Software Updates: Courgette

Contents

How Courgette works

As I described in *Smaller is faster (and safer too)*, we wrote a new differential compression algorithm for making Google Chrome updates significantly smaller.

We want smaller updates because it *narrows the window of vulnerability*. If the update is a tenth of the size, we can push ten times as many per unit of bandwidth. We have enough users that this means more users will be protected earlier. A secondary benefit is that a smaller update will work better for users who don't have great connectivity.

Rather than push put a whole new 10MB update, we send out a diff that takes the previous version of Google Chrome and generates the new version. We tried several binary diff algorithms and have been using bsdiff up until now. We are big fans of bsdiff - it is small and worked better than anything else we tried.

But bsdiff was still producing diffs that were bigger than we felt were necessary. So we wrote a new diff algorithm that knows more about the kind of data we are pushing - large files containing compiled executables. Here are the sizes in bytes for the recent 190.1->190.4 update on the developer channel:

Full update	10,385,920
bsdiff update	704,512
Courgette update	78,848

The small size in combination with Google Chrome's silent update means we can update as often as necessary to keep users safe.

Compiled code

The problem with compiled applications is that even a small source code change causes a disproportional number of byte level changes. When you add a few lines of code, for example, a range check to prevent a buffer overrun, all the subsequent code gets moved to make room for the new instructions. The compiled code is full of internal references where some instruction or datum contains the address (or offset) of another instruction or datum. It only takes a few source changes before almost all of these internal pointers have a different value, and there are a lot of them - roughly half a million in a program the size of chrome.dll.

The source code does not have this problem because all the entities in the source are *symbolic*. Functions don't get committed to a specific address until very late in the compilation process, during assembly or linking. If we could step backwards a little and make the internal pointers symbolic again, could we get smaller updates?

Courgette uses a primitive disassembler to find the internal pointers. The disassembler splits the program into three parts: a list of the internal pointer's target addresses, all the other bytes, and an 'instruction' sequence that determines how the plain bytes and the pointers need to be interleaved and adjusted to get back the original input. We call this an 'assembly language' because we can run an 'assembler' to process the instructions and emit a sequence of bytes to recover the original file.

The non-pointer part is about 80% of the size of the original program, and because it does not have any pointers mixed in, it tends to be well behaved, having a diff size that is in line with the changes in the source code. Simply converting the program into the assembly language form makes the diff produced by bsdiff about 30% smaller.

We bring the pointers under control by introducing 'labels' for the addresses. The addresses are stored in an array and the list of pointers is replaced by a list of array indexes. The array is a primitive 'symbol table', where the names of the symbols, or 'labels' are the integer indexes into the array. What we get from the symbol table is a degree of freedom in how we express the program. We can move the addresses around in the array provided we make the corresponding changes to the list of indexes.

How do we use this to generate a better diff? With bsdiff we would compute the new file, 'update' from the 'original' like this:

```
server:
   diff = bsdiff(original, update)
   transmit diff

client:
   receive diff
   update = bspatch(original, diff)
```

(The server would pre-compute diff so that it could be transmitted immediately)

Courgette transforms the program into the primitive assembly language and does the diffing at the assembly level:

```
server:
   asm_old = disassemble(original)
   asm_new = disassemble(update)
   asm_new_adjusted = adjust(asm_new, asm_old)
   asm_diff = bsdiff(asm_old, asm_new_adjusted)
   transmit asm_diff

client:
   receive asm_diff
   asm_old = disassemble(original)
   asm_new_adjusted = bspatch(asm_old, asm_diff)
   update = assemble(asm_new_adjusted)
```

The special sauce is the adjust step. Courgette moves the addresses within the asm_new symbol table to minimize the size of asm_diff. Addresses in the two symbol tables are matched on their statistical properties which ensures the index lists have many long common substrings. The matching does not use any heuristics based on the surrounding code or debugging information to align the addresses.

More than one executable, less than an executable

For the above to work, 'assemble' and 'disassemble' have to be strict inverses, and 'original' and 'update' have to be single well-formed executable files. It is much more useful if 'original' and 'update' can contain several executables as well as a lot of non-compiled files like JavaScript and PNG images. For Google Chrome, the 'original' and 'update' are an archive file containing all the files needed to install and run the browser.

We can think of a differential update as a prediction followed by a correction, a kind of guessing game. In its simplest form (just bsdiff / bspatch), the client has only a dumb guess, 'original', so the server sends a binary diff to correct 'original' to the desired answer, 'update'. Now what if the server could pass a hint that could be used to generate a better guess, but we are not sure the guess will be useful? We could insure against losing information by using the original and the guess together as the basis for the diff:

```
server:
   hint = make_hint(original, update)
   guess = make_guess(original, hint)
   diff = bsdiff(concat(original, guess), update)
   transmit hint, diff

client
   receive hint, diff
   guess = make_guess(original, hint)
   update = bspatch(concat(original, guess), diff)
```

This system has some interesting properties. If the guess is the empty string, then we have the same diff as with plain bsdiff. If the guess is perfect, the diff will be tiny, simply a directive to copy the guess.

Between the extremes, the guess could be a perfect subset of 'update'. Then bsdiff will construct a diff that mostly takes material from the perfect prediction and the original to construct the update. This is how Courgette deals with inputs like tar files containing both executable files and other files. The hint is the location of the embedded executables together with the asm_diff for each one.

Once we have this prediction / correction scheme in place we can use it to reduce the amount of work that the client needs to do. Executables often have large regions that do not contain internal pointers, like the resource section which usually contains string tables and various visual elements like icons and bitmaps. The disassembler generates an assembly language program which pretty much says 'here is a big chunk of constant data', where the data is identical to the original file. bsdiff then generates a diff for the constant data. We can get substantially the same effect by omitting the pointer-free regions from the disassembly and letting the final diff do the work.

Source Code

Everyone loves source, so you can find it here:
https://chromium.googlesource.com/chromium/src/courgette/+/master

Summary

Courgette transforms the input into an alternate form where binary diffing is more effective, does the differential compression in the transformed space, and inverts the transform to get the patched output in the original format. With careful choice of the alternate format we can get substantially smaller updates.

We are writing a more detailed paper on Courgette and will post an update when it is ready.

Comments

You do not have permission to add comments.