Naïve Differences of Executable Code

Colin Percival
Computing Lab, Oxford University
colin.percival@comlab.ox.ac.uk

Abstract

The increasing frequency with which serious security flaws are discovered and the increasing rapidity with which they are exploited have made it necessary for programs to be updated far more frequently than in the past. While binary updates are generally far more convenient than source code updates, the distribution of pointers throughout executable files makes it much harder to produce compact patches.

In contrast to earlier work which relies upon knowledge of the internal structure of a particular platform's executable files, we describe a naïve method which produces competitively small patches for any executable files.

1 Introduction

Historically, binary patches have been constructed using two basic operations, copying and insertion. Using either substring matching or hashing techniques [Ma00], portions of the new file are matched with portions of the old file; those regions are copied, while the remaining "new" bytes are stored in the patch file and inserted. Patches generated in this manner can therefore be considered as programs consisting of two instructions, COPY and INSERT.

Unfortunately, any source code modification will usually cause changes throughout an executable file. Adding or removing a small number of bytes of code or data will change the relative position of blocks of code, adjusting the displacement of relative branches which jump over the modified region; similarly, any data located after the modified region will have a different address, causing data pointers to be modified throughout the file. This causes patches generated with the traditional copy-and-insert method to be much larger than necessary; a one-line source code patch in a 500kB executable could translate into a 50kB patch file.

One solution to this problem relies upon knowledge of the internal structure of an executable file. If a pointer to address A in the old executable file changes to point at address B in the new executable file, it is very likely that other pointers to address A will also change in the same manner. As a result, by effectively disassembling the entire file and recording the first instance of each such substitution, one can predict future substitutions, thereby obviating the need to record them [BMM99]. However, the necessary disassembly means that any tools using this approach will be entirely platform-dependent.

2 BSDiff

In order to solve the 'pointer problem' in a portable manner, we make two important observations: First, in the regions of an executable file not directly affected by a modification, the differences will generally be quite sparse. Not only will the modified addresses constitute only a small portion of the compiled code, but addresses are most likely to only change in their least significant one or two bytes. Second, data and code tends to be moved around in blocks; consequently, locality of reference will lead to a large number of different (nearby) addresses being adjusted by the same amount. These two observations lead to the important fact that if the regions in two versions of an executable program which correspond to the same lines of source code are matched against each other, the bytewise differences will be mostly zero, and even when non-zero will take certain values far more often than others — in short, the string of bytewise differences will be highly compressible.

We now construct binary patches as follows. First, we read the old file and perform some sort of indexing, either based on hashing [Tr99] or suffix sorting (e.g., [LS99]). Next, using this index, we pass through the new file and find a set of regions which match exactly against regions of the old file. For reasons which will

become evident later, we only record regions which contain at least 8 bytes not matching the forward-extension of the previous match (i.e., if the previous match is $new[x \ldots x+k] = old[y \ldots y+k]$, we look for a match $new[x' \ldots x'+k'] = old[y' \ldots y'+k']$ with at least 8 distinct i such that $new[x'+i] \neq old[x'+i+(y-x)]$).

Conventional binary patch tools would translate this set of perfect matches directly into a patch file. Instead, we generate a pairwise disjoint set of "approximate matches" by extending the matches in each direction, subject to the requirement that every suffix of the forward-extension (and every prefix of the backwards-extension) matches in at least 50% of its bytes. These approximate matches will now roughly correspond to blocks of executable code derived from unmodified regions of source code, while the regions of the new file which are not part of an approximate match will roughly correspond to modified lines of source code. This process of extending the matches is why we ignore any matches which are not "better" than the previous match by 8 bytes.

The patch file is then constructed of three parts: First, a control file containing ADD and INSERT instructions; second, a 'difference' file, containing the bytewise differences of the approximate matches; and third, an 'extra' file, containing the bytes which were not part of an approximate match. Each ADD instruction specifies an offset in the old file and a length; the appropriate number of bytes are read from the old file and added to the same number of bytes from the difference file. INSERT instructions merely specify a length; the specified number of bytes is read from the extra file. While these three files together are slightly larger than the original target file, the control and difference files are highly compressible; in particular, bzip2 tends to perform remarkably well (probably due to the highly structured nature of these two files).

We have implemented this method in a tool named 'BSDiff'.

3 Performance

To evaluate the performance of BSDiff when moving between two versions of an executable, we use 19 pairs of DEC UNIX Alpha binaries used in the exposition of Exediff, the working of which is specific to that platform [BMM99]. In Table 1 we list for each of these pairs the original size of the new version, the compressed size of the new version, the size of patch produced by the currently pre-eminent free binary patch tool, Xdelta [Ma00], the size of patch produced by a widely used commercial tool, .RTPatch [Ps01], the size of patch produced by Exediff, and the size of patch produced by BSDiff. In the interest of a fair comparision, we recompressed Exediff's patches with bzip2 (a block sorting compressor) rather than gzip (a Lempel-Zif compressor) where it was advantageous.

