Compressing Differences of Executable Code

Brenda S. Baker[*], Udi Manber[†], and Robert Muth[‡]

April 22, 1999

Abstract

Programs change often, and it is important to bring those changes to users as conveniently as possible. The two most common ways to deliver changes are to send a whole new program or to send "patches" that encode the differences between the two versions, requiring much less space. In this paper, we address computation of patches for executables of programs. Our techniques take into account the platform-dependent structure of executables, We identify changes in the executables that are likely to be artifacts of the compilation process, and arrange to reconstruct these when the patch is applied rather than including them in the patch; the remaining changes that must be placed in the patch are likely to be derived from source lines that changed. Our techniques should be useful for updating programs over slow data lines and should be particularly important for small devices whose programs will need to be updated through wireless communication. We have implemented our techniques for Digital UNIX Alpha executables; our experiments show our techniques to improve significantly over previous approaches to updating executables.

1 Introduction

Computing file differences is a common practice for source code files, with two main purposes:

- Revision control; we keep the current version of the source and differences compared to all old versions so that the older versions can be recovered on demand.
- Patching; we use the differences (delta file) to construct a new version based on the old version.

The differences are typically much smaller than the original, leading to significant storage and transmission savings. This problem area is expected to become particularly important for small devices whose programs may need to be updated through slow expensive links (such as wireless).

Computing differences between two text files has been studied extensively, e.g. [5, 4, 6, 7, 9]), and many algorithms have been designed and are deployed daily for the two purposes above. Computing such a difference is usually done by viewing the files as sequences of lines and applying a sequence-comparison algorithm.

Differences of executables are more complex, because a small change in the source file can create major changes throughout the executables. Figure 1 illustrates this problem.

Figure 1 shows two executables, A and B. Let us assume that B was derived from A by inserting some additional code (shaded). The actual difference – byte-wise – is much bigger than the inserted code, due to many secondary changes caused by the main change. For example,

- a program counter relative branch that jumps over the inserted code will have a different branch displacement,

[*]Bell Laboratories, Lucent Technologies, 700 Mountain Avenue, Murray Hill, NJ 07974,bsb@bell-labs.com

[†]Department of Computer Science, University of Arizona, Tucson, AZ 85721, and Yahoo! Inc., 3400 Central Exp., Santa Clara, CA 95051, udi@cs.arizona.edu

[‡]Department of Computer Science, University of Arizona, Tucson, AZ 85721, muth@cs.arizona.edu

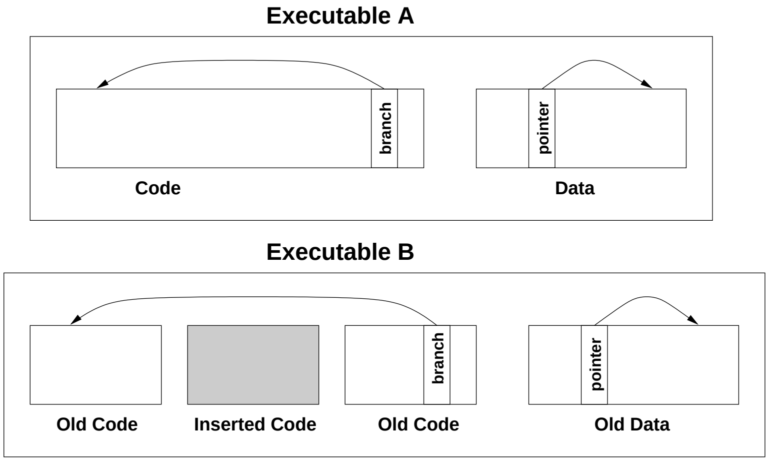

Figure 1: Two related executables

- an (absolute) pointer in the data segment pointing to some other place in the data segment will have a new value since all the addresses have moved up a little.

We have developed techniques for constructing a delta for two executables, which we call the *original* and the *upgrade*, such that it is possible to recover the upgrade from the original and the delta. We refer to the first process as *delta generation* and the second process as *delta application*. The goal is to minimize the size of the delta.

We do not assume a knowledge of the changes in the corresponding source files. But we do assume a knowledge of the architecture. Our techniques are designed to minimize the size of the differences that must be included in the delta by relying on recalculation of offsets and similar pieces of information where possible; a single item stored in the delta may lead to a cascading of computations through the upgrade to reconstruct multiple items. We believe that our techniques will be adaptable to many architectures.

