Of course, a small update can affect the instruction sequences of nearly all of the binary code in the executable due to compiler decisions. We must

minimize false positives for the results to be meaningful. The new code must also be associated with the correct part of the old code, or the importance of the code will not be assigned to the correct change.

In Chapter 2, we discuss related work in binary comparison. Chapter 3 describes our implementation of both execution tracing and binary analysis. Chapter 4 discusses the results of our approach for comparing multiple versions of two sets of tools, and Chapter 5 concludes the thesis.

CHAPTER 2

RELATED WORK

2.1 PatchAdvisor

PatchAdvisor [1] was created with the same goal as this work: to calculate the expected impact of a patch on a software system. PatchAdvisor uses IDA Pro [2] to extract the control flow graphs (CFGs) from each version of the binary. The prototype implementation is heavily dependent on the PaiMei [3] framework, which is used for both diffing the CFGs (using a modified version of PaiMeiDiff) and for attaching to the program to acquire traces. The intersection of the traces and the differences between the CFGs is used to infer the impact of the patch.

Several functions are proposed for determining the impact, including weighting areas of the CFG based on the execution traces and weighting based on proximity in the CFG to heavily executed code.

2.2 BMAT

BMAT [4] is a tool for matching basic blocks from two blocks of a program on x86, such that profile information from the old version can be propagated to the new one. Code blocks are matched by content and data blocks are matched by how the program accesses the data. To propagate the new information properly, BMAT performs fuzzy matching, so that blocks can be matched without being identical.

Basic blocks are matched in multiple stages. First, procedures are matched together by name if possible. If no matching name is found, BMAT searches for a similar name and may do a simplified block-matching pass over the candidate procedures. Second, basic blocks are matched only to blocks in the

other versions of the same procedures. Basic block matching is performed using a hashing-based algorithm. Each block is assigned a 64-bit hash value based on opcodes and operands. Blocks with the same hash value may be matched, though if multiple blocks are equivalent then the match is assigned based on heuristics and locations of the blocks. This hashing function is affected by instruction order.

To generate the best matching, BMAT uses multiple levels of fuzziness to match blocks. The lowest level, only used to match certain procedures, uses all information from the basic block's instructions, including register allocation and offsets. The next level excludes numerical offsets, and certain registers are considered to be equivalent. Several more levels exist, up to one where only the opcodes of instructions are considered. If a block has multiple match candidates, BMAT searches for one that passes the neighbor test: the block and the candidate must have adjacent blocks that also match.

BMAT was tested on several DLLs from Microsoft Windows 2000 and a DLL from Microsoft Internet Explorer 5.0, running on a computer with a Pentium II 200MHz Processor with 512 MB of RAM. Running time ranges from four seconds to four minutes, depending on the size of the executable and the number of weeks between the versions. BMAT was able to match and propagate over 98% of code blocks on average.

2.3 DarunGrim2

DarunGrim2 [5] is a binary diffing tool using various diffing algorithms. Like in many tools, symbol names are used as a starting point for block matching. Basic blocks are then assigned a fingerprint based on code features, which is used as a hash table key. Fingerprints are generated using opcodes and operands. The problem with this approach is that changes in order will change the hash value even if the block remains functionally the same. An instruction order normalization feature is added such that reordering only causes changes if the reordered instructions are dependent on each other. To avoid hash collisions, information about neighboring blocks can be used.

Functions are matched by checking how many basic blocks match, and the function with the highest number of matches is chosen. Matching blocks within these functions is performed by again hashing basic block fingerprints.

In this case, the search space is much smaller.

After using the fingerprinting method to match functions and blocks, structural analysis attempts to match the unmatched blocks using locality with blocks that were already matched. These blocks must be compared for similarity if they are not identical, which involves converting the fingerprint to an ASCII string and using string matching algorithms.

DarunGrim2 is tested on various Microsoft binaries, which are generally not obfuscated and contain small security changes. Symbols are present, which can simplify the matching process, though binaries without symbols can also be compared.

2.4 BinSlayer

BinSlayer [6] seeks to find variations of malware, as new malware binaries are frequently the result of a small change on existing malware. This creates difficulty for the anti-virus industry, as new malicious binaries must be classified as malware as quickly as possible to be effective. To combat this problem, BinSlayer combines two comparison algorithms to improve binary matching accuracy. Structural analysis [7] attempts to match all functions and basic blocks, and another graph matching algorithm [8] is used to improve the set of matches.

2.4.1 Structural Matching

Structural matching attempts to construct the CFG of each binary. This overall CFG contains the call graph (CG), in which graph the nodes are functions. Each function has a corresponding CFG of basic blocks in the function. Each basic block and function is given a tuple. In the case of a function, this tuple contains the number of basic blocks, the number of edges within the function's CFG, and the number of function calls in the CFG. For a node in a graph, a set of nodes from another graph is selected and whichever node matches will be returned, if such a match exists. BinDiff first finds an initial set of unique matches by comparing all nodes in each graph. Matches are then propagated by matching neighbors of matched nodes until no more matches can be found.

2.4.2 Hungarian Algorithm

The Hungarian algorithm generates assignments between two sets to create a solution with the lowest overall cost. With block matching, the aim is still to uniquely assign members of two sets with minimal cost. The Hungarian algorithm requires both sets to be the same size, creating a square matrix. This condition is unlikely between two versions of a binary. If one version contains n nodes and the other contains m nodes, the matrix will be (n + m) x (m + n), and each quadrant of the matrix corresponds to different kinds of graph edits.

2.4.3 BinSlayer

The structural matching algorithm is able to match nodes of the CG and CFGs accurately, but often leaves many functions unmatched. The Hungarian algorithm is applied to the unmatched nodes. This can introduce errors, so a validator is used to check function assignments. Depending on the similarity of nodes, certain edit costs are increased so they are less likely to occur, and the Hungarian algorithm is reapplied. BinSlayer was tested on coreutils programs. To determine how well functions were matched, they made an upper bound of symbols to match based on symbols included in the binaries. They do this only on the function level, not the basic block level, as functions can generally be validated automatically.

2.5 CoP

In addition to malware, plagiarism is a common reason for code obfuscation. CoP [9] is a method to find similarity between two binaries based on semantic equivalence, rather than similarity of the binary code itself. CoP generates the set of inputs and outputs using symbolic execution. A theorem prover is used to determine if two basic blocks are similar. A block is considered to be equivalent if a certain percentage of the output variables are semantically equivalent. This differs from other symbolic execution diffing methods in that fuzzy matching is used; obfuscation techniques can easily make blocks semantically different from the originals. There is some room for noise, since

part of the obfuscation could be to add extra code. Of course, similar basic blocks are common, so for plagiarism to be considered likely, what matters is a similar code path. The longest common subsequence of semantically equivalent blocks is found to calculate the actual similarity between functions.

CoP was tested on several programs based on different versions of the code, as well as the same versions of the source but compiled using different compilers or optimization levels. Unrelated programs were also compared to confirm that false positives are low.

2.6 BinHunt

BinHunt [10] addresses the limitations of traditional block matching tools by using symbolic execution to identify semantic differences between versions on x86. Each binary is disassembled and converted to an intermediate representation (IR). This is used to create CFGs and CGs, and compares those graphs to match functions and basic blocks.

In their implementation, binaries are disassembled using a plugin for IDA Pro. The IR is used to generate a CG for each binary. A CG is a directed graph with functions within the binary as nodes, connected by calls between the functions. A CFG is created for every function, with basic blocks as nodes. CGs and CFGs are compared using a graph isomorphism algorithm to find the maximum common subgraph, such that nodes in the common subgraph are matched.

Basic blocks must still be compared, as it is entirely possible for graphs to be isomorphic but for the nodes to represent functionally different code. For each basic block, they find inputs and outputs, and use symbolic execution to represent the output values. A theorem prover is then used to test if blocks are equivalent based on those outputs. With this method, binary changes such as instruction reordering will not be identified as changes unless they should change the functionality.

In case studies, BinHunt was able to match functions and basic blocks with high confidence. With `gzip`, which contained over 8,500 instructions, all 75 non-empty functions contained some syntactic differences, despite the change in source code being very small. Analysis took about an hour on a 2.1 GHz CPU. With `tar`, each binary contained more than 41,000 instructions,

but many functions were identical so analysis took about 30 minutes.

2.6.1 iBinHunt

While the subgraph isomorphism approach of BinHunt is resilient against typical compiler decisions like register allocation and instruction reordering, certain obfuscation techniques can make function boundaries unclear. This makes it difficult to match blocks within matching functions. iBinHunt [11] seeks to mitigate this difficulty using deep taint and inter-procedural control flow graphs (ICFGs).

Deep taint involves monitoring the execution of both binaries on the same input and recording which basic blocks are used in processing different parts of the input. Basic blocks receive different taint tags depending on which part of the input they process. Basic blocks are compared for matching if they have the same taint tag, which allows inter-procedure comparisons while still limiting the search space. Candidate blocks are compared in a similar manner to BinHunt, but the blocks must either be semantically equivalent or have matching predecessor or successor blocks.

iBinHunt was tested on program versions with large source code changes. The `gzip` versions used have over 25% of code lines changed, and about 90% of basic blocks were matched. Over 67% of matched basic blocks contained the same taint tags.

2.7 MalwareHunt

Symbolic execution and theorem proving is an effective way to find equivalent basic blocks, but tends to be a performance bottleneck. MalwareHunt [12] is built on top of iBinHunt, but implements features to reduce the number of times that theorem proving must be performed.

Basic blocks are normalized to undo certain obfuscation techniques. Blocks can be broken up such that the code is still executed in the same order, but with jumps in between. To address this, basic blocks with only one predecessor and one successor are merged into a single block. Normalization is performed on IR, with `NOP` instructions removed and address values replaced with zero values. MD5 values are then calculated for each block. Before in-

voking the theorem prover on a block, MalwareHunt checks if an equivalent block has the same hash value or if the blocks belong to the same subset of basic blocks. If blocks from different subsets are equivalent, then those subsets are merged.

The optimizations of MalwareHunt have been shown in experiments to provide a speedup of 2.8X to 5.3X (average 4.1X) and reduce invocations of the solver by a factor of 3X to 6X (average 4.5X). MalwareHunt also often has better accuracy than iBinHunt alone due to its normalization techniques.

2.8 Matching Execution Histories

Another existing approach matches the binaries of two versions of a program using their execution histories [13]. This can be helpful for many purposes, including piracy detection and comparing optimized versions against unoptimized to determine whether the source of erroneous behavior is the original version or the optimizer. A detailed trace format called Whole Execution Trace (WET) is used.

