Emoji shellcoding in RISC-V

Hadrien Barral[1], Georges-Axel Jaloyan[§1,2], and David Naccache[1]

[1]DIENS, École normale supérieure, CNRS, PSL University, Paris, France
[2]Amazon Web Services, Seattle, USA

***Abstract*—Shellcodes are short, executable code fragments that are utilized in various attack scenarios where code execution is possible. When they are injected through the program's inputs, they may require to be validated by filters, the most common of which is a restriction on the allowed character set. This paper explains how to design RISC-V shellcodes capable of running arbitrary code whose UTF-8 representation uses only Unicode emojis.**

Our approach to this problem is inspired by code-reuse attacks and involves the use of small, reusable code snippets called gadgets. By chaining these gadgets together, we are able to build a shellcode that can bypass the constraints imposed by filters, making it more versatile and effective in a wider range of attack scenarios.

I. Introduction

Memory corruption vulnerabilities, particularly stack-based overflow attacks [1], remain a popular method for hackers to gain unauthorized access to a program. Hackers often use *shellcodes*, short executable code snippets, as a means of injecting their payload. While it is commonly believed that shellcodes can be easily distinguished from legitimate data using methods such as signature-based detection [18] or executable space protection [21], they still present a significant threat, particularly in mobile and low-power embedded systems.

RISC-V, a popular architecture in embedded systems, is becoming increasingly attractive to low-level attackers, whose tools are traditionally geared towards x86 platforms. However, the use of filters accepting only the allowed character set, common in text-based applications, can make it more difficult for attackers to successfully inject their shellcodes.

In this context, it is important to note that low-level segments of code such as Java Native Interface in mobile applications, as well as low-power embedded devices (IoT, coprocessors) are particularly susceptible to buffer overflow exploitation and often lack memory protection mechanisms.

This paper aims to address the problem of designing shellcodes that can bypass validation filters, specifically for RISC-V architecture in the context of text-based applications. This will be done by introducing a new and more generic approach to shellcoding under constraints, inspired by code-reuse attacks, which builds the shellcode by chaining several small reusable code snippets called gadgets.

[§]This work was carried out prior to joining Amazon Web Services.

A. Our contribution

Our research presents the first analysis of utilizing emojis as a means for writing shellcodes and a framework for automatically generating such shellcodes in both 32 and 64-bit RISC-V architecture.

Our design relies on a multi-staged shellcode, with the first stage written solely using emoji code sequences, allowing for arbitrary code execution in a generic manner by pushing the complexity onto the unpacker instead of the payload.

Additionally, our approach differs from previous work by utilizing a generic method inspired by code-reuse attacks for generating low-level primitives, making the shellcodes more resistant to signature-based detection. We include examples of emoji shellcodes in RISC-V and an automated tool for building them in the appendix.

II. Background and previous work

A. Shellcodes and exploitation

In a typical *arbitrary code execution* (ACE) scenario, attackers can run a short program, known as a *shellcode*, to gain control of a system and execute additional programs. This can occur through a vulnerability such as a stack-based overflow, where an application allows writing beyond the allocated space of an array, resulting in the overwriting of stack frame data [1]. The shellcode is written in the array and executed as if it were the program's own instructions.

To be effective, shellcodes must be concise and comply with any constraints imposed by the application. Additionally, modern protections such as *Address Space Layout Randomization* (ASLR) [25], stack-smashing protection [9], and non-executable stack space [21] make shellcode design challenging.

Embedded and mobile devices, where protections are often partially implemented or not at all, offer a less restrictive environment for shellcode development. Furthermore, these devices host many third-party applications that may not adhere to secure coding practices, increasing the risk of vulnerabilities. Previous research on constrained shellcoding [17, 27] has mainly focused on the alphanumeric subset. Tools to generate alphanumeric shellcodes on the x86 platform [4] are now a standard component of attack frameworks including Metasploit (`msfvenom`) and UPX1. Three techniques are usually used: compilation, virtualization, and packing.

B. Compilation

Shellcode compilation uses a *compiler* to translate the payload directly into a constrained target-language. When applicable, compilation is very efficient: compilers such as *movfuscator* [15, 16] and *higher subleq* [20] have been provided for *one instruction set computers*, reduced ISA subsets made of only one instruction. Popescu [26] released in 2019 a compiler producing a null-free shellcode in x86 and x64. Similarly, compile-time gadget reduction techniques such as G-Free [23] also leverage compilation into a constrained target language (`c3`-free binaries).

Furthermore, such methods often rely on *syntax-directed translation schemes*, which hinders their usability in the context of shellcoding. Indeed, our constraints lie mostly on the operands, which cannot be expressed in the abstract syntax recursive translation scheme without a lot of special casing.

C. Emulation

Emulation, as used by Younan et al. for 32-bit ARMv7 alphanumeric shellcoding [30], requires the design of a bytecode and an interpreter, both compatible with the limited instruction set, and powerful enough to mount a realistic attack — beyond Turing-completeness, we need to perform system calls or other mechanisms to evade the virtual environment. Emulation presents a huge runtime overhead as well as a committed engineering effort.

The first automated tool using emulation was provided by Younan et al. in 2011 for the ARMv5 platform, relying on a BF interpreter and bytecode [31]. The technique however does not carry over to more recent architectures such as ARMv8.

D. Unpacking

Packing consists in splitting the exploit into a multi-staged shellcode, where each stage unpacks the next one before executing it. By convention, the first stage to execute is called *stage 1*. Packers can provide additional functionalities such as compression or encryption, which we do not explore here. However, unpacking requires the ability to execute self-modifying code, which may be hindered by the presence of executable space protection mechanisms like DEP [21], PaX [25] or NX-bit [19]. Moreover, self-modifying code causes cache issues which need to be handled on a target-specific basis.

