MY FIRST COREWAR BOOK
by Steven Morrell

PREFACE

This book is an introductory collection of corewar warriors with commentary. It assumes an acquaintance with the ICWS ’88 redcode language (See M.Durham’s tutorial.1 and tutorial.2 for details). Unless otherwise noted, all redcode is written in ICWS ’88 and is designed for a coresize of 8000, process limit 8000. All documents referred to in this text are available by anonymous FTP at ftp.csua.berkeley.edu in one of the subdirectories of pub/corewar/.

After a brief introduction, each chapter presents warriors by subject. I then pontificate on the merits of these various warriors and give some hints for successful implementation. I mention credits and give references to other warriors worth further investigation. Unless otherwise indicated, these warriors are archived in warrior10.tar in the redcode/ directory.

The presentation of each warrior follows roughly the same format. First, the parameters of the warrior are given. These include the name, author, attack speed, effective size, durability, and effectiveness, and score against the Pizza Hill. The effective size is the size of the executing code during the attack phase, taking into account regenerative code. Next, self-contained source code is given, followed by a brief description of the warrior. Finally, a detailed technical description of how the warrior runs is given.

I hope that this helps. If you have questions or comments, send them to morrell@math.utah.edu, where you can reach me until June,1994.

Steven Morrell

Chapter 1: Imp-Rings

On October 14, 1992, A.Ivner posted a warrior that revolutionized the game of corewar. "The IMPire strikes back" scored about 170 on the Intel hill and only suffered 10% losses, putting it firmly in first place. A.Ivner had invented a way to kill other programs with imps -- the world’s first imp-ring. D.Nabutovsky improved the launch code a bit by making an imp-spiral and adding a stone in his "Impressive", which lost only 2% and scored 195 when it started on the hill (for more information on stones, see chapter 2). Since that time, most warriors on the hill have either been imps or something hostile to imps.

This chapter deals with imps, from the basic imp proposed by A.K.Dewdney in the original Scientific American articles to the modern-day imp-spiral we see as a component of many successful warriors.

--1--

```
Name:           Wait
Speed:          None
Size:           1
Durability:     Strong
Effectiveness:  None
Score:

wait JMP wait
end wait
```

Wait is the simplest warrior. Its small size makes it difficult to locate. However, it has no attack, so it only wins if the enemy program self-destructs. We shall be using this program for fodder.

--2--

```
Name:           Imp
Author:         A.K.Dewdney
Speed:          100% of c (sequential)
Size:           1
Durability:     Strong
Effectiveness:  Poor
Score:

imp MOV imp, imp+1
end imp
```

Imp presents the enemy with a small, moving target that will not die without a direct hit. It ties a lot, and is vulnerable to the imp-gate. (See program 3)

HOW IT WORKS: When Imp is loaded and before it executes, it looks like this:

```
MOV 0,1  (1)
```

(The (1) shows which instruction will execute on the first cycle.) When process (1) executes, it first copies its instruction to the next address and then moves to the next instruction:

```
MOV 0,1       ;This is the original.
MOV 0,1  (2)  ;This is the copy.
```

Process (2) now executes. Since all addressing is relative, the process copies its instruction to the next address and advances.

```
MOV 0,1
MOV 0,1
MOV 0,1  (3)  ;This is the second copy.
```

And so it goes, overwriting anything in its path with MOV 0,1 instructions. So when it encounters enemy code, it replaces the enemy code with MOV 0,1 instructions, turning the enemy processes into imps. Note that although the enemy code is gone, the enemy processes live on, so imps do not win unless the enemy code self-destructs.

--3--

```
Name:           Imp Gate
Speed:          None
Size:           1
Durability:     Strong
Effectiveness:  Excellent against imps, Extremely Poor against others
Score:

gate equ wait-10
wait JMP wait,<gate
end wait
```

Imp Gate waits and destroys imps that happen to pass 10 instructions before it. It is seldom overrun by imps and its small size makes it difficult to locate. The imp gate is defensive by nature, and will not win against a stationary enemy unless this enemy self-destructs.

HOW IT WORKS: The process running at _wait_ jumps to the A-value of this command, i.e. back to _wait_. However, it also decrements the B-field of _gate_. Thus, the B-field of _gate_ is decremented every turn. When an enemy imp comes by this is what happens:

```
MOV 0,1   (x)  ;here comes the imp
DAT 0,-5       ;here is the gate

The imp copies itself and advances onto the gate:

MOV 0,1
MOV 0,1   (x+1)  ;here is the gate
```

The gate decrements:

```
MOV 0,1
MOV 0,0   (x+1)  ;here is the gate
```

The imp copies this instruction to itself (effectively doing nothing) and advances, falling off the end:

```
MOV 0,1
MOV 0,0        ;here is the gate
          (x+2)
```

The gate decrements again (but the damage has already been done.)

```
MOV 0,1
MOV 0,-1       ;here is the gate
          (x+2)
```

The enemy process executes an illegal instruction and dies.

--4--

```
Name:           Worm
Speed:          25% of c (linear)
Size:           1.75
Durability:     Very Strong
Effectiveness:  Poor
Score:

launch SPL b
       SPL ab
```

```
aa     JMP imp
ab     JMP imp+1
b      SPL bb
ba     JMP imp+2
bb     JMP imp+3
imp    MOV imp,imp+1
end launch
```

Worm is a symbiotic collection of imps. The only vulnerable parts of the worm is the tail instruction and the instruction about to execute, hence the effective size of 1.75 (25% of the time, the tail instruction is the instruction about to execute.) It is very difficult to kill, because each imp must be disposed of individually. However, it is still vulnerable to imp gates. As with Imp, Worm overwrites enemy code but preserves enemy processes.

HOW IT WORKS: First, we launch the worm using a binary launch:

```
SPL 4,0  (1)
SPL 2,0
JMP 5,0
JMP 5,0
SPL 2,0
JMP 4,0
JMP 4,0
MOV 0,1
```

The first process splits into processes (2) and (3):

```
SPL 4,0
SPL 2,0  (2)
JMP 5,0
JMP 5,0
SPL 2,0  (3)
JMP 4,0
JMP 4,0
MOV 0,1
```

Process (2) splits into processes (4) and (5):

```
SPL 4,0
SPL 2,0
JMP 5,0  (4)
JMP 5,0  (5)
SPL 2,0  (3)
JMP 4,0
JMP 4,0
MOV 0,1
```

Process (3) splits:

```
SPL 4,0
SPL 2,0
JMP 5,0  (4)
JMP 5,0  (5)
SPL 2,0
JMP 4,0  (6)
JMP 4,0  (7)
MOV 0,1
```

Process (4) jumps:

```
SPL 4,0
SPL 2,0
JMP 5,0
JMP 5,0  (5)
SPL 2,0
JMP 4,0  (6)
JMP 4,0  (7)
MOV 0,1  (8)
```

