

UNITED STATES DISTRICT COURT
DISTRICT OF MASSACHUSETTS
EASTERN DIVISION

RED BEND LTD., and
RED BEND SOFTWARE INC.,
Plaintiffs,

v.

GOOGLE INC.,
Defendant.

Civil Action No. 09-cv-11813-DPW

REPLY DECLARATION OF STEPHEN
A. EDWARDS IN SUPPORT OF
PLAINTIFFS’ MOTION FOR A
PRELIMINARY INJUNCTION
ENJOINING GOOGLE’S
INFRINGEMENT

Contents


1	Introduction	5
2	Claim Construction	6
2.1	Data Table	6
2.2	Modified Data Tables and Programs	8
2.3	Invariant References	9
2.4	Executable Program	10
3	The Operation of Courgette	11
3.1	The Windows Portable Executable Format	11
3.2	Disassembly	12
3.3	The Adjustment Step	13
3.4	The Encoded Program	17
3.4.1	Experiments	17
3.5	Delta Encoding and BSDiff	23
4	Courgette Infringes the ’552 Patent	24
4.1	Executable Programs; Compact Different Results; Reference Entries	24

4.2	Modified Old and New Data Tables	26
4.3	“Substantially Each Reference”	26
4.3.1	“Due to Delete/Insert Modifications”	28
4.3.2	“Substantially Each” as an Engineering Term	28
4.3.3	Courgette Considers Most Instructions With References	30
4.3.4	Experiments Suggest Very Few Rel8 References Go Un- detected	35
4.4	“Reflected as Invariant References”	37
4.5	“Generating Said Compact Difference Result”	38
5	Courgette Could Be Applied to Other Instruction Sets	38
6	Wetmore’s Invention	40
6.1	Wetmore’s Vectorization Process	42
6.2	Wetmore’s Patches	43
7	Wetmore Does Not Anticipate the ’552 Patent	43
7.1	Wetmore Does Not Start From An Executable Program	44
7.2	Wetmore Does Not Generate A Compact Difference Result	45
8	The ’552 Patent is Not Obvious	46
8.1	It Would Be Difficult to Transform Wetmore to ’552	46
8.1.1	Leaving a Vectorized Executable as the Final Result	47
8.1.2	Exactly Recovering the New Executable	48
8.2	██	49
8.3	██	50
8.4	The Inventor of BSDiff Respects the ’552 Patent	50
8.5	Website Comments	51
8.6	Red Bend’s vRapidMobile Product Practices the ’552 Patent	51
9	Non-Party Use	51

List of Figures

1	The preferred embodiment of '552 uses symbolic code	7
2	Excerpts from <code>assembly_program.h</code>	14
3	Instructions in an <i>AssemblyProgram</i>	15
4	Courgette's algorithm for locating <code>rel32</code> references	16
5	A comment explaining the purpose of Courgette's adjustment step	17
6	Code that selects an adjustment method	18
7	Courgette's algorithm for representing 32-bit numbers	19
8	Courgette uses this algorithm to delta-encode the <code>Abs32</code> and <code>Rel32</code> address streams	20
9	An example of delta encoding	20
10	Highlights from the definition of <i>EncodedProgram</i>	21
11	A description of the vectors in the <i>EncodedProgram</i> object	22
12	Courgette performs the adjustment step by supplying both the old and new programs to an adjustment method.	23
13	Claim 42 of the '552 Patent	25
14	A visualization of the limitations of "substantially each reference..."	27
15	Intel control transfer instructions with relative references	32

Exhibits

A	Annotated Claim Construction	53
B	Excerpts from the '552 File History	56
C	Excerpt from Appendix A of Vol. 2 of the <i>Intel Architecture Software Developer's Manual</i>	63
D		73
E	Source for old and new programs for the vector experiment	76
F	Annotated disassembly of the old and new versions of the "baz" function	78
G	EncodedPrograms for the old and new programs	80
H	EncodedPrograms for the modified "baz" and "foo" functions	82

I	An EncodedProgram from Walker’s Example	84
J	██	86
K	██	88
L	Results of My Reference Accounting Experiments	92
M	Excerpt from Red Bend’s “Principles of Updating Mobile Firmware Over-the-Air (FOTA)”	95
N	Excerpts from Brian Bershad’s patent, “Discovering code and data in a binary executable program”	97
O	██ ████████████████████	100
P	VisionMobile’s “100 Million Club”	105

I, Stephen A. Edwards, declare as follows:

1 Introduction

1. Counsel for RED BEND LTD. and RED BEND SOFTWARE INC. (“Red Bend”) have asked me to investigate whether Google Inc.’s (“Google”) compression algorithm “Courgette” infringes the claims of Red Bend’s U.S. Patent No. 6,545,552 (the “’552 Patent”) that are currently being asserted in this case. In particular, Red Bend’s counsel has asked me to focus on claims 8, 21, 42, and 55 (“the relevant claims”) of the ’552 Patent that are infringed by Courgette and the use of the Courgette executable and source code by others outside Google.
2. This declaration is intended as a rebuttal to Walker and as a supplement to my earlier declaration in this case (dated November 17, 2009). Since that time, I have been presented with many additional documents, including the expert declaration of Martin G. Walker (“Walker”), Google Inc.’s Opposition to Red Bend’s Motion for a Preliminary Injunction (“the opposition”), U.S. Patent No. 5,481,713 (“Wetmore”), [REDACTED] and numerous documents from Stephen R. Adams (“Adams”), who appears to be Courgette’s main developer, and other internal Google documents. I have also undertaken additional analysis of the Courgette source code and performed additional experiments.
3. I appreciate the detailed analysis Dr. Walker has done of Courgette and the ’552 patent and I broadly agree with his understanding of the operation of both inventions. Yet he and I come to very different conclusions largely because we interpret the claim language differently. This document is largely concerned with why I believe his interpretation is flawed.
4. I continue to find that under what I consider to be the correct definitions of terms (see the claim construction chart, Exhibit A), Courgette infringes on the relevant claims either literally or by equivalents. Furthermore, I disagree with the assertion that the ’552 Patent is invalid. In particular, I find it neither anticipated nor obvious. Below, I discuss the reasoning for my conclusions.

2 Claim Construction

5. Exhibit A is an annotated version of the claim construction from my declaration, which includes notes on how my constructions differ from those in Walker's Exhibit D. I discuss these differences in detail below.

2.1 Data Table

6. I reject Walker's construction that a "data table [...] cannot be source or other symbolic code" [Walker, Exhibit D-1] because it would cause the preferred embodiment in the specification of the '552 Patent to fall outside the scope of that patent. Such constructions make no sense to me, and indeed, counsel has informed me that "[s]uch an interpretation is rarely, if ever, correct."¹
7. Many of the data tables mentioned in the preferred embodiment of the '552 Patent would not be data tables if Walker's construction were adopted. While I agree that the data tables input to the invention in the '552 Patent are not intended to contain high-level language source code (the subject of much prior art), Walker reads too much into a paragraph of the '552 file history and asserts that data tables may not have any "symbolic code." The '552 Patent does not impose this constraint; the preferred embodiment actually modifies certain data tables to contain symbolic entries.
8. In construing data tables as never containing source or symbolic code, Walker relies on a part of the '552 Patent file history, the relevant page of which I attach as Exhibit B (RedBend0000151). Here, the examiner is comparing the application with Okuzumi et al., which deals exclusively with differences between versions of source code. To emphasize that the '552 invention is different, the applicant writes

In extracting diff² between 2 versions of executable files as defined in amended Claim 1, there is no source involved, and neither statements,

¹*Vitronics Corp. v. Conceptronic, Inc.*, 90 F.3d 1576, 1583 (Fed. Cir. 1996)

²"diff" is shorthand for "difference result"

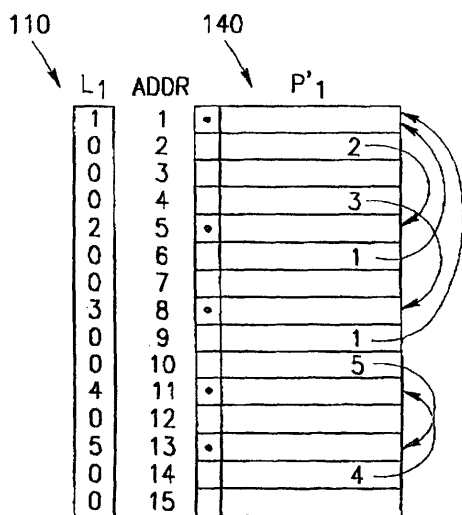


Figure 1: The preferred embodiment of the '552 patent uses symbolic code: this shows label list L_1 and table P'_1 from FIG. 2A of the '552 Patent. P_1 contains symbolic code because the numbers in P_1 are symbols, e.g., “2” refers to entry 5 because “2” appears in location 5 of L_1 .

nor any textual or other symbolic representation of the program even exist. [RedBend0000151, ¶3]

9. The applicant was only trying to convince the examiner that his invention *did not start from* source code, not that it avoided all forms of symbolic representation throughout its operation. Consistent with this, the applicant’s only amendment was to add “executable” to the “program” claims (8 and 21) [Exhibit B: RedBend0000160–3] to satisfy the examiner. Neither the data table claims (claims 42 and 55) nor the specification of the patent—including the glossary—were amended.
10. While Walker and I agree that the starting point for the '552 Patent is executable code, Walker’s construction insists that *all* data tables not contain symbols, which is not true of the preferred embodiment. In the preferred embodiment, the references in intermediate programs/data tables are often symbolic and thus not executable.
11. The table P'_1 created in the preferred embodiment of the '552 Patent employs symbolic references. As shown in Figure 1, the references have been replaced by index numbers (1, 2, etc.). The actual destination of such a reference comes from consulting the label table L_1 , e.g., the first reference in P'_1 is 2. Because entry 5

of L_1 has the value 2, this reference is referring to entry 5 of table P'_1 . Thus, the value 2 symbolizes 5 and *table P'_1 contains symbolic code*.

2.2 Modified Data Tables and Programs

12. Walker construes “modified old and new programs and data tables” as “[a] version of the actual program or data table in its original executable form, with certain portions replaced.” [Walker, Exhibit D-1] but this is vague and imprecise and inconsistent with the '552 patent, which does not require these to be “in original form.”
13. Again, Walker seems to be relying on the paragraph in the '552 file history I mentioned above, but this paragraph is only referring to the initial input to the invention and not to any intermediate steps.
14. His restriction of “in executable form” is vague and imprecise. During his deposition, Walker switched to the phrase “in executable format” and said that this meant it “had to look like an executable program”:

Q. So it's your view that the modified program or the modified data table, depending on which claim we're looking at—in either case, that thing that is the modified program or modified data table has to also be executable?

A. No, I didn't say that. I said it had to be in executable format. Had to look like an executable program, just as the examples we were going through before the lunch break were in executable program. [sic]

Q. So you—it is not your opinion that the modified versions have to actually be executable?

A. That's correct.

Q. But they have to be in executable format?

A. Right. [Walker Deposition, p. 116]

15. I find Walker's explanation that something “had to look like an executable

program” unhelpful since “look” usually refers to something’s appearance and data structures do not “look” like anything until someone chooses to draw them in a particular way.

16. In any case “in executable form” or “in executable format” is so vague that just about anything can be argued to be (or not to be) in executable form, rendering the construction effectively meaningless to a Computer Scientist.

2.3 *Invariant References*

17. Walker’s construction for “invariant references” as “excluded from the difference result” [Walker, Exhibit D–1] is overly narrow, although Courgette would still infringe under this construction.
18. Walker and I agree that one of the central observations in the ’552 Patent is that a large number of differences between two versions of an executable program are due to changes in the values of references due to the insertion or deletion of entries (e.g., code):

[A]pplying a known [...] file difference utility to an old program and a new program normally results in a relatively large amount of data, even if the modifications that were introduced to the old program [...] are very few. The present invention is based on the observation that the relatively large size of the difference result stems from the alterations of reference in reference entries as a result of other newly inserted entries (and/or entries that were deleted). [’552 Patent, 3:27–35]

19. It then explains how “invariant references” are the key trick in ameliorating this problem:

On the basis of this observation, the invention aims at generating a modified old program and a modified new program, wherein the difference in references in corresponding entries in said new and old programs [...] will be reflected as invariant entries in the modified old

and new programs. The net effect is that [these] *invariant reference entries* [...] will not appear in the difference result [...] [’552 Patent, 3:36–47, emphasis mine]

20. However, from this, Walker narrows his construction of “invariant references” to be “[v]alues made the same in the modified old and new programs (or data tables) for corresponding reference entries so that the reference addresses are excluded from the difference result” [Walker, Exhibit D–1].
21. While it is true that the reference addresses for such invariant references are excluded from the difference result in the preferred embodiment of the ’552 Patent, such a narrow construction makes no sense. If the applicant had intended “invariant references” to mean this, I would expect him to have defined them as such in the glossary or added additional language to the claims.
22. Instead, the applicant wrote the above paragraph to *explain* why invariant references are useful in the context of his invention, not to limit the scope of his invention.
23. To emphasize this point further, the applicant explains earlier in the file history that with existing binary difference utilities, “to simply reflect all the changed references when computing a difference, one must include them all,” but that in his invention, “such a need is reduced or eliminated,” [’552 file history, Red-Bend0000150] i.e., the references are not *always* excluded.

2.4 Executable Program

24. I did not feel it was necessary to construe the term “executable program” in my first declaration because I did not believe it would be in dispute, but have since found issue with the construction Walker proposes in his declaration.
25. Walker construes “executable program” to be

A program comprising machine language instructions and corresponding bytes of data used by the program that are ready to be run on a computer, excluding source or other symbolic code. [Walker, Exhibit D–1]

26. The “excluding source” clause is redundant; source never comprises machine language instructions, so to say this is unnecessary.
27. More troubling, executables often do include symbolic code, including those in Microsoft’s Portable Executable file format (see §3.1). Thus to construe “executable program” as “excluding other symbolic code” would contradict the standard definition of “executable” adopted by the industry and contradicts the description in the ’552 patent of “a relocation table attached to executable programs” [’552 Patent, 2:66], which I would consider to contain symbolic code.

3 The Operation of Courgette

28. When I prepared my first declaration, I relied partially on the source, but primarily on some Google webpages³ explaining the operation of Courgette, which indicated Courgette met the limitations of the relevant claims and thus infringed. My assumption was that these descriptions were accurate, and this has proved correct: while analyzing the source code and its operation, I have found nothing that contradicts these web pages. However, because Google raised many objections and Walker’s description of Courgette contains numerous errors and erroneous statements, I describe Courgette’s operation in detail here.

3.1 *The Windows Portable Executable Format*

29. Courgette can accept, but does not require, input programs in Microsoft’s Portable Executable (“PE”) format. When other files are supplied, Courgette reverts to using a variant of the BSDiff algorithm to produce patches.
30. A PE file contains different kinds of data. Perhaps most important are memory images for a program’s code and data, which can often be mapped directly into memory, i.e., without modification. Pietrek⁴ explains “[l]oading [a PE file]

³<http://blog.chromium.org/2009/07/smaller-is-faster-and-safer-too.html>, credited to Stephen Adams, and <http://dev.chromium.org/developers/design-documents/software-updates-courgette>

⁴I take many of these details from Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, Microsoft’s MSDN Magazine, February and March 2002.

executable into memory [...] is primarily a matter of mapping certain ranges of a PE file into the address space.”

31. But a typical PE file contains data in addition to an in-memory image of the code. In particular, it contains multiple sections: the “.text” section contains executable code; “.rdata” holds read-only data, such as strings; “.data” contains initialized data that can be written, such as the initial contents of arrays; and the “.reloc” section contains “base relocations”: instructions for modifying the code should it need to be loaded at a different address.
32. The base relocation section amounts to a list of numbers that identify all the absolute references in the code. The loader, if it needs to relocate the code, goes to the point in the code given by each base relocation entry and, to the 32-bit reference address it finds there, adds an offset equal to the difference between the desired destination of the code and its default destination. The loader ignores the base relocation section if the code is to be loaded at its default address.
33. A PE file contains many symbols.⁵ The section names are strings (e.g., “text”) and the Imports Table consists of a list of imported executables (typically other PE files—dynamically linked libraries or “DLLs”) and for each imported executable, a list of symbols (names) to be imported.

3.2 Disassembly

34. Walker and I agree that “Courgette begins by inputting an old and new program and then parses the programs to create unique data structures apparently developed by Google to represent each program.” [Walker, ¶ 36]. Courgette represents both the old and new programs first as *AssemblyProgram* objects (defined in *assembly_program.h*), which are generated by the *ParseWin32X86PE* method⁶. Figure 2 shows a comment in the Courgette source explaining how this data struc-

⁵Oddly, under Walker’s definition of “executable program,” which “excludes source or other symbolic code,” the Windows Portable Executable file format would not be executable.

⁶*ParseWin32X86PE* [disassembler.cc:404] calls *Disassemble* [disassembler.cc:68], which calls *ParseAbs32Relocs* [disassembler.cc:89], *ParseRel32RelocsFromSections* [disassembler.cc:107], and finally *ParseFile* [disassembler.cc:232].

ture is used.

35. An *AssemblyProgram* object contains a vector of simple instructions, which I list in Figure 3, and two hash tables⁷ linking addresses in the code to *Label* objects, which consist of an address (reference) and an index value. (Figure 2)
36. The disassembly step identifies two types of references: “Abs32,” which are 32-bit absolute memory addresses, and “Rel32,” which are 32-bit numbers which refer to memory addresses relative to the program counter of the processor.
37. Walker and I agree that “Courgette recognizes all absolute addresses.” [Walker, ¶ 34] It does so by interpreting the contents of base relocation section as addresses to every absolute 32-bit reference in the code segment.⁸
38. Courgette uses a heuristic to identify the Rel32 references—see Figure 4. These references are not part of the base relocation table because they do not need to change if the code is relocated; if the code is relocated, the program counter is changed accordingly and the relative references remain valid.
39. This is a heuristic because it may mis-identify a piece of data as a Rel32. Because it does not attempt to determine whether the bytes it is considering are at the beginning of an instruction, it may identify bytes within an instruction that happen to follow these patterns. I feel this is a reasonable heuristic: such mis-identification may make the final difference result a little larger, but it is very difficult to identify which bytes start instructions. [REDACTED]

3.3 The Adjustment Step

40. Once Courgette parses the old and new programs, their indexes are re-arranged⁹ to make them similar using a so-called “adjustment method” (Figure 12). A com-

⁷A hash table is a kind of dictionary: it provides a way to connect “words” to “definitions” (here, addresses and labels).

⁸To obtain the location of each “Abs32” reference, *ParseAbs32Relocs* [disassembler.cc:89] calls *ParseRelocs* [image_info.cc:348], which returns the base relocation section as vector of addresses.

⁹technically permuted

```

// A Label is a symbolic reference to an address. Unlike a conventional
// assembly language, we always know the address. The address will later be
// stored in a table and the Label will be replaced with the index into the
// table.
class Label {
public:
    static const int kNoIndex = -1;
    Label() : rva_(0), index_(kNoIndex) {}
    explicit Label(RVA rva) : rva_(rva), index_(kNoIndex) {}
    RVA rva_; // Address referred to by the label.
    int index_; // Index of address in address table, kNoIndex until assigned.
};

// An AssemblyProgram is the result of disassembling an executable file.
//
// * The disassembler creates labels in the AssemblyProgram and emits
// 'Instructions'.
// * The disassembler then calls DefaultAssignIndexes to assign
// addresses to positions in the address tables.
// * [Optional step]
// * At this point the AssemblyProgram can be converted into an
// EncodedProgram and serialized to an output stream.
// * Later, the EncodedProgram can be deserialized and assembled into
// the original file.
//
// The optional step is to modify the AssemblyProgram. One form of modification
// is to assign indexes in such a way as to make the EncodedProgram for this
// AssemblyProgram look more like the EncodedProgram for some other
// AssemblyProgram. The modification process should call UnassignIndexes, do
// its own assignment, and then call AssignRemainingIndexes to ensure all
// indexes are assigned.
class AssemblyProgram {
    // [method declarations]
private:
    uint64 image_base_; // Desired or mandated base address of image.

    std::vector<Instruction*> instructions_; // All the instructions in program.

    // These are lookup maps to find the label associated with a given address.
    RVAToLabel rel32_labels_;
    RVAToLabel abs32_labels_;
};

```

Figure 2: Excerpts from assembly_program.h (Goog-00028077-79)

Origin <rva> sets the current disassembly address

MakeRelocs generates a base relocation table

Byte <value> emits a single byte literal

Rel32 <label> emits a Rel32-encoded reference from a *Label* object

Abs32 <label> emits an Abs32-encoded reference from a *Label* object

Figure 3: Instructions in an *AssemblyProgram* [assembly_program.cc]

ment in the Courgette code explains the purpose of this: see Figure 5. Once a PE file is parsed, *Disassemble* [disassembler.cc:81] calls *DefaultAssignIndexes* [assembly_program.cc:187] to assign indexes to labels (references in the program) in a simple increasing order.