In 2016, Barral et al. introduced the first tool capable of compiling arbitrary ARMv8 code into alphanumeric executable code [2], which they extended to alphanumeric RISC-V in 2019 [3]. We decided to follow this approach as it is conceptually simpler, much easier to check for correctness, and well-suited to our target platform. Though, in the context of emoji shellcoding, we now need to find a new technique to generate the low-level primitives used in stage 1, as the previous ones are not applicable anymore.

E. Code-reuse attacks

The introduction in 2004 of *Data Execution Prevention* (DEP) [21] made straightforward code injection attacks almost impossible, as injected data could not be executed anymore. Instead, malware developers started using a technique reusing executable code already present in memory, called *Return-Oriented Programming* (ROP). The first ROP attack was publicly presented in 2001 by Nergal [22], and academically studied in 2007 by Shacham [28].

ROP bypasses DEP by injecting into the stack a succession of call frames. Each call frame will result in the execution of a *gadget*: a small snippet of legitimate code containing a small number of instructions ending with a `ret`. When the `ret` instruction is reached, the address of the next gadget is popped from the stack into the program counter, yielding the control flow to the next gadget in the chain. Provided that enough different gadgets are available in the target executable, arbitrary code may be executed by chaining those gadgets.

JIT-spraying [6] is a technique leveraging the output of a Just-In-Time compiler to add gadgets into the produced executable code. The generated gadgets are then reused through a ROP attack bypassing DEP.[1]

F. Unicode

Unicode is a standard aiming for a consistent encoding and representation of written characters and text. It is the *de facto* standard for most modern software stacks. For the rest of this paper, we refer specifically to Version 14.0 [11], published in 2021.

We summarize in the following paragraphs the main Unicode terminology definitions used in the rest of the paper:

Codepoint: an integer in the Unicode *codespace* range. *Assigned* codepoints usually represent abstract characters (an exception being UTF-16 surrogates).

Encoding Form: Codepoints being only an abstraction, systems must have a way to represent them with bits. This is called *encoding*. Unicode defines three encodings: UTF-8, UTF-16 and UTF-32. In the following, we only use the UTF-8 encoding (which is the most common). For more information, we refer the reader to chapter 2.5 of the Unicode Standard [11].

Emoji: Unicode defines in Unicode Technical Standard #51 [12] some codepoint sequences to be *emojis*. These emojis can have multiple representations. *Fully-qualified emojis* and *minimally-qualified emojis* are meant to be represented in a colorful and/or animated way, called *emoji presentation*. *Unqualified emojis* are meant ot be represented in a more 'font-like' way. E.g., `U+2618 U+FE0F` is the fully-qualified shamrock ☘️ whereas `U+2618` is the unqualified shamrock ☘. In this work, the set of emojis is defined to be set of *fully-qualified emojis* and *minimally-qualified emojis*. A list of such emojis is given in [10].

[1]JIT-spraying also uses heap-spraying to bypass ASLR, which is not within the scope of this paper.

G. *RISC-V*

RISC-V [29] is an *Instruction Set Architecture* (ISA) developed since 2010. It is based on the concept of *Reduced Instruction Set Computer* (RISC) [24], targeting simplicity by providing few, limited computer instructions. RISC ISAs have become increasingly popular with the advent of embedded devices such as smartphones, tablets, and other IoT devices. RISC-V is the fifth RISC ISA published by UC Berkeley, and it is completely free and open-source. It features 32-bit and 64-bit little-endian variants (designated as `RV32` and `RV64`), with a planned extension to 128-bit.

RISC-V splits its instruction set between a mandatory core set (*RV64I*) and different optional extensions, each designated by a capitalized string. For example, the compressed instruction set designated with the letter `C`. The general-purpose ISA, which includes `IMAFDZicsrZifencei` bears the letter `G`. In this paper, we focus on the `RV64GC` ISA, which is the one agreed upon by Debian and Fedora developers, as well as members of the RISC-V Foundation. Additionally, the foundation intends to provide "*a profile for standard RISC-V Unix platforms that will include* `C` *extension as mandatory*".[2]

The `RV64GC` ISA features 32-bit and 16-bit instructions, aligned on 16 bits. It has 31 general-purpose 64-bit registers (`x1-x31`), 32 floating-point registers (`f0-f31`), a program counter (`pc`), as well as various control-and-status registers. The pseudo-register `x0` designates the zero constant.

We adopt the terminology defined in the RISC-V Instruction Set Manual [29] for the remainder of this paper. Assembly instructions are written in the format `add x1,x2,x3` where `add` is the *opcode* and `x1`, `x2`, `x3` are the *operands*. Specifically, `x1` is the *destination register*, `x2` is the *first source register*, and `x3` is the *second source register*. When a source register is replaced with a constant, it is referred to as an *immediate*. In addition to these conventions, we also use the slicing notation $K[x:y]$ (where K is a register and $x<y$) to denote the slice of bits from x to y of K, with the lowest bit denoted as bit 0. We also follow the register naming convention of RISC-V ELF psABI [13], as shown in Table I.

Register	**ABI Mnemonic**	**Meaning**
`x0`	`zero`	Zero
`x1`	`ra`	Return address
`x2`	`sp`	Stack pointer
`x3`	`gp`	Global pointer
`x4`	`tp`	Thread pointer
`x5-x7`	`t0-t2`	Temporary registers
`x8-x9`	`s0-s1`	Callee-saved registers
`x10-x17`	`a0-a7`	Argument registers
`x18-x27`	`s2-s11`	Callee-saved registers
`x28-x31`	`t3-t6`	Temporary registers

Table I: RISC-V register naming convention, as per psABI [13].