Processes (5), (6) and (7) jump:

```
SPL 4,0
SPL 2,0
JMP 5,0
JMP 5,0
SPL 2,0
JMP 4,0
JMP 4,0
MOV 0,1  (8)
         (9)
         (10)
         (11)
```

The worm will now start crawling though memory. Note that if processes (9), (10) or (11) executed right now, they would execute an illegal instruction and die. But process (8) executes, copying the MOV instruction to where process (9) is going to execute:

```
SPL 4,0
SPL 2,0
JMP 5,0
JMP 5,0
SPL 2,0
JMP 4,0
JMP 4,0
MOV 0,1
MOV 0,1  (9) (12)
         (10)
         (11)
```

Now process (9) executes, copying the MOV instruction to process (10).

```
SPL 4,0
SPL 2,0
JMP 5,0
JMP 5,0
SPL 2,0
JMP 4,0
JMP 4,0
MOV 0,1
MOV 0,1  (12)
MOV 0,1  (10) (13)
         (11)
```

And after (10) and (11) have executed, the worm has crawled forward an instruction, leaving a slimy MOV 0,1 trail behind.

```
SPL 4,0
SPL 2,0
JMP 5,0
JMP 5,0
SPL 2,0
JMP 4,0
JMP 4,0
MOV 0,1
MOV 0,1  (12)
MOV 0,1  (13)
MOV 0,1  (14)
MOV 0,1  (15)
```

--5--

```
Name:           Ring
Speed:          100% of c (mostly linear)
Size:           1
Durability:     Average
Effectiveness:  Fair
Score:

c      JMP imp-2666
launch SPL c
       SPL imp+2667
imp    MOV 0,2667
end launch
```

Ring is a symbiotic collection of three imps distributed through core. It has the capability to destroy enemy processes it overruns, if the enemy is running only one or two processes. This code will run correctly only in a coresize of 8000, although the constants may be tweaked to run in any coresize not divisible by 3. Ring is an example of a 3-pt imp.

HOW IT WORKS: The launching code is a very small binary startup:

```
JMP -2663,   0
SPL     0,   0  (1)
SPL  2668,   0
MOV     0,2667
```

The first process splits:

```
JMP -2663,   0  (3)
SPL     0,   0
SPL  2668,   0  (2)
MOV     0,2667
```

The second process splits:

```
JMP -2663,   0  (3)
SPL     0,   0
SPL  2668,   0
MOV     0,2667  (4)
...
                (5)  ;this location is 2667 instructions after the imp
```

The third process jumps:

```
MOV 0,2667  (4)
...
            (5)  ;this location is 2667 instructions after the imp
...
            (6)  ;this location is 2667 instructions after process (2)
```

Now the fun begins. Process (4) executes, copying the imp instruction to process (5) and becoming process (7):

```
MOV 0,2667
            (7)
...
MOV 0,2667  (5)
...
            (6)
```

(5) executes, copying the imp instruction to process (6):

```
MOV 0,2667
            (7)
...
MOV 0,2667
            (8)
...
MOV 0,2667  (6)
```

And now (6) executes, copying the imp instruction back to process (7):

```
MOV 0,2667
MOV 0,2667  (7)
...
MOV 0,2667
            (8)
...
MOV 0,2667
            (9)
```

The cycle starts all over again, and the ring creeps forward.

Let’s see what happens when Ring fights Wait (Program 1). Wait executes JMP 0,0 until eventually Ring overwrites this instruction with MOV 0,2667.

```
MOV 0,2667  (1)
```

Wait executes this instruction and advances:

```
MOV 0,2667
            (2)
```

Since Ring takes 3 cycles to move the next command into place, Wait’s process now executes an illegal instruction and dies.

So Ring slowly advances through core, and if the enemy is running a single process, it falls off the end of the imp ring.

--6--

```
Name:           Spiral
Speed:          37.5% of c (mostly linear)
Size:           1.875
Durability:     Very Strong
Effectiveness:  Fair
Score:

step equ 2667
launch  SPL 8
        SPL 4
        SPL 2
        JMP imp
        JMP imp+step
        SPL 2
        JMP imp+(step*2)
        JMP imp+(step*3)
        SPL 4
        SPL 2
        JMP imp+(step*4)
        JMP imp+(step*5)
        SPL 2
        JMP imp+(step*6)
        JMP imp+(step*7)
imp     MOV 0,step
end launch
```

Spiral crosses the durability of a worm with the effectiveness of a ring. Spiral is resistant to most conventional attacks, and since it is an 8-process imp-ring, it kills any enemy it overwrites if the enemy has less than 8 processes running. The only vulnerable parts of the spiral are the tail and the process that is currently running. Spiral is vulnerable to imp gates, however.

HOW IT WORKS: After a binary launch, the processes are arranged as follows:

```
MOV 0,2667  (16)
            (19) ;this process is 2667 instructions after process (18)
            (22)
...
            (17) ;this process is 2667 instructions after process (16)
            (20)
            (23)
...
            (18) ;this process is 2667 instructions after process (17)
            (21)
```

Now the spiral worms along: (16) copies the imp to (17), which copies it to (18), and so on. All the processes advance 1 instruction as this happens, and then the imp-passing instructions begin again.

A step-by step analysis of how imp gates destroy spirals would be lengthy and unnecessarily complicated. The key idea is this: The imp gate is constantly being modified. As the imp overruns the imp gate, no imp instructions are left intact to copy to the next processes' location. This next process executes an illegal instruction and dies. This scenario repeats until the entire spiral moves through the imp gate and disintegrates.

--7--

```
Name:           Gate Crashing Spiral
Speed:          12.5% of c (mostly linear)
Size:           5.875
Durability:     Very Strong
Effectiveness:  Good
Score:

step1 equ 2667
step2 equ 2668
start    SPL lnch1
         SPL lnch3

lnch2    SPL 8
         SPL 4
         SPL 2
         JMP imp2+(step2*0)
         JMP imp2+(step2*1)
         SPL 2
         JMP imp2+(step2*2)
         JMP imp2+(step2*3)
         SPL 4
         SPL 2
         JMP imp2+(step2*4)
         JMP imp2+(step2*5)
         SPL 2
         JMP imp2+(step2*6)
         JMP imp2+(step2*7)

lnch3    SPL 8
         SPL 4
         SPL 2
         JMP imp3+(step2*0)
         JMP imp3+(step2*1)
         SPL 2
         JMP imp3+(step2*2)
         JMP imp3+(step2*3)
         SPL 4
         SPL 2
         JMP imp3+(step2*4)
         JMP imp3+(step2*5)
         SPL 2
         JMP imp3+(step2*6)
         JMP imp3+(step2*7)

lnch1    SPL 8
         SPL 4
         SPL 2
         JMP imp1+(step1*0)
         JMP imp1+(step1*1)
         SPL 2
         JMP imp1+(step1*2)
         JMP imp1+(step1*3)
         SPL 4
         SPL 2
         JMP imp1+(step1*4)
         JMP imp1+(step1*5)
         SPL 2
         JMP imp1+(step1*6)
         JMP imp1+(step1*7)
```

```
imp1     MOV 0,step1
         DAT #0
         DAT #0
         DAT #0
imp2     MOV 0,step2
         MOV 0,step2
imp3     MOV 0,step2
         MOV 0,step2
end start
```

Gate Crashing Spiral is a collection of three spirals that work together to kill imp gates. The first is a standard imp spiral and the other two are slightly modified, interleaved for greater protection against split bombs. The large size of its launch code makes it vulnerable to fast attacks.