41. This simple-minded mechanism for assigning index numbers is correct but may produce a larger-than-needed difference result because indexes that should correspond between the two programs may not. Adams recognized this and developed at least two adjustment methods to try to fix this problem.
42. Exhibit G and Exhibit H illustrate the effect of the adjustment step. Compare the index streams of the three programs: those in the old program and new program before adjustment have some similarities, but differ in many places. I highlighted in yellow the index values that differ. After adjustment, however, the streams in the new program are much closer to those in the old program.
43. Exhibit I, which I generated from Walker’s exemplar, shows the effect of the adjustment step even more clearly: every one of the indexes shown in the “Rel32” stream is different in the new program before adjustment; after adjustment, only two still differ. These probably do not correspond.
44. The Courgette source actually contains three adjustment methods: a “null” method, which leave the indexes unchanged [adjustment_method.cc:38]; a “trie” method, which performs some sort of graph matching [adjustment_method.cc:582];

```

// Heuristic discovery of rel32 locations in instruction stream: are the
// next few bytes the start of an instruction containing a rel32
// addressing mode?
const uint8* rel32 = NULL;

if (p + 5 < end_pointer) {
    if (*p == 0xE8 || *p == 0xE9) { // jmp rel32 and call rel32
        rel32 = p + 1;
    }
}
if (p + 6 < end_pointer) {
    if (*p == 0x0F && (*(p+1) & 0xF0) == 0x80) { // Jcc long form
        if (p[1] != 0x8A && p[1] != 0x8B) // JPE/JPO unlikely
            rel32 = p + 2;
    }
}
}

```

Figure 4: Courgette’s algorithm for locating rel32 references in the code. It identifies sixteen patterns: E8, E9, 0F80, ..., 0F89, 0F8C, ..., 0F8F. [disassembler.cc:182–198] These are the opcodes for CALL, JMP, and Jcc, each with 32- or 16-bit relative offsets—see the Intel opcode map (Exhibit C) and Figure 15.

and a “shingle” method [adjustment_method_2.cc:1250], which is the method used by default. In my declaration, I had mis-identified the “trie” method as the one in use. Both the “trie” and “shingle” methods attempt to do the same thing (both adjustment_method.cc and adjustment_method_2.cc contain the comment in Figure 5) and it is clear the code is designed to make it easy to switch among them (Figure 6).




```
// The purpose of adjustment is to assign indexes to Labels of a program 'p' to
// make the sequence of indexes similar to a 'model' program 'm'. Labels
// themselves don't have enough information to do this job, so we work with a
// LabelInfo surrogate for each label.
```

Figure 5: A comment explaining the purpose of Courgette’s adjustment step. [appears twice: `adjustment_method.cc:46` and `adjustment_method_2.cc:187`]

3.4 The Encoded Program

45. Courgette transforms each *AssemblyProgram* object¹⁰ into an *EncodedProgram*,¹¹ which consists of eight vectors. Walker describes this in ¶ 40 of his declaration and gets it largely correct. An *EncodedProgram* object consists primarily of eight vectors that, together, are ultimately used to reconstruct the vector of executable code bytes and the base relocation table. I describe the contents of these vectors in Figure 11.

46. The two address vectors are created by iterating through both hash tables of labels in a *AssemblyProgram* object. For each label (which consists of an address and an index) the procedure inserts the address into the vector at the location given by its index. [`assembly_program.cc:306–307`]

47. To create the other vectors, the *Encode* method iterates through the *AssemblyProgram*’s instructions [`assembly_program.cc:310–342`], which have a nearly one-to-one correspondence to those in an *EncodedProgram*. The one exception: sequences of Byte instructions are coalesced into single Copy instructions. [`encoded_program.cc:185–217`]

3.4.1 Experiments

48. To understand the contents of these vectors, I created two versions of a small

¹⁰The old program is encoded first, then the new program is adjusted, and finally the adjusted new program is encoded [`win32_x86_generator.h:77–99`].

¹¹*Encode* [`assembly_program.cc`] calls *Encode* [`assembly_program.cc:302`], which does the actual conversion.

```

// Factory methods for making adjusters.

// Returns the adjustment method used in production.
static AdjustmentMethod* MakeProductionAdjustmentMethod() {
    return MakeShingleAdjustmentMethod();
}

// Returns and adjustment method that makes no adjustments.
static AdjustmentMethod* MakeNullAdjustmentMethod();

// Returns the original adjustment method.
static AdjustmentMethod* MakeTrieAdjustmentMethod();

// Returns the new shingle tiling adjustment method.
static AdjustmentMethod* MakeShingleAdjustmentMethod();

```

Figure 6: Code that selects an adjustment method. The author made it easy to select among them by changing what *MakeProductionAdjustmentMethod* calls.

program (Exhibit E), compiled them into DLLs using Microsoft’s Visual Studio 2008 Express, and ran Courgette on the resulting PE files. To simplify the example, I made only a small change: the insertion of a single “printf” call in the *baz()* function. Exhibit F shows the result of disassembling the old and new versions of the *baz()* function.¹² The code for the new program makes one additional call to *printf* function.

49. Exhibit G shows an excerpt from the beginning of the encoded program vectors for the old version of this program, the new version of the program before adjustment, and the new version of the program after adjustment.¹³ In these ta-

¹²I used a version of the freely available *objdump* program adapted to work with PE files and manually added comments.

¹³To obtain these, I added a vector printing function patterned after the *AssembleTo* function [encoded_program.cc:356–482]. I added an additional encode step to Courgette to produce the “new program before adjustment” vectors; Courgette normally does not bother to encode this

```

// Write the base-128 digits in little-endian order. All except the last digit
// have the high bit set to indicate more digits.
inline uint8* Varint::Encode32(uint8* destination, uint32 value) {
    while (value >= 128) {
        *(destination++) = value | 128;
        value = value >> 7;
    }
    *(destination++) = value;
    return destination;
}

```

Figure 7: Courgette uses this algorithm to represent 32-bit numbers in fewer bytes. E.g., the 32-bit number “00002318” is represented as “4698,” saving two bytes. [streams.cc:115]

bles, the “RVA” column indicates where the bytes will be placed in memory (it is not a vector); the “OP” column shows abbreviated opcodes; the “Bytes” column shows either bytes from the copy_bytes vector (“..” indicates bytes were elided for clarity), or bytes that would be generated by the Abs32 or Rel32 operations (in parentheses). Thus, the Bytes column lists all the data forming the executable program code. The “A32 Ind” and “R32 Ind” columns show the indexes in the abs32_ix and rel32_ix vectors, and “Origin” shows the addresses in the origins vector. Not shown in these tables for clarity are the two address vectors; their effect appears in parentheses in the “Bytes” column.

50. I also reproduced Walker’s experiment on vectors; Exhibit I shows an excerpt from the beginning of the vectors for the example program in Walker’s declaration. The “Old Program” vectors in Exhibit I correspond to those in Walker’s Exhibit K, although I print the origin numbers in hexadecimal.
51. Perhaps to de-emphasize the effect of the adjustment step, Walker only showed vectors for one version of the program; Courgette actually produces two sets of vectors internally: one for the old program and one for the adjusted new pro-

version of the program; the adjustment step works on its antecedent.

```

// Serializes a vector, using delta coding followed by Varint32 coding.
void WriteU32Delta(const std::vector<uint32>& set, SinkStream* buffer) {
    size_t count = set.size();
    buffer->WriteVarint32(count);
    uint32 prev = 0;
    for (size_t i = 0; i < count; ++i) {
        uint32 current = set[i];
        uint32 delta = current - prev;
        buffer->WriteVarint32(delta);
        prev = current;
    }
}

```

Figure 8: Courgette uses this algorithm to delta-encode the Abs32 and Rel32 address streams. As a result, only the first address ever appears verbatim in the output stream. [encoded_program.cc]

42	43	90	93	103
42	1	47	3	10

Figure 9: An example of delta encoding, which Courgette uses to encode the two address streams in the encoded program. The sequence of numbers above the line is transformed into the sequence below the line by taking the difference between each pair. When the input sequence is closely spaced, the deltas tend to be small. Note that only the initial “42” above the line appears in the delta-encoded sequence.

```

// Binary assembly language operations.
enum OP {
    ORIGIN,    // ORIGIN <rva> - set address for subsequent assembly.
    COPY,      // COPY <count> <bytes> - copy bytes to output.
    COPY1,     // COPY1 <byte> - same as COPY 1 <byte>.
    REL32,     // REL32 <index> - emit rel32 encoded reference to address at
                // address table offset <index>
    ABS32,     // ABS32 <index> - emit abs32 encoded reference to address at
                // address table offset <index>
    MAKE_BASE_RELOCATION_TABLE, // Emit base relocation table blocks.
    OP_LAST
};

// Binary assembly language tables.
uint64 image_base_;
std::vector<RVA> rel32_rva_;
std::vector<RVA> abs32_rva_;
std::vector<OP> ops_;
std::vector<RVA> origins_;
std::vector<int> copy_counts_;
std::vector<uint8> copy_bytes_;
std::vector<uint32> rel32_ix_;
std::vector<uint32> abs32_ix_;

// Table of the addresses containing abs32 relocations; computed during
// assembly, used to generate base relocation table.
std::vector<uint32> abs32_relocs_;

```

Figure 10: Highlights from *EncodedProgram* showing definitions for the opcodes and the eight streams. [encoded_program.h:57–88]

ops : a sequence of instructions for reconstructing the program's code stream

ORIGIN sets the current code generation address to the next address in the *origin* vector

COPY takes the next count from the *copy_counts* vector and copy that many bytes from the *copy_bytes* vector into the code.

COPY1 copies a single byte from the *copy_bytes* vector into the code.

REL32 takes the next index from the *rel32_ix* vector, call it *i*, and uses it to fetch the *i*th address from the *rel32_rva* vector. It subtracts this from the current RVA plus four and writes the four-byte result into the code vector. [encoded_program.cc:414]

ABS32 takes the next index from the *abs32_ix* vector, call it *i*, and uses it to fetch the *i*th address from the *abs32_rva* vector. It adds the current value of *image_base* to this address and writes the four-byte result into the code.

MAKE_BASE_RELOCATION_TABLE reminds Courgette to generate a base relocation table

OP_LAST terminates the reconstruction process

copy_counts : indicates to COPY how many bytes to copy

copy_bytes : holds the bytes for COPY and COPY1 to copy into the code

rel32_ix : from which REL32 takes indexes

abs32_ix : from which ABS32 takes indexes

rel32_rva : from which REL32 takes addresses based on an index. These are unique addresses within the program (i.e., no duplicates).

abs32_rva : from which ABS32 takes addresses based on an index. These are unique addresses within the program (i.e., no duplicates).

origins : indicates to ORIGIN the starting address of later code

Figure 11: A description of the vectors in the *EncodedProgram* object

```

// Adjust is called in win32_x86_generator.h:91
    Status adjust_status = Adjust(*old_program, new_program);

// Adjust is defined in adjustment_method.cc:698
Status Adjust(const AssemblyProgram& model, AssemblyProgram* program) {
    AdjustmentMethod* method = AdjustmentMethod::MakeProductionAdjustmentMethod();
    bool ok = method->Adjust(model, program);
    method->Destroy();
    if (ok)
        return C_OK;
    else
        return C_ADJUSTMENT_FAILED;
}

```

Figure 12: Courgette performs the adjustment step by supplying both the old and new programs to an adjustment method.

gram.¹⁴

3.5 *Delta Encoding and BSDiff*

52. Once Courgette has parsed, adjusted, and encoded the old and new programs, it writes the encoded old and new programs to streams and runs the BSDiff algorithm on them.¹⁵ The Courgette source tree includes a modified copy of the BSDiff source.

53. Along with other data, Courgette packs the eight vectors from the two encoded programs into single stream to be passed to the BSDiff algorithm. It treats each of the eight vectors slightly differently, according to their type. Most are encoded as a count followed by numbers; both the count and following numbers are expressed as a sequence of base-128 digits encoded in bytes whose high bit is set if there are

¹⁴The old program is encoded first [win32_x86_generator.h:77], then the adjusted new program [win32_x86_generator.h:99].

¹⁵The BSDiff algorithm is described in an unpublished paper: Colin Percival, *Naïve differences of executable code*, <http://www.daemonology.net/bsdiff/>, 2003.

more digits. Figure 7 shows the code.

54. Courgette treats the address vectors differently: before using the above-mentioned byte encoding, it delta-encodes the addresses, meaning that it only stores the *differences* between numbers rather than the numbers themselves. Figure 8 shows this code; Figure 9 shows an example.

4 Courgette Infringes the '552 Patent

55. I find that Google's use of Courgette infringes the '552 Patent literally, and if not literally, by equivalents, because it contains all the limitations of the relevant claims (e.g., Claim 42, which I reproduce in Figure 13). That is, I find it generates a compact difference result between data tables that contain reference entries that refer to other entries in the table, generates modified and old and new data tables in which substantially each reference entry in the old table that is different because of delete/insert modifications is invariant, and that it finally generates a compact difference result using the old and new tables.

56. My understanding is that one can infringe a Patent's claims either literally or by equivalents. Literal infringement requires that the accused system or method contain all the limitations of the claim exactly or inherently. My understanding is that a claim is infringed under the doctrine of equivalents if the accused system or method contains only insubstantial changes from the claims' limitations and/or performs substantially the same function, in substantially the same way, to achieve substantially the same result.

57. In this section, I present one way Google's use of Courgette contains the limitations of the claim. My understanding is that infringement requires at least one such finding, but not necessarily only one.

4.1 Executable Programs; Compact Different Results; Reference Entries

58. I do not believe there is a dispute over whether Courgette contains the limitations of the preamble ("A method for generating..."). Nevertheless, I present my reasoning below. First, Walker and I agree that Courgette generates compact difference results between executable programs:

42. A method for generating a compact difference result between an old data table and a new data table; each data table including reference entries that contain reference that refer to other entries in the data table; the method comprising the steps of:

- (a) generating a modified old data table utilizing at least said old data table;
- (b) generating a modified new data table utilizing at least said new data table, said modified old data table and modified new data table have at least the following characteristics:
 - (i) substantially each reference in an entry in said old data table that is different than [the] corresponding entry in said new data table due to delete/insert modifications that form part of the transition between said old data table and new data table are reflected as invariant references in the corresponding entries in said modified old and modified new data tables;
- (c) generating said compact difference result utilizing at least said modified new data table and modified old data table.

Figure 13: (Independent) Claim 42 of the '552 Patent. The other relevant claims differ by using of "program" for "data table" and "system" for "method."

It is undisputed that the Courgette code as it exists today creates compact difference results only¹⁶ for executable programs written for the Intel x86 instruction set and stored in the Windows PE executable file format. [Walker, ¶ 74]

59. Walker also does not appear to dispute that the data tables contain references that refer to other entries in the data tables.

Generally, machine instructions of the type described by Dr. Edwards may include several types of memory references. One type, explicitly discussed by Dr. Edwards is references to other executable instructions. [Walker, ¶ 13]

In particular, Courgette recognizes all absolute addresses but only certain relative addresses that are used as references. [Walker, ¶ 34]

4.2 *Modified Old and New Data Tables*

60. The '552 Patent's limitation on "generating a modified old data table utilizing at least said data table" is deliberately broad and can be found throughout Courgette (just about any data derived from the old PE file would qualify). However, I will say for this analysis that the "AssemblyProgram" version of the old program, which I described in §3.2, is the modified old data table: it is derived from the old PE file.

61. Similarly, for this section, I say the modified new data table is the "AssemblyProgram" version of the new program after adjustment. It is derived by taking the parsed version of the new PE file and adjusting it (Figure 12). Thus, it is generated "utilizing at least said new data table."

4.3 *"Substantially Each Reference"*

62. Walker argues that Courgette does not consider substantially each relevant

¹⁶Actually it is disputed: "only" is incorrect here. When given non-PE files, Courgette compares them using BSDiff. Since BSDiff produces smaller difference results than using techniques available before the '552 patent, it can be said to produce "compact difference results."

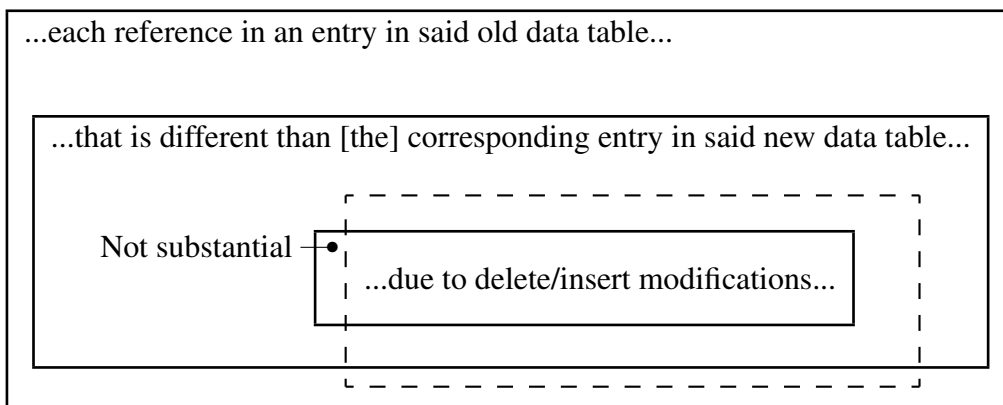


Figure 14: A visualization of the limitations of the “substantially each reference...” sentence of Claim 42 of the ’552 Patent (the other relevant claims have equivalent language). For something to infringe, all except for some not “substantial” number of references that are different “due to delete/insert modifications” must be “reflected as invariant.” The dashed rectangle represents all references “reflected as invariant” in the old and new tables. Provided this is true, the limitations remain met regardless of how many additional references, if any, are reflected (the portion of the dashed rectangle outside the innermost box).

reference as defined in part (b) (i) of the relevant claims. But I find Courgette does meet this limitation for a wide variety of reasons, including my understanding of “substantially” as an engineering term, a detailed analysis of the source code that found the few references Courgette leaves undetected are rarely relevant, and finally experiments that quantify how rarely Courgette leaves relevant references undetected.

63. First, I find it helpful to draw a diagram to illustrate the limitations of this 60-word sentence fragment. Its objective is to precisely specify which references need to be “reflected as invariant” in the modified old and new data tables. I drew the diagram in Figure 14 to illustrate how it does this. First, it is placing a constraint only on references in the old data table; the outermost box represents such references.

64. Next, it says to only consider those “that [are] different than [the] corresponding entry in said new data table,” i.e., if the corresponding entries are not different (the same), they do not need to be “reflected as invariant references.” The second-innermost box represents this set.

65. Finally, it says to only consider those that are different “due to delete/insert modifications that form part of the transition between said old data table and new data table.” The innermost box—a subset of the other two boxes—represents this.

66. Thus, to meet these limitations, Courgette must “reflect as invariant” “substantially each” of the references in the innermost box. To meet these limitations, Courgette must do at least this, but it may do more than this. In particular, it is free to “reflect as invariant” as many references outside the innermost box in Figure 14 as it wishes and still meet the limitations. I drew the dashed rectangle in Figure 14 larger than the innermost box to illustrate this.

4.3.1 “Due to Delete/Insert Modifications”

67. Walker reads “due to delete/insert modifications” and erroneously concludes

[T]he above claim element¹⁷ requires looking for any references that are different due to insert/delete modifications. [Walker, ¶ 47]

68. Had the applicant of the ’552 patent meant an invention must *look* for such references, he would have included this as an explicit limitation. I would have expected much different language in the claims, perhaps a step that “identifies differences due to delete/insert modifications.” The ’552 patent includes no such language in any form, and for good reason. The applicant recognized that reducing differences due to delete/insert modifications was at the center of the invention, *but did not limit his claims to a particular mechanism for identifying them, if at all.*

4.3.2 “Substantially Each” as an Engineering Term

69. Whether Courgette meets the limitations of “substantially each,” hinges in part

¹⁷He is referring to part (b) (i), which begins “substantially each reference...” [Walker, ¶ 45]

on the phrase itself, which the applicant surely chose because it was imprecise.¹⁸ I consider it highly relevant that he chose to use it instead of a precise term such as “all,” “half,” or “90%.” If any of those captured the essence of the invention, no doubt the application would have used them.

70. While in certain algorithms it is necessary to consider exactly all of something to be considered “correct,” other algorithms are free to over- or under-count things without affecting the correctness of the algorithm; the quality of the result may suffer. For example, if a spelling algorithm ignored the first letter of each word, you would consider it simply incorrect. However, if an algorithm designed to produce small diffs produced 101 still-correct bytes instead of the ideal 100-byte result, but did so in 1/10th the time, you would probably consider that acceptable.
71. Software engineers make such tradeoffs all the time. Algorithms usually have separate correctness and quality dimensions; programmers strive to produce correct algorithms that produce the highest-quality results within the available resources such as programmer or processor time. Usually a programmer will reach a point of diminishing returns, where additional effort can still improve the algorithm, but the effort required is not worth the slight improvement it would produce.
72. Differencing algorithms such as that described in the '552 patent have this character: while they ultimately must produce a byte-for-byte identical new program, they have many choices about exactly which diff to produce. Thus, I believe the author of the '552 Patent had in mind something like “consider enough references to produce a good-quality result, but do not waste time trying to identify all of them” when he wrote “substantially each.”
73. The author of Courgette also appears to have understood such tradeoffs. He calls his Rel32 identification algorithm “heuristic”¹⁹ (Figure 4) [REDACTED]
[REDACTED]
[REDACTED]

¹⁸I also consider it relevant that the patent examiner could have objected to such language but did not.

¹⁹In Computer Science, a heuristic produces good results in practice but is not mathematically guaranteed to produce the best possible.

74.

[REDACTED]

4.3.3 Courgette Considers Most Instructions With References

75. As I mentioned above, I do not believe counting the number of instructions in the instruction set whose references Courgette treats as invariant is the right metric for evaluating the limitations of “substantially each,” but Walker does. Besides being simply the wrong thing to do, Walker is sloppy in his analysis and accounting.

76. The Intel architecture’s instruction set is perhaps the largest ever produced. Walker appears to have been intimidated by it:

Q. What did you use to look up these [Intel] instruction types?

A. Let’s see. So there’s—I looked at a listing on the Web of the—that listed the instructions. You know, it started for the—it was a Wikipedia article.

Q. Is there some sort of definitive place one could go to identify what the instructions are in the x86 instruction set?

A. Well, I would have thought that the—Intel would provide a reference, but the Intel reference for this is 300 pages or something²⁰ long, and it’s pretty intractable to come up with a list of instructions, so I used these alternative sources. [Walker deposition, p. 133]

77. I attach as Exhibit C the “opcode map” Intel provides as a guide for decoding instructions. Intel uses variable-length instructions that can range from one to at

²⁰The definitive reference from Intel, the *Intel Architecture Software Developer’s Manual* consists of three volumes. The second, which describes the instruction set, is actually 854 pages.

least fifteen bytes. While a complete discussion of Intel instruction encoding is outside the scope of this document, it is worth understanding this table.

