Disassembling Terabytes of Random Data with Zig and Capstone to Prove a Point

Jacob Strieb
jstrieb@alumni.cmu.edu

***Abstract*—We use Zig and Capstone to decompress and disassemble terabytes of random data. In doing so, we prove that raw Thumb instructions are more likely to appear in random byte streams than DEFLATE-compressed Thumb instructions. Successful disassembly of Thumb instructions (65% of 128-byte buffers) was significantly more common than successful decompression (0.5% of buffers) or decompression followed by disassembly (0.18% of buffers). This result is partially due to Thumb's high code density, and partially due to decompression failures caused by invalid DEFLATE headers and metadata. Even under conditions contrived to maximize decompression success, disassembly remained more likely. Zig made it easy to write high-performance code with minimal effort over the course of the experiments.**

Introduction

I had a friendly disagreement the other day. Imagine looking for ARM (Thumb[1] mode) instructions in a stream of random bytes. A friend claimed that the random stream is more likely to contain DEFLATE-compressed[2] Thumb instructions than it is to contain uncompressed Thumb instructions.[3] [4] I disagreed.

The gist of his argument was that random bytes have high Shannon Entropy[5] in expectation, and compressed bytes have higher entropy than raw assembled instructions. Thus, compressed streams of assembled instructions will be more likely to occur than raw assembled instructions in sequences of random bytes. I don't debate the premise, but the conclusion does not follow.

I argued that the joint probability of random bytes being a valid DEFLATE stream—*and* that stream inflating to valid instructions—is lower than the probability of random bytes being valid instructions. Having written a DEFLATE/INFLATE implementation from scratch, I know how many ways there are for compressed data to be invalid. I also know that ARM's Thumb instruction set is designed for high code density, which directly implies that a large proportion of possible byte sequences disassemble as valid Thumb instructions. Just because the entropy of typical Thumb code in the wild is lower than that of typical compressed data doesn't mean that the theoretical maximum entropy of valid Thumb is lower too. He was not convinced. I was nerd sniped.[6]

Since he has a PhD in an applied field of computer science, I figured he is most likely to be compelled by strong experimental results in a poorly-written, LaTeX-typeset paper with lots of graphs that he can skim and pretend to have read fully. (I'm only mostly joking.)

This is that paper.[7] [8]

AI Usage Disclosure

Large Language Models (LLMs) were **not** used for any part of this research. The code and prose are entirely human-written. I wrote a separate post detailing some opinions about LLMs that you should read if you want to know more.[9]

Methods

We write Zig[10] code to generate random byte sequences. Then we try to disassemble the bytes using Capstone.[11] We also try to inflate and then disassemble the bytes. We record successes and failures to output statistics at the end.

We test code samples with Zig version 0.14.1.[12] Zig changes fast, so it is likely that code samples will need to be modified to work on other versions.

The full code repository is available on GitHub.[13]

Note that all of the code snippets look much better in the web version of this paper.[14]

[1]https://en.wikipedia.org/wiki/ARM_architecture_family#Thumb

[2]https://en.wikipedia.org/wiki/Deflate

[3]He was actually arguing that ZLIB-compressed Thumb instructions (the format that wraps DEFLATE and is found in PNG files, not the zlib library) would be most likely, but I'm refuting a stronger version of his argument. A DEFLATE stream without headers and checksums is strictly more likely to randomly appear than a stream with them.

[4]https://www.rfc-editor.org/rfc/rfc1950.html

[5]https://en.wikipedia.org/wiki/Entropy_(information_theory)

[6]https://www.explainxkcd.com/wiki/index.php/356:_Nerd_Sniping

[7]There is also a web version of this paper.

[8]https://jstrieb.github.io/posts/random-instructions

[9]https://jstrieb.github.io/posts/llm-thespians/

[10]https://ziglang.org/

[11]https://www.capstone-engine.org/

[12]Version 0.14.1 was the latest at the time of writing, but not the latest at the time of publishing.