HOW IT WORKS: Each spiral has its own binary launch. The first spiral launches first and crawls forward an instruction by the time the other two spirals have launched. Core then looks like this (after resetting the counter for clearer exposition):

```
MOV 0,2667       ;This is label imp1 | MOV 0,2667       | MOV 0,2667
MOV 0,2667  (17)                     | MOV 0,2667  (18) | MOV 0,2667  (19)
MOV 0,2667  (20)                     | MOV 0,2667  (21) | MOV 0,2667  (22)
DAT #0,#0   (23)                     |             (24) |
MOV 0,2668   (1) ;This is label imp2 |                  |
MOV 0,2668                           |              (2) |
MOV 0,2668   (9) ;This is label imp3 |                  |              (3)
MOV 0,2668                           |             (10) |
             (4)                     |                  |             (11)
                                     |              (5) |
            (12)                     |                  |              (6)
                                     |             (13) |
             (7)                     |                  |             (14)
                                     |              (8) |
            (15)                     |                  |
                                     |             (16) |
```

The imps then move forward via the usual instruction juggling.

When a gate crashing spiral overruns a gate, the second or third spirals hit first:

```
MOV 0,2668  (x) ;imp gate here
```

The gate decrements:

```
MOV 0,2667  (x)
```

The wounded spiral copies this instruction 2667 ahead:

```
MOV 0,2667
            (x+24)
...
MOV 0,2667
```

The second and third spirals now fall off the end and die, and then the first spiral hits the gate:

```
MOV 0,2667  (y) ;imp gate here
...
MOV 0,2667  (y+1)
```

The gate decrements:

```
MOV 0,2666  (y)
...
MOV 0,2667  (y+1)
```

Process (y) executes, and can't copy the imp to process (y+1), but this is okay, because process (y+1) executes the imp instruction from the two spirals gone before. The spiral crawls through the gate and goes on to kill the enemy processes.

--8--

```
Name:           Nimbus Spiral
Speed:          50% of c (somewhat linear)
Size:           1.992
Durability:     Very Strong
Effectiveness:  Fair
Score:

step equ 127
imp    MOV 0,step
launch SPL 1     ;1 process
       SPL 1     ;2 processes
       SPL 1     ;4 processes
       SPL 1     ;8 processes
       SPL 1     ;16 processes
       MOV -1,0  ;32 processes
       SPL 1     ;63 processes
       SPL 2     ;126 processes
spread JMP @spread,imp
       ADD #step,spread
end launch
```

Nimbus Spiral launches a 63-point spiral with two processes per point. Because a binary launch would exceed the 100-instruction limit, Nimbus Spiral uses what is called a Nimbus-type launch. The code for this type of launch is obviously smaller, but the time it takes to launch spirals is roughly doubled.

HOW IT WORKS: Each SPL 1 command doubles the number of processes acting in tandem at the next instruction. The first process that executes the MOV -1,0 command does not split, but all subsequent processes execute a SPL 1 command. Hence, before execution of the SPL 2 command, core looks like this (with counter reset):

```
MOV 0,127
SPL 1,0
SPL 1,0
SPL 1,0
SPL 1,0
SPL 1,0
SPL 1,0
SPL 1,0
SPL 2,0      (1)-(126)
JMP @0,-9
ADD #127,-1
```

After execution of the SPL 2 command:

```
MOV 0,127
SPL 1,0
SPL 1,0
SPL 1,0
SPL 1,0
SPL 1,0
SPL 1,0
SPL 1,0
SPL 2,0
JMP @0,-9    Odd processes
ADD #127,-1  Even processes
```

We reset the processes again. Process (1) now executes, jumping to the location of the B-operand of the JMP instruction:

```
MOV 0,127    (253)   ;this came from process (1)
SPL 1,0
SPL 1,0
SPL 1,0
SPL 1,0
SPL 1,0
SPL 1,0
SPL 1,0
SPL 2,0
JMP @0,-9    Odd processes greater than 1
ADD #127,-1  Even processes
```

Process (2) now executes, adding 127 to the B-operand of the JMP instruction:

```
MOV 0,127    (253)
SPL 1,0
SPL 1,0
SPL 1,0
SPL 1,0
SPL 1,0
SPL 1,0
SPL 1,0
SPL 2,0
JMP @0,118   Odd processes greater than 1
ADD #127,-1  Even processes greater than 2
             (254)                ;this came from process (2)
```

And it continues. Process (3) jumps to a new location. The even processes modify the jump vector, and the odd processes do all of the jumping. By the time process (127) is ready to execute, we have the following situation:

```
MOV 0,127    (253)
SPL 1,0      (379)
SPL 1,0
SPL 1,0
SPL 1,0
SPL 1,0
SPL 1,0
SPL 1,0
SPL 2,0
JMP @0,-134
ADD #127,-1
             Even processes
...
Odd processes broadcast throughout core
```

The odd processes form an imp spiral and the even processes execute illegal instructions and die, leaving just the spiral to crawl through memory.

--Conclusion--

Two questions beg to be answered: When should you add an imp to your favorite warrior, and how do you kill imps?

Most of today's fighters have some resistance to imps, so pure imp programs seldom are successful. But imps are easy to add to code that has multiple processes running, like today's stones, vampires, or paper. The most successful imp warriors use most of their process time in a more conventional attack, and rely on the imp-ring as a backup. Whether an imp is a good idea in your program depends on the program; you may lose less, but you may win less. About the only thing you can be sure of is tying more. But testing your warrior always helps.

Killing imps is difficult, but not impossible. Imp gates work well against most imps, but should only be executed after the rest of your code has done its stuff. Imp gates of the form

```
SPL 0,<gate
DAT <gate,<gate
```

can sometimes kill even gate-crashing imps. Fast bombing programs can occasionally catch the launching code before it has completed, especially with fancier imps. Code with a long enough bombing run (e.g. Charon v8.1) can hit and destroy all the imp instructions if it is done right. Dropping a single MOV 0,<1 bomb on the tail (or vulnerable instruction soon after the tail) of an imp-ring will kill the entire ring off. Dropping a MOV <2667,<5334 instruction on a 3-point imp ring can kill as many as 9 imp instructions, and is extremely effective in a stream (which is sequential bombing of memory). Some programs use an imp trap tailor-made for stunning imp-rings by dropping SPL 0 bombs on the imp-ring using a step size of 2667, so that the ring is attacked from the tail forward.