78. The core of most Intel instructions is a one- or two-byte opcode; the opcode map in Exhibit C is the key to interpreting these. For example, to decode an instruction with the opcode “E8,” go to the the row marked “E” on the One-byte Opcode Map and find the column marked “8.” The cell there is labeled “CALL Jv,” which means it is a procedure call that contains a “relative offset to be added to the instruction pointer register” (p. A–2 of Exhibit C) that can be a “[w]ord or doubleword, depending on operand-size attribute” (p. A–3). In most PE files, the operand size is 32 bits by default; the operand size can be set to 16 bits if the opcode is preceded by the operand size prefix “66.”
79. First, Walker and I agree that “Courgette recognizes all absolute addresses” [Walker, ¶ 34] so Courgette considers all data instructions that contain absolute references. No data instruction in the Intel architecture uses an instruction-pointer-relative addressing mode, so Courgette considers all the relevant data instructions.
80. Many control-transfer instructions contain relative references that refer to other instructions and thus are at issue when considering the limitations of the “substantially each reference” language. I list these instructions and how Courgette treats them in Figure 15.
81. From Figure 15, Courgette considers fourteen distinct “Jcc long” instructions. In his declaration Walker miscounts them: “Jcc (conditional jump) includes approximately 30 sub instructions of which Courgette recognizes most but not all” [Walker, footnote 1 in ¶ 34] but corrected his understanding during his deposition with a little help:

Q. Where did the—you get the number approximately 30 sub instructions from in that footnote?

A. I looked up a list of the Jcc instructions and I counted them.

[...]

Q. So [the code in disassembler.cc] would detect 0F80 through [0F8F]?

Instruction	Opcodes	Courgette's Treatment
Jcc short	70–7F	Not considered; rarely change due to insert/delete modifications.
Jcc long	0F80–0F8F	Considers all but JPE (0F8A) and JPO (0F8B) (Figure 4).
CALL	E8	Considered (Figure 4).
JMP near	E9	Considered (Figure 4).
JMP short	EB	Not considered; rarely changes.
LOOP/E/NE short	E0–E2	Not considered; rarely change.
JCXZ short	E3	Not considered; rarely changes.

Figure 15: Control transfer instructions in the Intel Architecture that contain relative references. Courgette considers all but those that rarely change due to insert/delete operations.

A. Yes.

Q. How many opcodes would that be?

A. That would be 16.

Q. Okay. And then it excludes two?

A. Excludes two.

Q. So then it would detect 14 opcodes?

A. Yes, but we don't—there's—that's correct. [Walker deposition, pp. 136,138]

82. Courgette deliberately ignores the JPE and JPO instructions (0F8A and 0F8B) because they occur very rarely in compiler-generated code. Courgette's author acknowledges as much in the comment in Figure 4, and Walker agreed with it during his deposition:

Q. How often does JPE and JPO appear in a typical Windows executable file compiled from C?

A. I think the comment is it's unlikely, and I agree with the comment.
[Walker Deposition, p. 136]

83. JPE and JPO are “conditional branch on (odd or even) parity²¹” and cannot be expressed directly in any high-level language I know of, including C and C++. They are accessible from assembly language, but such usage is rare, generally restricted to libraries, and probably would rarely change “due to delete/insert modifications.”
84. Courgette does ignore short jumps (70–7F, EB), LOOP instructions (E0–E2), and JCXZ (E3), but does so while still reflecting as invariant “substantially each reference...due to delete/insert modifications” because *the offsets in these instructions rarely change due to delete/insert modifications*.
85. The opcodes that Courgette ignores each use a (signed) 8-bit displacement as a reference, meaning that each of these instructions is restricted to only transferring control forward 127 bytes or backward 128 bytes: at most about 127 instructions, and usually far fewer. Compilers generally use such short-distance control transfer instructions for loops and *if* statements, provided the code within these statements consumes fewer than 127 bytes.
86. Such short relative references *only change if an delete/insert operation occurs between the instruction and its target* and thus *do not tend to form a substantial fraction of all the references that change due to delete/insert modifications*. In particular, they do not usually exhibit the behavior identified in the '552 Patent as being the central problem:

In fact, insertion of only one new entry may result in the plurality of altered reference entries which will naturally be reflected in the

²¹The parity of a machine word is the number of 1's it contains. Parity tests are sometimes used to verify the integrity of data.

difference result and obviously will inflate its volume. [’552 patent, 2:6–9]

87. Only when a group of short relative reference entries send control *across* the same insertion/deletion point can a single such point alter a plurality of references. While this can happen, the number of relative references that are affected tends to be small (in normal code, I would expect at most ten short relative references to change due to a single insert/delete operation; the pathological worst case²² is only about 128) compared to the total number of references in a large program (in the 100s of thousands).
88. In other words, short relative references behave very differently than absolute references or even long-range relative references. A single insert/delete can affect as many as all the absolute or long-range relative references, half of all of them in the program on average (this is certainly substantial); a single insert/delete usually changes ten or fewer short-range relative references (which would not be a substantial portion of the 100s of thousands of references in a large program).²³
89. I believe all of this explains why Adams did not consider Rel8 references in his code: he did not feel it was worth the effort to identify and handle them. It would have been technically straightforward for him to introduce an additional “Rel8” stream in the encoded program and identify them using a heuristic discovery procedure similar to his algorithm for identifying Rel32 references. I can only assume he understood this was possible yet not worth his time because the improvement would not have been substantial.

²²This occurs when you insert a byte in the middle of a (nonsensical) sequence of short branch instructions that all send control across the insertion point. All of these would have to change, but there are at most 128 of them because that all that could “fit” within the space of an 8-bit displacement. I would never expect to see such a code sequence in a real program.

²³In fact, the impact of short relative references relative to long-range references shrinks as the program grows because delete/insert operations can affect proportionately fewer of the short relative references.

4.3.4 Experiments Suggest Very Few Rel8 References Go Undetected

90. As described above, I believe the (Rel8) relative references Courgette does not detect do not constitute a substantial portion of the references that are different “due to delete/insert modifications.” I performed some experiments to try to quantify this further. Exhibit L lists my results. Courgette routinely leaves undetected less than 1%—often much less—of all references that changed due to delete/insert modifications—not a substantial number. Below, I describe these experiments.
91. For a particular run of Courgette, I wanted to compare the number of corresponding Rel8 references that changed due to delete/insert modifications that Courgette leaves undetected to the total number of corresponding references that changed due to delete/insert operations. I decided to go looking for this information in the patch stream generated by BSDiff since every change to the program must be accounted for there. Furthermore, Courgette feeds programs to BSDiff in the eight-vector form (§3.4), so I interpreted the BSDiff stream as such an encoded program.
92. First, I went looking for changed references that Courgette leaves undetected. We know that Courgette detects all the Abs32 references and most of the Rel32 references (§4.3.3), so Courgette only leaves the Rel8 references undetected. These would not be treated as invariant and hence will appear in the difference result. I went looking for data that could be opcodes for control-flow instructions with short (one-byte or “rel8”) relative references. Courgette ignores these opcodes (Figure 15).
93. To the portion of BSDiff that reconstructs the instruction stream, I added code that looks for modified rel8 references. Specifically, in each change region²⁴ in the new program, I look for an opcode with a rel8 offset (70–7F, E0–E2, and EB)

²⁴BSDiff’s difference results consists of a series of instructions about how to change the old file into the new. Each instruction describes a “change region” that copies a series of bytes from the old file into the new file with modification.

that was also there in the old program²⁵ followed by a changed (rel8) reference. This is a heuristic algorithm similar to that used in Courgette: if I find a byte with the right value, I assume it is an opcode. Of course the byte might be data that just happens to look like an opcode, so my accounting is generous—I will always report more changed rel8 references than there actually are; never fewer.

94. Next, I tried to determine the total number of corresponding references that are different due to delete/insert modifications. Here, I collected statistics by observing how BSDiff reconstructs the index streams. I wrote code that counted, for each index BSDiff corrects, whether it was (1) copied from elsewhere in the index vector, and therefore certainly a corresponding entry found by the adjust step; or (2) modified by BSDiff, suggesting that while it probably corresponded, Courgette’s adjust step may not have been able to correct it.

95. We are only interested in corresponding references that are different, so I added code that counted all the indexes whose associated address did not change between the old and new program. None of the indexes’ addresses would change if Courgette were fed identical old and new programs; based on the observations in the ’552 patent, we expect small changes will lead to very few indexes with unchanged addresses because even a single delete/insert modification can force a change in many references.

96. I combine these four numbers to estimate the total number of references that Courgette should reflect as invariant: the sum of corresponding indexes, those indexes that probably correspond but changed for some reason, and the rel8 references that changed. From this, I subtract the indexes with unchanged addresses, which obviously cannot be due to delete/insert modification.

97. Exhibit L shows the result of my experiments. In addition to comparing my tiny “reva” program to “revb” Exhibit E and “setup1.exe” to “setup2.exe” in the Courgette distribution, I also downloaded twelve incremental releases of the Chrome browser, extracted the main “Chrome.dll” from them, and ran compar-

²⁵If this opcode were added to the new program, there was no corresponding instruction in the old program with a reference that could have changed so I do not count it.

isons between various pairs.

98. The two “Results” columns list the total number of relevant references and what percentage of relevant rel8 references Courgette did not detect. This fraction is generally quite small (1% or lower for successive Chrome releases), but larger for the bigger jumps, which necessarily involve more changes. From this, I again conclude that the fraction of undetected relevant (rel8) references is not substantial.²⁶

4.4 “Reflected as Invariant References”

99. The claim insists that “substantially each reference...[is] reflected as [an] invariant reference.” Courgette’s adjustment step is designed to do exactly this and thus meets the limitations of the claim.
100. Courgette parses PE files and disassembles them into *AssemblyProgram* objects, each of which contains an instruction stream that contains symbolic references (§3.2). Each of these references (a *Label* object; Figure 2) consists of an index and the address to which the index refers. Courgette’s adjustment step then permutes the indexes of the new program to make them similar to the old one. The index vectors from Walker’s example program (Exhibit I) illustrate this clearly: before adjustment, all of the indexes at the beginning of the Rel32 vector do not match those in the old program; after adjustment, only two different ones remain in this part of the vector.
101. Courgette performs the adjustment step to make the index vectors similar so that when they are later run through BSDiff, their differences are smaller. When they are made invariant, their values do not appear in BSDiff’s output stream, i.e., “[t]he net effect is that [these] invariant reference entries [...] will not appear in

²⁶I believe the test cases I ran for this series of experiments are representative: I would expect Courgette to behave similarly when given most other programs and hence will still infringe either literally or by equivalents. Specifically, Courgette will still perform substantially the same function (i.e., generate a compact difference result starting from old and new executable programs) in substantially the same way (by generating modified old and new data table with the relevant references reflected as invariant) to obtain substantially the same result (a compact difference result.)

the difference result [...]” [’552 Patent, 3:36–47]

102. As I mentioned in §2.3, Walker uses this language from the ’552 patent to erroneously conclude that the reference addresses from invariant references must be excluded from the difference result [Walker, Exhibit D–1]. I disagree with this construction but even if the court adopts it, Courgette would still reflect substantially each reference as invariant because *most addresses do not appear in Courgette’s difference result*. This is because the address vectors are delta-encoded §3.5. In particular, only the first address in each vector could appear; the rest are represented as differences. Adams understood this (Exhibit D).

4.5 “Generating Said Compact Difference Result”

103. Courgette meets the final limitation of Claim 42 because it generates “said compact difference result utilizing at least said modified new program and modified old program.” It encodes both the modified old program and modified (adjusted) new program (*AssemblyProgram* objects—see §4.2), transforms them into streams, then runs BSDiff on the result (§3.5).
104. BSDiff generates compact difference results because it produces smaller results than existing tools at the time of the ’552 patent.²⁷

5 Courgette Could Be Applied to Other Instruction Sets

105. In my earlier declaration, I stated that the code in Courgette “is written such that it is easily adaptable to processing executable files for other platforms, such as those found in mobile devices.” [Edwards Decl, ¶ 24]. Walker called that statement “wrong” [Walker, ¶ 74] because Courgette uses the base relocation table of the PE format and heuristics for identifying certain Intel opcodes, but neither of these are especially unique. Other executable formats contain information like the base relocation table and in any case, its contents can be guessed at much like Courgette guesses the location of the Rel32 offsets currently. Furthermore, it is easy to apply similar heuristics to other instruction sets. Brian Bershad, Adams’s

²⁷Colin Percival, *Naïve differences of executable code*, <http://www.daemonology.net/bsdiff/>, 2003.

boss, holds a patent that points out analyzing code for reduced instruction set computers, such as the ARM processor common in mobile devices, is quite a bit easier than analyzing the Intel instruction set. (Exhibit N, 1:64–2:1)

106. [REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]

[REDACTED]
[REDACTED]

107. [REDACTED]

[REDACTED]
[REDACTED]

108. [REDACTED]
[REDACTED]

[REDACTED]
[REDACTED]

109. [REDACTED]
[REDACTED] [REDACTED] [REDACTED]

[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]

110. I agree with [REDACTED] the above. In fact it is easy for a person of ordinary skill

[REDACTED]
[REDACTED]
[REDACTED]

in the art, given any processor's instruction set, to come up with a fairly good set of reference entries that should be made invariant because the set of reference entries does not have to be precise: missing a few or including things that are not references merely affects the quality of the result, not its correctness. In particular, one simple technique, which is nearly instruction-set-agnostic, is to treat as a reference *any* group of bytes that could conceivably be a reference. This will, of course, over-count references, but will probably miss few and still lead to compact difference results.

6 Wetmore's Invention

111. Wetmore³⁰ describes "A method and apparatus for generating patching resources in an information processing system having operating instructions on a Read Only Memory Device." [Wetmore, abstract] This is already a substantial departure from the '552 Patent, which is "[a] method for generating a compact difference result between an old program and a new program." ['552 Patent, abstract] Red Bend goes to some length to explain their product, which practices the '552 Patent, does not perform "patching." (see Exhibit M) So it is not at all clear that someone considering how to generate compact difference results between program revisions would even think to consult Wetmore.

112. Wetmore's focus on Read-Only Memory³¹ ("ROM") is an enormous departure from the '552 patent. The contents of ROM, as the name suggests, can only be written once, typically when the chip is manufactured, but read many times. ROM holds its contents indefinitely and is therefore available immediately after a computer is powered on. Wetmore explains he is targeting systems that "provide as much operating system functionality into ROM as possible. This has the desired effect of freeing up RAM for application programs. This approach is used for the operating system for the Apple Macintosh family of computers." [Wetmore 1:29–34] As of 2010, the role of system ROM has been largely supplanted by Flash

³⁰Russ Wetmore et al. "Method and Apparatus for Patching Code Residing on a Read Only Memory Device," U.S. Patent No. 5,481,713, Jan. 2, 1996.

³¹The Wetmore patent mentions ROM over 100 times,

memory, which is non-volatile³² like ROM, but is capable of changing its own contents many times.

113. Wetmore’s independent claims all make it clear that his invention is limited to a ROM implementation:

1. A method for applying patches to code residing on a Read Only Memory (ROM) device [...]

7. In an information processing system having a random access memory (RAM) device and a first version of operating software stored in a vectorized Read Only Memory (ROM) device [...]

11. A method for generating patch resource files for a plurality of previous versions of operating software based on a new version of operating software, wherein said new version of operating software is stored in a storage device, wherein each of said plurality of previous versions of operating software is stored in a vectorized Read Only Memory (ROM) device [...]

16. A method for updating a first operating software stored in a read only memory (ROM) device [...] [Wetmore 12:22–15:58]

114. Wetmore even defines “patching” in his context with respect to ROM, a concept absent from the ’552 patent:

The term patching is used to describe the process for creating and installing patches to the ROM that add functionality or fix bugs. [Wetmore 10:9–12]

115. Wetmore does mention that his invention would apply to systems that use Flash memory, but only if the code therein was “static”:

³²preserves its contents even when powered down

It should be noted that the fact that this code is to reside on ROM is not meant to limit the scope of the present invention. Any system that utilizes “static” code on, for example “FLASH” memory magnetic or optical disk media or other storage devices, could be utilized without causing departure from the spirit and scope of the present invention. [Wetmore 5:3–9]

116. This seems like a contradiction to me since Flash is not static by design and suggests to me the applicant was unfamiliar with Flash memory technology, which is consistent.³³ In 1993, when Wetmore was filed, Flash memory technology was relatively new, uncommon, and fairly expensive.

117. Because ROM cannot be altered once it leaves the factory, there are stringent constraints on how it can be “patched.” Wetmore’s solution is to “vectorize” the code in ROM *before any patches are released* and insist that any code that wishes to use ROM routines (including any code in the ROM itself) first consult a “vector table” to determine if a patched version of the resource that should be used instead was placed in normal memory.³⁴

6.1 *Wetmore’s Vectorization Process*

118. Wetmore’s vectorization process does not start with an executable. Instead, he makes it clear that he starts with symbolic information from a group of object files:

First, the source files are compiled (in the case of a high level language) or assembled (in the case of assembler language source) to create object files [...]. The object files are then vectorized to create vectorized object files [...]. It is significant that only the object files are modified. The source files are not touched. Object files contain a series of defined records, each one containing specific items such as

³³He also wrote “FLASH” in all caps and in quotes, which is very non-standard usage since it never was an acronym.

³⁴i.e., Random Access Memory (“RAM”), which can be altered.

the object code for a routine, the name of a routine, external reference from one routine to another, or comments. In object files the references to other routines have not been resolved. Therefore object files are an ideal place to alter the code without modifying the source code files. [Wetmore 6:48–60]

119. Only after this “vectorization” does Wetmore teach creating an executable:

The object files are then linked together to create the final binary values which will be written to ROM [...]. This is performed through a traditional linkage editing step. Finally, after the object files have been “linked” together to create the final binaries, the ROM image is created [...] [Wetmore 6:63–67]

6.2 *Wetmore’s Patches*

120. Wetmore writes,

A patch resource is comprised of a plurality of entries, each of which defines a vector table address, an offset into the vector table and the routine to be inserted. By comparing routines between the ROM versions, routines which are different or new are identified. These routines will become patch resource entries. [Wetmore, abstract]

121. This is another enormous departure from the ’552 patent, whose difference results are at the granularity of entries in a data table (e.g., bytes), and not routines, which are rarely under, say, 100 bytes.

7 **Wetmore Does Not Anticipate the ’552 Patent**

122. I am informed by counsel for Red Bend that for a claimed invention to be anticipated, each claim limitation must expressly or inherently be disclosed in a single prior art reference arranged as in the claim.³⁵ “The prior art reference must

³⁵*Net Moneyin, Inc. v. Verisign, Inc.*, 545 F.3d 1359, 1371 (Fed. Cir. 2008) (citations omitted)

clearly and unequivocally disclose the claimed invention or direct those skilled in the art to the invention without any need for picking, choosing, and combining various disclosures not directly related to each other by the teachings of the cited reference.”³⁶ A difference between the prior art reference and claimed invention, however slight, invokes the question of obviousness, not anticipation.³⁷

123. I am further informed by counsel that “anticipation by inherent disclosure is appropriate only when the reference discloses prior art that must *necessarily* include the unstated limitation.”³⁸

7.1 *Wetmore Does Not Start From An Executable Program*

124. I find Wetmore does not anticipate Claim 8³⁹ of the ’552 Patent because I cannot find the limitations of its preamble in Wetmore, i.e.,

A method for generating a compact difference result between an old executable program and a new executable program [’552 Patent, 16:19–21]

125. Wetmore is clear that his preferred embodiment utilizes “object files,” which are not executable. Walker agreed with this during his deposition:⁴⁰

Q. Okay. Well, what does [Wetmore] disclose is in the content of an object file in the section that we’re referring to now?

A. “Object files contain a series of defined records, each one containing specific items such as [the] object code for a routine, the name of a

³⁶Id., citing *In re Arkley*, 455 F.2d 586, 587

³⁷*Net Moneyin* at 1371

³⁸*Transclean Corp. v. Bridgewood Services Inc.*, 290 F.3d 1364, 1373 (*Fed. Cir.* 2002) (emphasis in original) (citation omitted)

³⁹A similar argument applies to Claim 21.

⁴⁰Walker had misconstrued my equating of “object code” with “executable” in my declaration as extending to “object files.” [Walker, ¶ 66] He appears to correct himself here and now agrees that object files are not the same as executables.

routine, external reference from one routine to another, or comments.”

[Wetmore, 6:54–57]

Q. Would such an object file as described there be executable?

A. This file would not be executable until a linker touched it and loaded it into memory. [Walker Deposition, p. 209]

126. In fact, Wetmore’s invention relies on being able to enter into the compilation process *before* an executable is generated so that it can understand and modify the (non-executable) object files. To emphasize this point, he writes “object files are an ideal place to alter the code without modifying the source code files” [Wetmore 6:59–60]

127. Walker, during his deposition, claimed Wetmore suggested vectorizing an executable, in particular a static program that is to be installed in ROM,⁴¹ but I find this reading inconsistent because Wetmore makes it very clear that his vectorization process starts from object files. I discussed this in more detail in §6.1.

7.2 *Wetmore Does Not Generate A Compact Difference Result*

128. I find Wetmore does not anticipate any of the relevant claims of the ’552 Patent because they each speak of a “method” or “system” for “generating a compact difference result,” and Wetmore does not generate a compact difference result. This is because we have construed, and Google agrees, that the proper construction for “compact difference result” is “a difference result of *a smaller size as compared to a conventional difference result obtained by using techniques in existence prior to the invention of the patent-in-suit*” and Wetmore’s invention tends to generate much larger patches than existing difference tools would. Thus Wetmore creates a difference result, but it is not compact.