Two concepts are central to our method: *preliminary matching* (or *pre-matching*) and *value recovery*. Both will be explained in subsequent sections.

Pre-matching is an attempt to identify items, such as instructions, in the original and the update that correspond. They may look different in the executables due to secondary changes. The idea is to first ignore the secondary changes, find a good correspondence, and fix things later. Pre-matching results in an alignment of certain items in the original and upgrade executables, together with a list of edit operations, e.g. insertions and deletions, representing primary changes.

Value recovery tries to deal with secondary changes by predicting values in the upgrade, eg. jump displacements. It is used both in delta generation and in delta application. We usually proceed from smaller to higher addresses (offsets), so that when trying to recover a particular value, we can exploit both the alignment and previously recovered values.

In delta generation, we predict an upgrade value from the available information and then check whether it agrees with the actual known value. If it disagrees, we store it explicitly in the delta. If the value recovery scheme is good, the number of those items should be small. The predictions are based on several heuristics; the type of the item (e.g. register or pointer) determines which heuristics are tried and in which order. It is not necessary to include bits in the delta to encode which heuristic is finally used to predict each value, because this information is also recoverable at delta application time. To protect against possible globally deleterious effects from local pre-matching and prediction choices, the value recovery phase self-tunes to reduce the delta size through several iterations of adjusting the alignments and the predictions.

During delta application, values that were stored explicitly in the delta are copied into the upgrade, while other values are recovered. To recover a value, the same sequence of heuristics as in delta generation are used to predict the value, and the value is copied into the upgrade. Once computed, recovered values may be used as the basis for subsequent recoveries. Thus, a single value stored in the delta may have a cascading effect, enabling the computation of later values in the upgrade.

Two related approaches have also addressed the problem of reducing the download time for new versions of executables. The first is to produce a delta from two arbitrary executables without using domain knowledge. We know of two programs that take this approach: bindiff [1] and PocketSoft's RTPatch [8]. The second approach is to compress just the target file using domain knowledge (cf. [3]). This would be advantageous when the old executable is not available. However, it would be disadvantageous in the case of a software distributor who would like to make a patch easily available, say on the Web, but only usable by people who have already licensed the original executable.

We have implemented our techniques for Digital UNIX Alpha binaries. We have compared the sizes of compressed delta files produced by our techniques with the compressed upgrade files and with the compressed deltas produced by bindiff. In nearly all cases our techniques substantially improved upon both. In fact, we usually beat bindiff by a factor of 2 to 5.

2 Overview of Delta Generation (Exediff)

The program that generates a delta from the original and upgrade executables is called Exediff. Exediff relies on the techniques of pre-matching and value recovery: pre-matching for an initial correspondence between items in the original and upgrade executables, and value recovery to identify which changes do not need to be explicitly stored in the delta.

Pre-matching has two purposes: to identify probable primary changes in the upgrade executable compared to the original executable, and to align the remaining instructions or data items that seem to correspond, even though these may not be identical because of secondary changes. The approach is heuristic in nature, since we do not assume access to source code here. However, we envision the deltas being generated by the owner/compiler of the source, and consequently the delta generation process could have access to additional information which is suitable to improve the pre-matching and potentially reduce the size of the delta. This possibility is left for further research.

To simplify the presentation we will regard an executable as a sequence of machine instructions. Domain knowledge about the platform tells us what types of information in instructions could be affected by secondary changes. For example, insertion of instructions could change jump displacements across the inserted code, and we may need to ignore jump displacements while we compute an alignment.

Consequently, pre-matching is achieved by transforming the original sequences into two new sequences and then computing the longest common subsequences of the new sequences. The transformation will usually be lossy, ie. it throws information away, although the original form is reserved for further analysis later. This approach will increase the number of matched items (e.g. instructions or data bytes) since items that are equal before the transformation will also be equal afterwards. It will also increase the number of spurious matches and we will describe later how to deal with this.

Pairs of items aligned in the pre-matching are identical with respect to the transformed sequences, but not necessarily with respect to the actual executables. For example, in application to the Alpha, described in Section 5, our transformation on instruction sequences preserves special registers but maps all others to the same value; two matching instructions may match in opcode but differ in actual registers used.