The WET format contains information such as control flow, values produced, relative addresses used, system calls, and more. Instructions are assigned signatures based on local execution history. Dynamic data dependence graphs are created, with nodes for each instruction that has been executed and edges for data dependencies that have been used at least once. Root nodes are matched, followed by nodes that are dependent on the roots.

Testing is performed on two versions of programs, optimized and unoptimized. The optimized versions contain fewer instructions than the unoptimized ones, so 95.2% of optimized instructions and 82.4% of unoptimized instructions are matched. While some false matches exist, very few matches are missed.

CHAPTER 3

IMPLEMENTATION

3.1 Overview

Like PatchAdvisor [1], we use execution traces to improve static analysis of two program versions. We extract basic block lists from each executable using IDA [2], attempt to match the basic blocks of the two versions, and use the branch traces to assign importance to the basic blocks. See Figure 3.1 for an overview of inputs and outputs.

3.2 Execution Tracing

Intel offers multiple ways to capture control flow, some of which can be used at the granularity of all branches. We acquire execution traces using the Intel Branch Trace Store [14] (BTS) feature. Traces are only needed from the old version of the program, so analysis can begin as soon as a new update is available if the current version has already been traced. BTS is enabled through the `IA32_DEBUGCTL` Model Specific Register (MSR) and configured in the DS save area, which is selected by setting the `IA32_DS_AREA` MSR. When BTS is enabled, all branches are stored to an allocated buffer as defined in the DS save area. The BTS buffer can be managed by either generating an interrupt when the buffer is full, or by using it as a circular buffer with the BTS index field of the DS save area. Each branch entry contains three fields: the location of the branch, the target of the branch, and whether the branch was predicted.

We use BTS with a circular buffer, which is accessed by a kernel module. A user-level program can read the branches through a proc file. When this file is read, the kernel module saves the current BTS index and copies all branches

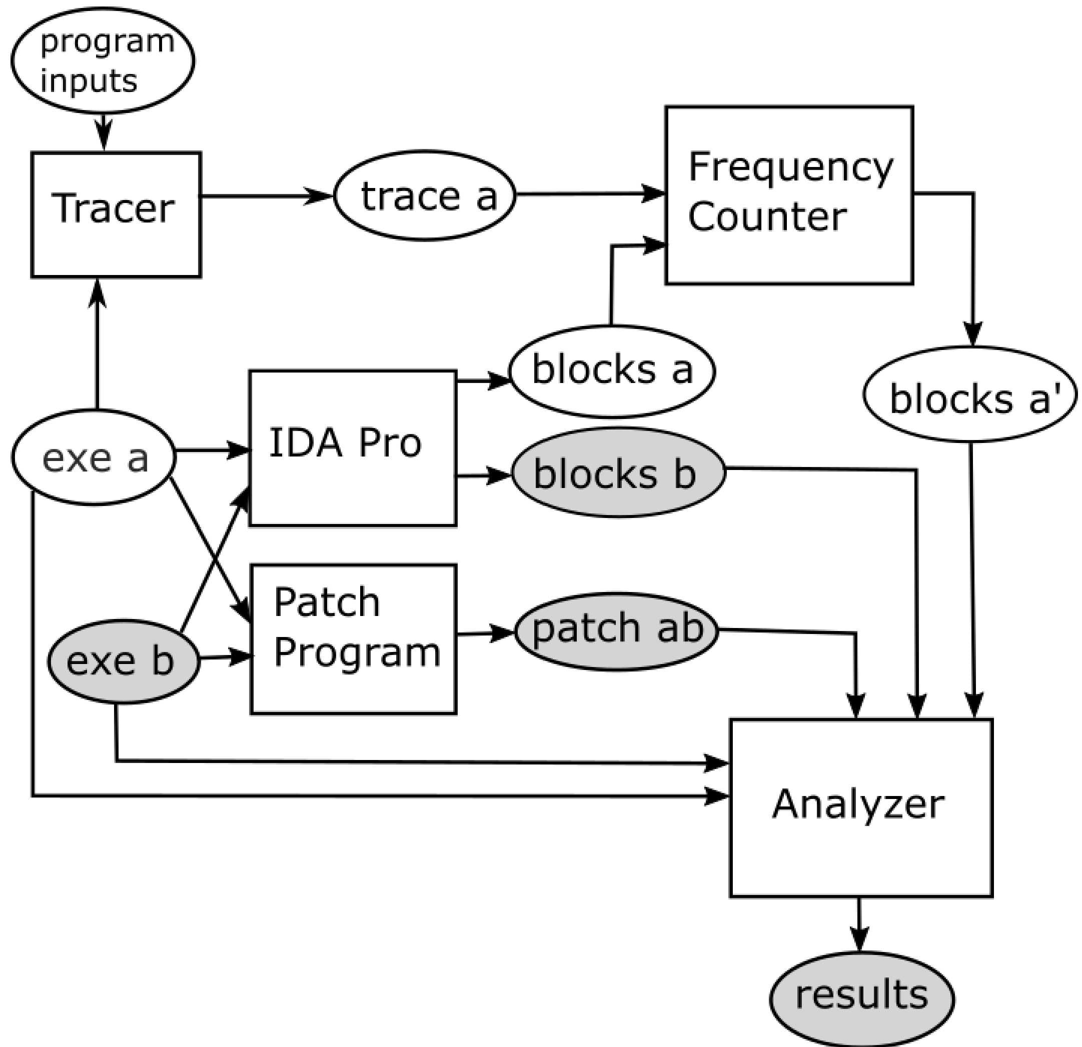


Figure 3.1: Architecture of the analysis system. Shaded areas indicate files involving the new version of the binary.

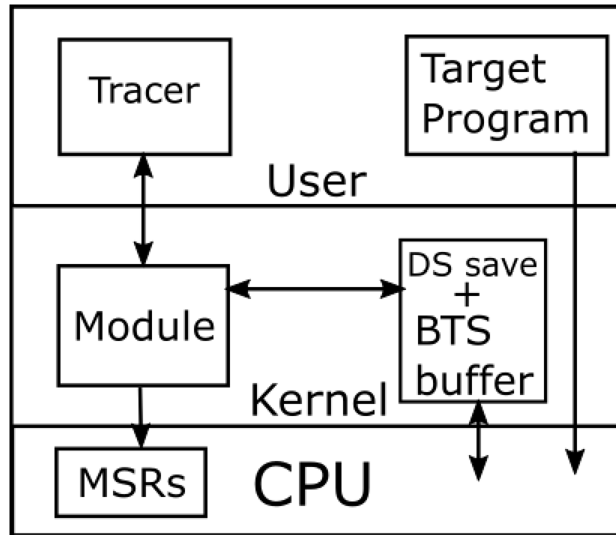


Figure 3.2: Overview of trace generation.

from the last index until the current index. The last index then becomes the current index so that we are ready for the next read. For simplicity, we perform tracing on only one CPU. This core is isolated by changing process affinities. Once BTS is enabled, we run the desired program on the isolated core and repeatedly read branches until the program has completed. This tracing system is summarized by Figure 3.2.

It would also be possible to modify the scheduler such that tracing may be performed on any core. In this case, the scheduler would enable BTS when the target process is scheduled, on the core where it is scheduled, and disable BTS when its time slice is over. This could also be used in a hypervisor by enabling BTS when the page directory base register corresponds to the page directory of the target guest process.

3.3 Basic Blocks

Basic blocks are defined as sequences of instructions for which there is only one entrance and one exit. This means that a block may be defined by the presence of branch instructions as well as instructions being the targets of other branch instructions. Typically, `call` instructions do not divide basic blocks and therefore may exist in the middle of a basic block. As a `call` instruction implies that control flow will return after a `ret` instruction, this

is a generally useful definition. We include `call` instructions with the other branching instructions because a `call` is not guaranteed to return to the call site.

We use IDA Pro to collect lists of basic blocks for both the old and new versions of executables. We also flag blocks with details such as whether a block is at the start of a function, or if the block is located in the Procedure Linkage Table (PLT). The PLT is used to call external functions that are not known when the executable is created, so while basic blocks exist there, they do not contain useful information for diffing and can cause false positives.

3.4 Code Paths

Once we have the basic block list for the old version of the executable, we can use our trace from Section 3.2 to count the usages of each block. BTS only logs branches that are taken, so to reconstruct the path, we must also count the implied block executions from the blocks located between a branch target and the next branch location.

3.5 Patching and Reconstruction

3.5.1 Patch Generation

Next, a binary patch is created from the old and new versions of the executable. Currently two patch formats are supported: those from ExamDiff Pro and those from bsdiff [15]. ExamDiff Pro, a commercial diffing tool, creates intuitive human-readable patches, but takes substantially longer to run. These patches are also significantly larger than those from bsdiff. For example, a binary patch for the coreutils program `du`, between version 8.22 and 8.23, is 35kB with bsdiff and 1,158kB with ExamDiff Pro. For more details, see Appendix B.

3.5.2 Rebuilding New Version

To gain a starting point for basic block matching, we use the old executable and the binary patch to reconstruct the new executable. New and old file positions are translated into virtual addresses. If the virtual addresses correspond to basic blocks, then those blocks are tentatively matched. Once all code sections of the new executable have been rebuilt, the patching stops. We do not attempt to match data blocks. Extra care must be taken when we are near a section boundary (see Appendix C).

With a patch from ExamDiff Pro, file offsets are explicitly listed in order, making initial matches simple. bsdiff builds the new executable in file offset order, but may skip forwards and backwards in the old binary (and does so frequently), so changes such as deletions are not explicit. Occasionally, the bsdiff patch may use part of a data block at the end of the old executable to create an early code block in the new version.

3.6 Basic Block Matching

3.6.1 Initial Pass

Once the new executable has been reconstructed, we sweep over all basic blocks in the old executable and attempt to guess a matching address range when possible. If the block is missing the start of the range, we check if the previous block in the old executable is adjacent to this block and has a match. If it does, we assign the start of the current match with the end of the last match. If the block is at the start of a function, we instead match to the first function start block after the previous block's match. Similarly, if the end of the range is missing, we use the start of the match of the next block if it is adjacent. If the block has a matching range, it may be adjusted based on neighboring blocks' matches and the basic block boundaries in the new executable.

For every block with a match range, we also store the block's address in its match block. When a match is adjusted, we also try to assign a similarly sized range. If the matching block is much smaller than the current block, the block after the match may also be added, unless this addition would

make the range unreasonably large. This is important not only for better comparisons, but also because this reduces the number of match candidates in the new executable, which becomes important in Section 3.6.3.