129. The key reason Wetmore’s patches are larger than those from existing difference tools is that Wetmore’s patches are performed *at the granularity of routines* whereas existing difference tools, such as Coppieters⁴² work at the granularity of

⁴¹He cited Wetmore, 5:18–21.

⁴²Kris Coppieters, “A Cross-Platform Binary Diff,” Dr. Dobb’s Journal, May 1995.

bytes. This means that if a single byte of a routine changes, Wetmore considers the entire routine different and *includes all the code of that routine* in the patch. This is not sloppiness on the part of Wetmore; the restrictions comes from the use of ROM: once control is passed to a ROM routine, it cannot be made to jump out from an arbitrary point to execute modified code. Instead, any routine with any change—no matter how small that change—must be placed in RAM and thus form part of the difference result.

130. By contrast, Coppieters strives to work with “chunks” in the 20–80 byte range, which he tries to match between old and new files. The generated diff file consists of a series of instructions that either copy chunks from the original file (perhaps to a new location) or insert new chunks included in the diff.⁴³ Since such chunks are generally much smaller than routines (which I expect to be in the 100s or 1000s of bytes), modifying a single byte in a routine would result in a diff roughly the size of a chunk; Wetmore’s patch would include the whole routine.

131. Thus, Wetmore neither describes a diff nor the creation of one that would inherently be smaller than Coppieters, and so does not generate a “compact difference result” as we have construed the term.

8 The ’552 Patent is Not Obvious

132. I find the ’552 Patent would not have been obvious under 35 U.S.C. § 103 because it would not have been obvious at the time to a person of ordinary skill in the art. There are many arguments against it being obvious to such a person, some of which I list below.

8.1 It Would Be Difficult to Transform Wetmore to ’552

133. One consideration in deciding whether a new thing is non-obvious relative to an older thing is how challenging it would be to transform the old into the new. Below, I consider transforming Wetmore into the claimed inventions of the ’552 Patent and conclude it would actually be very challenging and not at all obvious.

⁴³From the source code for BinDiff: <ftp://66.77.27.238/sourcecode/ddj/1995/9505.zip>

134. Wetmore's invention targets ROM-based systems and inherits many limitations from that: its reliance on modifying the program before it is made executable, the coarseness of operating on a per-routine basis, the impossibility of inserting changes into the ROM code, and the difficulty of sending control back and forth between ROM and RAM.

135. It would not be obvious to a person of ordinary skill in the art how to adapt an algorithm designed to work in such an environment to one in which the program was assumed to be in memory that could easily be changed. Even less obvious would be how to transform it to operate efficiently at the instruction level instead of the routine level.⁴⁴

136. It is clear Wetmore targets his invention to a setting with ROM; the the '552 patent does not mention ROM and *would not work in a context with ROM*. The '552 patent's assumption about how to apply the difference result it produces inherently assumes that the patched program resides in memory that can be modified. I feel making this jump would also be non-obvious.

137. Since the '552 Patent starts from an executable and Wetmore requires object files with precise symbolic, reference information, any technique would have to start by transforming an executable back to object files to apply Wetmore.

8.1.1 Leaving a Vectorized Executable as the Final Result

138. A vectorized program is executable, so one approach to transforming Wetmore into the inventions of the '552 Patent would be to leave the vectorized program running on the client. This technique would involve

1. *Exactly* disassembling the old and new executables since any mistakes made here would result in a non-operational final vectorized program. This is extremely difficult because it may not make mistakes like Courgette's heuristic disassembler.⁴⁵

⁴⁴Wetmore's preferred embodiment includes a very simple matching algorithm that simply decides if routines are absolutely identical or have changed. The '552 patent is much more subtle.

⁴⁵There is an extensive literature on trying to perform such "reverse compilation." It is costly and can only be performed under certain restrictive conditions.

2. Vectorize the old and new programs' object files, this time using not just routine entry points but substantially all references. This would result in an enormous executable with a very large vector table to match as most programs consist of only a handful of sequential instructions followed by a control-transfer instruction, which would lead to a large number of references that would need to be vectorized.
 3. Perform a diff on the old and new vectorized programs as Wetmore suggests. The result may not be especially compact since it would include the new vector table.
139. On the client side, the vectorized executable would be enormous and slow because every label would be vectorized. Wetmore teaches away from this, observing that vectored code has potential efficiency issues and should be used carefully (Wetmore 5:63–64)
140. Furthermore, Wetmore teaches simply adding his patches to RAM since ROM cannot be changed, so the memory on the client side will hold both the old version of the program as well as the patches.

8.1.2 *Exactly Recovering the New Executable*

141. An alternative to having a large, slow executable on the client would be to exactly reconstruct the new executable program on the client side. This, too, is challenging, for it would involve
1. Approximately disassembling the old and new executables to turn them into object files. Making mistakes here is not critical since they can be corrected later, although it will affect the patch size.
 2. Vectorizing the old and new programs from their object files, again using substantially all references.
 3. Performing a diff between the vectorized old and new programs.
 4. Applying the resulting diff to the old vectorized executable.

5. Undoing the vectorization from this patched old vectorized executable, which will almost certainly deviate from the new executable. Wetmore says nothing about how to undo vectorization; I consider this to be not at all obvious.
6. Generating a second diff between the unvectorized patched old executable and the new executable to capture any mistakes made in the whole process.
7. On the client side, disassembling the old executable
8. Vectorizing the old executable
9. Applying the first diff to the old vectorized executable
10. Unvectorizing the patched, vectorized old executable
11. Applying the second patch to it to obtain, finally the new executable on the client.

8.2 [REDACTED]

142. [REDACTED]
[REDACTED]
[REDACTED] [REDACTED] [REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]

143. [REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]

8.3 [REDACTED]

144. [REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]

145. [REDACTED]
[REDACTED]
[REDACTED]
[REDACTED]

8.4 *The Inventor of BSDiff Respects the '552 Patent*

146. Colin Percival, inventor of the BSDiff algorithm, briefly mentions the '552 Patent in his thesis⁴⁹ and treats it with respect:

What remains is the problem of encoding the second-order changes identified. As noted before, platform-specific information can make this task much easier; one approach [the '552 Patent] involving complete disassembly of the executables into assembly language removes these second-order differences completely, since upon re-assembly, the new addresses are used. [Percival Thesis, p. 39]

147. Percival does not treat every patent with respect, so if he thought it was obvious, I doubt he would have written the above. By contrast, on the page in his bibliography with the '552 Patent, Percival writes about a different patent,

We are surprised that this patent [US 6,496,974] was granted, given that it does not appear to cover any methods not previously published in [Hunt et al., "Delta algorithms: an empirical analysis," ACM

[REDACTED]
⁴⁹Colin Percival, *Matching with Mismatches and Assorted Applications*, University of Oxford, 2006.

Trans. on Software Engineering and Methodology, 7(2):192–214, 1998. [Percival Thesis, p. 73]

8.5 Website Comments

148. I have seen many Internet postings from people outside Google praising the release of Courgette in open-source form (e.g., GRB–4020). If it were obvious, people would ignore it (most released open-source software does not garner praise).

8.6 Red Bend’s vRapidMobile Product Practices the ’552 Patent

149. I have been informed by counsel that one factor to consider in the analysis of “obviousness” is whether the patented technology is commercially successful. My understanding is that Red Bend’s vRapidMobile software practices one or more claims of the ’552 Patent and is a commercial success (Exhibit P).

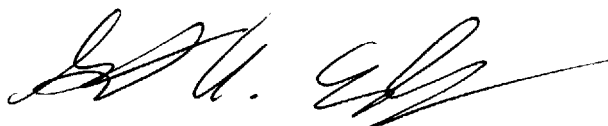
150. I have examined the source code of the vRapidMobile (rev. 6.1.7.18) software and found that meets at least the limitations of Claim 42 of the ’552 Patent. In particular, it begins with old and new data tables containing references (two executable programs), reflects at least “substantially each” of these as invariant references, and uses these two tables to generate a compact difference result. I have also been informed that this code has been used more-or-less unchanged in many earlier revisions of the software, including those when the product was named “vCurrentMobile” and thus users of these products have been practicing the relevant claims of the ’552 Patent as well.

9 Non-Party Use

151. Since my first declaration, it has come to my attention that numerous non-parties have downloaded and made use of Courgette. I have seen clear evidence that a developer of the “splayer” media player downloaded Courgette, compiled it, ran it to conduct experiments, incorporated the Courgette source into the source for splayer (RedBend–00011078–82), enabled the splayer executable to run Courgette (RedBend–11094–97), and distributed executables containing Courgette code, effectively enabling other non-parties to practice the claims of the ’552 Patent.

152. I have seen strong evidence that another party had downloaded and used Courgette to run experiments comparing the operation of Courgette to “deltarpm,” another binary diff tool (RedBend-0010682). In doing so, this party would have had to practice at least one claim of the '552 Patent.

I declare under penalty of perjury under the laws of the United States of America that the foregoing is true and correct and that this declaration was executed on March 24, 2010 in New York, New York by

A handwritten signature in black ink, appearing to read "S.A. Edwards", with a long horizontal flourish extending to the right.

Stephen A. Edwards

Exhibit A

Annotated Claim Construction

Red Bend v. Google
U.S. Patent No. 6,546,552 Claim Construction

Term	Definition	Notes
Data Table	A table of entries, where an entry is an addressable unit within the data table. An executable program is one example of a data table.	§2.1 and '552 patent glossary: 2:33–36, 2:61–63. Walker construes a data table to “not be other symbolic code” but the “modified old data table” of the '552 patent is symbolic.
Address	A number which is uniquely assigned to a single entry by which that entry is accessed.	'552 patent 2:37–38. Walker contends no definition is necessary.
Compact difference result	A difference result of a smaller size as compared to a conventional difference result obtained by using techniques in existence prior to the invention of the patent-in-suit.	'552 patent 3:30–46, 14:5–14. Walker accepts this definition.
Old data table	A data table (or portion of a data table) that is to be updated.	Glossary of '552 patent 2:51–54, 2:61–63
Reference	Part of the data appearing in an entry in the data table which is used to refer to some other entry from the same data table. A reference can be either an address or a number used to compute an address.	Glossary of '552 patent 2:42–45. Walker’s construction for this term also improperly included the '552 patent’s definition of “reference entry.”
Reference entry	An addressable unit containing data that includes a reference.	Glossary of '552 patent 2:35–36, 2:46–47. Walker’s construction deviates from the glossary and does not apply to claims referring to data tables.

Modified old data table	A data table generated using the old data table.	§2.2 and '552 patent claims 42, 55. Walker construes both old and new modified data tables as “a version of the actual program or data table in its original executable form,” yet this cannot be correct because the modified old and new data tables in the preferred embodiment of the '552 patent are not in executable form.
Modified new data table	A data table generated using the new data table.	'552 patent claims. See above.
Invariant	Unvarying, invariable, constant.	<i>Random House Webster's Unabridged Dictionary</i> (2nd ed. 1998)
Invariant references	References that are the same.	§2.3 and '552 patent 10:10–15. From this text, Walker also construes these to mean “so that the reference addresses are excluded from the difference result,” but again, this is not true in the preferred embodiment of the '552 patent.
Executable program	A program comprising machine language instructions and corresponding bytes of data used by the program that are ready to be run on a computer.	This is most of Walker's definition. §2.4

Exhibit B

Excerpts from the '552 File History

files. A major problem arises when applying these methods to executable program files aiming to extract the difference at the byte-level, regarding the files as a list of data bytes rather than list of text lines. The problem arises from the fact that executable programs are generated from sources and in that process many references are inserted into these executable files. These references do not refer symbolically to other location of the program, as may be the case in source files, but they refer to addresses - sequential locations in the program file. When re-generating an executable file from a modified source file, not only the actual changes are being reflected in the executable file, also the majority if not all the inserted references are modified as well since their referred addresses have changed their location in the executable file as a result of the actual changes. This phenomenon leads to a significant increase in the amount of differences. This problem is discussed in the "Background of the Invention" section of the specification, Page 2, lines 11 to 18, and exemplified on Page 2, lines 18 to 30.

Consider, for example, an extreme example where a change in the first source of line may lead to actual change of some first executable file, in which few bytes were added but also all references must change since they refer to locations that now have been moved farther for the amount of bytes added at the beginning. To simply reflect all the changed references when computing a difference, one must include them all. In accordance with the present application such a need is reduced or eliminated, and what is required, is just to send the first few modified bytes while computing the modified references at the time of reflecting the update on another old copy. Thus, in accordance with typical embodiments of the present application, a pre-processing is applied to both old and new files (giving rise to modified old and modified new programs) and applying a diff extraction utility to the modified old and modified new programs. The modification is effected in such a way that references become 'invariant' (see page 5, lines 22 to 29), thereby reducing the diff result and rendering it compact.

Okuzumi et al. explicitly mention 'source', and even more, this prior art reference contains a step of sorting 'statements' (a common name for an element of a source program) according to their 'character strings', such as in 0011.

In contrast, the present invention, according to amended Claim 1, defines an executable program. Accordingly, obtaining a compact difference result in the manner defined in the claims of the present application is not suggested even remotely in Okuzumi, which is not surprising considering the fact that in source programs there is no need to generate a compact result since the difference result is a priori compact.

In extracting diff between 2 versions of executable files as defined in amended Claim 1, there is no source involved, and neither statements, nor any textual or other symbolic representation of the program even exist.

Another major characteristic disclosed by Okuzumi et al. is that it deals with a special problem of updating an old source program (Okuzumi 0001, 0002 & 0008 & purpose section) that is being modified differently for 2 different copies and one is required to reflect both changes. For example, (see Okuzumi 00003) there is a package program (source program) that has undergone version change (first modified source). The same package program has been independently customized for a client use (second modified source). The version change of the first modified source need to be reflected also in the customized copy (second modified source). Note that the second modified source is not derived from the first modified source.

In contrast, in accordance with the present application the new program is derived from the old program (see for example page 18, lines 10 to 16, where P_1 is the old program, P_2 is the new program "which was generated (or could have been generated) by the sequence of modifications as depicted in the imaginary memory table (80)). Said sequence of modifications (either real or imaginary), constitutes a transition sequence between P_1 and P_2 ".

VERSION WITH MARKINGS TO SHOW CHANGES MADE
BY AMENDMENT

4. A method for generating a compact difference result between an old executable program and a new executable program; each program including reference entries that contain reference that refer to other entries in the program; the method comprising the steps of:

- (a) scanning the old program and for substantially each reference entry perform steps that include:
 - (i) replacing the reference of said entry by a distinct label mark, whereby a modified old program is generated;
- (b) scanning the new program and for substantially each reference entry perform steps that include:
 - (i) replacing the reference of said entry by a distinct label mark, whereby a modified new program is generated;
- (c) generating said difference result utilizing directly or indirectly at least said modified old program and modified new program.

5. A method for performing an update in an old executable program so as to generate a new executable program; each program including reference entries that contain reference that refer to other entries in the program; the method comprising the steps of:

- (a) receiving data that includes a compact difference result; said compact difference result was generated utilizing a modified old program and a modified new program;
- (b) scanning the old program and for substantially each reference entry perform steps that include:
 - (i) replacing the reference of said entry by a distinct label mark, whereby the modified old program is generated;
- (c) reconstituting the modified new program utilizing at least said compact difference result and said modified old program; said modified new program is differed from said new program at least in that substantially each reference entry in said new program is replaced in said modified new program by a distinct label mark;

- (d) reconstituting said new program utilizing directly or indirectly at least said compact difference result and said modified new program.

8. A method for generating a compact difference result between an old executable program and a new executable program; each program including reference entries that contain reference that refer to other entries in the program; the method comprising the steps of:

- (a) generating a modified old program utilizing at least said old program;
- (b) generating a modified new program utilizing at least said new program, said modified old program and modified new program have at least the following characteristics:
 - (i) substantially each reference in an entry in said old program that is different than corresponding entry in said new program due to delete/insert modifications that form part of the transition between said old program and new program are reflected as invariant references in the corresponding entries in said modified old and modified new programs;
- (c) generating said compact difference result utilizing at least said modified new program and modified old program.

12. A method for performing an update in an old executable program so as to generate a new executable program; each program including reference entries that contain reference that refer to other entries in the program; the method comprising the steps of:

- (a) receiving data that includes a compact difference result; said compact difference result was generated utilizing a modified old program and a modified new program;
- (b) generating a modified old program utilizing at least said old program;
- (c) reconstituting a modified new program utilizing directly or indirectly at least said modified old program and said compact difference result; said modified old program and modified new program have at least the following characteristics:
 - (i) substantially each reference in an entry in said old program that is different than corresponding entry in said new program due to delete/inset modifications that form part of the transition between said old program and new program are reflected as invariant references in the corresponding entries in said modified old and modified new programs;
- (d) reconstituting said new program utilizing directly or indirectly at least said

compact difference result and said modified new program.

14. A system for generating a compact difference result between an old executable program and a new executable program; each program including reference entries that contain reference that refer to other entries in the program; the system comprising a processing device capable of:

- (a) scanning the old program and for substantially each reference entry perform steps that include:
 - (i) replacing the reference of said entry by a distinct label mark, whereby a modified old program is generated;
- (b) scanning the new program and for substantially each reference entry perform steps that include:
 - (i) replacing the reference of said entry by a distinct label mark, whereby a modified new program is generated;
- (c) generating said difference result utilizing directly or indirectly at least said modified old program and modified new program.

18. A system for performing an update in an old executable program so as to generate a new executable program; each program including reference entries that contain reference that refer to other entries in the program; the system comprising a processing device capable of:

- (a) receiving data that includes a compact difference result; said compact difference result was generated utilizing a modified old program and a modified new program;
- (b) scanning the old program and for substantially each reference entry perform steps that include:
 - (i) replacing the reference of said entry by a distinct label mark, whereby the modified old program is generated;
- (c) reconstituting the modified new program utilizing at least said compact difference result and said modified old program; said modified new program is differed from said new program at least in that substantially each reference entry in said new program is replaced in said modified new program by a distinct label mark;
- (d) reconstituting said new program utilizing directly or indirectly at least said compact difference result and said modified new program.

21. A system for generating a compact difference result between an old executable program and a new executable program; each program including reference entries that contain reference that refer to other entries in the program; the system comprising a processing device capable of:

- (a) generating a modified old program utilizing at least said old program;
- (b) generating a modified new program utilizing at least said new program, said modified old program and modified new program have at least the following characteristics:
 - (i) substantially each reference in an entry in said old program that is different than corresponding entry in said new program due to delete/insert modifications that form part of the transition between said old program and new program are reflected as invariant references in the corresponding entries in said modified old and modified new programs;
- (c) generating said compact difference result utilizing at least said modified new program and modified old program.

25. A system for performing an update in an old executable program so as to generate a new executable program; each program including reference entries that contain reference that refer to other entries in the program; the system comprising a processing device capable of:

- (a) receiving data that includes a compact difference result; said compact difference result was generated utilizing a modified old program and a modified new program;
- (b) generating a modified old program utilizing at least said old program;
- (c) reconstituting a modified new program utilizing directly or indirectly at least said modified old program and said compact difference result; said modified old program and modified new program have at least the following characteristics:
 - (i) substantially each reference in an entry in said old program that is different than corresponding entry in said new program due to delete/inset modifications that form part of the transition between said old program and new program are reflected as invariant references in the corresponding entries in said modified old and modified new programs;
- (d) reconstituting said new program utilizing directly or indirectly at least said compact difference result and said modified new program.

Exhibit C

Excerpt from Appendix A of Vol. 2 of the *Intel Architecture Software Developer's Manual*

APPENDIX A OPCODE MAP

The opcode tables in this chapter are provided to aid in interpreting Intel Architecture object code. The instructions are divided into three encoding groups: 1-byte opcode encodings, 2-byte opcode encodings, and escape (floating-point) encodings. The 1- and 2-byte opcode encodings are used to encode integer, system, MMX™ technology, and Streaming SIMD Extensions. The opcode maps for these instructions are given in Table A-2 through A-6. Section A.2.1., “One-Byte Opcode Instructions” through Section A.2.5., “Opcode Extensions For One- And Two-byte Opcodes” give instructions for interpreting 1- and 2-byte opcode maps. The escape encodings are used to encode floating-point instructions. The opcode maps for these instructions are given in Table A-7 through A-22. Section A.2.6., “Escape Opcode Instructions” gives instructions for interpreting the escape opcode maps.

The opcode tables in this section aid in interpreting Pentium® processor object code. Use the four high-order bits of the opcode as an index to a row of the opcode table; use the four low-order bits as an index to a column of the table. If the opcode is 0FH, refer to the 2-byte opcode table and use the second byte of the opcode to index the rows and columns of that table.

The escape (ESC) opcode tables for floating-point instructions identify the eight high-order bits of the opcode at the top of each page. If the accompanying ModR/M byte is in the range 00H through BFH, bits 3 through 5 identified along the top row of the third table on each page, along with the REG bits of the ModR/M, determine the opcode. ModR/M bytes outside the range 00H through BFH are mapped by the bottom two tables on each page.

Refer to Chapter 2, *Instruction Format* for detailed information on the ModR/M byte, register values, and the various addressing forms.

A.1. KEY TO ABBREVIATIONS

Operands are identified by a two-character code of the form Zz. The first character, an uppercase letter, specifies the addressing method; the second character, a lowercase letter, specifies the type of operand.

A.1.1. Codes for Addressing Method

The following abbreviations are used for addressing methods:

- A Direct address. The instruction has no ModR/M byte; the address of the operand is encoded in the instruction; and no base register, index register, or scaling factor can be applied (for example, far JMP (EA)).
- C The reg field of the ModR/M byte selects a control register (for example, MOV (0F20, 0F22)).