[2] https://wiki.debian.org/RISC-V

III. Informal overview

The initial approach of attempting to disassemble arbitrary emoji sequences[3] does not produce any meaningful results, as only 10 emojis yield valid RV64GC instructions. Even using pairs of emojis does not improve the outcome, as the disassembler fails to find any valid executable instructions starting at the sequence. This is due to traditional linear disassemblers such as `objdump` assuming that the first instruction starts at offset 0.

Instead, we adopt a different approach searching for RISC-V instructions that could be part of any emoji sequence. We refer to this subset $\mathbb{E}$ of RISC-V instructions as *emoji-compatible*. Although this approach yields significantly more instructions than the first method, these instructions alone cannot be used to write a shellcode as the processor requires what comes before and after this instruction to be valid RISC-V code.

To address this issue, we use a recursive approach. We prepend and append other emoji-compatible instructions to a given instruction of $\mathbb{E}$, whose hexadecimal representations when concatenated form a valid emoji sequence.

As an example, consider the emoji-compatible RV64GC instruction `auipc ra,0x979ff`, whose little-endian hexadecimal representation is `97 F0 9F 97`. Without loss of generality, consider splitting the instruction `auipc ra,0x979ff` as `97` (left part) and `F0 9F 97` (right part). We then solve independently each part, by finding emoji sequences whose UTF-8 hexadecimal representation ends with `97` (resp. starts with `F0 9F 97`).

a) Solving the left part: As an example, consider the emoji ❗ (in hex. `E2 9D 97`), the first two bytes, `E2 9D` correspond to the RV64GC instruction `add s11,s8`. Assuming that there is a solution to the second part, we can start the executable sequence containing our `auipc` instruction at the expense of trashing the `s11` register. The corresponding emoji sequence starts with ❗.

Another possible emoji could be the 🆗 emoji (in hex. `F0 9F 86 97`). Unfortunately, there is no RISC-V instruction whose last three bytes coincide with the first three bytes of 🆗 (`F0 9F 86`). This means that if we want 🆗 to part of our shellcode, we must jump directly to the byte `97` for our shellcode to remain valid RISC-V code.

b) Solving the right part: The approach is similar. If we consider the 🗓️ emoji, (in hex. `F0 9F 97 93 EF B8 8F`), its last four bytes correspond to the instruction `ori t6,a7,-0x705`. We can end the second part with 🗓️. Indeed, there is no need to continue further, since any subsequent emoji can be viewed as the beginning of the first part of another instruction.

If we consider the 🗑️ emoji (in hex. `F0 9F 97 91 EF B8 8F`), then the first two bytes (`91 EF`) correspond to the instruction `bnez a5,+0x1C`, which is a jump forward if `a5`$\neq 0$. Assuming that the jump is taken, we do not need to check whether the last two bytes (`B8 8F`) are valid RISC-V

[3] Given that RISC-V instructions are at most 4 bytes and emojis at least 3 bytes, this means that we can restrict ourselves to sequences of only one or two emojis, greatly reducing the computation cost.

instructions, as they will not be executed. Besides, we limit ourselves to forward direct jumps having an offset large enough to jump out of the executable sequence.

As a result, the following four emojis sequences can be built to execute our original `auipc ra,0x979ff` instruction: ❗🗓️, 🆗🗓️, ❗🗑️, and 🆗🗑️. Depending on the surroundings of this instruction, taking into account whether the preceding sequence ends with a jump or not, we may choose either of those four possibilities.

c) Assembling the sequences: The method of generating small sequences of emojis reusable in a modularly makes the shellcode writing process similar to well-known code-reuse techniques such as Return-oriented programming [22, 28], Jump-oriented programming [7] or JIT-spraying [6] (which uses code-injection to write gadgets into memory). Likewise, we reuse the term *gadget* to designate our sequences. The main difference with ROP lies in the gadget chaining method, as it now depends on whether we need to jump into the next gadget or simply let the flow of execution proceed.

IV. Generating gadgets

In this section, we formally describe how to generate the emoji gadgets described in Section III.

We define our gadget as being a sequence of emojis, whose UTF-8 representation encompasses a sequence of RISC-V instructions. Borrowing terminology from code-reuse attacks, we call those sequences respectively the *emoji path* and the *execution path*.[4] Generating gadgets in this context implies finding all emoji and execution paths coinciding on their hexadecimal representation.

We generate the gadgets with Algorithm 1 and detail it in the next paragraphs. We remind the notation $P[a:b]$, denoting a slice of the array from indexes a (included) to b (excluded). As in Python, we allow negative integers to designate an index relative to the array's end, and use the absence of an integer to designate either the start or the end of the array.