Consequently, after the matching we distinguish 3 classes of items with respect to the original and upgrade executables:

- unmatched items, considered to represent primary changes
- matched but unequal items, considered to represent secondary changes
- matched and equal items

Subsequently, we apply value recovery techniques to determine which changes must be stored into the delta, and which can be recomputed once earlier values have already been determined. Only matched items are considered during value recovery either as producer or consumer of information.

For example, if the aligned items include ten instances of changing a value A to B, we need to store only the first change into the delta, provided that we can deduce at later occurrences of A in the original executable that we should substitute B in the upgrade executable. The first instance is a pair of *unrecoverable matched items* while the others are *recoverable matched items*. The value recovery techniques used for each item depend on domain knowledge and on the type of item, e.g. register, pointer, or integer.

The first instance may also be recoverable if B is equal to A.

When value recovery is unable to recover certain matched values, it is sometimes caused by a bad alignment, eg. spurious matches. Exediff keeps track of matched pairs that cause the value recovery to repeatedly fail. In such a case the alignment will be changed and value recovery restarted. This process is described in more detail in Section 4.2. Through this iterative process, we eventually arrive at a final alignment, a list of primary changes (unmatched items), and a list of secondary changes (unrecoverable matched items). We generate the delta as follows:

- We store the primary changes as a sequence of insertion and deletion commands.
- We store the secondary changes by encoding the location and value of the unrecoverable matched items.

We do not need to store the alignment of matching items explicitly in the delta because it is implicitly available from the insertions and deletions commands.

3 Overview of Delta Application (Exepatch)

The program that applies the delta is called Exepatch. From the original executable and the delta, Exepatch must construct the upgrade.

From the primary changes stored in the delta, Exepatch can deduce the final alignment found by Exediff. This alignment provides a "hull" with no values yet. Exediff fills values into the hull as follows:

- It fills the hull with the values of the primary changes stored in the delta.
- It fills the hull with the values for the unrecoverable matched items stored in the delta, i.e. the secondary changes.
- It recovers the values for the remaining (recoverable) matched items.

The value recovery heuristics are the same ones used by Exediff; the heuristics predict an upgrade value from a value in the original executable and previous matched and recovered values. However, where Exediff would compare the predicted value to the upgrade to determine whether to put it into the delta, Exepatch merely places the predicted value into the delta.

These heuristics are guaranteed to produce the correct upgrade. In particular, the value recovery techniques must produce the correct values for the upgrade, because by construction, if a value predicted by the value recovery techniques were not that of the upgrade, Exediff would have placed it into the delta. Our value recovery techniques are described in the next section.

4 Value Recovery Techniques

Value recovery uses the pre-matching alignment to reconstruct the values of the upgrade. Values are divided into types based on domain knowledge about the file format and the data within the file. A value may be a tuple representing an object such as an instruction that has multiple parts, e.g. opcode and registers. However, in this section we give only a general description of the techniques for simple types of values. The next section describes how these techniques apply to Alpha executables.

For the moment, we focus on the application of the delta information. Figure 2 shows the data structures representing the matching.

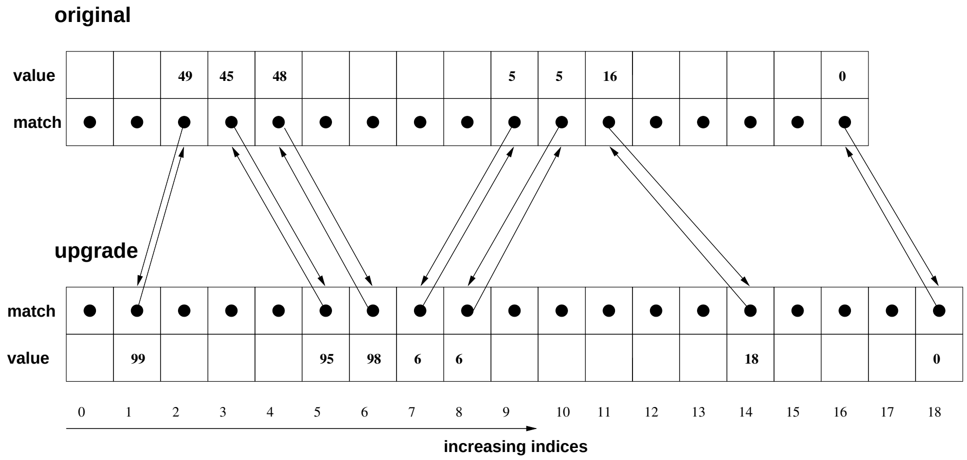

Figure 2: The data structures used by our techniques

The concrete task of the value recovery is to reconstruct those values of the upgrade (`upgrade[t].value`) which are part of a matching (`original[s].match` equals t for some s). The values of the upgrade which are not part of the matching are not reconstructible and we need to store them explicitly in the delta (primary changes)