- D The reg field of the ModR/M byte selects a debug register (for example, MOV (0F21,0F23)).
- E A ModR/M byte follows the opcode and specifies the operand. The operand is either a general-purpose register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.
- F EFLAGS Register.
- G The reg field of the ModR/M byte selects a general register (for example, AX (000)).
- I Immediate data. The operand value is encoded in subsequent bytes of the instruction.
- J The instruction contains a relative offset to be added to the instruction pointer register (for example, JMP (0E9), LOOP).
- M The ModR/M byte may refer only to memory (for example, BOUND, LES, LDS, LSS, LFS, LGS, CMPXCHG8B).
- O The instruction has no ModR/M byte; the offset of the operand is coded as a word or double word (depending on address size attribute) in the instruction. No base register, index register, or scaling factor can be applied (for example, MOV (A0–A3)).
- P The reg field of the ModR/M byte selects a packed quadword MMX™ technology register.
- Q A ModR/M byte follows the opcode and specifies the operand. The operand is either an MMX™ technology register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement.
- R The mod field of the ModR/M byte may refer only to a general register (for example, MOV (0F20-0F24, 0F26)).
- S The reg field of the ModR/M byte selects a segment register (for example, MOV (8C,8E)).
- T The reg field of the ModR/M byte selects a test register (for example, MOV (0F24,0F26)).
- V The reg field of the ModR/M byte selects a packed SIMD floating-point register.
- W An ModR/M byte follows the opcode and specifies the operand. The operand is either a SIMD floating-point register or a memory address. If it is a memory address, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement
- X Memory addressed by the DS:SI register pair (for example, MOVS, CMPS, OUTS, or LODS).
- Y Memory addressed by the ES:DI register pair (for example, MOVS, CMPS, INS, STOS, or SCAS).

A.1.2. Codes for Operand Type

The following abbreviations are used for operand types:

- a Two one-word operands in memory or two double-word operands in memory, depending on operand-size attribute (used only by the BOUND instruction).
- b Byte, regardless of operand-size attribute.
- c Byte or word, depending on operand-size attribute.
- d Doubleword, regardless of operand-size attribute.
- dq Double-quadword, regardless of operand-size attribute.
- p 32-bit or 48-bit pointer, depending on operand-size attribute.
- pi Quadword MMX™ technology register (e.g. mm0)
- ps 128-bit packed FP single-precision data.
- q Quadword, regardless of operand-size attribute.
- s 6-byte pseudo-descriptor.
- ss Scalar element of a 128-bit packed FP single-precision data.
- si Doubleword integer register (e.g., eax)
- v Word or doubleword, depending on operand-size attribute.
- w Word, regardless of operand-size attribute.

A.1.3. Register Codes

When an operand is a specific register encoded in the opcode, the register is identified by its name (for example, AX, CL, or ESI). The name of the register indicates whether the register is 32, 16, or 8 bits wide. A register identifier of the form eXX is used when the width of the register depends on the operand-size attribute. For example, eAX indicates that the AX register is used when the operand-size attribute is 16, and the EAX register is used when the operand-size attribute is 32.

A.2. OPCODE LOOK-UP EXAMPLES

This section provides several examples to demonstrate how the following opcode maps are used. Refer to the introduction to Chapter 3, *Instruction Set Reference*, in the *Intel Architecture Software Developer's Manual, Volume 2* for detailed information on the ModR/M byte, register values, and the various addressing forms.

A.2.1. One-Byte Opcode Instructions

The opcode maps for 1-byte opcodes are shown in Table A-2 and A-3. Looking at the 1-byte opcode maps, the instruction and its operands can be determined from the hexadecimal opcode. For example:

Opcode: 030500000000H

LSB address					MSB address
03	05	00	00	00	00

Opcode 030500000000H for an ADD instruction can be interpreted from the 1-byte opcode map as follows. The first digit (0) of the opcode indicates the row, and the second digit (3) indicates the column in the opcode map tables. The first operand (type Gv) indicates a general register that is a word or doubleword depending on the operand-size attribute. The second operand (type Ev) indicates that a ModR/M byte follows that specifies whether the operand is a word or doubleword general-purpose register or a memory address. The ModR/M byte for this instruction is 05H, which indicates that a 32-bit displacement follows (00000000H). The reg/opcode portion of the ModR/M byte (bits 3 through 5) is 000, indicating the EAX register. Thus, it can be determined that the instruction for this opcode is ADD EAX, mem_op, and the offset of mem_op is 00000000H.

Some 1- and 2-byte opcodes point to “group” numbers. These group numbers indicate that the instruction uses the reg/opcode bits in the ModR/M byte as an opcode extension (refer to Section A.2.5., “Opcode Extensions For One- And Two-byte Opcodes”).

A.2.2. Two-Byte Opcode Instructions

Instructions that begin with 0FH can be found in the two-byte opcode maps given in Table A-4 and A-5. The second opcode byte is used to reference a particular row and column in the tables. For example, the opcode 0FA405000000003H is located on the two-byte opcode map in row A, column 4. This opcode indicates a SHLD instruction with the operands Ev, Gv, and Ib. These operands are defined as follows:

- Ev The ModR/M byte follows the opcode to specify a word or doubleword operand
- Gv The reg field of the ModR/M byte selects a general-purpose register
- Ib Immediate data is encoded in the subsequent byte of the instruction.

The third byte is the ModR/M byte (05H). The mod and opcode/reg fields indicate that a 32-bit displacement follows, located in the EAX register, and is the source.

The next part of the opcode is the 32-bit displacement for the destination memory operand (00000000H), and finally the immediate byte representing the count of the shift (03H).

By this breakdown, it has been shown that this opcode represents the instruction:

SHLD DS:00000000H, EAX, 3

The next part of the SHLD opcode is the 32-bit displacement for the destination memory operand (00000000H), which is followed by the immediate byte representing the count of the shift (03H). By this breakdown, it has been shown that the opcode 0FA4050000000003H represents the instruction:

SHLD DS:00000000H, EAX, 3.

Lower case is used in the following tables to highlight the mnemonics added by MMX™ technology and Streaming SIMD Extensions.

A.2.3. Opcode Map Shading

Table A-2 contains notes on particular encodings. These notes are indicated in the following Opcode Maps (Table A-2 through A-6) by superscripts.

For the One-byte Opcode Maps (Table A-2 through A-3), grey shading indicates instruction groupings.

A.2.4. Opcode Map Notes

Table A-1 contains notes on particular encodings. These notes are indicated in the following Opcode Maps (Table A-2 through A-6) by superscripts.

Table A-1. Notes on Instruction Set Encoding Tables

Symbol	Note
1A	Bits 5, 4, and 3 of ModR/M byte used as an opcode extension (refer to Section A.2.5., "Opcode Extensions For One- And Two-byte Opcodes").
1B	These abbreviations are not actual mnemonics. When shifting by immediate shift counts, the PSHIMD mnemonic represents the PSLLD, PSRAD, and PSRLD instructions, PSHIMW represents the PSLW, PSRAW, and PSRLW instructions, and PSHIMQ represents the PSLQ and PSRLQ instructions. The instructions that shift by immediate counts are differentiated by ModR/M bytes (refer to Section A.2.5., "Opcode Extensions For One- And Two-byte Opcodes").
1C	Use the 0F0B opcode (UD2 instruction) or the 0FB9H opcode when deliberately trying to generate an invalid opcode exception (#UD).
1D	Some instructions added in the Pentium® III processor may use the same two-byte opcode. If the instruction has variations, or the opcode represents different instructions, the ModR/M byte will be used to differentiate the instruction. For the value of the ModR/M byte needed to completely decode the instruction, see Table A-6. (These instructions include SFENCE, STMXCSR, LDMXCSR, FXRSTOR, and FXSAVE, as well as PREFETCH and its variations.)

Table A-2. One-byte Opcode Map (Left)

	0	1	2	3	4	5	6	7	
0	ADD						PUSH ES	POP ES	
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv			
1	ADC						PUSH SS	POP SS	
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv			
2	AND						SEG=ES	DAA	
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv			
3	XOR						SEG=SS	AAA	
	Eb, Gb	Ev, Gv	Gb, Eb	Gb, Ev	AL, Ib	eAX, Iv			
4	INC general register								
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI	
5	PUSH general register								
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI	
6	PUSHA/ PUSHAD	POPA/ POPAD	BOUND Gv, Ma	ARPL Ew, Gw	SEG=FS	SEG=GS	Opd Size	Addr Size	
7	Jcc, Jb - Short-displacement jump on condition								
	O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A	
8	Immediate Grp 1 ^{1A}				TEST		XCHG		
	Eb, Ib	Ev, Iv	Ev, Ib	Ev, Ib	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv	
9	NOP	XCHG word or double-word register with eAX						eSI	eDI
		eCX	eDX	eBX	eSP	eBP			
A	MOV								
	AL, Ob	eAX, Ov	Ob, AL	Ov, eAX	MOVS/ MOVSB Xb, Yb	MOVS/ MOVSW/ MOVSD Xv, Yv	CMPS/ CMPSB Xb, Yb	CMPS/ CMPSW/ CMPSD Xv, Yv	
B	MOV immediate byte into byte register								
	AL	CL	DL	BL	AH	CH	DH	BH	
C	Shift Grp 2 ^{1A}		RETN lw	RETN	LES Gv, Mp	LDS Gv, Mp	Grp 11 ^{1A} - MOV		
	Eb, Ib	Ev, Ib					Eb, Ib	Ev, Iv	
D	Shift Grp 2 ^{1A}				AAM lb	AAD lb		XLAT/ XLATB	
	Eb, 1	Ev, 1	Eb, CL	Ev, CL					
E	LOOPNE/ LOOPNZ Jb	LOOPE/ LOOPZ Jb	LOOP Jb	JCXZ/ JECXZ Jb	IN		OUT		
					AL, Ib	eAX, Ib	Ib, AL	Ib, eAX	
F	LOCK		REPNE	REP/ REPE	HLT	CMC	Unary Grp 3 ^{1A}		
							Eb	Ev	

Table A-3. One-byte Opcode Map (Right)

8	9	A	B	C	D	E	F	
OR						PUSH CS	2-byte escape	0
Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv			
SBB						PUSH DS	POP DS	1
Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv			
SUB						SEG=CS	DAS	2
Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv			
CMP						SEG=DS	AAS	3
Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	eAX, Iv			
DEC general register						eSI	eDI	4
eAX	eCX	eDX	eBX	eSP	eBP			
POP into general register						eSI	eDI	5
eAX	eCX	eDX	eBX	eSP	eBP			
PUSH Iv	IMUL Gv, Ev, Iv	PUSH Ib	IMUL Gv, Ev, Ib	INS/INSB Yb, DX	INS/INSW/INSD Yv, DX	OUTS/OUTSB DX, Xb	OUTS/OUTSW/OUTSD DX, Xv	6
Jcc, Jb- Short displacement jump on condition								
S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G	7
MOV				MOV Ew, Sw	LEA Gv, M	MOV Sw, Ew	POP Ev	8
Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev					
CBW/CWDE	CWD/CDQ	CALLF Ap	FWAIT/WAIT	PUSHF/PUSHFD Fv	POPF/POPFD Fv	SAHF	LAHF	9
TEST		STOS/STOSB Yb, AL	STOS/STOSW/STOSD Yv, eAX	LODS/LODSB AL, Xb	LODS/LODSW/LODSD eAX, Xv	SCAS/SCASB AL, Yb	SCAS/SCASW/SCASD eAX, Xv	A
AL, Ib	eAX, Iv							
MOV immediate word or double into word or double register								
eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI	B
ENTER lw, lb	LEAVE	RETF lw	RETF	INT 3	INT lb	INTO	IRET	C
ESC (Escape to coprocessor instruction set)								
D								
CALL Jv	near JV	JMP far AP	short Jb	IN AL, DX eAX, DX		OUT DX, AL DX, eAX		E
CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp 4 ^{1A}	INC/DEC Grp 5 ^{1A}	F

GENERAL NOTE:

All blanks in the opcode maps A-2 and A-3 are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-4. Two-byte Opcode Map (Left) (First Byte is OFH)

	0	1	2	3	4	5	6	7
0	Grp 6 ^{1A}	Grp 7 ^{1A}	LAR Gv, Ew	LSL Gv, Ew			CLTS	
1	movups Vps, Wps movss (F3) Vss, Wss	movups Wps, Vps movss (F3) Wss, Vss	movlps Wq, Vq movhlps Vq, Vq	movlps Vq, Wq	unpcklps Vps, Wq	unpckhps Vps, Wq	movhps Vq, Wq movhlps Vq, Vq	movhps Wq, Vq
2	MOV Rd, Cd	MOV Rd, Dd	MOV Cd, Rd	MOV Dd, Rd				
3	WRMSR	RDTSC	RDMSR	RDPMC	SYSENTER	SYSEXIT		
4	CMOVcc, (Gv, Ev) - Conditional Move							
	O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
5	movmskps Ed, Vps	sqrtps Vps, Wps sqrtps (F3) Vss, Wss	rsqrtps Vps, Wps rsqrtps (F3) Vss, Wss	rcpps Vps, Wps rcpps (F3) Vss, Wss	andps Vps, Wps	andnps Vps, Wps	orps Vps, Wps	xorps Vps, Wps
6	punpcklbw Pq, Qd	punpcklwd Pq, Qd	punpckldq Pq, Qd	packsswb Pq, Qq	pcmpgtb Pq, Qq	pcmpgtw Pq, Qq	pcmpgtd Pq, Qq	packuswb Pq, Qq
7	pshufw Pq, Qq, Ib	pshimw ^{1B} Pq, Qq (Grp 12 ^{1A})	pshimd ^{1B} Pq, Qq (Grp 13 ^{1A})	pshimq ^{1B} Pq, Qq (Grp 14 ^{1A})	pcmpeqb Pq, Qq	pcmpeqw Pq, Qq	pcmpeqd Pq, Qq	emms
8	Jcc, Jv - Long-displacement jump on condition							
	O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
9	SETcc, Eb - Byte Set on condition							
	O	NO	B/C/NAE	AE/NB/NC	E/Z	NE/NZ	BE/NA	A/NBE
A	PUSH FS	POP FS	CPUID	BT Ev, Gv	SHLD Ev, Gv, Ib	SHLD Ev, Gv, CL		
B	CMPXCHG Eb, Gb		LSS Mp	BTR Ev, Gv	LFS Mp	LGS Mp	MOVZX Gv, Eb	
		Ev, Gv					Gv, Ew	
C	XADD Eb, Gb	XADD Ev, Gv	cmpps Vps, Wps, b cmpss (F3) Vss, Wss, Ib		pinsrw Pq, Ed, Ib	pextrw Gd, Pq, Ib	shufps Vps, Wps, Ib	Grp 9 ^{1A}
D		psrlw Pq, Qq (Grp 12 ^{1A})	psrld Pq, Qq (Grp 13 ^{1A})	psrlq Pq, Qq (Grp 14 ^{1A})		pmullw Pq, Qq		pmovmskb Gd, Pq
E	pavgb Pq, Qq	psraw Pq, Qq (Grp 12 ^{1A})	psrad Pq, Qq (Grp 13 ^{1A})	pavgw Pq, Qq	pmulhuw Pq, Qq	pmulhw Pq, Qq		movntq Wq, Vq
F		psllw Pq, Qq (Grp 12 ^{1A})	pslld Pq, Qq (Grp 13 ^{1A})	psllq Pq, Qq (Grp 14 ^{1A})		pmaddwd Pq, Qq	psadbw Pq, Qq	maskmovq Ppi, Qpi

GENERAL NOTE:

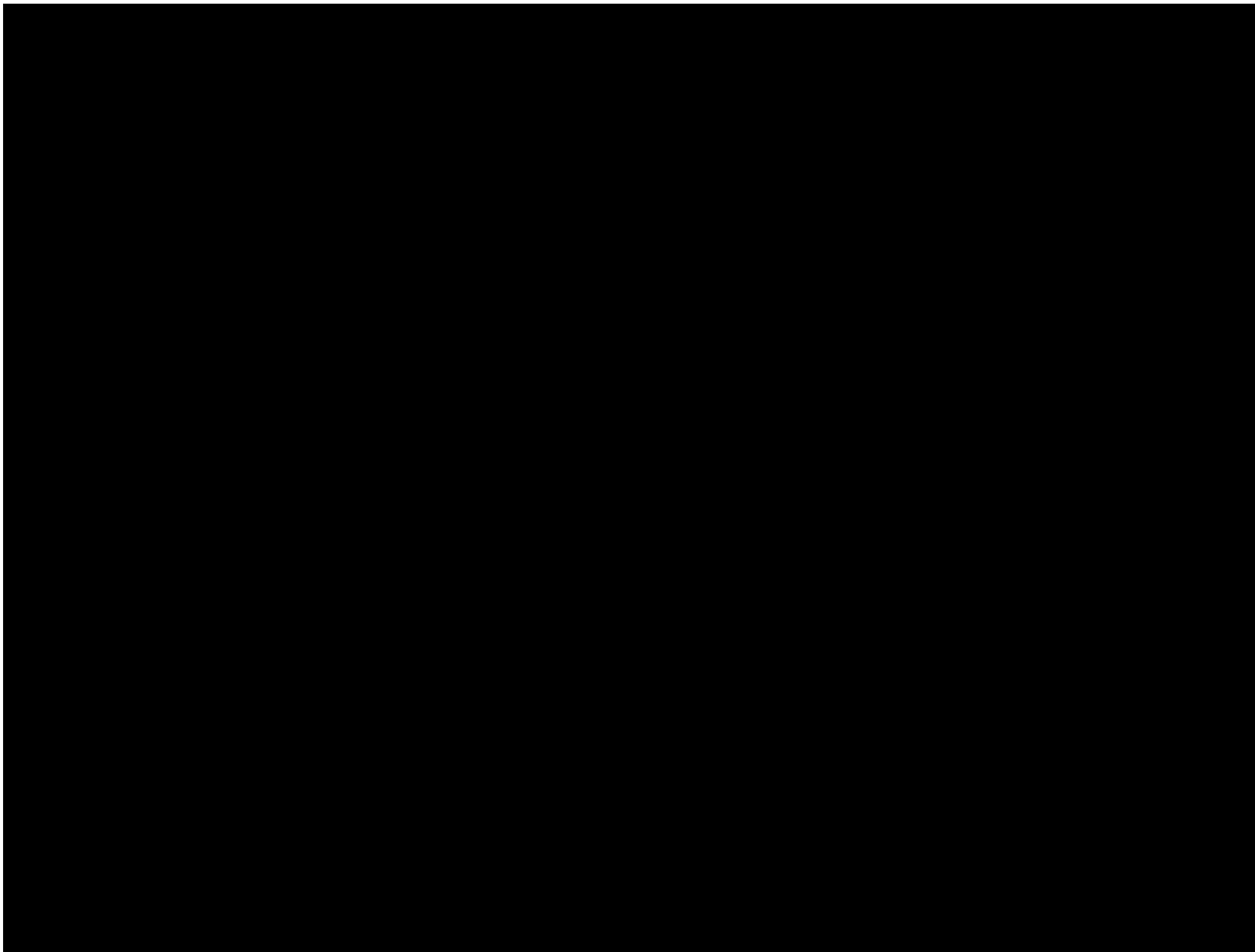
All blanks in the opcode maps A-4 and A-5 are reserved and should not be used. Do not depend on the operation of these undefined opcodes.