The algorithm takes as input a single RISC-V instruction,[5] and splits it in all possible ways. E.g., there are five ways to split a four-byte $B_0B_1B_2B_3$ instruction: ▌$B_0B_1B_2B_3$, B_0▌$B_1B_2B_3$, B_0B_1▌B_2B_3, $B_0B_1B_2$▌B_3, and $B_0B_1B_2B_3$▌. We then independently compute any possible gadget start and end for this instruction splitting choice. The final gadgets set is just the Cartesian product of the sets of gadget starts and gadget ends (line 39 of Algorithm 1). As gadget length can be potentially unbounded, we add a heuristic in our implementation to limit the gadgets' length to at most 7 instructions.[6]

```
Input: $B_0 ... B_3$, a RV64G valid instruction
Result: $G$, the set of all emoji RV64GC gadgets
        containing $B_0 ... B_3$
1  Function Append($P_{😃}$, $P_{💻}$) is
       // $P_{😃}$ is the emoji path
       // $P_{💻}$ is the executable path
2      $S = \varnothing$
3      if NumberOfInstructionsIn($P_{💻}$) > 3
4          return $\varnothing$
5      else if $|P_{😃}| = |P_{💻}|$
6          return $\{P_{💻}\}$
7      else if $|P_{😃}| > |P_{💻}|$
           /* expand $P_{💻}$ with any possible
              emoji-compatible RISC-V instruction      */
8          for $I \in \mathbb{E}$ do
               // w.l.o.g.: $|P_{😃}| - |P_{💻}| < 4$
9              if $I[0:|P_{😃}| - |P_{💻}|] \neq P_{😃}[|P_{💻}|:|P_{😃}|]$
                   // $I$ is not a valid extension of $P_{💻}$
10                 continue
11             $S := S \cup Append(P_{😃}, P_{💻} + I)$
12         return $S$
13     else
           /* expand $P_{😃}$ with any possible emoji      */
14         for $E \in Emojis$ do
15             if $E[0:|P_{💻}| - |P_{😃}|] \neq P_{💻}[|P_{😃}|:|P_{💻}|]$
                   // $E$ is not a valid extension of $P_{😃}$
16                 continue
17             if $P_{💻}[-1]$ is a jump
                   // We can stop extending the gadget
18                 $S := S \cup \{P_{💻}\}$
19             else
20                 $S := S \cup Append(P_{😃} + E, P_{💻})$
21         return $S$

22 Function Prepend($P_{😃}$, $P_{💻}$) is
23     $S = \varnothing$
24     if NumberOfInstructionsIn($P_{💻}$) > 3
25         return $\varnothing$
26     else if $|P_{😃}| = |P_{💻}|$
27         pass
28     else if $|P_{😃}| > |P_{💻}|$
29         for $I \in \mathbb{E}$ do
30             if $I[|P_{💻}| - |P_{😃}|:] \neq P_{😃}[:|P_{😃}| - |P_{💻}|]$
31                 continue
32             $S := S \cup Prepend(P_{😃}, I + P_{💻})$
33     else
34         for $E \in Emojis$ do
35             if $E[|P_{😃}| - |P_{💻}|:] \neq P_{💻}[:|P_{💻}| - |P_{😃}|]$
36                 continue
37             $S := S \cup Prepend(E + P_{😃}, P_{💻})$
38     return $S \cup \{P_{💻}\}$

39 $G := \bigcup_{i=0}^{4} Prepend([], B_0..B_{i-1}) \times Append([], B_i..B_3)$
```

Algorithm 1: Algorithm generating all emoji-gadgets.

[4] In ROP or JIT-spray attacks, we overlap two executable sequences, called the Main and the Hidden execution paths (MEP and HEP).

[5] Without loss of generality, we only consider 4-byte instructions of RV64G. The process is similar for the 2-byte instructions of the C extension.

[6] We exclude the initial jump for dangling-start gadgets from this count.

A. *Ending the gadget (Append)*

The basic idea is to extend either the emoji or the executable path (resp. $P_{😃}$ and $P_{💻}$) with an emoji or a RISC-V instruction, whichever is shorter. The only criterion being that the chosen emoji (resp. RISC-V instruction) must not contradict the executable path $P_{💻}$ (resp. emoji). I.e. that the first bytes of the chosen emoji must be equal to the last bytes of the executable path (*mutatis mutandis* the chosen RISC-V instruction). This is done by line 15 (resp. 9) of Algorithm 1.

We perform the previous step recursively, cutting any branch for which we reach a point at which there is no possible emoji (resp. instruction) extending $P_{😃}$.

Two additional conditions allow us to stop successfully and return a result, either by reaching a point when $P_{😃}$ and $P_{💻}$ both have the same size, or when the last instruction added in $P_{💻}$ is a jump. Indeed, in the former case, we found a point at which we can write what follows in the shellcode independently, while in the latter case, we jump out of our gadget, thus we can stop and leave the problem of finding the next executable instruction to another gadget.

In code-reuse terminology, the two conditions are thoroughly studied under the notion of *Point of Interest.* In the context of emoji gadgets, our points of interests encompass all jumps (indirect as in ROP and direct as in JOP), and *synchronization points* between the emoji and execution paths.[7] In this paper, we restrict ourselves to only direct jumps with positive offsets to prevent spaghetti shellcode.

B. *Starting the gadget (Prepend)*

The gadget's start is computed similarly by performing the same extension operation, with the exception of replacing all appends by prepends. The terminal conditions are slightly different, as we now allow the gadget to start in two ways. On top of synchronization points between the emoji path $P_{😃}$ and execution path $P_{💻}$, we also allow *dangling gadget starts* (line 38), as another gadget may directly jump on the first non-synchronized instruction of the gadget. Though, a dangling start now requires the gadget to be positioned at the right offset from the end of the previous gadget.

C. *Unclogging*

When assembling gadgets, there is often a gap between them. This typically happens for $G_{\texttt{slti\_t0}}$ (used to zero register `t0`, detailed in Figure 1), which, being a *dangling-start gadget*, needs another gadget to jump to the executable part of it. Moreover, $G_{\texttt{slti\_t0}}$ ends with a small forward jump instruction. The jump offset is typically larger than the bytes needed to finish the emoji path, leaving *undefined* byte chunks. We call such a gap *clog.* Figure 1 shows a clog of size 24, and a clog of size 22.

Since clogs are part of the shellcode, they must also have emoji representations. We call *unclogging* the process

[7]In code-reuse terminology, those are the points at which the HEP merges into the MEP.

✅	E2 9C	`add s9,s9,s8`
©	85 C2	`beqz a3,+0x20`
	A9 EF B8 8F	`.fill 0xa9efb88f`
	…	`.clog 0x18`
	F0 9F	`.fill 0xf09f`
📢	93 A2 E2 81	`slti t0,t0,-2018`
⁉️	89 EF	`bnez a5,+0x1a`
	B8 8F	`.fill 0xb88f`
	…	`.clog 0x16`

Figure 1: Details of gadget $G_{\texttt{slti\_t0}}$ (📢⁉️) with a preceding *jump* gadget (✅©) to account for $G_{\texttt{slti\_t0}}$'s dangling start

finding an emoji-representation for a given clog. Hereafter, we provide an unclogging method (whenever possible) using three and four-byte emojis using Bézout's lemma. Minimizing the number of emojis can be done with dynamic programming, which helps reduce the length of the shellcode, counted in characters.