To illustrate this, let us consider the case of the ordinary delta generation by diff: if two items (text lines) match, their values must be identical. So the value recovery is trivial:

`upgrade[u].value := original.[upgrade[u].match].value`

In the case of our pre-matching, `upgrade[t].value` and `original.[upgrade[t].match].value` may or may not be equal. Consequently, we have developed several simple schemes for value recovery:

1. **MatchValue**. This scheme replaces the value in the upgrade by the value matched to it in the original, as described above for diff.

2. **TranslateAddress**. If a value denotes an address within a file, the index corresponding to that address can be computed, and if the item at that index has been matched we can derive the new address directly from the matching. “Address” has a very broad meaning here. Anything that can point us to some item within a matching can be considered as an address, eg. offsets, table indices, file positions, etc..

   Example. Consider the item `upgrade[14]` in Figure 2. We know from the file format that this value is an address. In order to recover it using the TranslateAddress scheme we look at the value of the matching item (`original[11]`). For simplicity, assume that addresses and indices are identical, so `original[11]` contains the address of the last item (16) in `original`, which is matched with the last item (18) in `upgrade`. Hence the scheme correctly derives 18 as the value of item `upgrade[14]`.

3. **EqualValue**. If a value in the original also occurs in the original in a different matching for which we have already recovered the corresponding upgrade value, we can use the previously recovered value from that match. In our implementation, we restrict our attention to a set O of indices in the original; O includes only indices for which we have previously recovered upgrade values, but the particular set depends on the type of item.

   Example. Consider the item `upgrade[8]` in Figure 2. In order to recover it using the EqualValue scheme we search for the value of the matching item (`original[10]`). Let us assume that $O = \{2,3,4,9\}$. Among those indices only item `original[9]` agrees in value. Its match, `upgrade[7]`, contains the value 6, which we assume has been previously recovered. This is also the correct value for `upgrade[8]`.

4. **CloseValue**. This is similar to the previous scheme, but more general, because we do not restrict ourselves to find an equal value within the index set O but just a close value. This is useful in cases where we cannot find an equal value.

Example. Consider the item `upgrade[6]` in Figure 2. In order to recover it using the CloseValue scheme, we search for a value close to the value 48 of the matching item (`original[4]`). Let us assume that $O = \{2,3\}$. Among those indices, item `original[2]` is 49, only one more than 48. Its match, `upgrade[1]`, contains the value 99, which we assume has been previously recovered. Thus, we calculate 98 correctly as the value for `upgrade[6]`.

The last two schemes rely on other values that must have been recovered earlier. The simplest strategy is to recover the value for item `upgrade[u]` in order while restricting the set O to smaller indices, ie. subsets of $[1 : o-1]$, where o is the current index in the original file. Since matchings never "cross", i.e. the lines connecting matching pairs as in Figure 2 never cross, this achieves the desired convergence. Other strategies could also achieve convergence. EqualValue and CloseValue must operate deterministically in order to assure that while computing and applying delta information the same choices are made.

4.1 Chaining of Value Recovery Schemes

The different methods can be chained. If one scheme fails to predict the correct value or is not able to predict any value in the upgrade, we can try a different scheme. The important point is that the order of application of schemes is fixed in the program for each value type, so that the same predicted value will be obtained in the generation of the delta and in the application of the delta. Furthermore, as long as the program can determine from available information such as the alignment whether a particular scheme failed, no bits are needed in the delta to encode which scheme to use to recover the value when chaining occurs, eg. the EqualValue scheme might not find any proper previously matched value and hence will not predict any value.