3.6.2 Basic Block Diffing

Basic Block Representation

To compare basic blocks, we use the disassembler library Udis86 to create an array of instruction units. An instruction unit comprises a mnemonic identifier (ID) and operand type IDs. All `NOP` instructions are excluded from the array. Operand types include various types of addresses, constants, and registers.

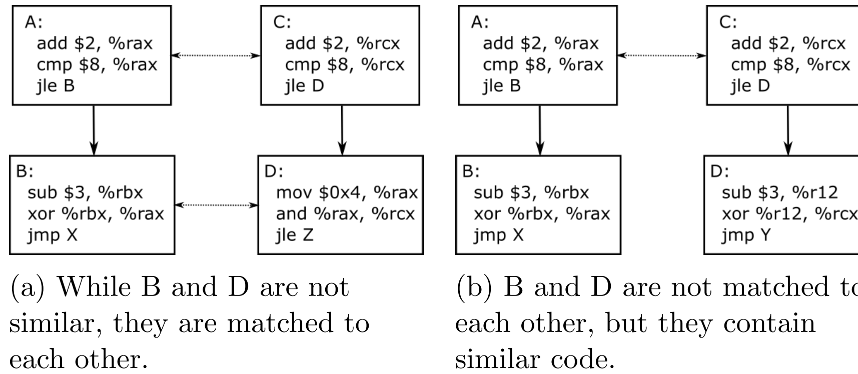
Caller save registers and callee save registers are treated as separate types. While register allocation changes very often, the decisions tend to stay the same with respect to calling convention. An example of this can be found in Appendix D. The mnemonic ID and operands are packed into a 32-bit value, with 16 bits for the mnemonic and 4 bits for each operand. If an instruction uses fewer operands, the first operand fields are used and the remaining ones are set to zero.

Block Diff Algorithm

To diff the instruction arrays, we use Myers' diffing algorithm [16], since our instruction units may be checked for equality as integers and basic blocks are typically short sequences. It is important to note that any similar algorithm could be substituted here, as long as it works on sequences of simple values, so most string diffing algorithms could be used.

In the Myers algorithm, sequence edit distances are described only as insertions and deletions. The shortest edit distance is the smallest number of edits to transform the first sequence into the second. This algorithm is very unforgiving for instruction reordering.

It would be interesting to compare results against an algorithm that is more generous with reordering, but this would not necessarily give better results, since some instruction reorderings affect functionality and some do



(a) While B and D are not similar, they are matched to each other.

(b) B and D are not matched to each other, but they contain similar code.

Figure 3.3: Two cases demonstrating each possibility for branches in A and C to be equivalent. Blocks A and B are in the old executable and C and D are in the new one. Dashed lines indicate an assigned matching.

not. This exposes a shortcoming of this block diffing approach: without some form of dependency analysis, it is possible that we are flagging blocks as modified when they are functionally equivalent.

Special Cases

Occasionally, conditional branches will appear to change, but remain logically equivalent. A change in operand types usually indicates a substantive change in a basic block, but if the order is merely reversed in a comparison, then depending on the branch condition, there may be no change. Table 3.1 lists the pairs of conditional branches that are equivalent when the operands are switched.

Due to our method of block comparison, branches whose locations are changed are considered to be equivalent instructions. This is generally desirable, as offsets are very unlikely to remain unchanged between versions. It can be informative, however, to compare the branch targets of two blocks.

Table 3.1: Conditional branch pairs.

je	je
jne	jne
jg	jle
jge	jl
ja	jbe
jae	jb

Additionally, this information can be used to break ties when a block has multiple potential matches.

Suppose we have four blocks: A, B, C, and D. A and B are in the old version of the executable, with A branching to B. C and D are in the new version, with C branching to D. A matches with C, and is equivalent except for the operands of their branch instructions. If block B is already matched to block D, then A is equivalent to C, as in Figure 3.3a. Otherwise, if block B and block D have a low difference score, A and C are still equivalent, as in Figure 3.3b. If blocks B and D are a poor match, block A is flagged as having a different branch target. Note that for this type of comparison, branch targets of blocks B and D are not compared.

3.6.3 Block Rematching

After diffing all blocks that have match guesses, we try to assign matches to any block that still has no good match. For every block in the old executable without a good match, we add it to the list of bad matches. If that block has an assigned match, all blocks corresponding to the match range are added to the candidate list. All blocks in the new executable without matches are also added to the candidate list. Any block made only of `NOP` instructions is not added.

Candidates are then compared to the bad match blocks. The best match result for each candidate is stored within the candidate, so old blocks are not reassigned unless they are the best match for a candidate. If multiple best matches exist, we choose the one that is closest. One limitation is that this is performed on the single block granularity, so block splitting is not accounted for. This step can be repeated as desired, but after a certain number of iterations is unlikely to produce accurate matches.

3.7 Binary Analysis

Finally, the patch is scored based on the block matches. A total score for the executable is calculated as the sum of all block scores. Blocks are scored as the product of the block size in bytes and the number of times the block has been executed. The modified score is the contribution of block scores for

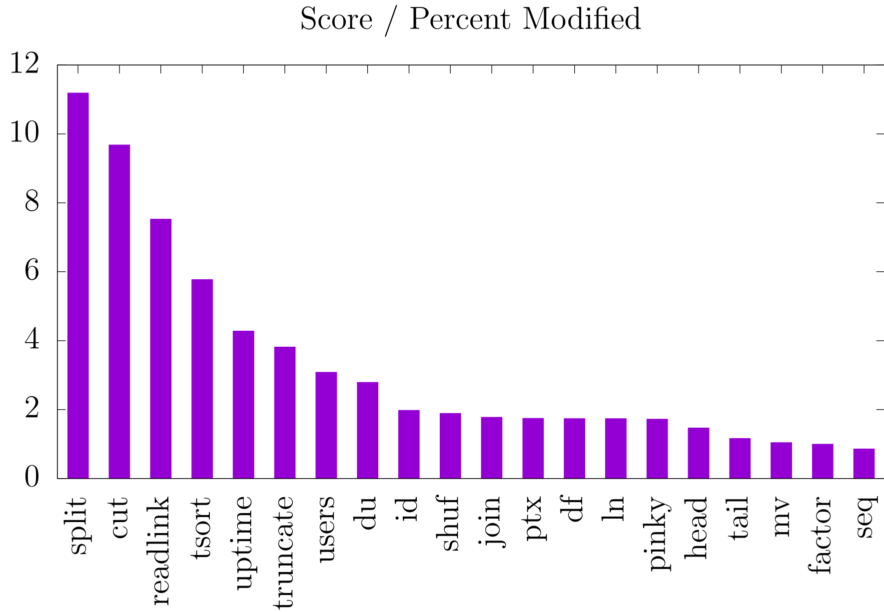


Figure 3.4: Top 20 score:block change ratios in coreutils, 8.22-8.23.

only blocks that have been modified. If only the branch target changes, the score is increased only negligibly. The final score is the modified percentage of the total score.

A high final score means that the patch is likely to have a large impact on the behavior of the program. If the patch is scored multiple times using different execution traces, the highest score identifies the profile that should be the most affected, and therefore is the best for testing the update. A low score means that the patch can be applied with little risk.

Scores can be high in one of two ways: the modified blocks are important, or a large number of less important blocks are modified. Either case implies that users will likely be impacted by the update, but another interesting metric is the final score divided by the percentage of modified blocks, which can help show which case applies to the programs. If the score is significantly higher than the block percentage, it may be worth checking if different inputs produce similar scores, which can provide a hint to which parts of the program have changed the most. Figure 3.4 shows the top results where the score is much higher than the fraction of modified blocks for one pair of coreutils versions (see Section 4.3.1 for more details).

This approach is limited in that it would be very ineffective against mali-

cious updates. For instance, an update could intentionally affect a lesser-used code path to exploit a small number of users. Additionally, only the typical `NOP` patterns are recognized, so obfuscation techniques could easily insert useless code and it would be treated as a normal sequence of instructions, reducing the match quality.

CHAPTER 4

EVALUATION

4.1 Test Cases

4.1.1 Branch Trace Checking

To confirm that BTS captures all taken branches, we perform tracing on a simple program that does nothing but branch between evenly spaced blocks through a jump table (see Figure 4.1). Using a known seed, each block calls `rand`. The pseudorandom result is masked to create a block index, and we jump to the corresponding block until the desired number of branches has occurred. Once the trace file is created, we can use the same seed to generate the expected sequence of branches.

4.1.2 Patch Analysis

Analysis was performed mainly on the programs in coreutils, a package of over 100 basic tools. Coreutils was chosen because they are commonly used but are also relatively small binaries, so it was easy to analyze a large number of different programs in a short period of time. Many coreutils programs take no arguments, and many others can be used on any text file. Unless a certain format was more interesting, we traced coreutils with a large text file of the book *Moby Dick*, acquired from Project Gutenberg [17], as the main input.

Binutils contains programs used mainly for dealing with executables and object files in various ways. This package has fewer programs but those programs contain about 30X more basic blocks than coreutils on average. Multiple versions of the programs are tested using the same sets of inputs. Final scores will vary depending on inputs. Binutils programs were mainly traced by running them on binutils binaries.

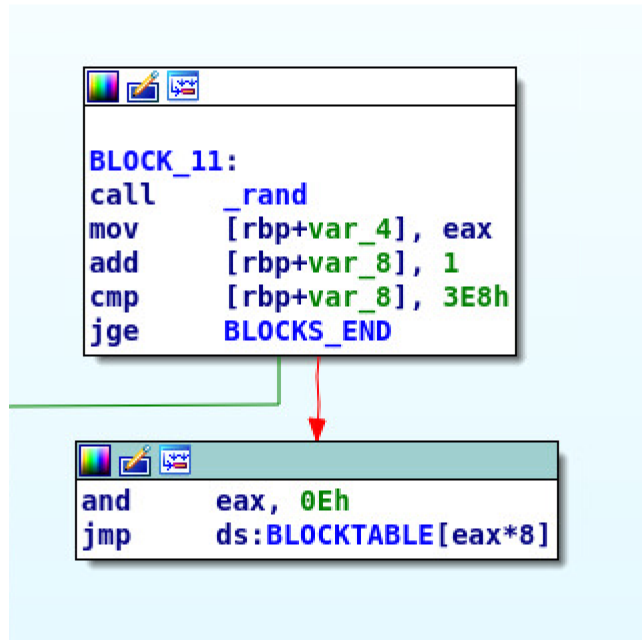


Figure 4.1: A pseudorandom branch code block.

All binary patches were generated using bsdiff.

4.2 Performance

Tests were performed using a computer with an Intel i7-4910MQ 2.90GHz CPU and 16GB of RAM.