With the exception of two cases where the the difference was exceptionally small, the Apache 1.2.4 → 1.3.0 case, where none of the methods were superior to simply compressing the new binary, and the Apache 1.3.0 → 1.3.1 case, where BSDiff was slightly superior to Exediff, there is a very clear pattern: XDelta gives the largest patches, and Exediff gives the smallest patches, while BSDiff and .RTPatch come in second smallest and second largest respectively. Considering all 19 pairs, and taking the arithmetic mean weighted by the square root of the original file size, we find that bzip2 gives 2.8-fold compression, XDelta gives 5.2-fold compression, .RTPatch gives 10.2-fold compression, Exediff gives 13.7-fold compresion, and BSDiff gives 11.6-fold compression on average. Excluding the Apache 1.2.4 → 1.3.0 upgrade (we note that those two versions share less than half of their source code, so the large patches are not surprising), XDelta gives an average of 5.3-fold compression, .RTPatch gives an average of 11.6-fold compression, Exediff gives 16.8-fold compression, and BSDiff gives 13.0-fold compression.

More important than the above consideration of upgrading between versions ("feature updates"), however, is security updates. These are fundamentally different, in that the source code modifications are typically extremely small — often as small as a single line. We take the i386 build of FreeBSD 4.7-RELEASE and a snapshot of the RELENG_4_7 security branch as a corpus for comparison here. In total, there are 97 modified binaries, with a total size of 36397575 bytes, which bzip2 can compress to 13566233 (a factor of 2.7). Xdelta produces patches totalling 3288540 bytes (a factor of 11.0), and .RTPatch produces patches totalling 749710 bytes (a factor of 47.7), while BSDiff produces patches totalling 621277 bytes, a reduction by a factor of 58.3.

4 Conclusions

We have presented an algorithm for generating binary patches which, applied to two versions of an executable

Program	Uncompressed	Compressed	Xdelta	.RTPatch	Exediff	BSDiff
alto: identical binaries	466944	148024	137	n/a	155	142
alto: gcc -O2 → gcc -O3	466944	148024	78390	34755	20793	33633
alto: changed reg. alloc.	450560	148024	97923	34571	15845	23246
alto: extra printf	466944	148024	50613	7524	6237	6299
agrep: 4.0 → 4.1	262144	114388	14631	5910	3531	6066
glimpse: 4.0 → 4.1	524288	222548	109252	37951	23200	31720
glimpseindex: 4.0 → 4.1	442368	193883	98632	25764	18473	21559
wgconvert: 4.0 → 4.1	368640	157536	75230	20712	15688	15806
agrep: 3.6 → 4.0	262144	114502	80346	58124	41554	53490
glimpse: 3.6 → 4.0	524288	222178	177434	140549	104350	130210
glimpseindex: 3.6 → 4.0	442368	193892	144927	105510	79085	97782
netscape: 3.01 → 3.04	6250496	2396661	1100430	351759	284608	302431
gimp: 0.99.19 → 1.00.00	1646592	642725	463878	301879	185962	284278
iconx: 9.1 → 9.3	548864	233056	139409	51195	38121	44961
gcc: 2.8.0 → 2.8.1	2899968	708301	549250	140284	76072	121371
rcc (lcc): 4.0 → 4.1	811008	221826	889	265	303	289
apache: 1.3.0 → 1.3.1	679936	180708	111421	48033	40460	38278
apache: 1.2.4 → 1.3.0	671744	179369	191920	216867	227233	180981
rcc (lcc): 3.2 → 3.6	434176	155090	84456	34098	22019	33136
Average Compression	100%	35.5%	19.4%	9.8%	7.3%	8.6%

Table 1: Sizes of updates produced by bzip2, Xdelta, .RTPatch, Exediff, and BSDiff

program, consistently generates patches considerably smaller than those produced by the currently preeminent binary patch tools; when applied to security updates, the patches produced are extraordinarily compact.

While its performance does not quite match that of a platform-specific tool, we believe that BSDiff probably attains close to the best possible performance from a platform-independent tool.

5 Acknowledgements

The author would like to sincerely thank Robert Muth for searching through four year old backups to find the corpus used in [BMM99].

The author would also like to acknowledge support from the Commonwealth Scholarship Commission, which is funding his studies at Oxford University.

6 Availability

BSDiff is available under an open source license from `http://www.daemonology.net/bsdiff/`

The files used for the performance comparisons above are available from the author on request.

References

[BMM99] B.S. Baker, U. Manber, and R. Muth, *Compressing Differences of Executable Code*, ACM SIGPLAN Workshop on Compiler Support for System Software, 1999.

[LS99] N.S. Larsson, K. Sadakane, *Faster Suffix Sorting*, LU-CS-TR:99-214, Department of Computer Science, Lund University, 1999.

[Ma00] J.P. MacDonald, *File System Support for Delta Compression*, Master's Thesis, University of California at Berkeley, 2000.

[Ps01] Pocket Soft Inc, *.RTPatch*, `http://www.pocketsoft.com`, 2001.

[Tr99] A. Tridgell, *Efficient Algorithms for Sorting and Synchronization*, Ph.D. Thesis, The Australian National University, 1999.