4.2 Eliminating Spurious Matches

In the case of the last three recovery schemes, certain matches may hurt value recovery in the sense that the eventual delta will be larger than if the items had not been not matched. Such matches may be spurious, in the sense that these items were accidentally matched due to the heuristic nature of the pre-matching. Conceptually, we could remove each match in turn and check the size of the generated delta; if it gets smaller, we would win by eliminating the match. In practice, we approximate this by computing the "benefits" of each matching during value recovery, eliminating the matches that "hurt", and repeating the recovery phase. Likely candidates for elimination are unrecoverable matched items, since they are part of the delta anyway. This step is iterated several times.

5 Application: DEC UNIX Alpha Executables

DEC UNIX Alpha executables are stored using the ECOFF object file format [2]. As with most UNIX systems, an executable is composed of three segments

1. **Text**. Contains mostly instructions plus read-only constants.
2. **Data**. Contains initialized data structures.
3. **Bss**. Contains zero initialized data structures.

Only the Text and Data Segment, together with other administrative information (eg. the size of the Bss Segment, symbol table information, relocation information, etc.), are stored in the executable.

We apply pre-matching and value recovery to the Data Segment and the code bearing parts of the Text Segment. For the remaining parts of the executable we use bindiff as described in [1] to compute the delta information.

5.1 Text Segment

For simplicity, we assume that the Text Segment consists entirely of instructions. Hence it can be regarded as a sequence of 32-bit-wide Alpha instruction words. An Alpha instruction typically has two operand registers and one result register. At the expense of one or more registers, an instruction might also include an immediate value. For simplicity, we assume that each instruction has three registers **and** an immediate value. The format is

$$opcode\ reg_1\ reg_2\ reg_3\ immediate$$

If an instruction has fewer than 3 registers, we substitute *zero* or $fzero$ for the missing ones. *Zero* is a register that is hardwired to the integer value 0, while $fzero$ is its floating point equivalent. Other special purpose registers are the stack pointer *sp* and the global pointer gp.

We define $AdminRegs := \{zero, fzero, gp, sp\}$

5.1.1 Pre-matching

We need to abstract away all the information that is likely to be subject of secondary changes, such as immediates and ordinary registers. The following transformation function is used:

$$opcode\ \ reg_1\ \ reg_2\ \ reg_3\ \ immediate \rightarrow opcode\ \ RFilter(reg_1)\ \ RFilter(reg_2)\ \ RFilter(reg_3)\ \ IFilter(immediate)$$

where

$$RFilter(r) := \text{ if } r \in AdminRegs \text{ then } r \text{ else } \varepsilon \text{ endif}$$

and

$$IFilter(i) := \text{ if } i \geq 0 \text{ then POS else NEG endif}$$

5.1.2 Value Recovery

Pre-matching for instructions is based on incomplete information about registers and immediates. Value recovery needs to reconstruct those values.

Register Recovery

Register recovery uses the MatchValue and the EqualValue schemes to recover the registers of an instruction. Since there are three registers per instruction a pre-matching of two instructions represents three pre-matchings of the corresponding registers in the following order: first operand register, second operand register, result register,

We employ MatchValue if the original register is in *AdminRegs* or if the instruction containing the register is function call or return. The latter case is motivated by the calling conventions, which dictate the register to be used by those instructions in most cases.

EqualValue is employed for all other cases. We divide the instruction (and hence register) matching into smaller matchings each representing one function within the executable. This is possible since the instructions belonging to one function are consecutive in the Text Segment and special instructions mark the beginning of a function. The subset O of indices is then chosen as the set of matched indices that are smaller than the current one and still lie within the current function. The idea here is that the choice of registers in one function rarely influences the choice of registers in another function. If EqualValue is unable to recover a register because it cannot find another match involving the same register we try MatchValue, i.e. we chain two recovery techniques as described in the Section 4.1. Note that no bit is needed in the delta to encode which scheme is used.

Immediate Recovery

We found it beneficial to classify the immediate value based on opcodes and registers in *AdminRegs* and then perform the value recovery within those classes. Pre-matching will only match instructions belonging to the same class. Often we limit our search to matches within the current function, as we did with register recovery. The classes and the corresponding recovery chains were found by experimenting. All four recovery heuristics are used. Because of space restriction we are only able to discuss two examples here.