An enhancement to the imp-launching routines is to add decrement statements to all the b-fields of the SPL and JMP commands. If you have a large binary launch, for example, you could decrement 63 instructions throughout core for free. Most of the original code I have based this chapter on has such b-fields.

Here is a list of imp-style programs worth investigating. Unless otherwise noted, they can be found in warrior10.tar in the 88 directory. Imp-stone combos will be listed in the back of chapter 2.

"The IMPire strikes back" by Anders Ivner (impire)
"Trident" by Anders Ivner (trident)
"Nimbus 1.2" by Alex MacAulay (nimbus12)
"Imps! Imps! Imps!" by Steven Morrell (contact morrell@math.utah.edu)

Program 2, Imp, was written by A.K. Dewdney for his Scientific American articles.

Program 3, Imp Gate, was suggested in its current form by B.Thomsen, and is often called a wimp in the literature.

Program 5, Ring, was stolen and modified from a _Push Off_ article from P.Kline, but it looks suspiciously like A.Ivner’s "Trident."

Program 7, Gate Crashing Spiral, was stolen and modified from P.Kline’s "Cannonade."

Program 8, Nimbus Spiral, was stolen and modified from A.MacAulay’s "Nimbus 1.2."

Follow this link to Chapter 2.

MY FIRST COREWAR BOOK

Chapter 2: Stones

If you are fast and small, you can find the enemy before the enemy finds you. This is the philosophy of pattern bombers, a group of warriors much maligned by frustrated corewar enthusiasts trying to make intelligent warriors. But the fact remains, frenzied maniacs can often kill the slow brooding kind.

Pattern Bombers are also refered to as stones, as part of the stone - scissors - paper analogy. Scissors, which includes vampires and scanners, are bigger than stones and therefore tend to get beat up by them more often. Paper, also known as a replicator, is a program that makes copies of itself throughout the core faster than a pattern bomber can destroy all of them. Stones are thus ineffectual against paper, or at least they were until W. Mintardjo stuck a two-pass core-clear on one of his stones.

--1--

```
Name:           Dwarf
Author:         A.K.Dewdney
Speed:          33.33% of c
Size:           4
Durability:     Weak
Effectiveness:  Average
Score:

bomb   DAT #0
dwarf  ADD #4, bomb
       MOV bomb, @bomb
       JMP dwarf
end dwarf
```

Dwarf bombs every fourth instruction with DAT instructions in hopes that enemy processes will execute this code and die. Since 4 divides coresize, Dwarf will never drop a bomb on itself. Because Dwarf only hits every fourth instruction, it is a mod-4 bomber.

HOW IT WORKS: Before anything executes, core looks like this:

```
DAT #0, #0          ;bomb
ADD #4, -1    (1)
MOV -2, @-2
JMP -2, 0
```

Then process (1) adds 4 to the B-field of bomb:

```
DAT #0, #4          ;bomb
ADD #4, -1
MOV -2, @-2   (2)
JMP -2, 0
```

Process (2) moves bomb 4 instuctions forward, where the B-field of bomb points to:

```
DAT #0, #4          ;bomb
ADD #4, -1
MOV -2, @-2
JMP -2, 0     (3)
DAT #0, #4
```

Process (3) simply makes the program loop back to the beginning.

```
DAT #0, #4          ;bomb
ADD #4, -1    (4)
MOV -2, @-2
JMP -2, 0
DAT #0, #4
```

Process (4) adds 4 to the B-field of bomb:

```
DAT #0, #8          ;bomb
ADD #4, -1
MOV -2, @-2   (5)
JMP -2, 0
DAT #0, #4
```

Process (5) drops the next bomb where the B-field of bomb is pointing.

```
DAT #0, #8
ADD #4, -1
MOV -2, @-2
JMP -2, 0     (6)
DAT #0, #4



DAT #0, #8
```

Process (6) loops back, and bomb after bomb are dropped forward through core.

--2--

```
Name:           Stone
Author:         Matthew Householder
Speed:          33.34% of c
Size:           4
Durability:     Weak
Effectiveness:  Average
Score:

start MOV <2, 3
      ADD d1, start
      JMP start
      DAT #0
d1    DAT #-5084, #5084
end start
```

Stone is a mod-4 bomber like Dwarf, but with two important improvements. First, the step-size has been increased somewhat for better distribution of bombs against larger opponents. Second, Stone decrements other adresses while it bombs. Decrementing opponent’s code may wound it so that DAT bombs can destroy it later.

HOW IT WORKS: Pre-decrement indirect addressing can be tricky, so we shall use the intuitive approach, even though it yields wrong results for weird instructions like MOV <0,<1. See "tutorial.2" or the ICWS ’94 standard for precise details.

When Stone is loaded, core looks like this below. The DAT #0,#0 instruction is used only as a spacer between the executable code and the other DAT statement, as we shall shortly see.

```
MOV <2, 3          (1)
ADD 3, -1
JMP -2, 0
DAT #0, #0
DAT #-5084, #5084
```

The B-field of the JMP instruction (pointed to by the A-field of the MOV instruction) is decremented, so that it now points to the ADD instruction. This ADD instruction is now moved to the DAT #0,#0 instruction (pointed to by the B-field of the MOV instruction). Core now looks like this:

```
MOV <2, 3
ADD 3, -1          (2)
JMP -2, -1             ;this has been decremented
ADD 3, -1              ;this has been copied
DAT #-5084, #5084
```

This last sequence may be a little misleading, because it looks like we are dropping ADD 3,-1 bombs throughout core. We shall see this is not usually the case.

We now come to the ADD 3,-1 instruction. Since this ADD is not immediate, as it was in Dwarf, the A-operand of the DAT instruction is added to the A-operand of the MOV instruction and the B-operand of the DAT instruction is added to the B-operand of the MOV instruction:

```
MOV <-5082, 5087
ADD 3, -1
JMP -2, -1         (3)
ADD 3, -1
DAT #-5084, #5084
```

The executing process now jumps back (the -1 in the B-field is ignored).

```
MOV <-5082, 5087   (4)
ADD 3, -1
JMP -2, -1
ADD 3, -1
DAT #-5084, #5084
```

Process (4) drops another bomb: the location -5082 behind the MOV instruction is decremented and whatever it points to is moved 5087 in front of the MOV instruction. The pattern continues until someone is killed or time runs out.

Stone, then, doesn’t really drop bombs as such, but rather moves instructions around core in a pseudo-random fashion. But since core is initialized to DAT 0,0, most of the instructions it moves are deadly DAT statements. This process is called transposition in the literature.