4.2.1 Branch Trace Store

Because BTS writes all taken branches to a buffer in memory, tracing incurs a large performance penalty. We calculated the average slowdown of various programs by timing the execution on the isolated core, with and without timing, 100 times. Average slowdown is the average runtime with BTS divided by average runtime without. Figure 4.2 shows the average slowdown for various coreutils programs. Tracing was performed using binaries from version 8.22.

Programs such as `cat` and `cp`, which consist mostly of sequential memory accesses, see a relatively small slowdown, only slightly over 2. Due to both hardware and software optimizations, sequential accesses tend to be

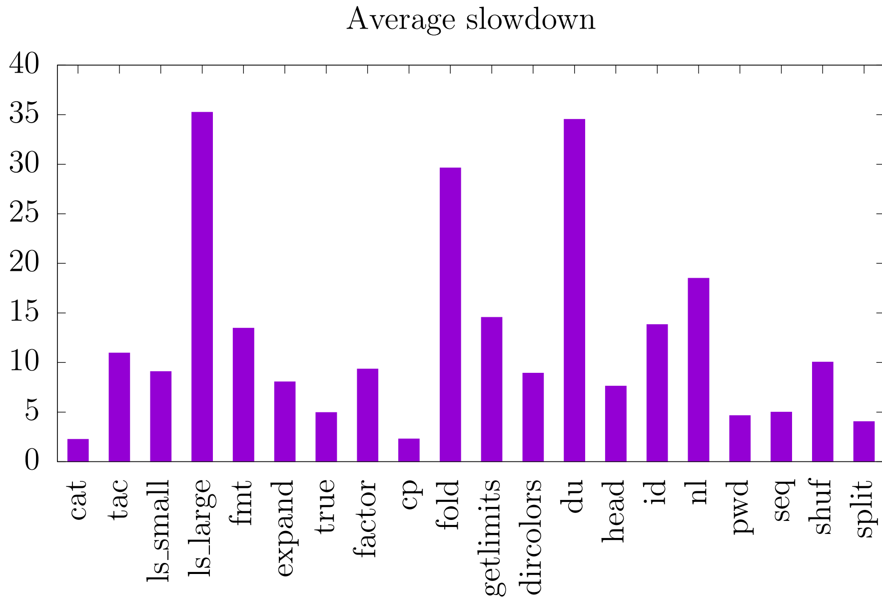


Figure 4.2: Average slowdown for selected coreutils programs.

very fast. If a copy function, for instance, is compiled with loop unrolling, then fewer branches will occur, which typically improves performance under normal circumstances. With BTS active, this improvement is even more significant because fewer branches are being written to memory.

Programs such as `tac` and `nl` also mostly read a file in a predictable order, but some extra work is performed compared to `cat` or `cp`. This would greatly increase the number of branches in the program. In fact, the individual trace files for `cat` and `tac`, run on the same input file, are 509kB and 38MB respectively.

Programs that work on directories rather than a single file, like `ls` and `du`, are also very slow, especially when run on large directories.

Programs such as `sort` are excluded from the graph because they are much slower. `sort` runs 90X slower with BTS than without. Tracing `sort` generates a file over 4GB in size.

4.2.2 Block Matching

Table 4.1 lists various runtimes for our analysis. This does not include steps like getting the basic block lists from IDA.

Table 4.1: Block analysis runtimes.

	Fastest	**Slowest**	**Typical**
coreutils 8.22-8.23	0.000s	6.13s	0.02s
coreutils 8.23-8.24	0.01s	23.7s	0.1s
binutils 2.26-2.27	0.01s	155m58s	31m6s
binutils 2.29-2.29.1	0.01s	3m37s	2m11s

Binutils shows that more match heuristics and optimizations will be necessary to be convenient for substantial updates to large programs. There is significant room for such improvements; coreutils ran so quickly that it was not a priority during initial development.

4.3 Results

In this section, we discuss the results for multiple updates to coreutils and binutils. To confirm that results make sense, we check the virtual addresses of modified blocks against IDA Pro to find the function. The blocks should correspond to some change in the source code of that function, or possibly an inlined function.

4.3.1 Coreutils 8.22-8.23

The top 20 final scores and modified block pecentages for coreutils 8.22-8.23 are shown in Figures 4.3 and 4.4 respectively. Most binaries see a very small change in this update, but a number of programs have scores that are significantly higher than the block change percentage (see Figure 3.4). One example is `cut`, which has many small changes to the `cut_fields` function. This function is used unless `cut` is invoked in byte mode, which we did not do while tracing.

`split` is a similarly extreme case. Only 2.24% of basic blocks are modified, but this program received the second highest score out of this version of coreutils. In this program, version 8.23 uses a different function to read from standard input, which also changes the error conditions to check for.

`numfmt` shows a high modification score, but a relatively low final score. A large part of the change is the removal of a large function called `vasnprintf`,

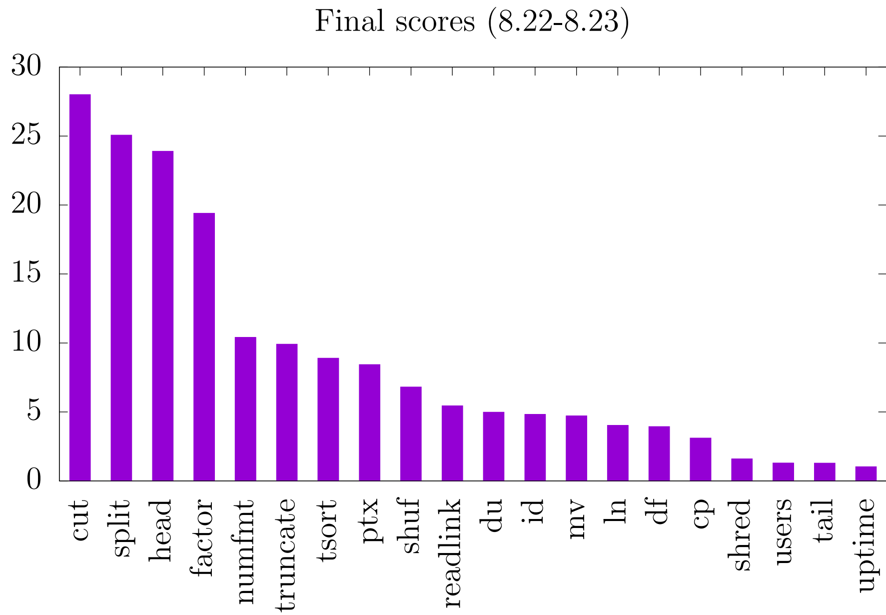


Figure 4.3: Top 20 scores for coreutils 8.22-8.23.

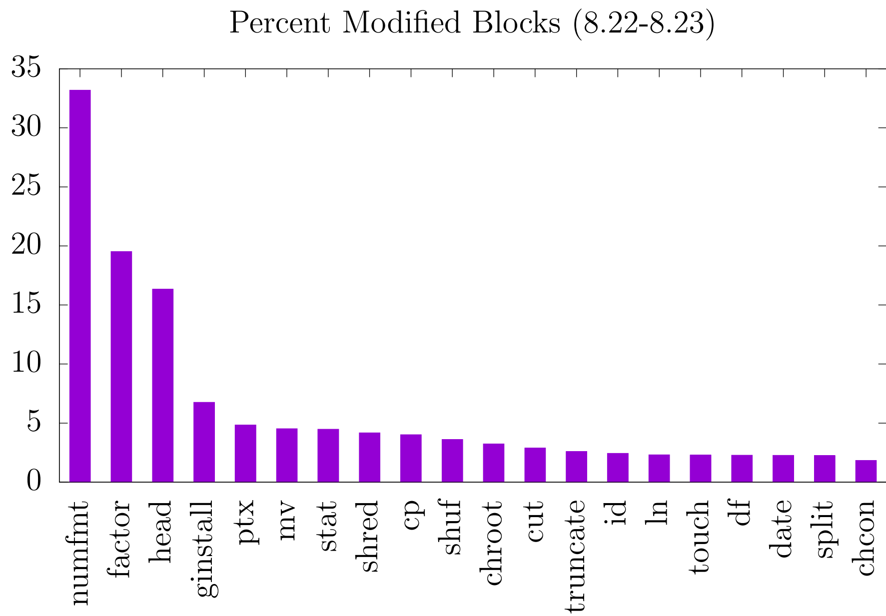


Figure 4.4: Blocks for coreutils 8.22-8.23.

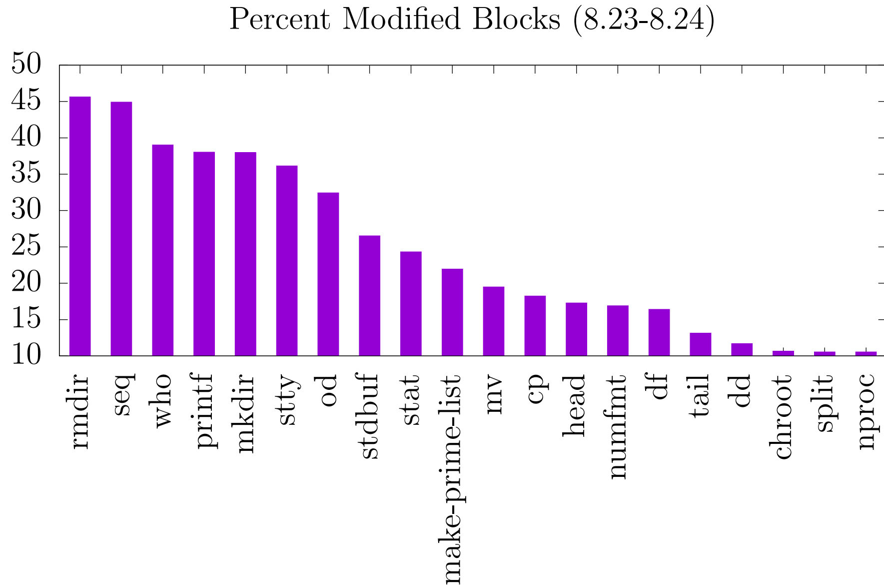


Figure 4.5: Blocks for coreutils 8.23-8.24.

which is present in version 8.22 but not used with our input so it does not contribute to the final score.

`factor` also has a large number of changed blocks, many of which are large unused functions, but a function called `print_factors_single`, which is actually used in 8.22, is not present in the 8.23 binary, so `factor` has a higher score.