- Class SpLoadStore. This class contains all loads and stores to and from the stack. The immediate value is the stack frame offset of the data item loaded or stored. The relative position of those data items on the stack is unlikely to change. Hence we apply the CloseValue heuristics first. (The O set contains all the matched indices that are smaller than the current one, are of class SpLoadStore, and lie within the current function). If CloseValue is unable to recover a offset we fall back to MatchValue.
- Class Branch. This class contains all conditional and unconditional branch instructions. The immediate value is a program-counter relative branch displacement. Since conversion between relative and absolute addresses is trivial, we will regard the value as an absolute address. We first attempt to recover the displacement using the TranslateAddress scheme. If this scheme does not provide an answer, we fall back to EqualValue. If this fails too, we resort to the MatchValue scheme.

5.2 Data Segment

The Data Segment contains initialized data structures either specified by the programmer or generated by the compiler/linker. It also contains 64-bit (8-byte) pointers into the Text, Data, and Bss Segments. Those pointers are 8-byte aligned. We can therefore determine with high probability whether an arbitrary sequence of 8 bytes represents a pointer or not.

5.2.1 Pre-matching

The transformation function is the identity function for most data. However, pointers receive special treatment:

- **Text Pointer**. The byte sequence points into the Text Segment to instruction i.

  $$byte_0\ byte_1\ byte_2\ byte_3\ byte_4\ byte_5\ byte_6\ byte_7 \rightarrow \text{TEXT}\ \ hash(opcode_i, opcode_{i+1}, opcode_{i+2})$$

  The hash function is used to encode information about the target of the pointer, to make it more likely that pointers will be matched if they point to the corresponding regions of code.
- **Data Pointer**. The byte sequence points into the Data Segment to byte b .

  $$byte_0\ byte_1\ byte_2\ byte_3\ byte_4\ byte_5\ byte_6\ byte_7 \rightarrow \text{DATA}\ \ b$$

- **Other Pointer**. The byte sequence is a pointer but neither a text nor a data pointer.

  $$byte_0\ byte_1\ byte_2\ byte_3\ byte_4\ byte_5\ byte_6\ byte_7 \rightarrow \text{OTHER}$$

- **Ordinary Data**.

  $$byte_0\ byte_1\ byte_2\ byte_3\ byte_4\ byte_5\ byte_6\ byte_7 \rightarrow byte_0\ byte_1\ byte_2\ byte_3\ byte_4\ byte_5\ byte_6\ byte_7$$

5.2.2 Value Recovery

Text and Data Pointers are recovered using the following chain of recovery schemes: TranslateAddress, CloseValue (where O contains all the matched indices smaller than the current index of the same pointer type), MatchValue. The pre-matching alignment for the Text Segment is used when dealing with Text Pointers.

Other Pointers are recovered using the following chain of recovery schemes: CloseValue (where O contains all the matched indices smaller than the current index of Other Pointers), MatchValue.

Ordinary Data is recovered using MatchValue.

program	upgrade.gz	bindiff.gz		delta.gz		
	bytes	bytes	% of upgrade.gz	bytes	% of upgrade.gz	% of bindiff.gz
alto: identical versions	162470	54	0.0	155	0.1	287.0
alto gcc -O2 → gcc -O3	162470	83051	51.1	20793	12.8	25.0
alto: changed reg. alloc.	162549	109753	67.5	16813	10.3	15.3
alto: added a printf	162470	54819	33.7	6237	3.8	11.4
agrep 3.6 → 4.0	121849	92437	75.9	42468	34.9	45.9
agrep 4.0 → 4.1	121753	14535	11.9	3531	2.9	24.3
glimpse 3.6 → 4.0	235774	198059	84.0	109329	46.4	55.2
glimpse 4.0 → 4.1	235913	132726	56.3	23200	9.8	17.5
glimpseindex 3.6 → 4.0	205945	166122	80.7	82545	40.1	49.7
glimpseindex 4.0 → 4.1	206074	124173	60.3	18473	9.0	14.9
wgconvert 4.0 → 4.1	166789	93493	56.1	15688	9.4	16.8
netscape 3.01 → 3.04	2558478	1471610	57.5	284992	11.1	19.4
icalc 2.1b2 →2.2	589862	54819	9.3	6237	1.1	11.4
gimp 0.99.19 → 1.00.00	684725	495869	72.4	191657	28.0	38.7
iconx 9.0 → 9.3	242013	233730	96.6	38121	15.8	16.3
cc1 (gcc) 2.8.0 → 2.8.1	831626	847457	101.9	76313	9.2	9.0
rcc (lcc) 3.2 → 3.6	157685	99017	62.8	22019	14.0	22.2
rcc (lcc) 4.0 → 4.1	237033	645	0.3	303	0.1	47.0
apache 1.2.4 → 1.3.0	200413	201470	100.5	253300	126.4	125.7
apache 1.3.0 → 1.3.1	201529	119585	59.3	42038	20.9	35.2