Lemma 1. *Let* $\mathbb{N}$ *be the set of non-negative integers. Let* $\mathfrak{c}$ *be a clog , whose size in bytes is* $|\mathfrak{c}|$. *Unclogging* $\mathfrak{c}$ *is possible if and only if* $|\mathfrak{c}| \in \mathcal{C} = \mathbb{N} \setminus \{1, 2, 5\}$.

Proof. UTF-8 contains in particular 3 and 4-bytes emojis. Thus, since $\bigcup_{i,j\in\mathbb{N}} 3i + 4j = \mathcal{C}$, Bézout's lemma gives a trivial way to unclog $\mathfrak{c}$, except if $|\mathfrak{c}| \in \{1, 2, 5\}$.

Moreover, since no emoji has a UTF-8 representation of 1, 2, or 5 bytes, if $|\mathfrak{c}| \in \{1, 2, 5\}$, then unclogging $\mathfrak{c}$ is impossible. □

D. *Available instructions*

Using the gadget generation algorithm described above on `RG64GC`, we get $\mathcal{G}$, a set of emoji gadgets. In this subsection, we attempt to give a concise overview of what can be done with $\mathcal{G}$. As $\mathcal{G}$ is too big to be humanly searchable, we build $\mathcal{I}_{\mathcal{G}}$, the set of instructions appearing a least once in $\mathcal{G}$, then cluster $\mathcal{I}_{\mathcal{G}}$ into different instruction types.

Note that this approach yields an over-approximation of the instructions one could want to use. Imagine for example that there is a single gadget (g) containing the `li a0, 42` instruction. It could be that in g, this instruction is immediately followed by the `mv a0, s1` instruction, nullifying the effect of `li a0, 42`. Thus `li a0, 42` is of no use in $\mathcal{I}_{\mathcal{G}}$.

The result of clustering $\mathcal{I}_{\mathcal{G}}$ is presented below:

1) Data processing: We list here all available instructions which only operate on general-purpose registers.

- Register move: 76 `mv` (move) instructions allow to move data between registers.
- Several addition instructions are available: `add`, `addi`, `addiw`, `addw`. These allow to add multiple small constants in 32-bit and 64-bit variants.
- Likewise, `sub` and `subw` (subtraction) are present.

- The `auipc` (add upper immediate to program counter) allows to read the program counter indirectly, and may allow for position-independent shellcodes.
- Bitwise manipulation: 55 `andi` (bitwise AND with immediate), 4 `xor` (bitwise XOR) and 3 `ori` (bitwise OR with immediate), and 72 `srai`/`srli` (shift right arithmetic/logic with immediate).
- We get no `li` (load immediate) instruction, but we do have 25 `lui` (load upper immediate) variants.
- Set less than immediate: in both signed `slti` and unsigned version `sltiu`, a few hundreds of them can be found.

2) Control-flow instructions: Both conditional and unconditional, forward and backward jumps are available.

- We get 148 basic `j` (jump) instructions, and 16 `jal` (jump-and-link) instructions.
- For conditional jumps, three variants are available: `beq` (branch if two registers are equal, 4 of them), `beqz` (branch if register is zero, 16 of them), `bnez` (branch if register is not zero, 68 of them).

3) Memory processing:

- We have both `lb` (8-bit) and `lw`/`lwu` (32-bit) loads as well as `sw` (32-bit) and `sd` (64-bit) stores, for a total of 33 loads and 64 stores variants.
- A total of 1565 floating-point loads and stores (both 32-bit and 64-bit) are available. These are of little use since we have no floating-point data processing instructions.

4) Other instructions:

- One single control-status register manipulation instruction: `csrrs ra,0xbf8,gp`. It only appears in 🏴󠁧󠁢󠁥󠁮󠁧󠁿, 🏴󠁧󠁢󠁳󠁣󠁴󠁿 and 🏴󠁧󠁢󠁷󠁬󠁳󠁿 (all members of the Emoji subdivision-flag subgroup).

5) Difference with `RV32GC`: The set of available gadgets in `RG32GC` (denoted $\mathcal{G}_{32}$) is slightly different. We give in appendix C an overview of the differences.

V. Shellcode construction

We will now explain how we build a shellcode from the previously found emoji-compatible gadgets.

A. High-level overview

As hinted in section II-D, we will use *unpacking*. Let $\mathcal{P}$ be our final payload (this is the code the attacker wants to execute on the target).

Our shellcode high-level design is shown in Figure 2. The unpacker $\mathcal{U}$ will write $\mathcal{P}$ into memory, then jump to it. Contrary to [2], we do not use a main decoding loop: the unpacker $\mathcal{U}$ is entirely unrolled.

B. Stage 1 design

Stage 1 assumes no initial state. Specifically, this make the shellcode position independent. Therefore, it starts with an initialization phase, to set-up some registers.

For example, gadget $G_{\texttt{slti\_t0}}$ presented in Figure 1 is used twice to zero the `t0` register (it relies on previous

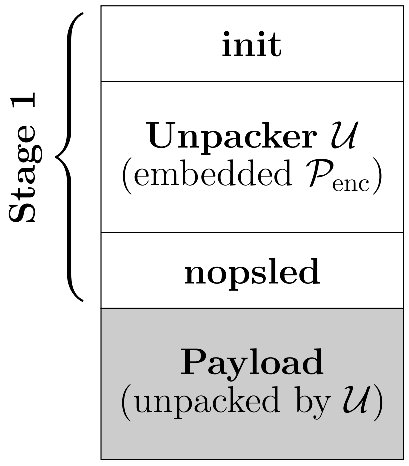

Stage 1
init
Unpacker $\mathcal{U}$ (embedded $\mathcal{P}_{\text{enc}}$)
nopsled
Payload (unpacked by $\mathcal{U}$)

Figure 2: High-level construction overview for our generic emoji shellcode.