4.3.2 Coreutils 8.23-8.24

In the coreutils 8.23-8.24 update, returns from `main` using the `exit` function are replaced with `return` statements. For programs such as `true` and `false`, where returning is all that they do, this is a high scoring change. Additionally, several functions that are present in 8.23 are no longer in the binary in 8.24, most notable being `vasnprintf`. Figure 4.5 shows that many programs contain more differences here than in the 8.22-8.23 update. Final scores are shown in Figure 4.6.

Programs using that function in 8.23, such as `who` and `od`, received high scores and relatively high block change rates. A function called by the coreutils `usage` functions is also changed, so if traces were performed with invalid

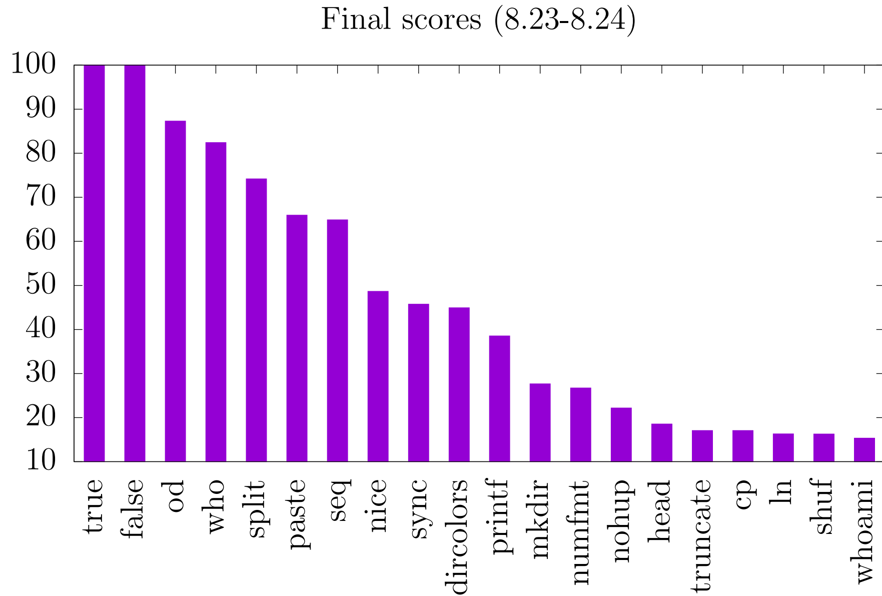


Figure 4.6: Scores for coreutils 8.23-8.24.

inputs, they would most likely receive high scores as well. To demonstrate this, we ran `nice` with "`--help`" and it received a score of almost 50% despite less than 6% of its basic blocks being modified.

Aside from those programs, `split` has the highest score. In this case, most execution takes place in main, which has several changes. Only 10.5% of the blocks are modified, but even fewer of these blocks contribute to the 74% final score.

4.3.3 Binutils 2.29-2.29.1

Versions 2.29 and 2.29.1 of binutils represent an update to the Binary File Descriptor library (BFD), which is mainly for parsing ELF binaries. Most of the source code updates appear to be minor changes for edge cases. Because this is a commonly used library for binutils, changes are expected to be approximately the same for most programs.

With the exceptions of `readelf` and `elfedit`, the tools show around 1,000 basic blocks with changes, which is about 1.5-2.5% of all basic blocks in the executables, as shown in Figures 4.7 and 4.8. The absolute number of modified blocks is very close for almost all of the programs. Scores (see Figure

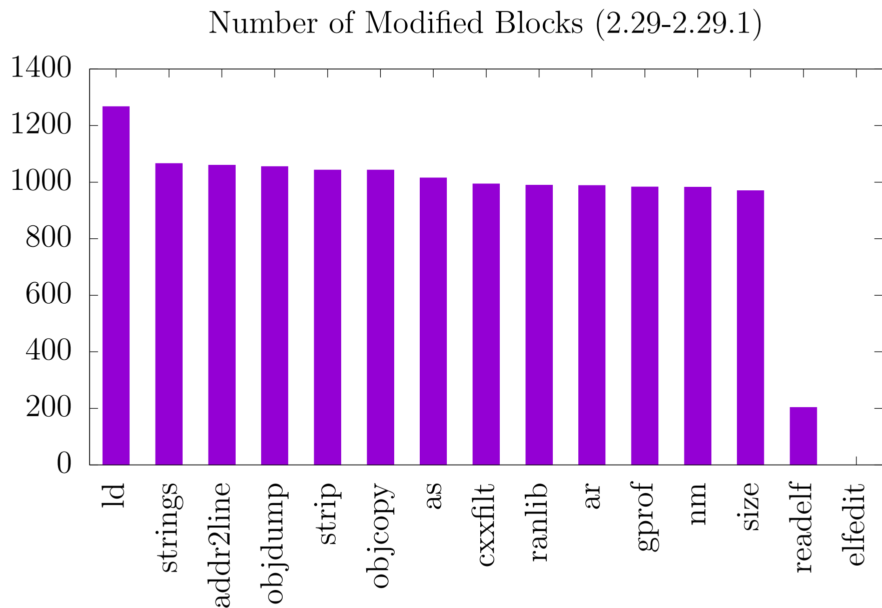


Figure 4.7: Blocks that changed in version 2.29.1 of binutils.

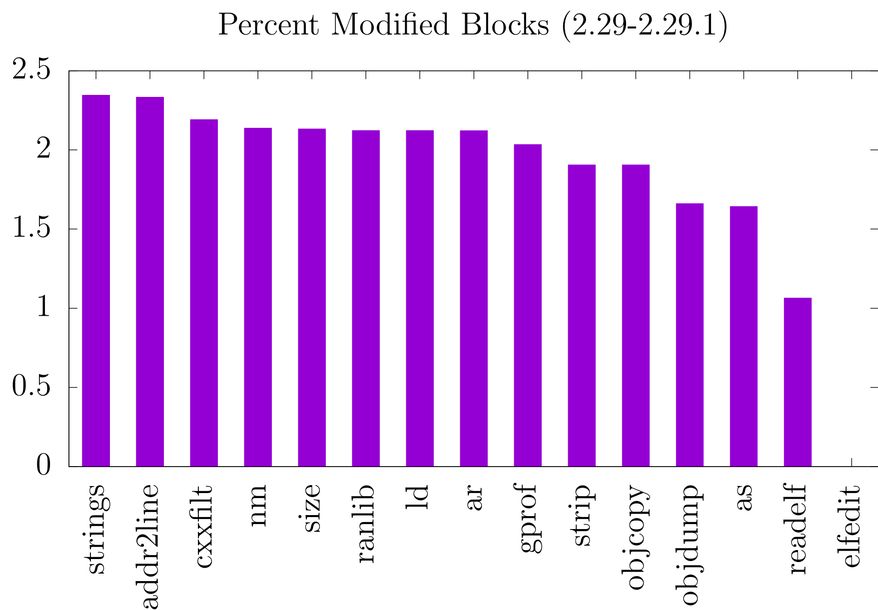


Figure 4.8: Fraction of blocks that changed version 2.29.1 of binutils.

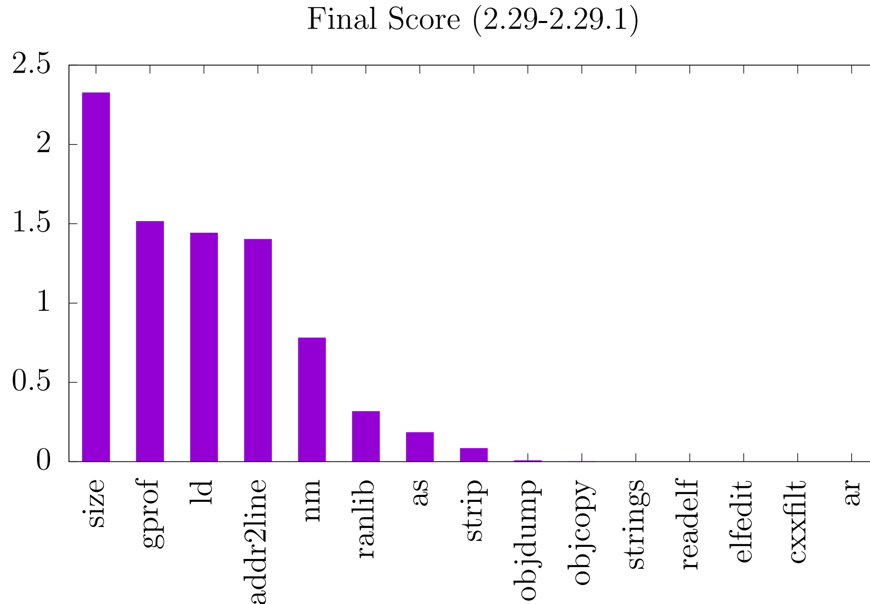


Figure 4.9: Score for changes in version 2.29.1 of binutils.

4.9) are much more variable, as only certain parts of BFD are changed, and they are very minor so it depends on how the program uses the library.

4.3.4 Binutils 2.26-2.27

Binutils 2.26-2.27 is a much larger update. Like in Section 4.3.3, there are many updates to the BFD library, which is predictable for a large update to a tool set like this, but with more impact. Figure 4.10 shows that most programs contain a somewhat consistent amount of basic blocks that are changed, which implies that shared code is changed, especially since the score distributions are again uneven in Figure 4.11. It appears that for most programs, whether they have high final scores or not, most executed changes are from files other than their main self-named files.

Programs like `objcopy` and `strings` show the typical amount of changed blocks, but extremely low final scores. Many changes exist throughout the binaries, but very few of them are executed. In `objcopy`, some executed changes can be traced to functions in the BFD file `elf.c`, but this is a small fraction of the overall execution.

The most executed changes to `objdump` draw attention to the `i386-dis.c`

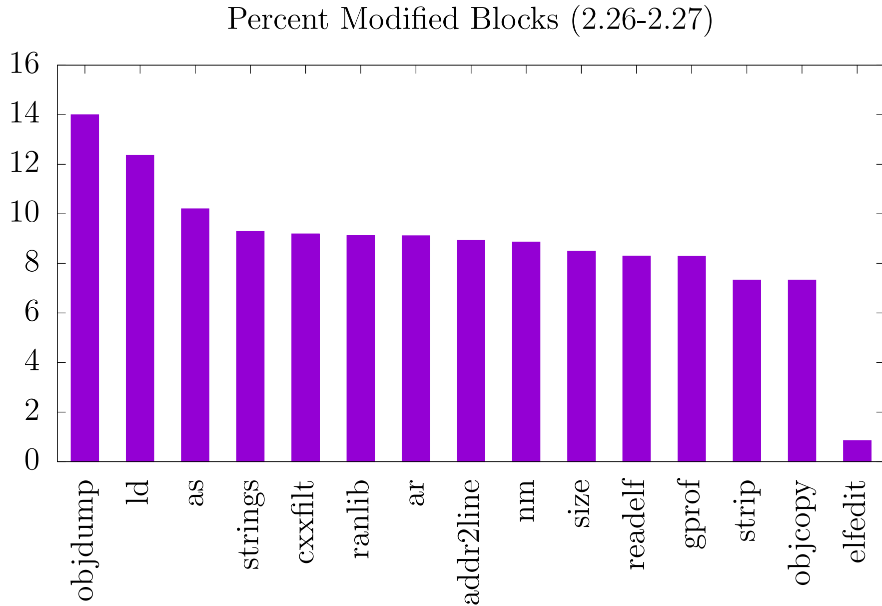


Figure 4.10: Percent modified blocks in versions 2.26-2.27 of binutils.

file, specifically to the function `print_insn`. Since this program disassembles files, this is expected. If we diff the source code for both versions of the source file, we find an added error check, which is located early in the function such that it should be executed every time the function is called.