Table 1: Experimental results for Set 1

6 Experimental Results

6.1 Set 1

We ran 20 experiments on pairs of executables for 12 distinct programs. The results are shown in Table 1. We compared the gzipped deltas produced by our techniques with the size of the gzipped upgrades and the gzipped deltas produced by bindiff.

The first four experiments were simple checks on behavior when the original source and upgrade source were identical or almost identical. In the first experiment, the original and upgrade executables were identical. In the second, only the compiler optimization level was changed to create the second executable. In the third, the compiler was forced to change the register allocation for the upgrade executable. In the fourth, a single `printf` statement was added to the source.

The remaining 16 experiments tested different versions of executables for the same program; the specific programs and versions are listed in the table. In every experiment except one, the size of the gzipped delta file produced by our techniques was substantially smaller than both the gzipped upgrade and the gzipped delta produced by bindiff. For upgrades that affected only the minor version number of a program we typically observed a fivefold reduction in the delta size (compared to bindiff), otherwise we observed a twofold reduction. The one exception was apache versions 1.2.4 and 1.3.0, for which the gzipped delta produced by our techniques was 26% larger than the gzipped upgrade; the gzipped delta produced by bindiff was just slightly (0.5%) larger than the gzipped upgrade. In examining the source files, we observed that the source files had been extensively reorganized. However, for apache versions 1.3.0 and 1.3.1, our techniques did produce substantial improvement.

6.2 Set 2

We ran experiments with four versions (V290,V320,V321,V332) of `less`, a pager for text files. The sizes in bytes of the gzipped executable and the gzipped source for the different versions are shown in Table 2

Program	program.gz	src.gz
V290	43416	107817
V320	47470	122318
V321	47473	122328
V334	49635	130563

Table 2: Characteristics of different `less` versions

Programs	src.diff.gz	delta.gz
V290 → V320	28418	21955
V290 → V321	28425	21971
V290 → V332	38179	28395
V320 → V321	254	163
V320 → V332	15518	14711
V321 → V332	15477	14717

Table 3: Experimental results for Set 2

In Table 3 we compare each version with all newer versions. We compare the size in bytes of the gzipped delta for the source code (determined using `diff -n`) with the gzipped delta produced by Exediff.

The delta of the executables code compares favorably with the diff of the source code especially since we only considered true source code and excluded makefiles and configuration scripts.

7 Conclusions

We have shown that our techniques can produce deltas for old and new versions of executables that are significantly smaller than the compressed new version, and also significantly smaller than the delta files produced by bindiff. We can even compete with source code patches when the changes are not too dramatic. In general, source code patches will tend to be smaller than executables patches, but for most applications, especially for small devices, compiling at the end user level is not feasible. Most software companies are also reluctant to release source code. Our techniques could make a significant difference in how patches are distributed. We plan to extend our techniques to other plattforms in particular java class files.

References

[1] Kris Coppieters. A cross-platform binary diff. *Dr. Dobb's Journal*, May 1995.

[2] Digital Equipment Corp. *Object File / Symbol Table Format Specification*. Digital Unix, July 1998.

[3] Jens Ernst, William Evans, Christopher W. Fraser, Steven Lucco, and Todd A. Proebsting. Code compression. *ACM SIGPLAN Notices*, 32(5):358–365, May 1997.

[4] D.S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, pages 664–675, Oct. 1977.

[5] J.W. Hunt and T.G. Szymanski. A fast algorithm for computing longest common subsequences. *Comm. ACM*, 20(5):350–353, 1977.

[6] Webb Miller and Eugene W. Myers. A file comparison program. *Software, Practice, and Experience*, 15(11):1025–1040, 1985.

[7] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[8] PocketSoft. .RTPatch Professional, Feb. 23, 1998. `http://www.pocketsoft.com/products.html`.

[9] Walter F. Tichy. The string-to-string correction problem with block moves. *ACM Trans. on Comput. Syst.*, 2(4):309–321, 1984.