✅ ♈	E2 9C	`add s9,s9,s8`
	85 E2	`bnez a3,+0x20`
	99 88	`.fill 0x9988`
	…	`.clog 0x1b`
➕	E2	`.fill 0xe2`
	9E 95	`add a1,a1,t2`

(a) Detail of gadget $G_{\texttt{a1++}}$, with a *jump* gadget to account for the dangling start

❌ ♋	E2 9D	`add s11,s11,s8`
	8C E2	`sd a1,0(a3)`
	99 8B	`andi a5,a5,6`

(b) Detail of gadget $G_{\text{store}_{64}}$

♉ ➖	E2 99	`add s3,s3,s8`
	89 E2	`bnez a3,+0x2`
	9E 96	`add a3,a3,t2`

(c) Detail of gadget $G_{\texttt{a3++}}$

Figure 3: Details of gadgets used in $\mathcal{U}$. Register `t2` is set to 1 during initialization

gadgets to set register `a3` to zero and `a5` to a non-zero value). Indeed, the only instruction which modifies `t0` is `slti t0,t0,-2018`. Per the RISC-V documentation [29]: "*SLTI (set less than immediate) places the value 1 in register rd if register rs1 is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to rd*". Hence, after the first call to $G_{\texttt{slti\_t0}}$, register `t0` is either 0 or 1. Since $1 > 0 \geq -2018$, it follows that after the second call to $G_{\texttt{slti\_t0}}$, `t0` is now equal to 0.

Once initialized, the decoder's main body comes next. The main body can be written with only 3 gadgets, whose semantics are roughly the following:

- $G_{\texttt{a1++}}$: `a1++` (see Figure 3a)
- $G_{\text{store}_{64}}$: `*(byte*)a3 = a1` (see Figure 3b)
- $G_{\texttt{a3++}}$: `a3++` (see Figure 3c)

Initialization has set `a1` to 0, and `a3` to the address where we want to write the payload. We note $\mathcal{P} = \mathcal{P}_0..\mathcal{P}_i..\mathcal{P}_n$, with $\mathcal{P}_i$ being $\mathcal{P}$'s i-th byte.

We then have the following algorithm to generate the sequence of gadgets which will re-create $\mathcal{P}$ in the target memory:

```
Input: $\mathcal{P} = \mathcal{P}_0...\mathcal{P}_n$, the payload to encode
Result: $\mathcal{U}$, the unpacked, which recreates $\mathcal{P}$ when run
1  Function Encode($\mathcal{P}$) is
2      $a_1 := 0$
3      $\mathcal{U} = []$
4      for $i$ from 0 to $n$ do
5          while $a_1 \neq \mathcal{P}_i$ do
6              $\mathcal{U}$.append($G_{\texttt{a1++}}$)
7              $a_1 = (a_1 + 1) \bmod 256$
8          $\mathcal{U}$.append($G_{\text{store}_{64}}$)
9          $\mathcal{U}$.append($G_{\texttt{a3++}}$)
10     return $\mathcal{U}$
```

Algorithm 2: Algorithm generating $\mathcal{U}$.

This approach results in quite big unpackers. Using another set of gadgets with almost no clog, we managed to reduce their size by a factor of up to 15. We show in Appendix A several examples of shortened shellcodes. The more general approach would consist in writing a three-staged shellcode, thoroughly explored in [3].

Finally, a forward jump and/or a nopsled is added at the end of the stage 1, so that the program counter slides down to the beginning of the unpacked payload.

C. Polymorphism

As previously discussed, to connect gadgets with dangling starts, we use small forward jumps, creating gaps known as clogs. Section IV-C describes two unclogging methods. This section presents an additional method to make the shellcode partially polymorphic.

A code is considered *polymorphic* if it can be modified into another code with the same functionality. In this case, we repurpose the *unclogger* as a *polymorphic engine*, which helps the shellcode to evade basic pattern-matching detection methods [8]. Other specific techniques can be used to bypass more recent intrusion detection systems [14].

The unclogger is modified to randomly select an emoji sequence of a specified length. This allows for randomizing a significant portion of the shellcodes: numbers are given next to the shellcodes samples in Appendix A.

D. 32-bit RISC-V variant

In Section V-B, gadget $G_{\text{store}_{64}}$ is used for the unpacker. This gadget uses the `sd a1,0(a3)` instruction, which is 64-bit specific, meaning that it is not available in `RV32`.[8]

Thus, we replace gadget $G_{\text{store}_{64}}$ by gadget $G_{\text{store}_{32}}$, shown in Figure 4.

[8]Its hexadecimal encoding `0x8ce2` corresponds to the `fsw fa1,0(a3)` instruction in `RV32`.

	E2 99	`add s3,s3,s8`
♌	8C C2	`sw a1,0(a3)`
©	A9 EF	`bnez a5,+0x5a`
	B8 8F	`.fill 0xb88f`
	…	`.clog 0x54`

Figure 4: Details of gadget $G_{\text{store}_{32}}$

Unfortunately, $G_{\text{store}_{32}}$ is larger (62 bytes) than $G_{\text{store}_{64}}$ (6 bytes), resulting in slightly larger shellcodes in `RV32`. All the other gadgets we used are common to both `RV32` and `RV64`.

VI. Evaluation

A. QEMU

We initially tested our emoji shellcodes on QEMU [5], a widespread open-source emulator. It emulates a HiFive Unleashed development board with `RV64GC` or `RV32GC` cores,[9] without some of its micro-architectural features like caches or timings. The payload is expected to print "*Hello world!*" on the serial device mapped at address `0x10013000`. After generating the corresponding shellcodes for both RV32IMC and RV64GC, we successfully executed them on QEMU. We provide in Appendix A information to easily reproduce this experiment.