Table A-5. Two-byte Opcode Map (Right) (First Byte is OFH)

8	9	A	B	C	D	E	F	
INVD	WBINVD		2-byte Illegal Opcodes UD2 ^{1C}					0
Prefetch ^{1D} (Grp 16 ^{1A})								1
movaps Vps, Wps	movaps Wps, Vps	cvtpi2ps Vps, Qq cvtssi2ss (F3) Vss, Ed	movntps Wps, Vps	cvttps2pi Qq, Wps cvtssi2si (F3) Gd, Wss	cvtps2pi Qq, Wps cvtssi2si (F3) Gd, Wss	ucomiss Vss, Wss	comiss Vps, Wps	2
								3
CMOVcc(Gv, Ev) - Conditional Move								4
S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G	
addps Vps, Wps addss (F3) Vss, Wss	mulps Vps, Wps mulss (F3) Vss, Wss			subps Vps, Wps subss (F3) Vss, Wss	minps Vps, Wps minss (F3) Vss, Wss	divps Vps, Wps divss (F3) Vss, Wss	maxps Vps, Wps maxss (F3) Vss, Wss	5
punpckhbw Pq, Qd	punpckhwd Pq, Qd	punpckhdq Pq, Qd	packssdw Pq, Qd			movd Pd, Ed	movq Pq, Qq	6
MMX UD						movd Ed, Pd	movq Qq, Pq	7
Jcc, Jv - Long-displacement jump on condition								8
S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G	
SETcc, Eb - Byte Set on condition								9
S	NS	P/PE	NP/PO	L/NGE	NL/GE	LE/NG	NLE/G	
PUSH GS	POP GS	RSM	BTS Ev, Gv	SHRD Ev, Gv, Ib	SHRD Ev, Gv, CL	(Grp 15 ^{1A}) ^{1D}	MUL Gv, Ev	A
	Grp 10 ^{1A} Invalid Opcode ^{1C}	Grp 8 ^{1A} Ev, Ib	BTC Ev, Gv	BSF Gv, Ev	BSR Gv, Ev	MOV SX Gv, Eb Gv, Ew		B
BSWAP								C
EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI	
psubusb Pq, Qq	psubusw Pq, Qq	pminub Pq, Qq	pand Pq, Qq	paddusb Pq, Qq	paddusw Pq, Qq	pmaxub Pq, Qq	pandn Pq, Qq	D
psubsb Pq, Qq	psubsw Pq, Qq	pminsw Pq, Qq	por Pq, Qq	paddsb Pq, Qq	paddsw Pq, Qq	pmaxsw Pq, Qq	pxor Pq, Qq	E
psubb Pq, Qq	psubw Pq, Qq	psubd Pq, Qq		paddb Pq, Qq	paddw Pq, Qq	paddd Pq, Qq		F

Exhibit D





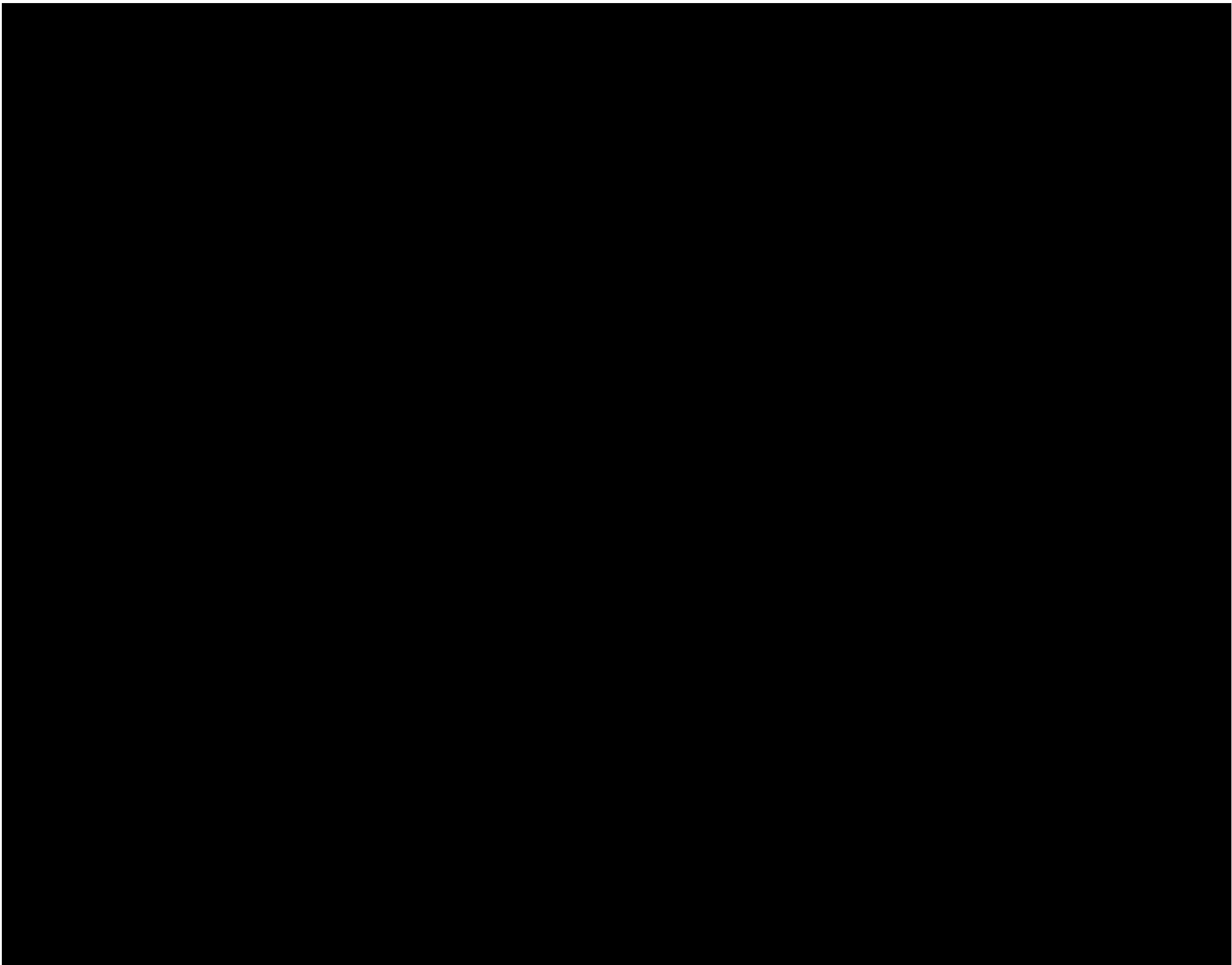


Exhibit E

Source for old and new programs for the vector experiment

```
// reva.cpp

#include "stdafx.h"
#include "stdio.h"

int baz()
{
    printf("In baz()\n");
    return 72;
}

int bar()
{
    printf("In bar()\n");
    int b = baz();
    return b + 10;
}

int foo()
{
    printf("Hello World!");
    int a = bar();
    return a + 42;
}
```

```
// revb.cpp

#include "stdafx.h"
#include "stdio.h"

int baz()
{
    printf("This is new code\n"); // INSERTED
    printf("In baz()\n");
    return 72;
}

int bar()
{
    printf("In bar()\n");
    int b = baz();
    return b + 10;
}

int foo()
{
    printf("Hello World!");
    int a = bar();
    return a + 42;
}
```

Exhibit F

Annotated disassembly of the old and new versions of the “baz” function

Old Version of baz() in “reva”

```

10011390: 55          push   %ebp
10011391: 8b ec      mov    %esp,%ebp
10011393: 81 ec c0 00 00 00 sub    $0xc0,%esp
10011399: 53        push   %ebx
1001139a: 56        push   %esi
1001139b: 57        push   %edi
1001139c: 8d bd 40 ff ff ff lea   -0xc0(%ebp),%edi
100113a2: b9 30 00 00 00 mov    $0x30,%ecx
100113a7: b8 cc cc cc cc mov    $0xcccccccc,%eax
100113ac: f3 ab     rep stos %eax,%es:(%edi)
100113ae: 8b f4     mov    %esp,%esi
100113b0: 68 3c 55 01 10 push   $0x1001553c # "In baz()\n"
100113b5: ff 15 64 82 01 10 call  *0x10018264 # printf
100113bb: 83 c4 04   add    $0x4,%esp
100113be: 3b f4     cmp    %esp,%esi
100113c0: e8 5d fd ff ff call   0x10011122 # __RTC_CheckEsp
100113c5: b8 48 00 00 00 mov    $0x48,%eax # 72 decimal
100113ca: 5f        pop    %edi
100113cb: 5e        pop    %esi
100113cc: 5b        pop    %ebx
100113cd: 81 c4 c0 00 00 00 add    $0xc0,%esp
100113d3: 3b ec     cmp    %esp,%ebp
100113d5: e8 48 fd ff ff call   0x10011122 # __RTC_CheckEsp
100113da: 8b e5     mov    %ebp,%esp
100113dc: 5d        pop    %ebp
100113dd: c3        ret

```

New Version of baz() in “revb”

```

10011340: 55          push   %ebp
10011341: 8b ec      mov    %esp,%ebp
10011343: 81 ec c0 00 00 00 sub    $0xc0,%esp
10011349: 53        push   %ebx
1001134a: 56        push   %esi
1001134b: 57        push   %edi
1001134c: 8d bd 40 ff ff ff lea   -0xc0(%ebp),%edi
10011352: b9 30 00 00 00 mov    $0x30,%ecx
10011357: b8 cc cc cc cc mov    $0xcccccccc,%eax
1001135c: f3 ab     rep stos %eax,%es:(%edi)
1001135e: 8b f4     mov    %esp,%esi
10011360: 68 48 55 01 10 push   $0x10015548 # "This is new code\n"
10011365: ff 15 64 82 01 10 call  *0x10018264 # printf
1001136b: 83 c4 04   add    $0x4,%esp
1001136e: 3b f4     cmp    %esp,%esi
10011370: e8 ad fd ff ff call   0x10011122 # __RTC_CheckEsp
10011375: 8b f4     mov    %esp,%esi
10011377: 68 3c 55 01 10 push   $0x1001553c # "In baz()\n"
1001137c: ff 15 64 82 01 10 call  *0x10018264 # printf
10011382: 83 c4 04   add    $0x4,%esp
10011385: 3b f4     cmp    %esp,%esi
10011387: e8 96 fd ff ff call   0x10011122 # __RTC_CheckEsp
1001138c: b8 48 00 00 00 mov    $0x48,%eax # 72 decimal
10011391: 5f        pop    %edi
10011392: 5e        pop    %esi
10011393: 5b        pop    %ebx
10011394: 81 c4 c0 00 00 00 add    $0xc0,%esp
1001139a: 3b ec     cmp    %esp,%ebp
1001139c: e8 81 fd ff ff call   0x10011122 # __RTC_CheckEsp
100113a1: 8b e5     mov    %ebp,%esp
100113a3: 5d        pop    %ebp
100113a4: c3        ret

```

Exhibit G

EncodedPrograms for the old and new programs

RVA	OP	Count	Bytes	A32 Ind	R32 Ind	Origin	RVA	OP	Count	Bytes	A32 Ind	R32 Ind	Origin	RVA	OP	Count	Bytes	A32 Ind	R32 Ind	Origin
0	CPY ORG	1024:	4D5A90..00			11000	0	CPY ORG	1024:	4D5A90..00			11000	0	CPY ORG	1024:	4D5A90..00			11000
11000	CPY	6:	CCCCC..E9				11000	CPY	6:	CCCCC..E9				11000	CPY	6:	CCCCC..E9			
11006	R32		(000025CC)		130		11006	R32		(000025DC)		130		11006	R32		(000025DC)		130	
1100A	CP1		E9				1100A	CP1		E9				1100A	CP1		E9			
1100B	R32		(00000E41)		58		1100B	R32		(00001781)		76		1100B	R32		(00001781)		58	
1100F	CP1		E9				1100F	CP1		E9				1100F	CP1		E9			
11010	R32		(00001A42)		91		11010	R32		(00001A52)		91		11010	R32		(00001A52)		91	
11014	CP1		E9				11014	CP1		E9				11014	CP1		E9			
11015	R32		(00002417)		117		11015	R32		(00002427)		117		11015	R32		(00002427)		117	
11019	CP1		E9				11019	CP1		E9				11019	CP1		E9			
1101A	R32		(000025AC)		128		1101A	R32		(000025BC)		128		1101A	R32		(000025BC)		128	
1101E	CP1		E9				1101E	CP1		E9				1101E	CP1		E9			
1101F	R32		(000019FD)		89		1101F	R32		(00001A0D)		89		1101F	R32		(00001A0D)		89	
11023	CP1		E9				11023	CP1		E9				11023	CP1		E9			
11024	R32		(000025B4)		131		11024	R32		(000025C4)		131		11024	R32		(000025C4)		131	
11028	CP1		E9				11028	CP1		E9				11028	CP1		E9			
11029	R32		(00000313)		33		11029	R32		(00000473)		36		11029	R32		(00000473)		33	
1102D	CP1		E9				1102D	CP1		E9				1102D	CP1		E9			
1102E	R32		(000005BE)		42		1102E	R32		(0000056E)		40		1102E	R32		(0000056E)		42	
11032	CP1		E9				11032	CP1		E9				11032	CP1		E9			
11033	R32		(00000ED9)		63		11033	R32		(00001819)		81		11033	R32		(00001819)		63	
11037	CP1		E9				11037	CP1		E9				11037	CP1		E9			
11038	R32		(00000E04)		57		11038	R32		(00001744)		75		11038	R32		(00001744)		57	
1103C	CP1		E9				1103C	CP1		E9				1103C	CP1		E9			
1103D	R32		(0000257D)		126		1103D	R32		(0000258D)		126		1103D	R32		(0000258D)		126	
11041	CP1		E9				11041	CP1		E9				11041	CP1		E9			
11042	R32		(000025E4)		144		11042	R32		(000025F4)		144		11042	R32		(000025F4)		144	
11046	CP1		E9				11046	CP1		E9				11046	CP1		E9			
11047	R32		(00001CC5)		103		11047	R32		(00001CD5)		103		11047	R32		(00001CD5)		103	
1104B	CP1		E9				1104B	CP1		E9				1104B	CP1		E9			
1104C	R32		(000025FE)		150		1104C	R32		(0000260E)		150		1104C	R32		(0000260E)		150	
11050	CP1		E9				11050	CP1		E9				11050	CP1		E9			
11051	R32		(000023ED)		118		11051	R32		(000023FD)		118		11051	R32		(000023FD)		118	
11055	CP1		E9				11055	CP1		E9				11055	CP1		E9			
11056	R32		(000025D6)		145		11056	R32		(000025E6)		145		11056	R32		(000025E6)		145	
1105A	CP1		E9				1105A	CP1		E9				1105A	CP1		E9			
1105B	R32		(00001961)		87		1105B	R32		(00001971)		87		1105B	R32		(00001971)		87	
1105F	CP1		E9				1105F	CP1		E9				1105F	CP1		E9			
11060	R32		(0000257E)		132		11060	R32		(0000258E)		132		11060	R32		(0000258E)		132	
11064	CP1		E9				11064	CP1		E9				11064	CP1		E9			
11065	R32		(00000E07)		59		11065	R32		(00001747)		77		11065	R32		(00001747)		59	
11069	CP1		E9				11069	CP1		E9				11069	CP1		E9			
1106A	R32		(00001DC2)		106		1106A	R32		(00001DD2)		106		1106A	R32		(00001DD2)		106	
1106E	CP1		E9				1106E	CP1		E9				1106E	CP1		E9			
1106F	R32		(000023E1)		121		1106F	R32		(000023F1)		121		1106F	R32		(000023F1)		121	

Old program

New program before adjustment

New program after adjustment

Exhibit H

EncodedPrograms for the modified “baz” and “foo” functions

RVA	OP	Count	Bytes	A32 Ind	R32 Ind	Origin	RVA	OP	Count	Bytes	A32 Ind	R32 Ind	Origin	RVA	OP	Count	Bytes	A32 Ind	R32 Ind	Origin
baz:							baz:							baz:						
1119F	CPY	530:	CCCCC..68				1119F	CPY	450:	CCCCC..68				1119F	CPY	450:	CCCCC..68			
113B1	A32		(1001553C)	22			11361	A32		(10015548)	23			11361	A32		(10015548)	22		
113B5	CPY	2:	FF15				11365	CPY	2:	FF15				11365	CPY	2:	FF15			
113B7	A32		(10018264)	173			11367	A32		(10018264)	174			11367	A32		(10018264)	173		
113BB	CPY	6:	83C404..E8				1136B	CPY	6:	83C404..E8				1136B	CPY	6:	83C404..E8			
113C1	R32		(FFFFFFD5D)		20		11371	R32		(FFFFFFDAD)		20		11371	R32		(FFFFFFDAD)		20	
113C5	CPY	17:	B84800..E8				11375	CPY	3:	8BF468				11375	CPY	3:	8BF468			
113D6	R32		(FFFFFFD48)		20		11378	A32		(1001553C)	22			11378	A32		(1001553C)	23		
113DA	CPY	71:	8BE55D..68				1137C	CPY	2:	FF15				1137C	CPY	2:	FF15			
							1137E	A32		(10018264)	174			1137E	A32		(10018264)	173		
							11382	CPY	6:	83C404..E8				11382	CPY	6:	83C404..E8			
							11388	R32		(FFFFFFD96)		20		11388	R32		(FFFFFFD96)		20	
							1138C	CPY	17:	B84800..E8				1138C	CPY	17:	B84800..E8			
							1139D	R32		(FFFFFFD81)		20		1139D	R32		(FFFFFFD81)		20	
							113A1	CPY	64:	8BE55D..68				113A1	CPY	64:	8BE55D..68			
foo:							foo:							foo:						
11453	CPY	62:	8BE55D..68				11413	CPY	62:	8BE55D..68				11413	CPY	62:	8BE55D..68			
11491	A32		(10015554)	24			11451	A32		(1001556C)	25			11451	A32		(1001556C)	92		
11495	CPY	2:	FF15				11455	CPY	2:	FF15				11455	CPY	2:	FF15			
11497	A32		(10018264)	173			11457	A32		(10018264)	174			11457	A32		(10018264)	173		
1149B	CPY	6:	83C404..E8				1145B	CPY	6:	83C404..E8				1145B	CPY	6:	83C404..E8			
114A1	R32		(FFFFFFC7D)		20		11461	R32		(FFFFFFCBD)		20		11461	R32		(FFFFFFCBD)		20	
114A5	CP1		E8				11465	CP1		E8				11465	CP1		E8			
114A6	R32		(FFFFFFBD8)		8		11466	R32		(FFFFFFC18)		8		11466	R32		(FFFFFFC18)		8	
114AA	CPY	21:	8945F8..E8				1146A	CPY	21:	8945F8..E8				1146A	CPY	21:	8945F8..E8			
114BF	R32		(FFFFFFC5F)		20		1147F	R32		(FFFFFFC9F)		20		1147F	R32		(FFFFFFC9F)		20	
114C3	CPY	31:	8BE55D..3D				11483	CPY	97:	8BE55D..25				11483	CPY	97:	8BE55D..25			

Old program

New program before adjustment

New program after adjustment

Exhibit I

An Encoded Program from Walker's Example

RVA	OP	Count	Bytes	A32 Ind	R32 Ind	Origin	RVA	OP	Count	Bytes	A32 Ind	R32 Ind	Origin	RVA	OP	Count	Bytes	A32 Ind	R32 Ind	Origin	
0	CPY	1024:	4D5A90..00				0	CPY	1024:	4D5A90..00				0	CPY	1024:	4D5A90..00				
	ORG					11000		ORG					11000		ORG						11000
11000	CPY	6:	CCCCC..E9				11000	CPY	6:	CCCCC..E9				11000	CPY	6:	CCCCC..E9				
11006	R32		(00001876)		85		11006	R32		(00001876)		80		11006	R32		(00001876)		85		
1100A	CP1		E9				1100A	CP1		E9				1100A	CP1		E9				
1100B	R32		(000026A7)		143		1100B	R32		(000026A7)		138		1100B	R32		(000026A7)		143		
1100F	CP1		E9				1100F	CP1		E9				1100F	CP1		E9				
11010	R32		(0000042C)		38		11010	R32		(0000276C)		160		11010	R32		(0000276C)		43		
11014	CP1		E9				11014	CP1		E9				11014	CP1		E9				
11015	R32		(00000DC7)		57		11015	R32		(00000DC7)		52		11015	R32		(00000DC7)		57		
11019	CP1		E9				11019	CP1		E9				11019	CP1		E9				
1101A	R32		(000003E2)		37		1101A	R32		(00002722)		159		1101A	R32		(00002722)		42		
1101E	CP1		E9				1101E	CP1		E9				1101E	CP1		E9				
1101F	R32		(00001973)		93		1101F	R32		(00001973)		88		1101F	R32		(00001973)		93		
11023	CP1		E9				11023	CP1		E9				11023	CP1		E9				
11024	R32		(000024C8)		127		11024	R32		(000024C8)		122		11024	R32		(000024C8)		127		
11028	CP1		E9				11028	CP1		E9				11028	CP1		E9				
11029	R32		(0000267D)		141		11029	R32		(0000267D)		136		11029	R32		(0000267D)		141		
1102D	CP1		E9				1102D	CP1		E9				1102D	CP1		E9				
1102E	R32		(0000192E)		91		1102E	R32		(0000192E)		86		1102E	R32		(0000192E)		91		
11032	CP1		E9				11032	CP1		E9				11032	CP1		E9				
11033	R32		(00002685)		144		11033	R32		(00002685)		139		11033	R32		(00002685)		144		
11037	CP1		E9				11037	CP1		E9				11037	CP1		E9				
11038	R32		(00000714)		47		11038	R32		(00000714)		42		11038	R32		(00000714)		47		
1103C	CP1		E9				1103C	CP1		E9				1103C	CP1		E9				
1103D	R32		(00000E5F)		62		1103D	R32		(00000E5F)		57		1103D	R32		(00000E5F)		62		
11041	CP1		E9				11041	CP1		E9				11041	CP1		E9				
11042	R32		(00000D8A)		56		11042	R32		(00000D8A)		51		11042	R32		(00000D8A)		56		
11046	CP1		E9				11046	CP1		E9				11046	CP1		E9				
11047	R32		(000018BB)		87		11047	R32		(000018BB)		82		11047	R32		(000018BB)		87		
1104B	CP1		E9				1104B	CP1		E9				1104B	CP1		E9				
1104C	R32		(0000264E)		139		1104C	R32		(0000264E)		134		1104C	R32		(0000264E)		139		
11050	CP1		E9				11050	CP1		E9				11050	CP1		E9				
11051	R32		(000026BB)		158		11051	R32		(000026BB)		153		11051	R32		(000026BB)		158		
11055	CP1		E9				11055	CP1		E9				11055	CP1		E9				
11056	R32		(00001AD6)		102		11056	R32		(00001AD6)		97		11056	R32		(00001AD6)		102		
1105A	CP1		E9				1105A	CP1		E9				1105A	CP1		E9				
1105B	R32		(0000267B)		149		1105B	R32		(0000267B)		144		1105B	R32		(0000267B)		149		
1105F	CP1		E9				1105F	CP1		E9				1105F	CP1		E9				
11060	R32		(00001BEA)		106		11060	R32		(00001BEA)		101		11060	R32		(00001BEA)		106		
11064	CP1		E9				11064	CP1		E9				11064	CP1		E9				
11065	R32		(00001E6D)		113		11065	R32		(00001E6D)		108		11065	R32		(00001E6D)		113		
11069	CP1		E9				11069	CP1		E9				11069	CP1		E9				
1106A	R32		(000026A8)		159		1106A	R32		(000026A8)		154		1106A	R32		(000026A8)		159		
1106E	CP1		E9				1106E	CP1		E9				1106E	CP1		E9				
1106F	R32		(00001A7D)		97		1106F	R32		(00001A7D)		92		1106F	R32		(00001A7D)		97		