[13]https://github.com/jstrieb/random-instructions/

[14]https://jstrieb.github.io/posts/random-instructions

Capstone and Zig

Zig's build system makes it easy to automatically clone, compile, and link against Capstone.[15] [16] Assuming a `build.zig` file already exists in the working directory (use `zig init` to create an empty project if not), the following command updates the `build.zig.zon` file with Capstone as a dependency.

```
zig fetch \
  --save=capstone \
  'git+https://github.com/'\
  'capstone-engine/capstone.git#5.0.6'
```

Then we add the following to `build.zig`.[17] [18] [19]

```
// There should already be a main executable
// created in the build script:
// const exe = b.addExecutable(.{ ... });

// Build Capstone
const capstone_dep = b.dependency(
  "capstone",
  .{},
);
const capstone_cmake = b.addSystemCommand(&.{
  "cmake",
  "-B",
  "build",
  "-DCMAKE_BUILD_TYPE=Release",
  "-DCAPSTONE_BUILD_SHARED_LIBS=1",
});
capstone_cmake.setCwd(capstone_dep.path(""));
const capstone_make = b.addSystemCommand(&.{
  "cmake",
  "--build",
  "build",
});
capstone_make.setCwd(capstone_dep.path(""));
capstone_make.step.dependOn(
  &capstone_cmake.step
);
exe.step.dependOn(&capstone_make.step);

// Add the Capstone lib and include dirs
// so we can import capstone.h and link
// against the Capstone shared object or DLL
exe.linkLibC();
exe.addLibraryPath(capstone_dep.path("build"));
exe.linkSystemLibrary("capstone");
exe.addIncludePath(
  capstone_dep.path("include")
);
```

In our `main.zig` file, we import and use Capstone.[20] [21] Our `Capstone.disassemble` method returns a `usize` that represents the percentage of disassembled instruction bytes (between 0 and 100, inclusive). We only care the proportion of bytes that successfully disassemble; we don't care about the disassembled instructions themselves.

```
const std = @import("std");
const capstone = @cImport({
  @cInclude("capstone/capstone.h");
});

fn Capstone(
  arch: capstone.cs_arch,
  mode: c_int,
) type {
  return struct {
    engine: capstone.csh,
    const Self = @This();

    pub fn init() !Self {
      var engine: capstone.csh = undefined;
      if (capstone.cs_open(
        arch,
        mode,
        &engine,
      ) != capstone.CS_ERR_OK)
      {
        return error.CapstoneInitFailed;
      }
      if (capstone.cs_option(
        engine,
        capstone.CS_OPT_SKIPDATA,
        capstone.CS_OPT_ON,
      ) != capstone.CS_ERR_OK) {
        return error.CapstoneInitFailed;
      }
      return .{ .engine = engine };
    }

    pub fn deinit(self: *Self) void {
      _ = capstone.cs_close(&self.engine);
    }

    pub fn disassemble(
      self: Self,
      b: []const u8,
```

[15] I could have used the version of Capstone ported to the Zig build system, but I didn't think to look for it when I first started this. Building it myself was the first thing I tried, and it worked.

[16] https://github.com/allyourcodebase/capstone

[17] The version of `build.zig` in the repo is more complicated because we cross-compile Capstone using Zig and (optionally) statically link, instead of building only for the host architecture and enforcing dynamic linking. The statically-linked version runs slower, likely due to known performance issues with the default musl libc allocator. Unsurprisingly, allocating in the hot path is bad for performance.

[18] https://github.com/jstrieb/random-instructions/blob/master/build.zig

[19] https://nickb.dev/blog/default-musl-allocator-considered-harmful-to-performance/

[20] It appears that Capstone does dynamic memory allocation when we call the `cs_disasm` function. Allocation in a hot loop is less than ideal, and we pray that whatever allocator they are using is thread-safe. The code runs fast enough for now, though, so no need to patch Capstone internals or use `cs_opt_mem`.

[21] https://github.com/capstone-engine/capstone/blob/accf4df62f1fba6f92cae692985d27063552601c/include/capstone/capstone.h#L242-L248

```
    ) usize {
      if (b.len == 0) return 0;
      var instructions: [*c]capstone.cs_insn =
        undefined;
      const count = capstone.cs_disasm(
        self.engine,
        @ptrCast(b),
        b.len,
        0,
        0,
        &instructions,
      );
      defer capstone.cs_free(
        instructions,
        count,
      );
      var instruction_bytes: usize = 0;
      for (instructions, 0..count) |i, _| {
        if (i.id != 0) {
          instruction_bytes += i.size;
        }
      }
      return 100 * instruction_bytes / b.len;
    }
  };
}

test "basic disassembly" {
  var cs: Capstone(
    capstone.CS_ARCH_ARM,
    capstone.CS_MODE_THUMB,
  ) = try .init();
  defer cs.deinit();

  try std.testing.expectEqual(
    100,
    cs.disassemble("\xe0\xf9\x4f\x07"),
  );
  try std.testing.expectEqual(
    100,
    cs.disassemble("\x00\x00"),
  );
  try std.testing.expectEqual(
    0,
    cs.disassemble("\x00"),
  );
  try std.testing.expectEqual(
    50,
    cs.disassemble("\xff\xff\x00\x00"),
  );
  try std.testing.expectEqual(
    50,
    cs.disassemble(
      "\x00\x00\xff\xff\xff\xff\x00\x00",
    ),
  );
}
```

