Special Issue | August 2017 ISSN: 2320-9984 (Online)

International Journal of Modern Engineering & Management Research

Website: www.ijmemr.org

All India Seminar on

Futuristic Trends in Telecommunication Engineering & Telecom Panorama – Fundamentals and Evolving Technology, with Particular Reference to Smart City on 5th – 6th August 2017

Organized by

The Institution of Engineers (India) Jabalpur Local Centre

7. Naive Differences of Executable Code

G. Shashank Rao
B.E.(Final Year),
Jabalpur Engineering College
Jabalpur (M.P.), [INDIA]
Email: Required

Jatin Beohar
Department of Information Technology Engineering,
Jabalpur Engineering College
Jabalpur (M.P.), [INDIA]
Email: Required

Jeevesh Kataria
Department of Information Technology Engineering,
Jabalpur Engineering College
Jabalpur (M.P.), [INDIA]
Email: Required

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 pro-duce 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 na¨ive 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 in-serted. 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 con-tain at least 8 bytes not matching the forward-extension of the previous match (i.e., if the previous match is new[x . . . x + k] = old[y . . . y + k], we look for a match new[x' . . . x' + k'] = old[y' . . . y' + k'] with at least 8 distinct i such that new[x' + i] ≠ 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.

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.

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).

Thus BSDiff can be used which produces patches with a reduction by a factor of approx 58.3.

3. CONCLUSIONS

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

REFERENCES:

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

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

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

[4] Pocket Soft Inc, .*RTPatch*,

[5] http://www.pocketsoft.com 2001.

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