`ld`, the highest scoring program, has a relatively large number of executed changes. Many of these changes are in the various `ld` files, but these blocks have fairly low execution counts each. The most executed changes appear to be in BFD files, including `linker.c` and `elflink.c`, which is unsurprising for a linker program. In `linker.c`, there are many changes to error checking in a function used to add symbols. `elflink.c` has many changes throughout the entire file, and its contributions appear to be among the most executed code in the binary. `elf64-x86-64.c` is a heavily modified file, but its contributions that are used by `ld` are mainly limited to a few functions dealing with symbols.

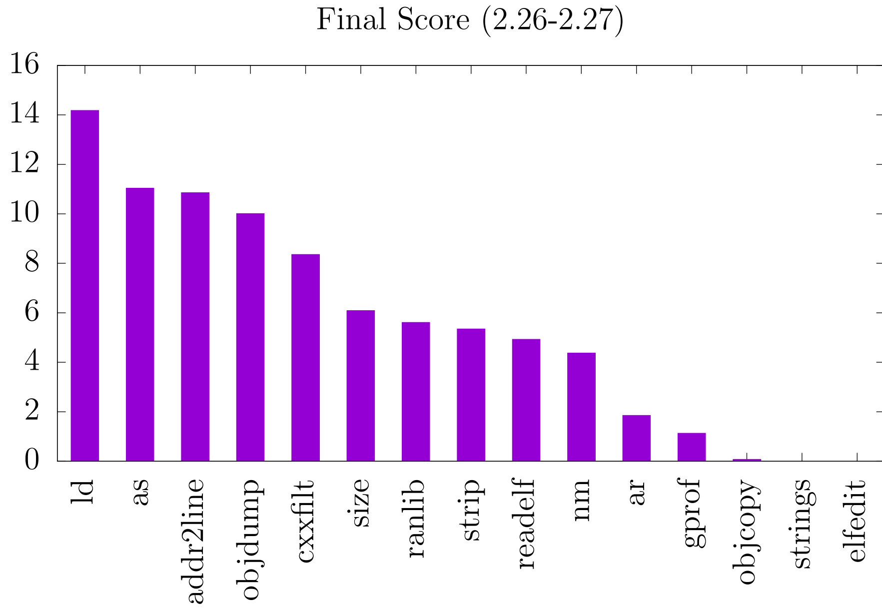


Figure 4.11: Score for versions 2.26-2.27 of binutils.

Table 4.2: Description of split options.

basic1000	No options. Default split into files of 1000 lines.
small200	Split into files of 200 bytes.
lines100	Split into files of 100 lines.
unbuffered	Default split, but copies to output without buffering.
chunks10	Split into 10 files.
chunks200	Split into 200 files.

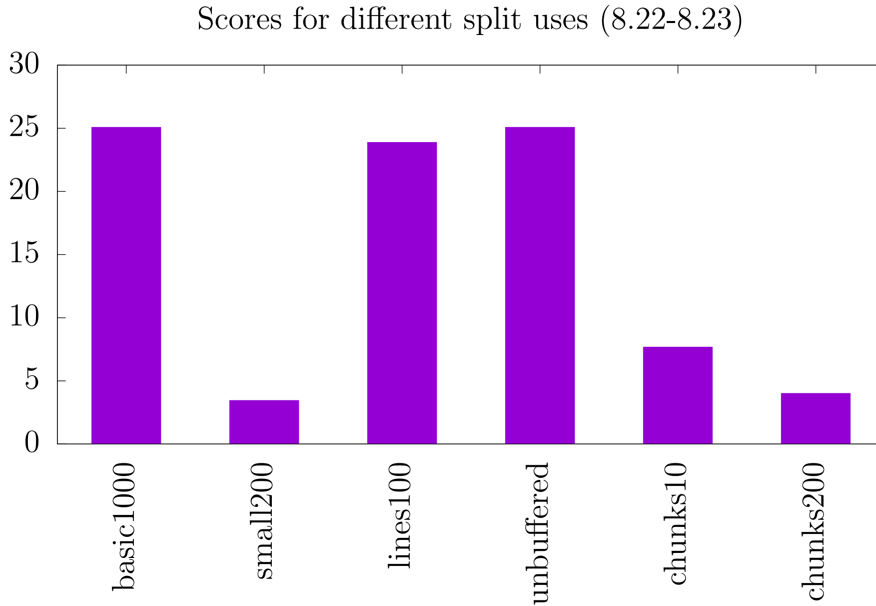


Figure 4.12: Score for split version 8.22-8.23 using different options.

4.4 Use Comparison: Split

Because `split` received such high scores in both coreutils updates, we investigate the nature of these updates and how they affect different users. The `split` program is used to generate many smaller files from one large file, by default into files of 1000 lines. In this section, we trace the versions of `split` using the same large text file as before, but with different options as described in Table 4.2. With the default option, the file is split into about 23 files. File size is variable due to line length but most are around 55KB. We used default options for split in Sections 4.3.1 and 4.3.2.

Both updates have similar score distributions for the different use cases. Figure 4.12 shows the scores for the 8.22-8.23 update, and Figure 4.13 shows the scores for the 8.23-8.24 update. It appears that both updates have significant impacts on users who split files by line boundary, and little to no impact if file size is used. Looking at the source code for split, splitting into specific file sizes and splitting into a specific number of files often call the same function, `bytes_split`, while splitting by line boundaries generally invokes `lines_split` or `lines_chunk_split`.

Diffing the source code of 8.23 and 8.24 shows many changes to the line

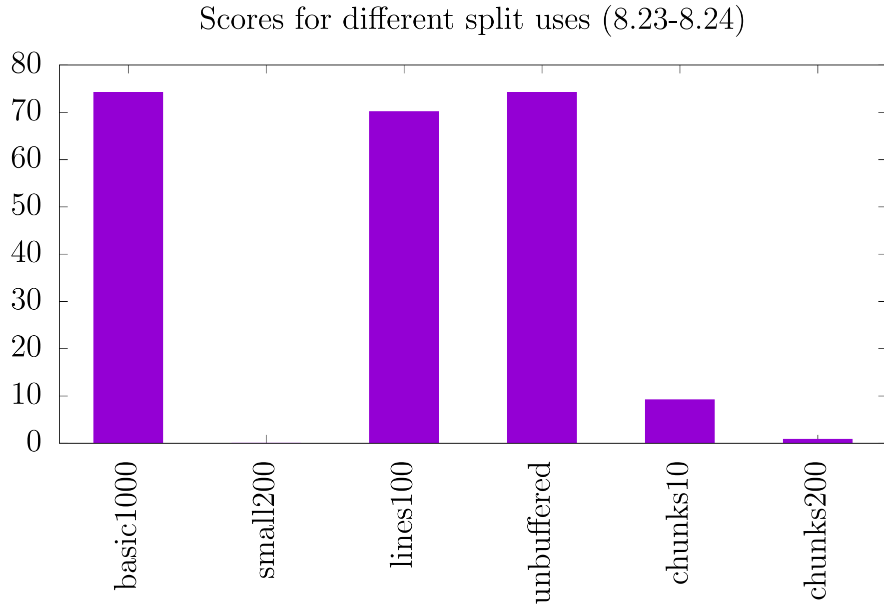


Figure 4.13: Score for split version 8.23-8.24 using different options.

splitting functions and almost none to `bytes_split`. The 8.22 to 8.23 update (see Section 4.3.1) appears less targeted, since it replaces most reading function calls. In fact, the default `split` version executes 11 modified blocks and small200 executes 15 modified blocks. Interestingly, the line splitting functions appear to be inlined while `byte_split` is not. It is possible that the inlined functions change more in commonly used blocks due to more available optimizations.

CHAPTER 5

CONCLUSION

In this work, we have presented a tool for determining if a software update introduces meaningful changes based on actual usage of the programs. While there is significant room for improvement, we are able to identify binary code changes and whether that code is frequently executed for a given workload.

5.1 Future Work

One of the largest drawbacks of this technique is the slowdown from BTS when acquiring traces. For command line programs, this performance reduction may be acceptable for occasional tracing, but for interactive programs, especially those with graphical user interfaces, the slowdown can be quite disruptive. For instance, web browsers are noticeably slower and many games become nearly unresponsive. One potential way to deal with this problem, without sacrificing completeness, would be to record user actions and replay them at a later time with BTS.

To improve the quality of block matching, some form of dependency analysis within basic blocks would help solve the instruction reordering issue. This reordering occurs more often in code near code with actual changes, so the results are usually not too skewed by these false positives, but if two nearby code paths have very different usage patterns, this can disproportionately increase the final score. Adding more heuristics for which blocks should be compared would improve performance on large programs and likely improve accuracy in most cases.

More options for code importance would also be useful, especially with analysis over multiple traces. For instance, blocks that are used only once or twice will be considered fairly unimportant under our current scoring function. If we look at a large number of traces and every one of them uses

these blocks (still a small number of times each), however, this would mean that all users are likely to be affected by changes to these blocks.

Occasionally, the compiler may change its decisions about which functions to inline. This will cause a large number of blocks to be flagged as modified. Usually, this happens when the inlined (or previously inlined) function or its caller changes significantly, so aside from potential restructuring of other basic blocks due to a large insertion or deletion, this is likely to call attention to actual changes. This could also result in large false positives, however, if about half of an inlined function is used, and changes to the unused half are why it is no longer inlined.

It would be interesting to see how results would change if we adapted this system to work with more different types of binary diffing tools, such as Courgette [18]. This would require a dramatically different approach, as that tool works by diffing modified assembly, but could potentially be very helpful with determining whether certain binary changes are meaningful.