Generating Random Bytes

We generate random bytes using ChaCha[22] seeded by cryptographic randomness. ChaCha is a fast CSPRNG,[23] so it outputs high-quality random bytes, but won't slow down our hot loop.

```
var random = random: {
  var seed: [
    std.Random.ChaCha.secret_seed_length
  ]u8 = undefined;
  std.crypto.random.bytes(&seed);
  var chacha = std.Random.ChaCha.init(seed);
  break :random chacha.random();
};

// Fill a buffer with random bytes
var in_buffer: [128]u8 = undefined;
random.bytes(&in_buffer);
```

Inflating Bytes

In Zig 0.14.1, the standard library INFLATE implementation[24] operates on reader and writer streams.[25] [26] [27] [28] We build those streams out of pre-allocated buffers, one of which contains the random input data.

```
var out_buffer: [128 * 1024]u8 = undefined;
var in_stream = std.io.fixedBufferStream(
  &in_buffer,
);
var out_stream = std.io.fixedBufferStream(
  &out_buffer,
);

try std.compress.flate.inflate.decompress(
  .raw,
  in_stream.reader(),
  out_stream.writer(),
)
```

Parallelizing

To maximize throughput, we run our main loop across all CPU cores. We keep statistics local to each thread, only compiling global results from the thread-local results when the loop terminates. That way the hot loop is lock-free, and we avoid overhead from contention. The downside of

[22]https://en.wikipedia.org/wiki/Salsa20
[23]https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator
[24]https://ziglang.org/documentation/0.14.1/std/#std.compress.flate.inflate.decompress
[25]In the latest version of Zig (0.15.2), released right before this was published, the INFLATE API was changed to accommodate the new reader and writer API. Those changes are probably related to the proposed async I/O implementation. We don't use them here because this code was written with an older version of Zig.
[26]https://ziglang.org/download/0.15.1/release-notes.html#reworked-stdcompressflate
[27]https://ziglang.org/download/0.15.1/release-notes.html#New-stdIoWriter-and-stdIoReader-API
[28]https://kristoff.it/blog/zig-new-async-io/

this strategy is that we can't print aggregated statistics while the trials are running.[29] [30]

```
const total_iterations: usize = 100_000_000;

var results: struct {
  disasm_count: u64 = 0,
  inflate_count: u64 = 0,
  inflate_disasm_count: u64 = 0,
  lock: std.Thread.Mutex = .{},

  const Self = @This();

  pub fn update(
    self: *Self,
    disasm_count: u64,
    inflate_count: u64,
    inflate_disasm_count: u64,
  ) void {
    self.lock.lock();
    defer self.lock.unlock();
    self.disasm_count += disasm_count;
    self.inflate_count += inflate_count;
    self.inflate_disasm_count +=
      inflate_disasm_count;
  }

  fn print(self: *Self) !void {
    // ...
  }
} = .{};

fn loop(iterations: usize) !void {
  var disasm_count: u64 = 0;
  var inflate_count: u64 = 0;
  var inflate_disasm_count: u64 = 0;
  // Other initialization

  for (0..iterations) |_| {
    // Do stuff...
  }

  results.update(
    disasm_count,
    inflate_count,
    inflate_disasm_count,
  );
}

pub fn main() !void {
  const thread_count = @min(
    std.Thread.getCpuCount() catch 1,
    1024,
  );
  var thread_buffer: [1024]std.Thread =
    undefined;
  const threads =
    thread_buffer[0..thread_count];
  const iterations =
    total_iterations / thread_count;
  for (threads) |*t| {
    t.* = try std.Thread.spawn(
      .{},
      loop,
      .{iterations},
    );
  }
  for (threads) |t| {
    t.join();
  }
  try results.print();
}
```

[29]If we really wanted to display statistics while the code runs, we could rewrite the loop to use lock-free atomics for each of the totals. Here, we elide progress updates in the interest of keeping the code simple.