B. Linux on HifiveU

Subsequently, we moved to the more realistic environment of Linux on a HiFive Unleashed board powered by a quad-core Freedom U540 `RV64GC` processor. It features a minimal busybox-based buildroot environment, for which we created a purposely vulnerable application executing its input data.

The first payload uses a `write` system call to print "*Hello world!*" on the standard output. As previously, we generated the emoji shellcode, and successfully executed it on the vulnerable application. In addition, we successfully tested the shellcode with two other payloads, one that spawns a shell using the `execve` system call, and one that prints on the standard output the contents of `/etc/shadow` file, using the `openat`, `read` and `write` system calls.

Furthermore, we did not observe any cache issue, as one could dread when using self-modifying code. This is explained by the use of `fence.i` as the payload's first instruction synchronizing the instruction cache.

C. Bare metal ESP32C3

As a third experiment, we used an Espressif board featuring an ESP32-C3 RV32IMC CPU. We flashed on the board a purposely vulnerable bare-metal application emulating a cryptocurrency wallet. Our emoji shellcode triggers the wallet's backup mechanism dumping the secret key to the serial output. As previously, we generated the emoji shellcode, and successfully executed it on the vulnerable application, with no caching issues.

[9]The full emulation documentation can be found on: https://qemu.readthedocs.io/en/v6.2.0/system/riscv/sifive_u.html

VII. Conclusion

We developed a method for creating versatile, polymorphic RISC-V shellcodes using emoji gadgets. We utilized a code-reuse attack strategy to generate these gadgets and then used them to construct an unpacker for arbitrary code execution. Additionally, we demonstrated how to incorporate polymorphism into our shellcodes using a modified unclogger.

As a demonstration, we provided examples of these shellcodes on the HiFive Unleashed board running a standard Linux operating system and created an automated tool for building them (for both 32 and 64-bit architectures). Our results support the effectiveness of our chosen unpacking approach for writing shellcodes in highly restricted ISA subsets. Future work includes exploring the potential of extending the gadget generation algorithm to other left-linear grammars for even more constrained shellcodes.

References

[1] Aleph-One. "Smashing The Stack For Fun And Profit". In: *Phrack* 49 (1996). URL: http://phrack.org/issues/49/14.html.

[2] Hadrien Barral et al. "ARMv8 Shellcodes from 'A' to 'Z'". In: *Proceedings of the 12th International Conference on Information Security Practice and Experience*. Springer-Verlag, 2016, pp. 354–377. URL: https://link.springer.com/chapter/10.1007/978-3-319-49151-6_25.

[3] Hadrien Barral et al. "RISC-V: #AlphanumericShellcoding". In: *Proceedings of the 13th USENIX Workshop on Offensive Technologies*. USENIX Association, 2019. URL: https://www.usenix.org/system/files/woot19-paper_barral.pdf.

[4] Aditya Basu, Anish Mathuria, and Nagendra Chowdary. "Automatic Generation of Compact Alphanumeric Shellcodes for x86". In: *Proceedings of the 10th International Conference on Information Systems Security*. Springer-Verlag, 2014, pp. 399–410. URL: https://link.springer.com/chapter/10.1007/978-3-319-13841-1_22.

[5] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the 2005 USENIX Annual Technical Conference*. USENIX Association, 2005, pp. 41–46. URL: https://www.cse.iitd.ernet.in/~sbansal/csl862-virt/2010/readings/bellard.pdf.

[6] Dion Blazakis. *Interpreter Exploitation: Pointer Inference and JIT Spraying*. 2010. URL: http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf.

[7] Tyler Bletsch et al. "Jump-oriented Programming: A New Class of Code-reuse Attack". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 30–40. URL: https://www.comp.nus.edu.sg/~liangzk/papers/asiaccs11.pdf.

[8] Vesselin Bontchev. "Future Trends in Virus Writing". In: *International Review of Law, Computers & Technology* 11.1 (1994), pp. 129–146. URL: https://bontchev.nlcv.bas.bg/papers/docs/trends.txt.

[9] Robert Connolly. *Stack Smashing Protector, and _FORTIFY_SOURCE*. Linux from scratch. 2007. URL: https://linuxfromscratch.org/hints/downloads/files/ssp.txt.

[10] The Unicode Consortium. *Data files for Unicode Emoji, Version 14.0*. 2021. URL: https://unicode.org/Public/emoji/14.0.

[11] The Unicode Consortium. *The Unicode Standard, Version 14.0*. 2021. URL: https://www.unicode.org/versions/Unicode14.0.0/UnicodeStandard-14.0.pdf.

[12] The Unicode Consortium. *Unicode Technical Standard #51, Version 14.0*. 2021. URL: https://www.unicode.org/reports/tr51/tr51-21.html.

[13] Palmer Dabbelt et al. *RISC-V ELF psABI Specifcation*. 2016. URL: https://github.com/riscv/riscv-elf-psabi-doc.

[14] Theo Detristan et al. "Polymorphic Shellcode Engine Using Spectrum Analysis". In: *Phrack* 61 (2003). URL: http://phrack.org/issues/61/9.html.

[15] Stephen Dolan. *mov is Turing-Complete*. 2013. URL: https://www.cl.cam.ac.uk/~sd601/papers/mov.pdf.

[16] Christopher Domas. *The M/o/Vfuscator*. 2015. URL: https://recon.cx/2015/slides/recon2015-14-christopher-domas-The-movfuscator.pdf.

[17] Riley Eller. *Bypassing MSB Data Filters for Buffer Overflow Exploits on Intel Platforms*. 2000. URL: https://securiteam.com/securityreviews/5dp140afpq.