--3--

```
Name:           Armadillo
Author:         Stefan Strack
Speed:          32.86% of c
Size:           5
Durability:     Strong
Effectiveness:  Average
Score:

bomb    SPL 0
loop    ADD #3039, ptr
ptr     MOV bomb, 81
        JMP loop
        MOV 1, <-1
end bomb
```

Armadillo drops SPL 0 bombs throughout core to stun the enemy, and then lays down a DAT carpet (also called a core-clear) to kill the enemy. This is one of the earliest bombers that used a core-clear to erase all of memory. It scores 100% wins against Wait (program 1, chapter 1) where Dwarf and Stone only score 25% wins and 75% ties. In my experience, SPL bombs are the most effective single-instruction bomb a warrior can drop. However, SPL bombs don't kill many programs cleanly, don't allow you to simultaneously bomb the rest of the core with decrements, and don't paralyze the opponent as well as the multi-instruction bombs that scanners drop.

Another innovation in Armadillo is the use of a SPL 0 instruction inside the warrior. If any of the other instuctions are hit with DAT bombs, the program may not operate correctly, but the bomb doesn't kill all of the processes. Additionally, this self-splitting code generates enough processes that imps cannot kill Armadillo by themselves.

HOW IT WORKS: When Armadillo is loaded into core, it looks like this:

```
SPL 0, 0       (1)
ADD #3039, 1
MOV -2, 81
JMP -2, 0
MOV 1, <-1
```

Process (1) splits into processes (2) and (3).

```
SPL 0, 0      (3)
ADD #3039, 1  (2)
MOV -2, 81
JMP -2, 0
MOV 1, <-1
```

Process (2) executes and process (3) splits.

```
SPL 0, 0      (6)
ADD #3039, 1  (5)
MOV -2, 3120  (4)
JMP -2, 0
MOV 1, <-1
```

Process (4) drops a split bomb, process (5) changes the bombing location, and process (6) splits.

```
SPL 0, 0      (10)
ADD #3039, 1   (9)
MOV -2, -1841  (8)
JMP -2, 0      (7)
MOV 1, <-1
```

Process (7) jumps back in order to conserve processes, (8) bombs, (9) changes the bombing location, and (10) splits.

```
SPL 0, 0      (15)
ADD #3039, 1  (14) (11)
MOV -2, 1198  (13)
JMP -2, 0     (12)
MOV 1, <-1
```

And so the process continues. The ever-lengthening string of processes executes the code (backwards!) that drops the SPL bombs. Eventually, a SPL 0,0 gets dropped on the JMP statement:

```
SPL 0, 0
ADD #3039, 1
MOV -2, 1
SPL 0, 0      (1)
MOV 1, <-1
```

The loop is broken, and all of the processes fall through to this second SPL instruction eventually. We examine this last bit of code as if there were only one process running at the SPL instruction, since the program doesn't depend on process order from this point on. Process (1) splits:

```
SPL 0, 0
ADD #3039, 1
MOV -2, 1
SPL 0, 0      (3)
MOV 1, <-1    (2)
```

Process (2) decrements the B-field of the SPL instruction (which the SPL instruction doesn't need) and moves the blank (DAT 0,0) instruction to where the SPL instruction points:

```
SPL 0, 0
ADD #3039, 1

SPL 0, -1    (3)
MOV 1, <-1
            (4)
```

Process (3) splits:

```
SPL 0, 0
ADD #3039, 1

SPL 0, -1    (6)
MOV 1, <-1   (5)
             (4)
```

Now process (4) executes an illegal instruction and dies, (5) decrements the SPL instruction again and bombs the next instruction backwards, and (6) splits:

```
SPL 0, 0


SPL 0, -2    (9)
MOV 1, <-1   (8)
             (7)
```

This pattern repeats until eventually the core clear wraps around and erases itself. Just before this erasure occurs, core looks like this:

```
SPL 0, 2     (23997)
MOV 1, <-1   (23996)
             (23995)
```

Process (23995) dies as usual, but this time, when process (23996) bombs, it erases the bombing instruction:

```
SPL 0, 2     (23997)

             (23998)
```

Now, if we ignore all of the dying processes, we see that this SPL command keeps splitting processes to itself, keeping the warrior alive.

--4--

```
Name:           Cannonade Stone
Speed:          24.51% of c
Size:           5
Durability:     Average
Effectiveness:  Good
Score:

      MOV &LT;6, 1
start SPL -1, <5144
      ADD 3, -2
      DJN -2, <5142
      DAT #0, #0
      MOV 190, <-190
end start
```

Cannonade Stone takes the idea of self-splitting code to another level. Altough it bombs somewhat slower than other bombers, it splits off processes so quickly that a stun attack on other components of the warrior will not halt the execution of the stone. The bombing run hits every fifth instruction, with a transposition at every tenth position and a decrement between each transposition. Additionally, a DJN-stream is laid through memory, giving another form of attack without increasing the size or speed of the program. At the end of the bombing run, Cannonade Stone converts into a core-clear and partial imp-gate.

HOW IT WORKS: When Cannonade Stone is first loaded into memory, it looks like this:

```
MOV <6, 1
SPL -1, <5144   (1)
ADD 3, -2
DJN -2, <5142
DAT #0, #0
MOV 190, <-190
```

Process (1) splits:

```
MOV <6, 1       (3)
SPL -1, <5144
ADD 3, -2       (2)
DJN -2, <5142
DAT #0, #0
MOV 190, <-190
```

Now processes (2) and (3) execute, adding and then bombing like every other stone.

```
MOV <196, -189
SPL -1, <5144   (5)
ADD 3, -2
DJN -2, <5142   (4)
DAT #0, #0
MOV 190, <-190
```

Process (4) usually jumps back to the SPL instruction (more on this in a moment), and the pattern repeats: each process at the SPL command splits into two processes, which add and bomb in rapid succession.

At the end of the bomb run, the bomber mutates itself into a core-clear. The SPL -1,<5144 instruction is overwritten with the MOV 190,<-190 instruction. The executng portion of code then looks like this:

```
MOV 190, <-190
ADD 3, -2
DJN -2, <5142
```

The first instruction performs the core-clear, the second instruction does nothing of strategic worth, and the third instruction loops processes back to the first instruction. Additionally, the decrement in the MOV command sets up a partial (33%) imp-gate 190 instructions before it, and the decrement in the DJN instruction sets up a second partial (33%) imp gate 2666 instructions before the first one. Since 2667 is the magic number for 3-point imps, these instructions defend the bomber against 3-point imps at roughly 67% efficiency.

Let us examine in more detail how the DJN -2,<5142 instruction works. When it is executed, the predecrement in the B-field decrements the instruction 5142 after the DJN intstruction, which is probably a DAT 0,0 command:

```
DJN -2, <5142
...
DAT 0, -1
```

The DJN instruction now decrements the instruction before that, which probably doesn't have a B-value of 1, so the executing process jumps back to the beginning of the loop:

```
DJN -2, <5142
...
DAT 0, -1        ;this was decremented by the DJN
DAT 0, -1        ;this was decremented by the <
```

The next time the DJN instruction is executed, the B-field 5142 after the instruction is decremented, and so is the instruction pointed by that B-field (2 before it):

```
DJN -2, <5142
...
DAT 0, -1        ;this was decremented by the DJN
DAT 0, -1
DAT 0, -2        ;this was decremented by the <
```

As the DJN instruction is repeatedly executed, a carpet of decrements is laid down backwards through core.