Old program

New program before adjustment

New program after adjustment

Exhibit J



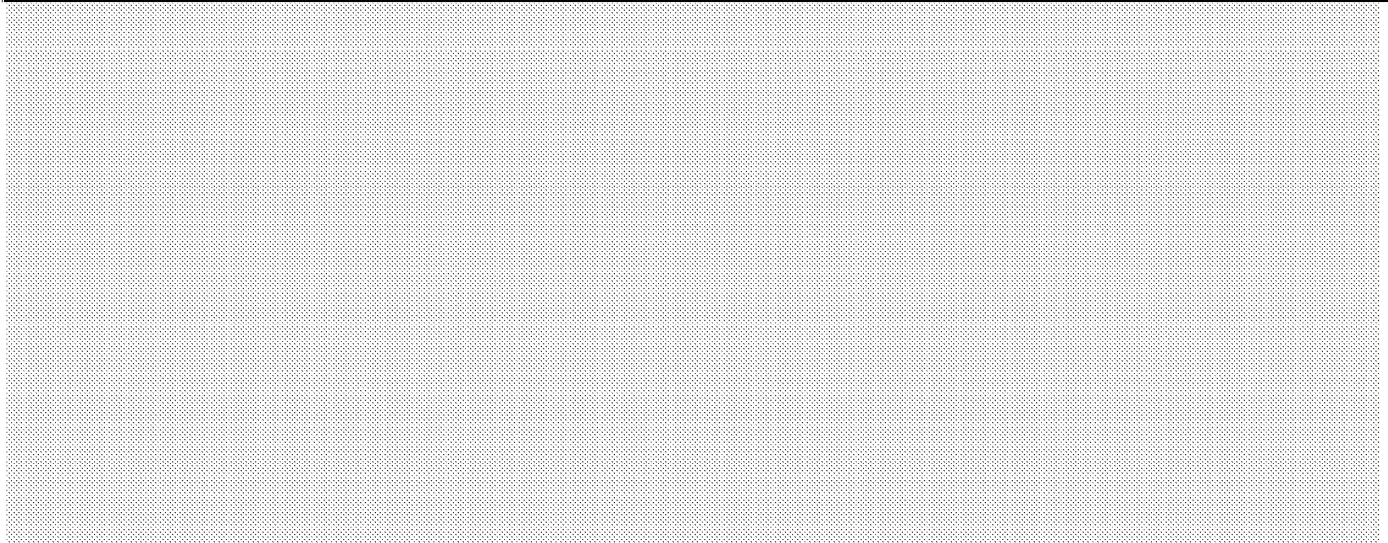
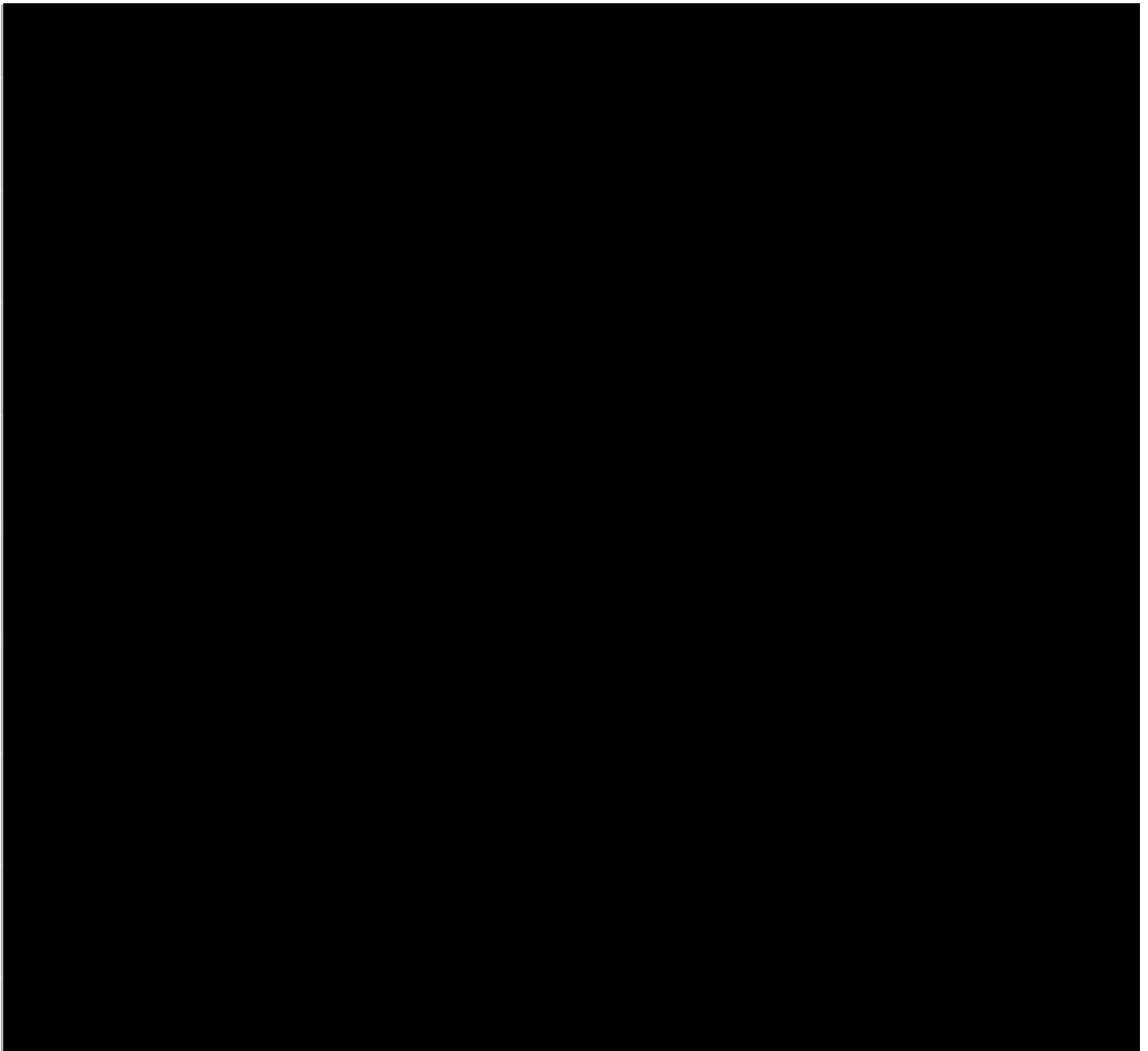
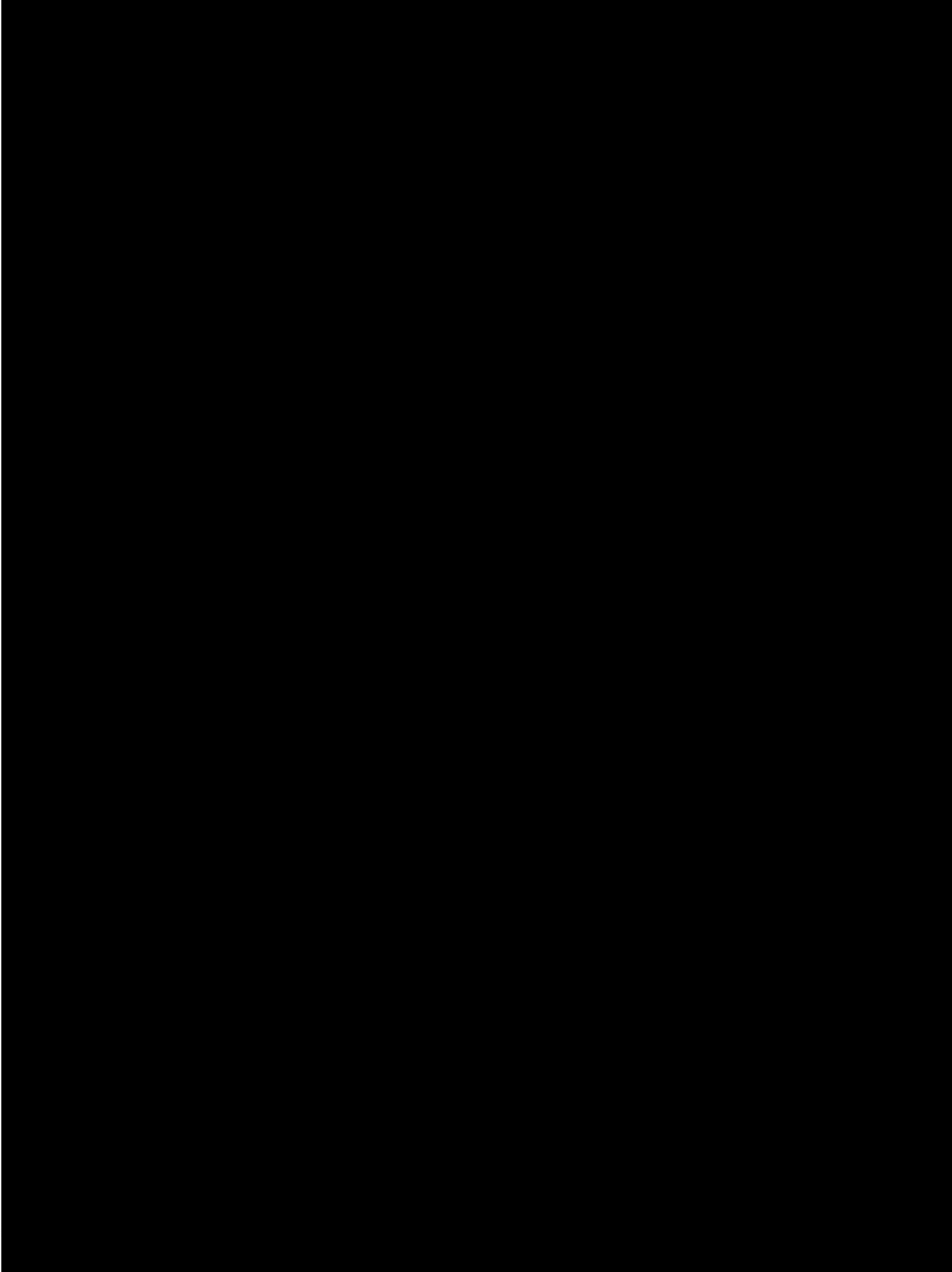


Exhibit K







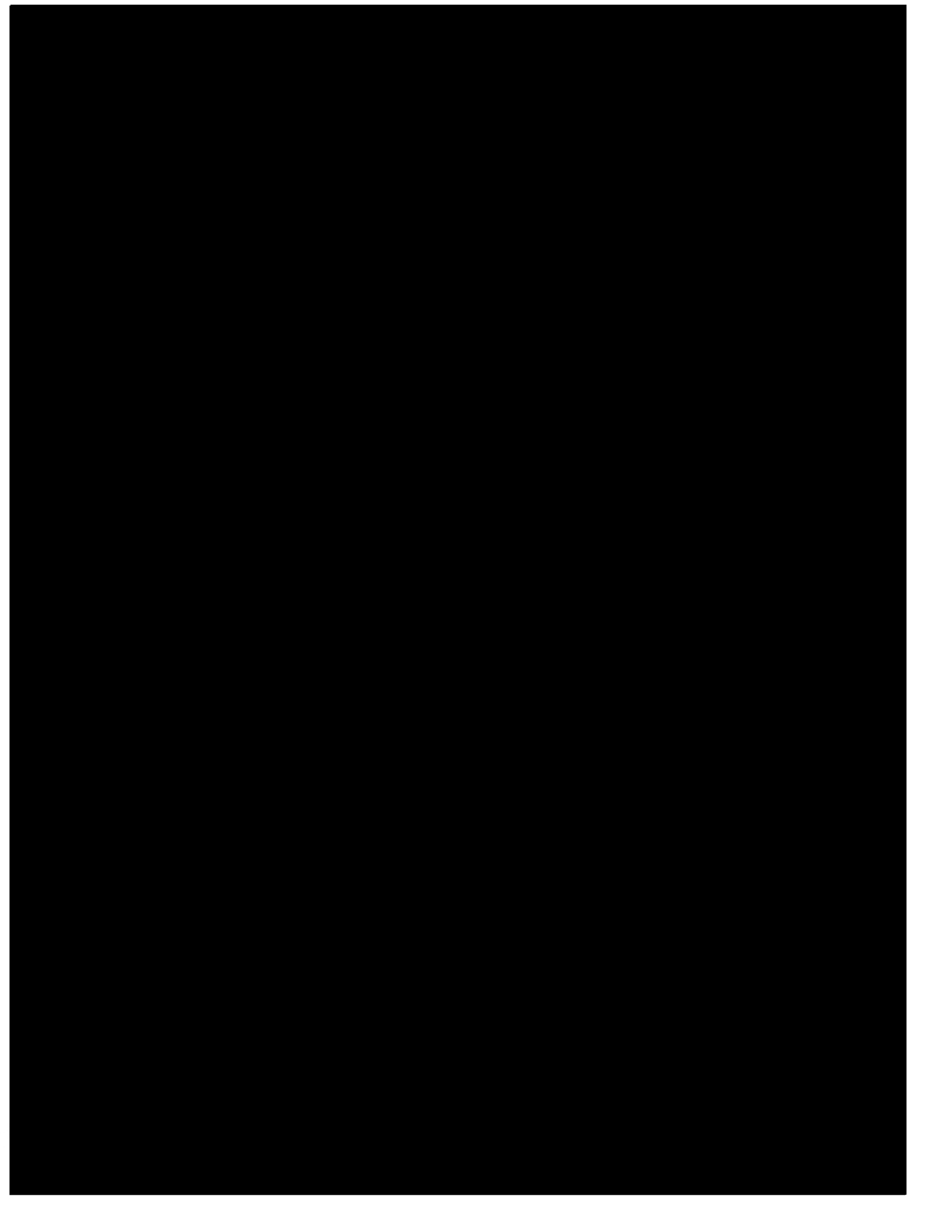


Exhibit L

Results of My Reference Accounting Experiments

Revisions Compared	Abs/Rel32 References			Rel8 Refs.	Results	
	Corresp. Indexes	Different	Unchanged Addresses	Changed	Total Refs Changed Del/Ins	% Rel8 Undetected
Successive Chrome Releases						
1.dll → 2.dll	582666	2339	480	499	585024	0.085
2.dll → 3.dll	557491	12677	680	7724	577212	1.338
3.dll → 4.dll	608555	3362	495	2152	613574	0.35
4.dll → 5.dll	608553	6447	499	2195	616696	0.355
5.dll → 6.dll	622528	1980	78495	148	546161	0.027
6.dll → 7.dll	617711	3943	692	3907	624869	0.625
7.dll → 8.dll	615004	7283	514	4425	626198	0.706
8.dll → 9.dll	653147	5393	511	2116	660145	0.32
9.dll → 10.dll	654225	8557	530	7119	669371	1.063
10.dll → 11.dll	677561	56	232899	17	444735	0.003
11.dll → 12.dll	667591	6275	555	2735	676046	0.404
Bigger Jumps Between Chrome Releases						
1.dll → 12.dll	534454	22537	572	17308	573727	3.016
3.dll → 12.dll	571535	21074	553	15748	607804	2.59
6.dll → 12.dll	588311	17928	735	14011	619515	2.261
9.dll → 12.dll	649099	11146	548	9101	668798	1.36
Test Cases						
reva.dll → revb.dll	499	6	280	2	227	0.881
setup1.exe → setup2.exe	21858	49	3366	13	18554	0.07

Abs/Rel32 References refers to reference entries treated by Courgette as indexes; **Corresp. Indexes** is the total number of indexes that moved in the index vectors without changing value, i.e., those that the adjustment step found corresponded; **Different** are indexes that changed, but probably corresponded; **Unchanged Addresses** are the number of indexes whose addresses did not change, i.e., corresponding, but not different. **Rel8 Refs. Changed** is the number of suspected rel8 refs in the difference stream that changed due to insert/delete modification but went undetected. **Total Changed Due Del/Ins** is the estimated number of all references that changed due to delete/insert modifications, that is, Abs/Rel32 references that corresponded plus those found to be different less those with unchanged addresses (did not change) plus all changed rel8 references; **% Rel8 Undetected** is the percentage of undetected Rel8 references that went undetected among all the corresponding references that changed. Part of the infringement question is whether these numbers are “substantial.”

The PE files compared

Filename	Source	Size (bytes)
1.dll	Chrome.dll from Chrome release 4.0.249.78	14,489,584
2.dll	Chrome.dll from Chrome release 4.0.249.89	14,492,144
3.dll	Chrome.dll from Chrome release 4.0.288.1	14,865,392
4.dll	Chrome.dll from Chrome release 4.0.295.0	14,921,728
5.dll	Chrome.dll from Chrome release 4.0.302.2	15,213,552
6.dll	Chrome.dll from Chrome release 4.0.302.3	15,214,064
7.dll	Chrome.dll from Chrome release 5.0.307.1	15,286,768
8.dll	Chrome.dll from Chrome release 5.0.317.2	16,734,192
9.dll	Chrome.dll from Chrome release 5.0.322.2	16,763,888
10.dll	Chrome.dll from Chrome release 5.0.335.0	16,892,400
11.dll	Chrome.dll from Chrome release 5.0.335.1	16,892,400
12.dll	Chrome.dll from Chrome release 5.0.342.1	17,015,352
setup1.exe	From Courgette distribution	967,168
setup2.exe	From Courgette distribution	965,632
reva.dll	My testcase	29,184
revb.dll	My testcase	29,184

Chrome.dll extracted from Chrome releases downloaded from
http://www.filehippo.com/download_google_chrome/

Exhibit M

Excerpt from Red Bend's "Principles of Updating Mobile Firmware Over-the-Air (FOTA)"

There are three approaches to address the ripple effect: padding, patching and computational.

- **Padding** — The firmware is split into several segments with extra spaces in between, called pads, which are designed to absorb some of the code shifts and reduce the ripple effect.
- **Patching** — Complete portions such code procedures, which need to be modified, are completely disabled and their new version is appended to the firmware, completely avoiding the ripple effect.
- **Computational** — This method removes the changed references out of the delta calculations and lets the receiving side to compute all these changes by itself.

From our experience and mainly from our customers' experience over the last five years, it is evident that padding and patching are either not sufficient or not practical to implement. Padding require re-organization of the firmware on a per version process, wastes expensive memory on the handset, and its behavior remains unpredictable. Patching is not scalable beyond one or two small updates. The computational approach, on the other hand, has been proven to provide optimum utilization of memory on the handset, a smaller delta package that utilizes less network bandwidth and storage space, and a simplified integration process.

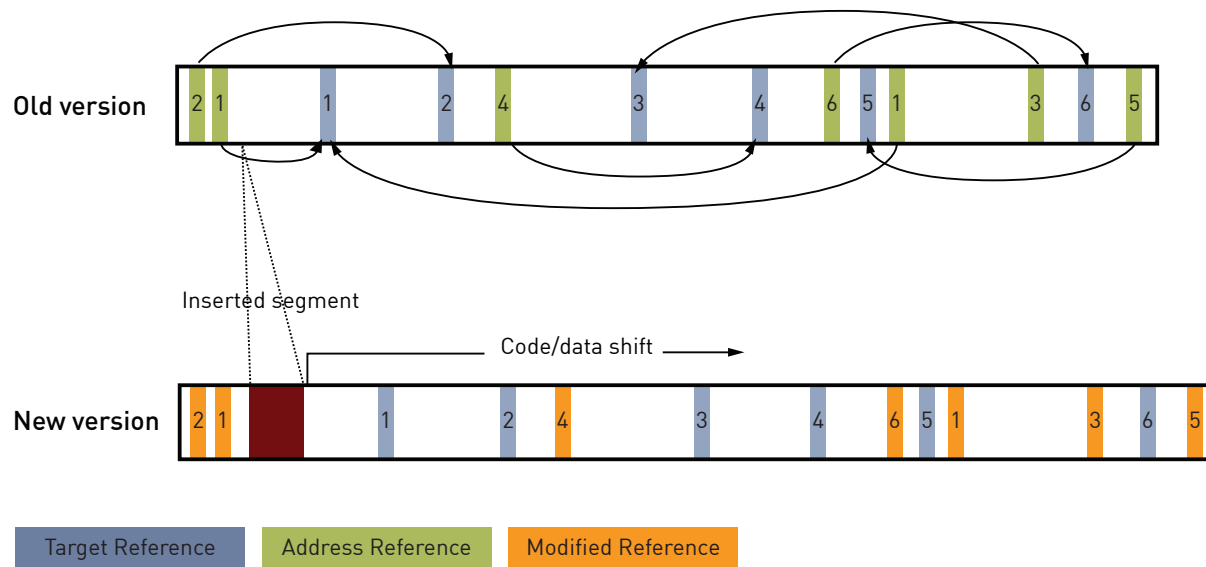


Figure 2: The "Ripple Effect"

Many modified references significantly increase the variance between versions and significantly complicate update calculation & application

Engineering Challenges on the Device

The challenge of delta-updating of mobile firmware is the update process on the device itself. While the generator running on a PC in the office can practically enjoy no resource limitations, the design of the update process on mobile devices must take into account the lack of any auxiliary data, lack of extra storage to be used as temporary buffers, lack of RAM in some cases and a much weaker CPU. Ineffective approaches could easily lead to conflicts between the various resources, and result in the "short blanket" effect – if you pull it from one side, the other side is left uncovered.

Exhibit N

Excerpts from Brian Bershad's patent, "Discovering code and data in a binary executable program"



US006014513A

United States Patent [19]
Voelker et al.

[11] **Patent Number:** **6,014,513**
[45] **Date of Patent:** **Jan. 11, 2000**

[54] **DISCOVERING CODE AND DATA IN A BINARY EXECUTABLE PROGRAM**

[75] Inventors: **Geoffrey Michael Voelker; Theodore H. Romer; Alastair Wolman; Dennis Chua Lee; Brian N. Bershad**, all of Seattle, Wash.; **John Bradley Chen**, Winchester, Mass.; **Henry M. Levy**, Seattle, Wash.; **Wayne Anthony Wong**, Hillsboro, Oreg.

[73] Assignee: **University of Washington**, Seattle, Wash.