APPENDIX A

BRANCH TRACE STORE

BTS is enabled mainly through the IA32_DEBUGCTL MSR, shown in Figure A.1 (adapted from the Intel Manual [14], Volume 3). Bit 7 enables the BTS feature and bit 6 enables branch trace messages so they can be stored. Bit 8 generates interrupts when the BTS buffer is full, but we keep this bit clear and use the buffer as a circular buffer. Bit 9 causes BTS to skip branches taken in ring 0 (privileged code). This MSR is also used to manage the Last Branch Record (LBR) feature and certain performance counter features.

The IA32_DS_AREA MSR holds a pointer to the DS save area, which we allocate using our kernel module. The DS save area, shown in Figure A.2, is where the BTS buffer is managed, as well as another feature called processor event-based sampling (PEBS). BTS Buffer Base points to the start of the BTS buffer, which we place in our allocated space past the DS save fields. BTS Index is a pointer to the next entry in the buffer, initially set to the start. BTS Absolute Maximum is the pointer to the byte past the end of the BTS buffer. BTS Interrupt Threshold is the pointer to the BTS buffer entry at which point an interrupt should be generated. Because we do not use the BTS interrupt feature, we set this past the BTS Absolute Maximum.

BTS Index is modified by the CPU when branches are logged. When we read branches from the BTS buffer, we save the current index, read branches starting at the last saved index, and stop when we reach the current index. The last saved index is then assigned the value of the current index.

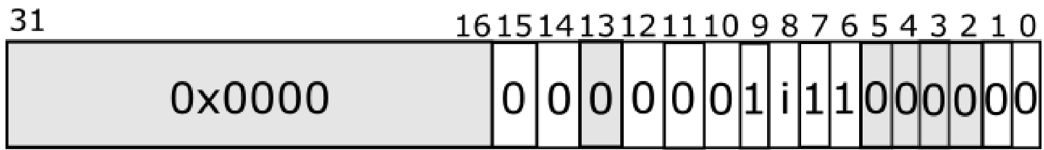


Figure A.1: IA32_DEBUGCTL MSR as used for BTS. Shaded bits are reserved.

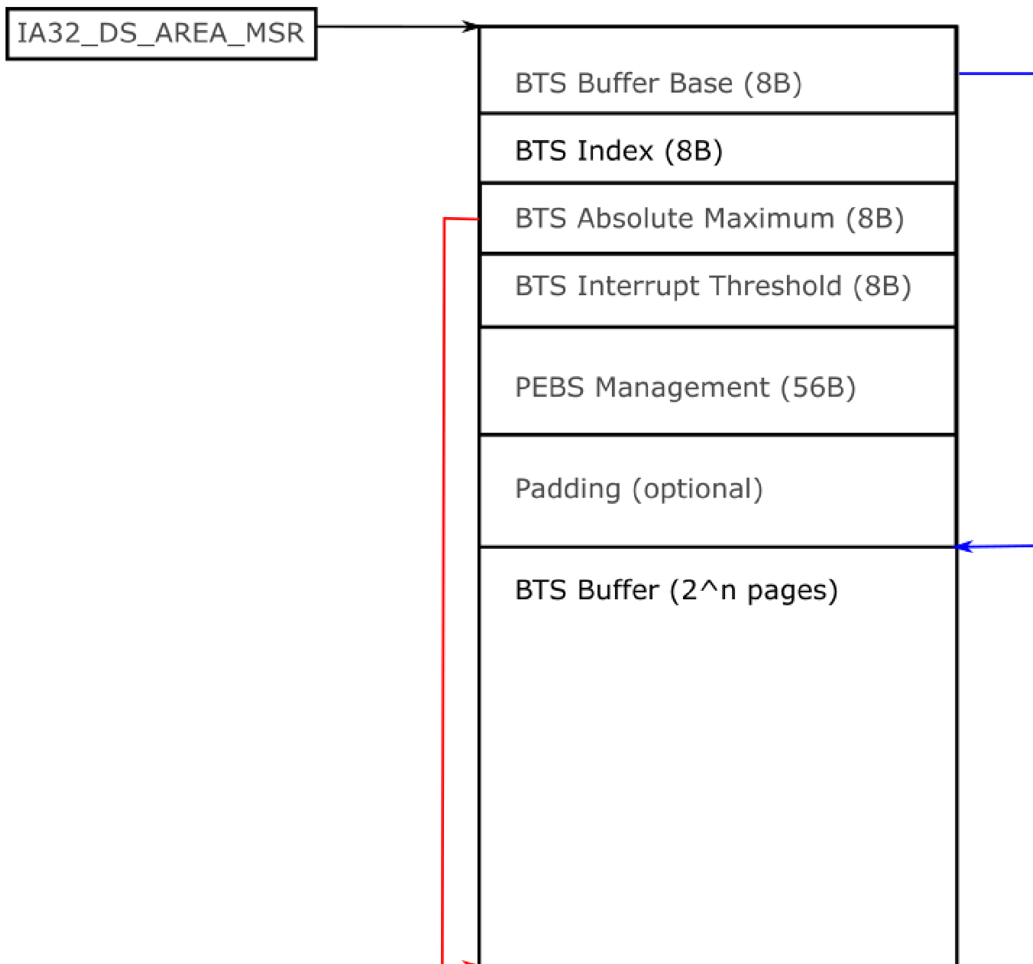


Figure A.2: DS save area. We pad the BTS buffer to start at a page boundary, but it is only required to start at a doubleword boundary.

APPENDIX B

BINARY PATCH TYPES

B.1 ExamDiff Pro

ExamDiff Pro is a commercial diffing tool for files and directories. Patches are large and slow to create, but intuitive. The program provides a visual comparison of binary files, and can output the diff to a human readable text file.

A patch file contains a list of file offsets and the changes that occur there: add, delete, and change. Add indicates bytes that are inserted into the new file, delete shows bytes that in the old file that are not present in the new file, and change is for same-length sequences of bytes where the bytes are replaced. The changes are listed in the order of file offsets.

In this section and Section B.2, we demonstrate how the patch format is used to generate a new file, using the same byte sequences. Note that these tools would not necessarily generate these exact patches. Figure B.1 shows how this program might diff the files. Three bytes are deleted, then the patch skips to the inserted bytes. The last byte is changed because they align.

```
File 1: 00 01 02 03 04 05 06 07 08 09 0A

File 2: 00 04 05 06 07 AA AB AC 08 09 AD

            1,3d1   8a5,7   10c10
            < 01    > AA    < 0A
Patch:      < 02    > AB    ---
            < 03    > AC    > AD
```

Figure B.1: A possible patch for the ExamDiff Pro file format.

```
File 1: 00 01 02 03 04 05 06 07 08 09 0A
File 2: 00 04 05 06 07 AA AB AC 08 09 AD
```

```
Control: [1, 0, 3][4, 3, 0][3, 0, 0]
Data: 00 00 00 00 00 00 00 A3
Extra: AA AB AC
```

Figure B.2: A possible patch for the bsdiff file format.

B.2 bsdiff

Our other supported format is created by bsdiff [15], a patch tool intended for binary files. The resulting patches are often significantly smaller than those made by other tools. A patch file consists of a list of control blocks, a data section, and an extra section. These sections are compressed using bzip2.

The first field of the control block specifies how many bytes to read from the old executable. Then, the same number of bytes is read from the data section and every pair of bytes is added. The second field is how many bytes to read from the extra section. The third field is how many bytes to skip in the old file. This number can be negative.

Figure B.2 shows an example with all positive numbers. The first control block indicates that one byte of the old version is taken and added to one byte from the data section. Since the data byte is 0, that byte is unchanged. No extra bytes are used, and we seek forward by three bytes in the old file. In this case, the seek indicates that those three bytes are deleted, but that is not universally true because we might seek backwards in the next control block. The second control block takes four bytes from the old file, again adding zeros, then the three extra bytes are inserted. In the last control block, we have three bytes left in each version of the file, so we add three data bytes to the old bytes. Note that the last data byte is nonzero, so this indicates a byte change.

B.3 Courgette

The Chromium team presents an algorithm to create smaller patches for Google Chrome as an alternative to bsdiff, which had previously generated the smallest patches. Instead of transmitting a patch made by bsdiff, they disassemble both binaries, use an "adjustment" step to resolve differences between addresses in the two versions, and use bsdiff to create a patch for the new adjusted assembly. To apply the patch, the old binary is disassembled, the disassembly is patched, and the result is assembled to create the new binary. We currently do not support binary comparisons using this algorithm.

APPENDIX C

EXECUTABLE FORMAT

C.1 Linux ELF

ELF files contain a header, a section table, a program header table, and section contents. For our purposes, we need to parse the ELF header and the section table to load the code sections for patching. The file header at the start of the file specifies locations and sizes of tables, including the section table.

Section header entries contain information needed to read each section and translate between file offsets and virtual addresses. For a full list of header fields, see Tables C.1 and C.2. `sh_offset` is the file offset of the start of the section, `sh_addr` is the corresponding virtual address, and `sh_size` is the size of the section in bytes. For a full description of the format, see [19]. The 32-bit format is very similar, but 64-bit fields are 32 bits.

C.2 Windows PE

To load sections and translate offsets to addresses, we must load the DOS header at the start of the file. At offset 60 (0x3C), the last field of the DOS header is the file offset pointing to the 4-byte PE signature. Immediately after is the COFF header, which contains the number of sections. Table C.3 lists the fields of this header, including the signature.

The optional header, located after the COFF header, is a large structure with the `ImageBase` field, which is the preferred virtual address for the program image. This field is found at offset 28 or 24 of the optional header for PE32 and PE32+ respectively. After the optional header is the section table. The format for a section table entry is shown in Table C.4. For more

Table C.1: List of 64-bit ELF header fields.

Name	**Size in bytes**	**Description**
e_ident	16	To identify the file as ELF.
e_type	2	Object file type.
e_machine	2	Architecture of the machine.
e_version	4	Version of file format.
e_entry	8	Virtual address of entry point.
e_phoff	8	File offset of program header table.
e_shoff	8	File offset of section header table.
e_flags	4	Processor-specific flags.
e_ehsize	2	Size in bytes of the ELF header.
e_phentsize	2	Size in bytes of program header table entry.
e_phnum	2	Number of entries in program header table.
e_shentsize	2	Size in bytes of section table entry.
e_shnum	2	Number of entries in section table.
e_shstrndx	2	Section header table index of name table.

Table C.2: List of 64-bit ELF section header fields.