This is not exactly the pattern that is laid down in core, because the SPL -1,<5144 command decrements the same B-field as the DJN instruction does. This adds gaps in the DJN-stream, making it more spread out and liable to hit the enemy program. Additionally, it turns the B-field into a better partial imp-gate.

We have made two assumptions: First, that the instruction 5142 after the DJN instruction is DAT 0,0; second, that the instruction pointed to by that instruction does not have a B-field of 1. If the first assumption fails, the worst that can happen is a non-zero B-field, in which case the DJN stream is laid somewhere else. If the second assumption fails, then the executing process does not jump back and proceeds instead to an illegal instruction. Fortunately, this is just one of many processes, so the bombing loop is not seriously affected. This result may be compounded, however, if the enemy has lots of B-fields with value 1.

--5--

```
Name:           Night Crawler Stone
Author:         Wayne Sheppard
Speed:          32.86% of c
Size:           4
Durability:     Strong
Effectiveness:  Good
Score:

start SPL 0, <-1001
      MOV <21, 1+2234
      SUB 1, -1
      DJN -2, <-2234
end start
```

Night Crawler Stone is a self-splitting mod-2 bomber with a DJN-stream. When it finishes its bombing run, it turns into code that performs an addition core-clear.

HOW IT WORKS: Night Crawler Stone bombs memory similarly to Stone, with the obvious improvements that Night Crawler Stone bombs in a tighter mod-2 pattern, is self-splitting, uses a DJN-stream, and embeds the bombing step size in the executing code, making it one instruction smaller.

After the SPL 0,<-1001 instruction has split off about 144 processes into the main loop, it is bombed, making the effective size of Night Crawler Stone only 3 instructions long. Just before the bomber hits the bombing loop, the SUB 1,-1 instruction is decremented, starting an addition core-clear.

Unlike traditional core-clears, the addition core-clear doesn’t overwrite core with DAT statements. Instead, it modifies the A- and B-fields of the instructions to mess up the enemy’s control structures. For example, a SPL 0 that survived the bombing run becomes a SPL 2 which will not hold processes by itself. An addition core-clear is only slightly less effective than a traditional core-clear, and requires no additional instructions to run.

Just before the addition core-clear takes effect, Night Crawler Stone looks like this:

```
DAT 0, -1
MOV <1, 3895     (12938) (12941) ...
SUB 1, -1        (12940) (12943) ...
DJN -2, <-2234   (12939) (12942) ...
```

Process (*38) executes, decrementing the SUB instruction and doing a copy:

```
DAT 0, -1
MOV <1, 3895     (12941) ...
SUB 1, -2        (12940) (12943) ...
DJN -2, <-2234   (12939) (12942) ...
```

Process (*39) executes, laying down another decrement in the DJN stream. Process (*40) then executes, changing the A- and B-operands of the DAT statement:

```
DAT 2, 2233
MOV <1, 3895     (12941) ...
SUB 1, -2        (12943) ...
DJN -2, <-2234   (12942) ...
```

Process (*41) executes, decrementing the SUB instruction again, and then (*43) modifies the operands of the next instruction back:

```
DAT 2, 2234
DAT 2, 2233
MOV <1, 3895
SUB 1, -3
DJN -2, <-2234
```

So goes the core-clear, until at the end the DJN instruction is hit and turns into DJN 0,<0, where all of the processes go and execute repeatedly, laying down a DJN stream until time expires.

--6--

```
Name:           Keystone Stone
Speed:          32.86% of c
Size:           5
Durability:     Strong
Effectiveness:  Good
Score:

step equ 2517
emerald SPL 0, <-25
        MOV <-step+1, 92
        SUB 2, -1
        DJN -2, <2002
        JMP step, <-step
wait    DJN 0, <-12
paper   DJN 0, <-12
boot    MOV emerald+4, paper-step
        MOV emerald+3, <boot
        MOV emerald+2, <boot
        MOV emerald+1, <boot
        MOV emerald,   <boot
        MOV wait, paper+3053
        JMP @boot
end boot
```

After initialization, Keystone Stone bombs with a mod-1 pattern which approximates mod-4. If paper is detected, processes are split to the label "paper," where some code can be inserted to withstand paper attacks. When the bombimg run is over, Keystone Stone turns itself into an imp gate. (P.Kline's Keystone uses this gate as a backup strategy. Under normal operation, an external core-clear erases this stone.)

HOW IT WORKS: To set things up, the imp-gate (labelled "wait") needs to be copied away from the main block of code. Rather than adding an instruction to the main block to do this, the boot-strapping code (imaginatively labelled "boot") copies the stone and the imp-gate away from itself.

This has two advantages when fighting warriors that search through memory for the enemy. First, the copied code containing the executing stone is kept small, making it more difficult to locate. Second, the original code acts as a decoy for the enemy. In fact, many programs pad the block of original code with nonsense instructions to make a larger decoy for the enemy to grapple with. Almost all modern stones use boot-strapping and decoys to slow down the enemy.

When the initialization is finished, the stone starts a typical bombing run. If a process executing the DJN instruction finds a B-operand of 1, it falls out of the loop, executes the JMP instruction, and ends up at the label "paper," where some paper-stomping code should be inserted. The rationale behind this is that typically only paper has a B-operand of 1.

The bombing run ends with the DJN -2,<2002 instruction being hit, but not with a typical DAT bomb. Because of clever planning, the imp-gate instruction overwrites the DJN -2 instruction. The bomber now looks like this:

```
SPL 0, <-25
MOV <-step+1, 2
SUB 2, -1
DJN 0, <-12
JMP 2517, <-2517
```

Nearly all of the processes in the stone end up executing the DJN 0 instruction, forming an imp-gate. Along with killing imps, this imp-gate lays down a DJN-stream for extra program mangling. And processes falling through the DJN instruction don't matter much, because the SPL 0 instruction slowly generates new processes.

--7--

```
Name:           Winter Werewolf
Author:         W. Mintardjo
Speed:          25% of c
Size:           7
Durability:     Weak
Effectiveness:  Excellent
Score:

step equ 153
init equ 152
n equ ((12*8)-2)
data    DAT <-4-n, #0
split   SPL 0, <-3-step-n
main    MOV jump, @3
        MOV split, <2
        ADD #step, 1
        JMP main, init
```

```
        MOV @-4, <n
jump    JMP -1, 1
boot    MOV main+5, -500+5
        MOV main+4, <boot
        MOV main+3, <boot
        MOV main+2, <boot
        MOV main+1, <boot
        MOV main,   <boot
        MOV main-1, <boot
        MOV data, boot-500-3-n
        JMP boot-500
end boot
```

Winter Werewolf is a mod-8 bomber more in the spirit of Armadillo than Stone -- it drops specialized bombs througout core to stun the enemy, and then kills the enemy with a core-clear. It outscores Armadillo in three major aspects: It drops a more effective SPL/JMP bomb, it uses a two-pass core-clear, and it degrades into a perfect imp-gate to mop up any stray imps. The first pass of the core-clear lays down a SPL 0 stream to make sure the enemy is Really Stunned, and the second pass of the core-clear lays down DAT statements that kill the enemy. Winter Werewolf was one of the first modern programs that could compete against imp-rings.