[21] Appl. No.: **08/996,839**

[22] Filed: **Dec. 23, 1997**

[51] **Int. Cl.**⁷ **G06F 9/445**

[52] **U.S. Cl.** **395/703; 395/701; 395/702; 395/704; 395/705; 395/706; 395/707**

[58] **Field of Search** **395/703, 704, 395/705, 709**

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,937,777	6/1990	Flood et al.	710/107
5,115,500	5/1992	Larsen	712/209
5,151,981	9/1992	Westcott et al.	714/50
5,214,763	5/1993	Blaner et al.	712/213
5,216,613	6/1993	Head, III	364/468.08
5,295,249	3/1994	Blaner et al.	712/213
5,603,043	2/1997	Taylor et al.	712/1
5,790,856	8/1998	Lillich	395/703
5,930,509	7/1999	Yates et al.	395/707

OTHER PUBLICATIONS

Ramsey et al., "Specifying Representations of Machine Instructions", ACM, pp. 492-524, May 1997.

Ramsey, "Relocating Machine Instruction by Currying", ACM, pp. 226-236, May 1996.

Hannan, "Operational Semantics-directed Compilers and Machine Architectures", ACM, pp. 1215-1247, Jul. 1994.

Cifuentes et al., "Intraprocedural Static Slicing of Binary Executables", IEEE, pp. 188-195, Oct. 1997.

Nichols et al., "Data Management and Control-flow Aspects of an SIMD/SPMD parallel Language/Compiler", IEEE, pp. 222-234, Feb. 1993.

Aho et al., "Compilers Principles, Techniques, and Tools", Addison-Wesley Publishing, pp. 10-15, 513-518, 522-538, 1988.

Primary Examiner—Tanq R. Hafiz

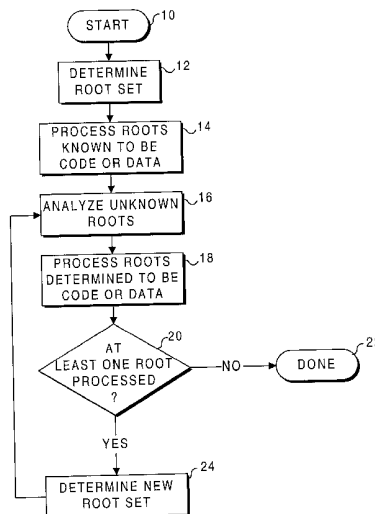
Assistant Examiner—Ted T. Vo

Attorney, Agent, or Firm—Ronald M. Anderson

[57] **ABSTRACT**

A computer software tool used for automatically identifying code portions and data portions of a binary executable software program in which the code portions include machine instructions that are of arbitrary length. Software products are typically distributed as binary, executable files, which comprise a string of binary values. In general, an executable file has no structure or meaning, except as determined by its behavior when dynamically executed, one instruction at a time, by a digital computer. The software tool determines a set of addresses for any known code and data portions. The tool is then used to disassemble machine instructions, beginning at a starting address for each known code portion, to identify the target addresses of other code portions and other data portions. Other sections of the binary executable software program that could be either code or data are then analyzed to identify additional code and data portions. As new portions are identified, the steps are repeated, until no further code or data portions are identifiable. The binary executable software program may include a plurality of executable modules. The entry addresses for each executable module and any addresses for code portions and data portions referenced and identified by any debug address, any export address, and any relocation address is added to the set of addresses. The binary executable software program is then executed to dynamically identify other executable modules so that the set of addresses can be further extended.

39 Claims, 5 Drawing Sheets



DISCOVERING CODE AND DATA IN A BINARY EXECUTABLE PROGRAM

FIELD OF THE INVENTION

The present invention generally relates to a method and system for identifying code and data portions of a binary executable program, and more specifically, to identifying the code and data in a binary executable program in which the code comprises arbitrary (i.e., variable or fixed) length machine instructions.

BACKGROUND OF THE INVENTION

Computer programs are typically written in a high level language or assembly language. The high level language listing of a computer program provides complete information about the program and its algorithms, and is easily read and understood by other programmers. In the process of producing an executable computer program in a high level language that is not interpreted, the high level language program instructions are first compiled into a relocatable object file, which is a binary version of the program. While an object file is not readable in the same sense as the high level language program, object files contain significant information that allows them to be understood and processed by other programs. For example, object files typically contain symbol definitions, types, and names for every function or global variable used in the program; these definitions, types, and names indicate whether the referent of the symbol is code or data. An object file may also contain debugging information relating the instructions and data in that file to source language constructs. It is thus relatively straightforward to process an object file to determine its components, based on the defining information provided in the object file.

In the final step to produce a distributable software program, the compiled object files of the program are linked into a binary executable program. In contrast to object files, a binary executable software program contains only a very small subset of the defining information contained in the corresponding object file(s). For example, a binary executable software program will have definitions only for functions and global variables explicitly exported by that program. The defining information in a binary executable software program does not include internal branch targets, includes only a subset of the functions and global variables, and does not provide any type information. In particular, a binary executable software program does not include any mechanism that distinguishes between code and data components.

Software programs are distributed in the form of binary executables because this is the format in which the program will be loaded and executed on a computer (to implement the functions defined by the program), and in part, in order to obscure many of the details of the program. Some binary executables are more difficult than others to understand, such as those targeted for the Intel Corporation's "x86" architecture, i.e., programs written to employ machine instructions that execute on the family of processors identified by the x86 suffix, such as the 80386, 80486, 80586 (or PENTIUM), etc. Because x86 machine instructions are not of a fixed length, an instruction for this family of processors can potentially start on any arbitrary byte boundary, making it extremely difficult to differentiate code portions from data portions, in contrast to reduced instruction set computer (RISC) processors, such as the Digital Equipment Corporation's ALPHA processors, for which the differentiation between code and data in an executable file is more straight-

forward. However, the need frequently arises, for reasons of analysis, performance evaluation, security, or error checking, to examine a binary executable software program (through software means), to understand its structure, and possibly to introduce changes, producing a modified binary executable software program that is related to the original program, because it provides the same functions, but is also able to provide additional functionality or operate more efficiently. Accordingly, it will be apparent that a method for determining the structure of arbitrary instruction length (e.g., based on x86 architecture) binary executable software programs is required in order to satisfy such needs. Currently, a solution to this problem does not appear to exist in the prior art.

SUMMARY OF THE INVENTION

In accord with the present invention, a method is defined for automatically identifying code portions and data portions of a binary executable software program, in which the code portions comprise machine instructions that are of arbitrary length. The method includes the step of determining a set of addresses in the binary executable software program that are for any known code portions and for any known data portions. Machine instructions at a starting address for each known code portion are disassembled, to identify a set of all possible control flow paths reachable from the starting address. From the control flow paths that are thus identified, a set of target addresses is determined so as to identify other code portions and other data portions. Beginning with bytes of the binary executable software program that are located at any address that could be a starting point for either a code portion or a data portion, the bytes from that point are analyzed to determine if they comprise a code portion. Addresses in the binary executable software program that have not yet been identified as being for code portions and for data portions are then reiteratively processed by repeating the previous steps to identify other code portions and data portions in the binary executable software program. This repetitive process continues until no further code portions and data portions are identifiable.

The binary executable software program may comprise a plurality of executable modules. If so, the step of determining the set of addresses in the binary executable software program that are for any known code portions and for any known data portions includes the step of identifying the plurality of executable modules. An executable module from the plurality of executable modules may include both code and data portions. The method then further includes the step of statically determining and adding an entry address for each of the plurality of executable modules to the set of addresses. Also, any addresses for code portions and data portions that are referenced and identified by any debug address, any export address, and any relocation address within the binary executable software program are added to the set of addresses. The method also includes the step of executing the binary executable software program to dynamically identify other executable modules of the plurality of executable modules while the binary executable software program is running. In this case, the method provides for determining and adding an entry address for each of the other executable modules. Similarly, any addresses for code portions and data portions, which are referenced and identified by any debug address, any export address, and any relocation address within the binary executable software program, are also added to the set of addresses.

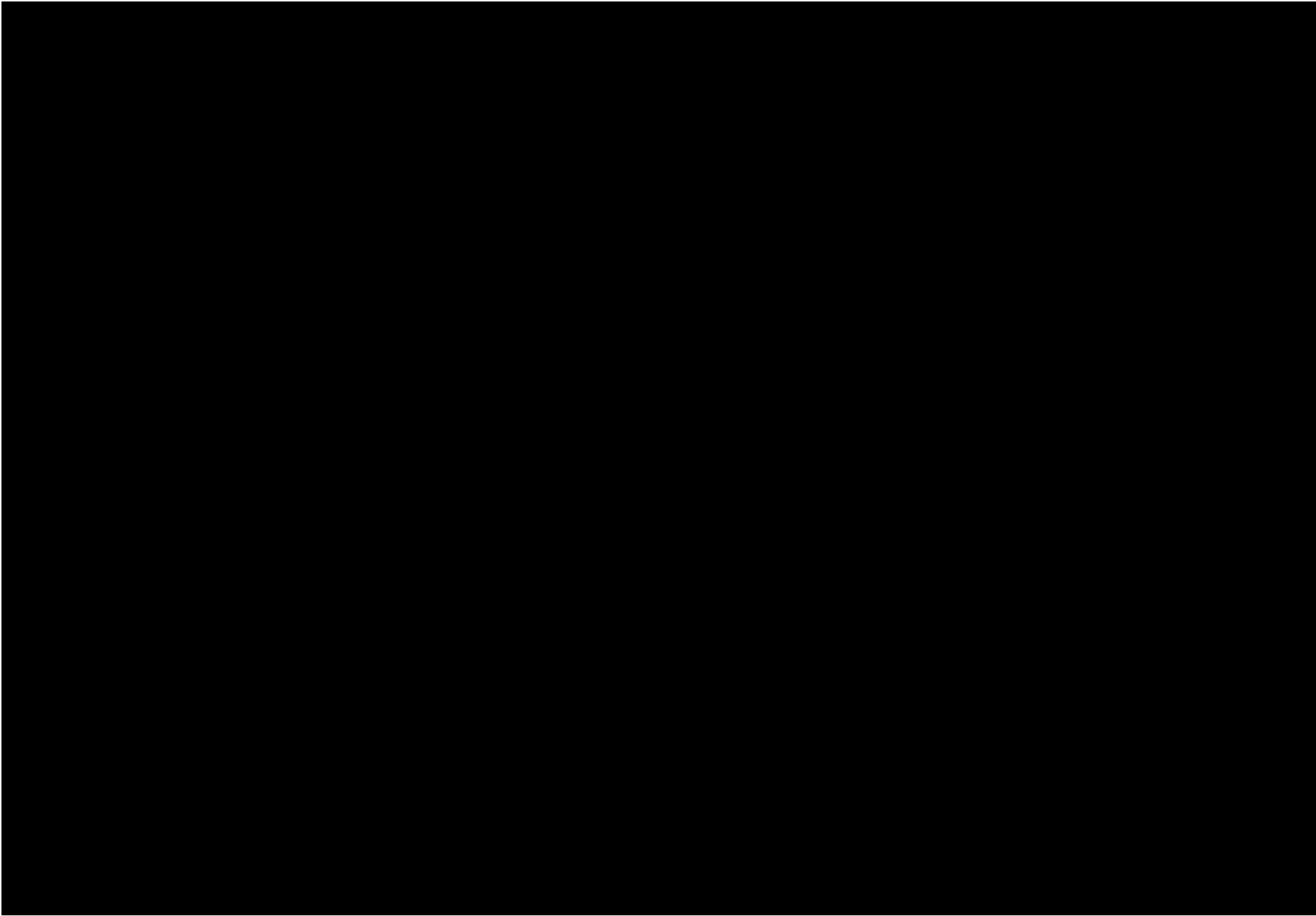
The method preferably further comprises the step of removing any addresses for data portions that have been

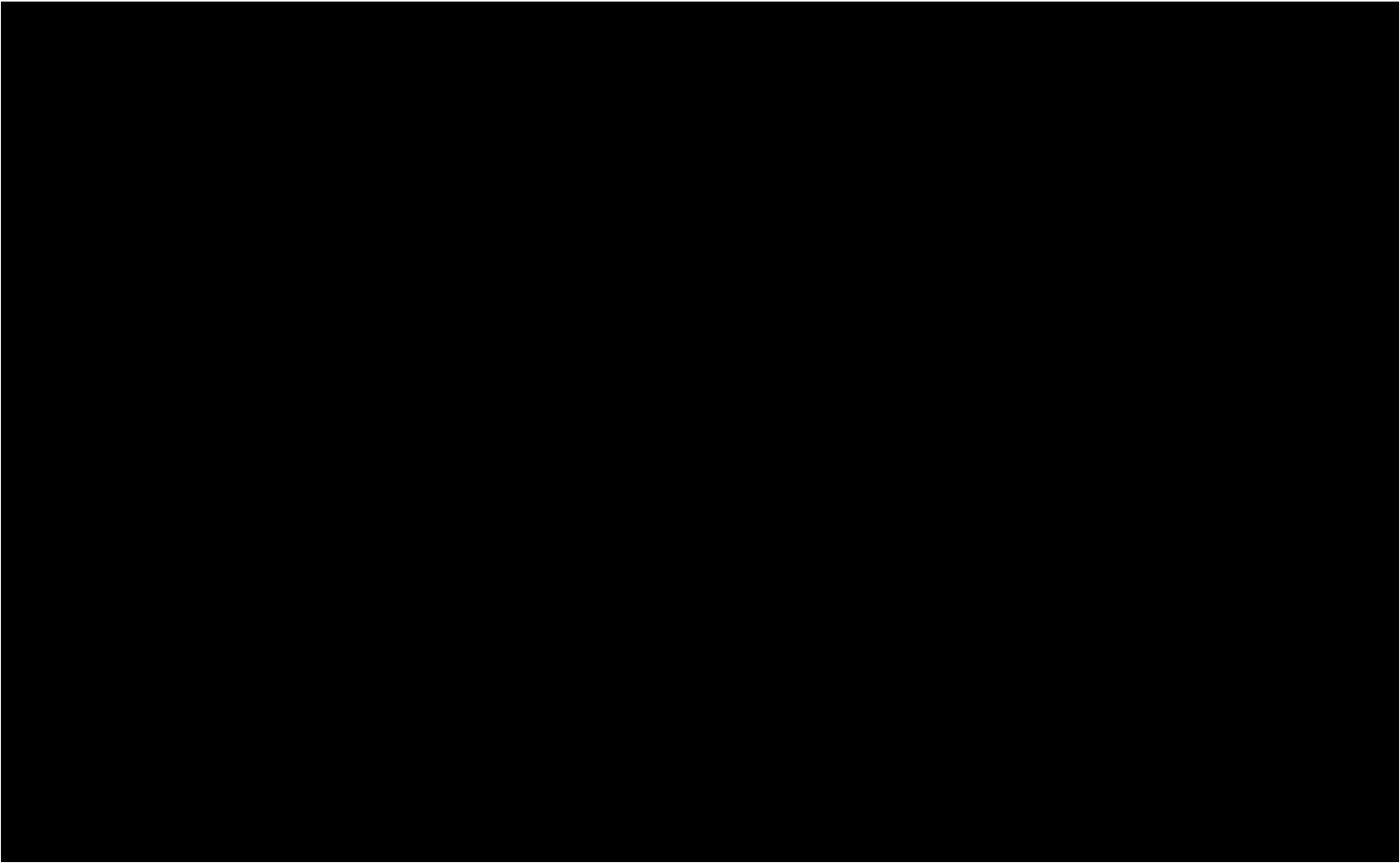
Exhibit O



[Redacted header text]

[Redacted main body text]





[Redacted header text]

[Redacted main body text]

Exhibit P

VisionMobile's "100 Million Club"



The 100 million club is the watchlist of software companies whose products have been embedded on more than 100 million mobile (cellular) handsets.

Updated semi-annually, the 100 million club celebrates software businesses who have succeeded in establishing a significant share in the mobile handset market.

Despite the apparent opportunity in the one-billion-a-year handset market, very few software companies have managed to overcome the commercial and technical challenges inherent in the mobile industry.

Based on our research, only 30 products from 24 companies have shipped on more than 100 million handsets up to the first half of 2009. These figures underscore the complexity of the mobile market, at a time when over 7.5 billion handsets have shipped as of the end of H1 09.

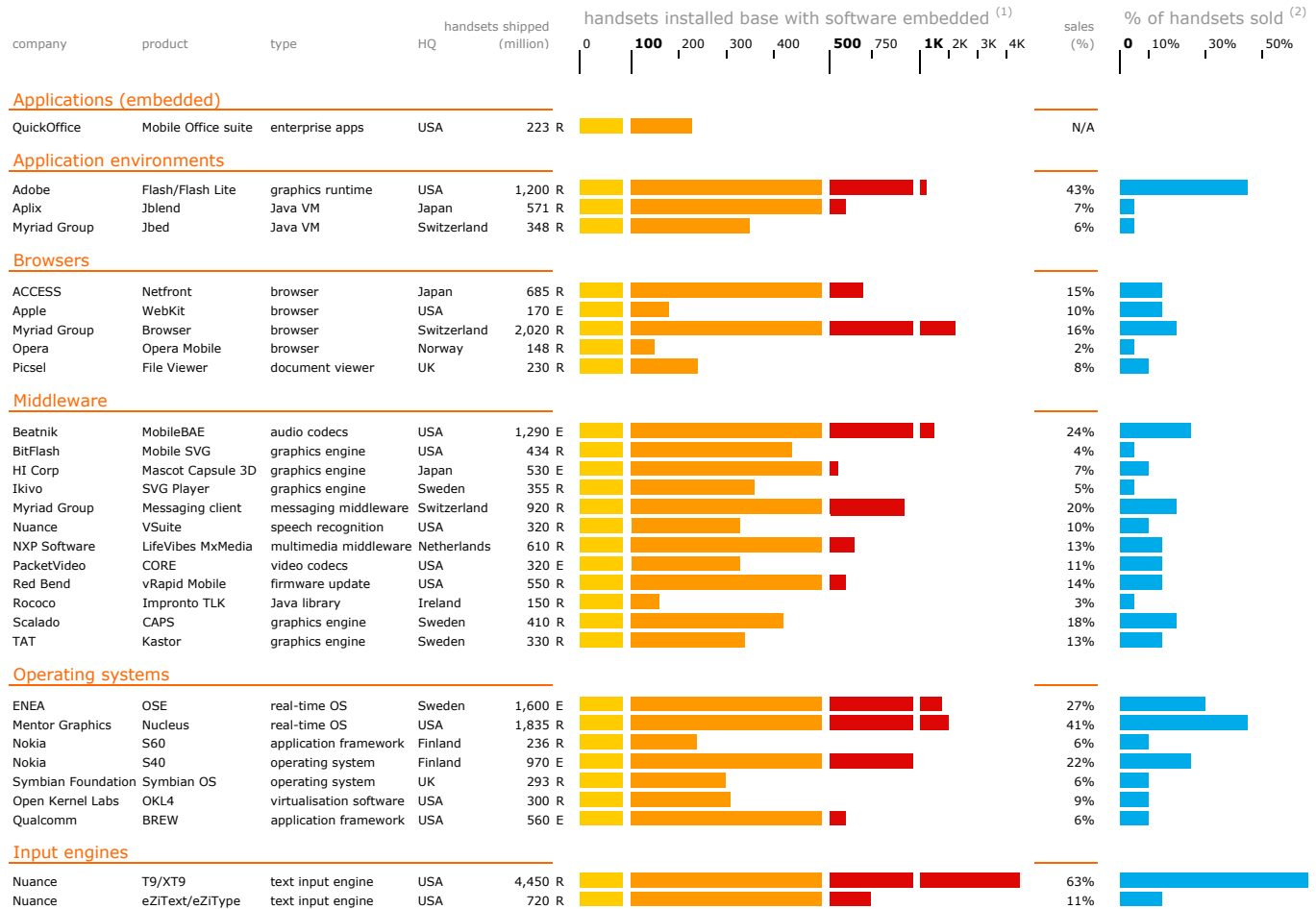
The companies featured in our 100 million club develop and license embedded software for mass-market phones, with products ranging from text input engines to application suites. These companies come from diverse backgrounds, with annual revenues ranging from \$10 million to over \$10 billion. Out of the 24 companies in our 100 million club, 11 are US-based, 6 are headquartered in the Nordics and 3 in Japan.

For more analysis and insights on the 100 million club visit www.100millionclub.com or email us at info@visionmobile.com

About VisionMobile

VisionMobile is an industry analysis firm focusing in mobile software and services. We offer competitive landscape analysis, industry maps, on-site training and due diligence on under-the-radar market sectors.

See also: *Industry Atlas*, the visual who's who of the mobile industry spanning 1,100 vendors.



sources: vendor data (R), VisionMobile estimates (E)

Research notes

- (1) Handset shipments refer to total licensed units, which have been pre-loaded on handsets shipped by the end of 1H09.
- (2) The % of handsets sold refer to the penetration of each software product in the base of handsets sold during 1H09.
- (3) The sum of shipments for all 100 million club members is 22 Billion, whereas only 7.5 Billion have been sold up to 1H09. This indicates that more the three 100 million club members' products exist within the same device, on average.
- (4) Our watchlist does not include ARM, InnoPath, Sony Ericsson JP, Sun's KVM and Tanla who did not disclose shipments.

Published in December 2009. Copyright VisionMobile. Some rights reserved. Licensed for use under a Creative Commons Attribution No Derivatives 3.0 license. VisionMobile believes the statements contained in this document to be based upon information that we consider reliable at the time of publication. The 100 million club is based on an original concept by Morten Grauballe.



CERTIFICATE OF SERVICE

I hereby certify that this document filed through the ECF system will be sent electronically to the registered participants as identified on the Notice of Electronic Filing (NEF) and paper copies will be sent to those indicated as non-registered participants on March 24, 2010.

By: */s/ Jennifer C. Tempesta*

Jennifer C. Tempesta