Name	**Size in bytes**	**Description**
sh_name	4	Offset in section name table.
sh_type	4	Section type.
sh_flags	8	Section attributes.
sh_addr	8	Virtual address of start of section.
sh_offset	8	File offset of start of section.
sh_size	8	Size in bytes of section.
sh_link	4	Section index of an associated section.
sh_info	4	Extra information.
sh_addralaign	8	Alignment of section.
sh_endsize	8	Size in bytes of entries, if fixed size.

Table C.3: List of COFF header fields.

Name	Size in bytes	Description
signature	4	Should be “PE” followed by two nul bytes.
machine	2	Target machine type.
numberOfSections	2	Number of sections.
timeDateStamp	4	Indicates when file was created.
symbolTablePointer	4	File offset of symbol table.
numSymbols	4	Number of symbols.
sizeOfOptional	2	Size of optional header.
characteristics	2	Flags for extra information.

Table C.4: List of PE section header fields. Some fields are intended for object files and not executables.

Name	Size in bytes	Description
name	8	String for name.
virtualSize	4	Size of section in memory.
virtualAddress	4	Offset from image base when loaded in memory.
sizeOfRawData	4	Size of section in file.
pointerToRawData	4	File offset for section data.
pointerToRelocations	4	File offset to start of relocation entries (zero).
pointerToLineNumbers	4	File offset to start of line number entries.
numberOfRelocations	2	Number of relocation entries (zero).
numberOfLineNumbers	2	Number of line number entries (zero).
characteristics	4	Flags for section characteristics.

details, see [20]. To translate a file offset to a virtual address, the `ImageBase` is added to the correct section's `virtualAddress` and the difference between the given offset and `pointerToRawData`.

APPENDIX D

SINGLE VARIABLE CHANGE

To test small changes to an input program with optimizations, we compiled a simple test program with several functions with the `-O3` gcc flag. Between the two versions, all functions remained exactly the same, except for one line of one function, shown in Figure D.1. We change line 16 so that instead of reading variable `j`, we read variable `k`.

Looking at the overall graph in Figure D.2, all actual changes take place in the same function, the one that was modified. Most of the changes involve function setup and teardown such as registers being initialized in a different order. The one basic block that clearly relates to the change (Figure D.3), in the middle of the function body, only differs in register allocation. Specific register allocations are not considered, but whether a register is caller-save or callee-save is less arbitrary, so we consider this to be different. It is important that this block be identified because although other blocks contain differences as a side effect of the different variable usage, those blocks are part of the entrance and exit to the function and are not representative of the actual changed code. Those blocks will be executed once per function, but the basic block in Figure D.3 is located in a `for` loop, and may be executed many times.

```
 1  int b(int i){
 2      int k, j = -4, l = 0;
 3      for(k = 0; k < 1; k++){
 4          j += a();
 5          l += j >> 1;
 6          printf(" ");
 7          if(j & 3){
 8              while(l & 3 == 3){
 9                  l--;
10                  j = d(j);
11              }
12          }
13          else if(l & 1){
14              l += a();
15          }
16          if((l + j) & 0xc1c1){
17              int m;
18              for(m = 0; m < (j & 0xf); m++){
19                  printf("hat%d\n", m);
20              }
21          }
22      }
23      return j + l;
24  }
```

Figure D.1: Test function.

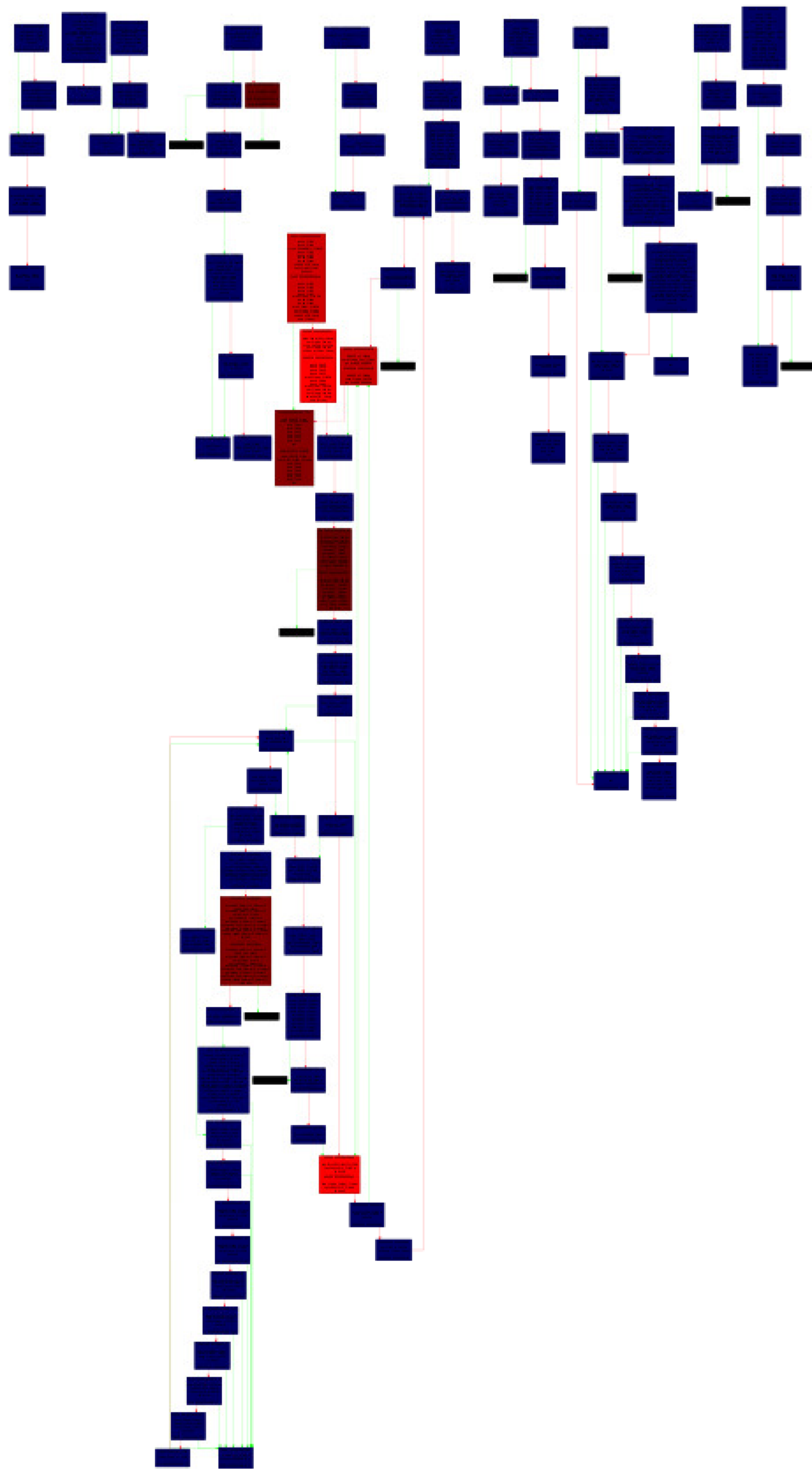


Figure D.2: Graph of the test program.

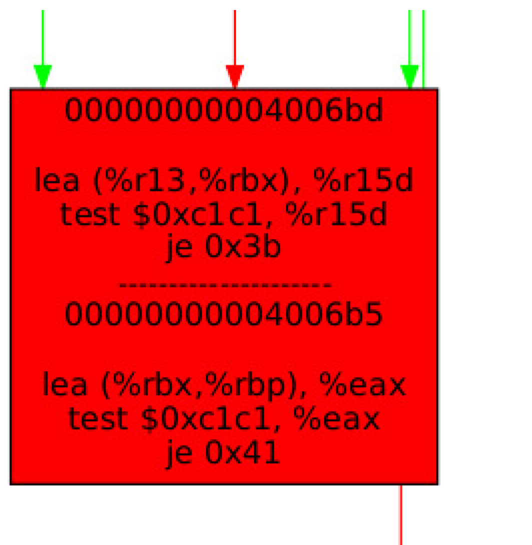


Figure D.3: Basic block reflecting change.

REFERENCES

[1] J. Oberheide, E. Cooke, and F. Jahanian, "If it ain't broke, don't fix it: Challenges and new directions for inferring the impact of software patches," in *Workshop on Hot Topics in Operating Systems (HotOS'09)*, 2009.

[2] Hex-Rays SA. (2017) IDA Pro Disassembler. [Online]. Available: http://hex-rays.com/idapro

[3] P. Amini. Paimei. [Online]. Available: http://pedramamini.com/PaiMei/docs/

[4] Z. Wang, K. Pierce, and S. McFarling, "BMAT: A binary matching tool for stale profile propagation," *J. Instruction-Level Parallelism 2*, 2000.

[5] J. Oh, "Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries," in *Black Hat USA*, 2009.

[6] M. Bourquin, A. King, and E. Robbins, "Binslayer: Accurate comparison of binary executables," in *Proc. ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW'13)*, 2013.

[7] H. Flake, "Structural comparison of executable objects," in *Proc. IEEE Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'04)*, 2004, pp. 161–173.

[8] J. Munkres, "Algorithms for the assignment and transportation problems," *J. of the Society for Industrial and Applied Mathematics*, vol. 5, pp. 32–38, 1957.

[9] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proc. ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*, 2014.

[10] D. Gao, M. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *Proc. International*

Conference on Information and Communications Security (ICICS'08), 2008, pp. 238–255.

[11] J. Ming, M. Pan, and D. Gao, "iBinHunt: Binary hunting with inter-procedural control flow," in *Proc. International Conference on Information Security and Cryptology (ICISC'12)*, 2012, pp. 92–109.

[12] J. Ming, D. Xu, and D. Wu, "Malwarehunt: Semantics-based malware diffing speedup by normalized basic block memoization," *J. Computer Virology and Hacking Techniques*, 2016.

[13] X. Zhang and R. Gupta, "Matching execution histories of program versions," in *Proc. European Software Engineering Conference (ESEC) held jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'05)*, 2005.

[14] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, 2016.

[15] C. Percival, "Naive differences of executable code," 2003, unpublished. [Online]. Available: http://www.daemonology.net/bsdiff/

[16] E. W. Myers, "An o(nd) difference algorithm and its variations," *Algorithmica*, pp. 251–256, 1986.

[17] Project Gutenberg. [Online]. Available: https://www.gutenberg.org/

[18] Software updates: Courgette. (2017) [Online]. Available: https://www.chromium.org/developers/design-documents/software-updates-courgette

[19] Elf-64 object file format. (1998) [Online]. Available: https://www.uclibc.org/docs/elf-64-gen.pdf

[20] PE format (windows). [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547(v=vs.85).aspx