[18] Vinod Kumar. "Signature Based Intrusion Detection System Using SNORT". In: *International Journal of Computer Applications & Information Technology* 1 (2012). URL: https://www.ijcait.com/IJCAIT/13/1314.pdf.

[19] David Maynor. *NX: How Well Does Say No to an Attackers eXecution Attempts?* 2005. URL: https://www.blackhat.com/presentations/bh-usa-05/bh-us-05-maynor.pdf.

[20] Oleg Mazonka. *Higher Subleq: Compiler into OISC language*. 2009. URL: http://mazonka.com/subleq/hsq.html.

[21] *Microsoft Security Toolkit Delivers New BlueHat Prize Defensive Technology*. Microsoft, 2012. URL: https://news.microsoft.com/2012/07/25/microsoft-security-toolkit-delivers-new-bluehat-prize-defensive-technology.

[22] Nergal. "Advanced Return-Into-Lib(c) Exploits (PaX Case Study)". In: *Phrack* 11.58 (2001). URL: http://phrack.org/issues/58/4.html.

[23] Kaan Onarligolu et al. "G-Free: Defeating Return-Oriented Programming Through Gadget-Less Binaries". In: *Proceedings of the 2010 Annual Computer Security Applications Conference*. ACM, 2010, pp. 49–58. URL: http://www.eurecom.fr/fr/publication/3235/download/rs-publi-3235.pdf.

[24] David A. Patterson and Carlo H. Sequin. "RISC I: A Reduced Instruction Set VLSI Computer". In: *Proceedings of the 8th annual symposium on Computer Architecture*. IEEE Computer Society Press, 1981, pp. 443–457.

[25] PaX Team. *PaX: Twelve Years of Securing Linux*. 2012. URL: https://pax.grsecurity.net/docs/PaXTeam-LATINOWARE12-PaX-linux-security.pdf.

[26] Ionut Popescu. *Shellcode Compiler*. GitHub repository. 2019. URL: https://github.com/NytroRST/ShellcodeCompiler.

[27] RIX. "Writing IA32 Alphanumeric Shellcodes". In: *Phrack* 57 (2001). URL: http://phrack.org/issues/57/15.html.

[28] Hovav Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. ACM, 2007, pp. 552–561. URL: https://hovav.net/ucsd/dist/geometry.pdf.

[29] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 20191213*. 2019. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

[30] Yves Younan and Pieter Philippaerts. "Alphanumeric RISC ARM Shellcode". In: *Phrack* 66 (2009). URL: http://phrack.org/issues/66/12.html.

[31] Yves Younan et al. "Filter-Resistant Code Injection on ARM". In: *Journal in Computer Virology* 7.3 (2011), pp. 173–188. URL: http://amnesia.gtisc.gatech.edu/~moyix/CCS_09/docs/p11.pdf.

Appendix

A. *Hello World Shellcodes*

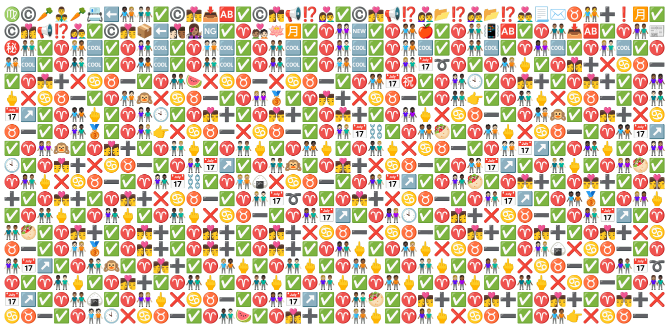
We provide ready-to-use demo emoji shellcodes, also available on the repository (Appendix B). They print "*Hello world!*" on the serial output, when executed on QEMU with the following command:

```
qemu-system-riscv64 -nographic -machine sifive_u
    -device loader,addr=0x80000800,cpu-num=0
    -device loader,file=shellcode.bin,addr=0x80000800
```

Version with only 4 gadgets for $\mathcal{U}$:

Note: 14677 out of the 20090 bytes were randomized using the polymorphic engine.

Version with more gadgets for $\mathcal{U}$:



Note: 3966 out of the 5980 bytes were randomized using the polymorphic engine.

Version heavily optimized for size in bytes:

Note: 84 out of the 1321 bytes were randomized using the polymorphic engine.

B. *Source code*

The full source code used for this article is available at: https://github.com/RischardV/emoji-shellcoding.
It contains all demos and tools used for this paper.

C. *Available gagdets*

Table II lists the number of different instances of instructions found in $\mathcal{G}$ and $\mathcal{G}_{32}$, as defined in section IV-D.

Instruction	**# in $\mathcal{G}$**	**# in $\mathcal{G}_{32}$**
`add`	89	89
`addi`	240	240
`addiw`	**312**	**0**
`addw`	**4**	**0**
`andi`	55	55
`auipc`	100	100
`beq`	4	4
`beqz`	16	16
`bnez`	68	68
`csrrs`	1	1
`fld`	624	624
`flw`	150	150
`fsd`	551	551
`fsw`	**240**	**272**
`j`	148	148
`jal`	**16**	**208**
`lb`	15	15
`lui`	25	25
`lw`	15	15
`lwu`	**3**	**0**
`mv`	76	76
`ori`	3	3
`sd`	**32**	**0**
`slti`	240	240
`sltiu`	210	210
`srai`	36	36
`srli`	39	39
`sub`	17	17
`subw`	**12**	**0**
`sw`	32	32
`xor`	4	4

Table II: Clustering of $\mathcal{G}$ and $\mathcal{G}_{32}$ per instruction. The first column is the instruction mnemonic. The second (resp. third) column give the number of different instances of the said mnemonic found in gadget $\mathcal{G}$ (resp. $\mathcal{G}_{32}$)