HOW IT WORKS: After the boot-strapping routine, Winter Werewolf looks like this:

```
DAT <-98, #0
...
SPL 0, <-250
MOV 5, @3     (1)
MOV -2, <2
ADD #153, 1
JMP -3, 152
MOV @-4, <94
JMP -1, 1
```

The next two instructions drop a SPL/JMP bomb. Fist the JMP -1,1 instruction is copied:

```
DAT <-98, #0
...
SPL 0, <-250
MOV 5, @3
MOV -2, <2    (2)
ADD #153, 1
JMP -3, 152
MOV @-4, <94
JMP -1, 1
...
JMP -1, 1
```

The next instruction decrements the bomb pointer and copies the SPL 0,<-250 instruction to the new location:

```
DAT <-98, #0
...
SPL 0, <-250
MOV 5, @3
MOV -2, <2
ADD #153, 1   (3)
JMP -3, 151
MOV @-4, <94
JMP -1, 1
...
SPL 0, <-250
JMP -1, 1
```

The next instruction changes the bomb pointer in preparation for dropping the next bomb.

```
DAT <-98, #0
...
SPL 0, <-250
MOV 5, @3
MOV -2, <2
ADD #153, 1
JMP -3, 302   (4)
MOV @-4, <94
JMP -1, 1
...
SPL 0, <-250
JMP -1, 1
```

Finally, the JMP instrction loops to bomb the next location. The B-operand of the JMP instruction is ignored, allowing it to be used as the bomb pointer. The bombing run hits every eighth locoation with one of these bombs. The big trick at this point is to have the program bomb itself without getting trapped in a SPL/JMP loop itselfÈthe bombing run is over, the program looks like this (if we reset the process counter):

```
DAT <-98, #0
...
SPL 0, <-250
MOV 5, @3     (1)
MOV -2, <2
ADD #153, 1
JMP -3, 0
MOV @-4, <94
JMP -1, 1
```

When this first instruction is executed, the bomb pointer is bombed with the JMP -1, 1 instruction.

```
DAT <-98, #0
...
SPL 0, <-250
MOV 5, @3
MOV -2, <2    (2)
ADD #153, 1
JMP -1,1
MOV @-4, <94
JMP -1, 1
```

But, since the B-field of the bomb pointer just got changed to 1, the next bomb hits the bomb pointer, too. Remember, first the pointer is decremented...

```
DAT <-98, #0
...
SPL 0, <-250
MOV 5, @3
MOV -2, <2
ADD #153, 1
JMP -1,0
MOV @-4, <94
JMP -1, 1
```

...and then the SPL bomb is dropped.

```
DAT <-98, #0
...
SPL 0, <-250
MOV 5, @3
MOV -2, <2
ADD #153, 1   (3)
SPL 0, <-250
MOV @-4, <94
JMP -1, 1
```

Now the most subtle command of the whole program executes: The B-field of the new SPL 0,<-250 command is altered. We shall see later why this is important.

```
DAT <-98, #0
...
SPL 0, <-250
MOV 5, @3
MOV -2, <2
ADD #153, 1
SPL 0, <-97   (4)
MOV @-4, <94
JMP -1, 1
```

Now the core-clear begins. The SPL 0,<-97 instruction splits off processes and the JMP -1,1 instruction speeds up the core-clear, but it is the MOV @-4,<94 command that does the actual core-clear, and this deserves further comment.

The A-field of the MOV @-4,<94 instruction points to the MOV 5,@3 command that points to the SPL 0,<-97 instruction. Since the A-field uses indirect addressing, we are carpteing the core with SPL 0,<97 instructions for now. If the B-field of the MOV @-4,<94 instruction pointed to an instruction with zero B-field, this would yeild a very short (93 instruction) core-clear before the MOV command erased itself. But because of the bombing run, the B-field points to a SPL 0,<-250 command. So the pointers look like this:

```
     <---------------------------------
...                                   |
DAT <-98, #0                          |
...                                   |
SPL 0, <-250                          |
MOV 5, @3    ---- <----               |
MOV -2, <2      |     |               |
ADD #153, 1     |     |               |
SPL 0, <-97 <---     |               |
MOV @-4, <94 A-field---  B-field---   |
JMP -1, 1                         |   |
...                               |   |
SPL 0, <-250 <---------------------  ---
```

After the first process executes the MOV @-4,<94 instruction, the pointers look like this:

```
SPL 0, <-97 <--------------------------
...                                   |
DAT <-98, #0                          |
...                                   |
SPL 0, <-250                          |
MOV 5, @3    ---- <----               |
MOV -2, <2      |     |               |
ADD #153, 1     |     |               |
SPL 0, <-97 <----     |               |
MOV @-4, <94 A-field---  B-field---   |
JMP -1, 1                         |   |
...                               |   |
SPL 0, <-251 <---------------------  ---
```

And after the second process executes this instruction, the pointers look like this:

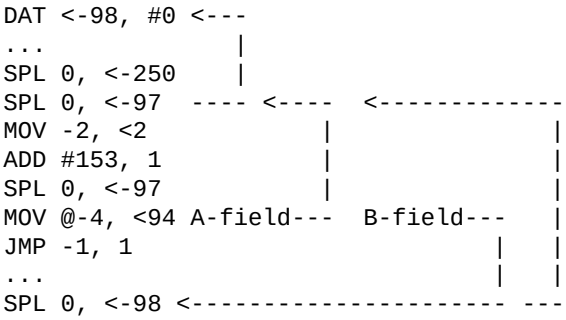
```
SPL 0, <-97 <--------------------------
SPL 0, <-97                           |
...                                   |
DAT <-98, #0                          |
...                                   |
SPL 0, <-250                          |
MOV 5, @3    ---- <----               |
MOV -2, <2      |     |               |
ADD #153, 1     |     |               |
SPL 0, <-97 <----     |               |
MOV @-4, <94 A-field---  B-field---   |
JMP -1, 1                         |   |
...                               |   |
SPL 0, <-251 <---------------------  ---
```

And so the core-clear goes, filling the entire core with SPL 0,<-97 commands, until the B-field pointer gets overwritten:

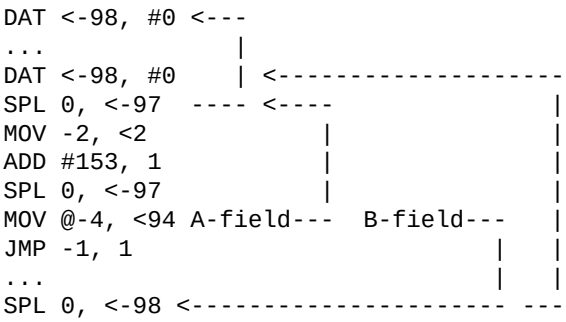
```
DAT <-98, #0
...
SPL 0, <-250
MOV 5, @3    ---- <----
MOV -2, <2      |     |  <-------------
ADD #153, 1     |     |               |
SPL 0, <-97 <----     |               |
MOV @-4, <94 A-field---  B-field---   |
JMP -1, 1                         |   |
...                               |   |
SPL 0, <-97 <---------------------  ---
```

This is where the ADD instruction mentioned above becomes so vital. If this pointer were overwritten by a SPL 0,<-250 command, the SPL core-clear would repeat, and the program would never get around to killing off the opponent.

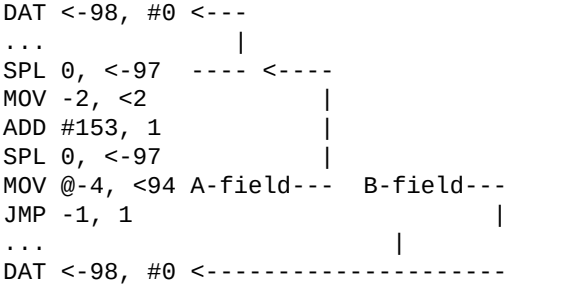
Note that the new pointer value skips over most of the core-clearing code, allowing the program to start a second core-clear. And the next iteration of MOV @-4,<94 does even more pointer magic, overwriting the A-field pointer with the SPL 0,<-97 instruction:



Because the A-field pointer now points to the DAT <-98,#0 command, this bomb is dropped next:



This starts the second core-clear, using DAT statements to finally kill the enemy processes. Like the first core-clear, this one continues until it wraps around and overwrites the B-field pointer again:



But this time, the new pointer does not cause the coreclear to skip the code. The core-clear continues until the MOV @-4,<94 instruction overwrites itself with the DAT <-98,#0 instruction:

```
SPL 0, <-97
MOV -2, <2
ADD #153, 1
SPL 0, <-97
DAT <-98, #0
DAT <-98, #0
```

The SPL 0,<-97 instruction keeps splitting processes to itself, keeping the program alive. The only other instruction that executes is the next one, which kills off all of the processes that execute it. Both of these instructions decrement the same instruction before the executing code, forming an imp-gate to kill off any remaining imp-spirals the enemy might have.

--Conclusion--

One factor that could mean the difference between a top-rate stone and an unsuccessful stone is the choice of step size. The program that manages to bomb the enemy first has a decided advantage, and some bombing step sizes are more efficient at scanning for the enemy than others. So what makes a good step size?

Ideally, it ought to hit every location in 8000 bombs, every other location in 8000/2=4000 bombs, every third location in 8000/3=2667 bombs, etc. Unfortunately, this is impossible, especially with a single step size, but it suggests a basic strategy -- go for the biggest programs first and then fill in the gaps.

One way of rating the efficiency of a step size is to find the length of the largest unbombed section of code after each bomb is dropped. By adding up all of these lengths, we get a number that tells us how big an average gap is. (Indeed, by dividing this number by the number of bombs dropped, we get the average gap size.) If we minimize this number over all step sizes, we get the "Optima Numbers." For a coresize of 8000, these optima numbers are:

```
mod-1  3359/3039  under-100 -> 73
mod-2  3094/2234  under-100 -> 98
mod-4  3364/3044  under-100 -> 76
mod-5  3315/2365  under-100 -> 95
mod-8  2936/2376
mod-10 2930/2430
```

The constant for Night Crawler Stone, for instance, is taken from this table.

Another common rating is how closely to in half the new bomb subdivides the old gap when it is dropped. By taking the differences between where the bombs fall and the middle of each gap and adding these distances up, we get an alternate method for testing efficiency.

Both of these methods are useful for finding general-purpose step sizes. But suppose you wanted to find a step size optimized for killing other stones. Since stones usually have four or five instructions, you would want a step size that would bomb every 4th and 5th instruction quickly, regardless of how it does in general.

Fortunately, there is a program in the public domain that calculates all of these things quicky. Corestep by Jay Han can be found as misc/corestep.c, and calcutates optima numbers and optimal step sizes. You will need a C compiler to use it, but it is otherwise self-contained. For more infromation, FTP a copy and read through it. The classic formula calculates optima numbers, the alternate formula calculates the sum of the distances between bombs and midpoints, and find-X calculates optimal step sizes against a specific program length.

If you don’t have access to a C compiler or want this for some other reason, P. Kline has compiled a list of all 8000 step-sizes with their mod, find-4, find-5, find-10, and find-13 numbers, along with imp-killing constants and imp-numbers. This table is designed for use in spreadsheets or databases. It is available in the misc/ directory under the name num8000.txt with documentation in num8000.doc. He used this on Keystone Stone to come up with a mod-1 constant with a low find-4 score, so that it would act like a mod-4 bomber but interfere with enemy scans (more about this in the next chapter).

Here is a list of successful stones. All of these can be found in warrior10.tar in the 88 directory, except for SJ-4A and Keystone t21, which are buried deep within the file feb94.txt.Z (in the newsgroup directory last time I checked.) Everything here by P.Kline has an anti-vamp component, which will be talked about in a later chapter.

"Leprechaun 1b" by Anders Ivner (leprechaun)
"Emerald 2" by P.Kline (emerald2)
"ExtraExtra 2" by P.Kline (extra2)
"Keystone t21" by P.Kline
"SJ-4A" by J.Layland
"Moonstone 1" by Dan Nabutovsky (moonstone)

Self-splitting stones with imp-rings can be very effective. Here is a list of imp-stone combos that are worth investigating. All of them except Cannonade can be found in warrior10.tar, and Cannonade can be found in the feb94.txt.Z file.

"Cannonade" by P.Kline
"Imprimis 6" by P.Kline (imprimis6)
"Night Crawler III" by Wayne Sheppard (nightcrawl)
"Sphinx 2.8" by W. Mintardjo (sphinx)

Program 1, Dwarf, was written by A.K. Dewdney for his Scientific American articles.

Program 2, Stone, was taken from the ICWS 1990 corewar tournament. It bears a remarkable resemblance to Rock by Scott Nelson, which was posted to the net a couple of months before the tournament. Strange, eh?

Program 4, Cannonade Stone was extracted from P.Kline’s Cannonade.

Program 5, Night Crawler Stone without the SPL 0 was submitted as "No Ties Allowed," and confused the experts as to how something so deadly could fit into 3 lines.

Program 6, Keystone Stone, was stolen from P.Kline’s "Keystone t21." The bootstrapping code in the example differs somewhat from the bootstrapping code used in Keystone.

Program 7, Winter Werewolf, originally did not copy the stone away from a decoy. I am led to speculate that the code as it exists here with a bigger decoy resembles Winter Werewolf 3, a program that was very successful on the hill.