[30]https://ziglang.org/documentation/0.14.1/std/#std.atomic.Value

Collecting Decompression Errors

To collect decompression errors, we use Zig compile-time (comptime) metaprogramming. At comptime, we build an array of all possible errors returned from the decompression function. We use that array to initialize a corresponding array of counts, where we tally the number of occurrences of each error.

```
const flate = std.compress.flate;
const decompress = flate.inflate.decompress;
const errors = errors: {
  const error_set = @typeInfo(
    @typeInfo(
      @TypeOf(result: {
        var in_stream =
          std.io.fixedBufferStream(&[_]u8{});
        break :result decompress(
          .raw,
          in_stream.reader(),
          std.io.NullWriter{
            .context = {},
          },
        );
      }),
    ).error_union.error_set,
  ).error_set.?;
  const num_errors = error_set.len;
  var result: [num_errors]anyerror =
    undefined;
  for (error_set, 0..) |err, i| {
    result[i] = @field(anyerror, err.name);
  }
  break :errors result;
};
var counts = [_]usize{0} ** errors.len;
```

Notice that to reify[31] an error from its name, we use

[31]https://en.wikipedia.org/wiki/Reification_(computer_science)

`@field(anyerror, name)`. This works because `anyerror` can be considered an enum that collects all errors reachable in the compilation unit.[32] [33] Hence, every possible Zig error is a field of that global enum.

Reifying errors from their names is important for inlining the mapping between returned errors and their corresponding index in the `counts` array. It is faster to check the equality of the received error and the expected error directly than to serialize the received error to a name and then do string comparison.

```
if (std.compress.flate.inflate.decompress(
  .raw,
  in_stream.reader(),
  std.io.NullWriter{ .context = {} },
)) {
  inflate_count += 1;
} else |e| {
  inline for (errors, 0..) |err, i| {
    if (err == e) {
      counts[i] += 1;
      break;
    }
  }
}
```

Final Script

The full script that combines each of these parts is available in the GitHub repository.[34]

Results

The most important result is that **successful disassembly is over 125x more common than successful decompression, and over 350x more common than decompression *and* disassembly**.

Disassembly Results

Given that about 65% of 128-byte buffers consist of at least 95% valid Thumb instructions, there are two natural questions: since most Thumb instructions are two bytes long, what proportion of 2-byte sequences are valid Thumb instructions? And what percentage of 128-byte buffers consist *entirely* of valid Thumb instructions?

According to our measurements, any 2-byte sequence has an 89.3% chance of disassembling as a valid Thumb instruction. Evidently, Thumb has very high code density.

Note that some few Thumb instructions assemble as four bytes,[35] so we cannot assume that every pair of bytes

Total	1,000,000,000
Disassembled	642,628,412
Decompressed	5,037,627
Decompressed and disassembled	1,810,170

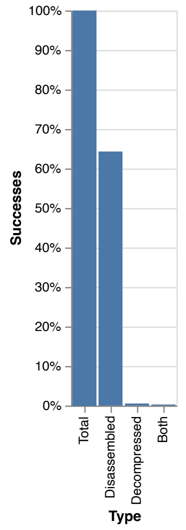

Fig. 1. Statistics for 1,000,000,000 random 128-byte buffers, counting instances that disassemble 95% or more of the bytes.

Total	1,000,000,000
Disassembled	892,800,538

Fig. 2. Statistics for 1,000,000,000 random 2-byte buffers, counting instances that disassemble completely.

has an independent, identically (Bernoulli[36]) distributed probability of being part of a valid assembly sequence. Our measurements indicate that 855,008,565 (or 85.5%) of 1,000,000,000 4-byte buffers disassembled completely as valid Thumb instructions, compared with the $0.892^2 \approx 79.7\%$ rate we would expect if all Thumb instructions were 2-bytes long.

Though we can't assume every pair of bytes is independent of every other pair, we may assume that every 4-byte sequence is independent and identically distributed in terms of the probability of successful disassembly. Using this assumption, we can approximately predict the proportion of fully disassembled 128-byte sequences.

$$0.855008565^{(128/4)} = 0.006653572 \approx 6.65\%$$

Compare our prediction with the actual, measured incidence of 4.47% of 128-byte buffers that completely disassemble.

Even when we require 100% of bytes to disassemble as valid Thumb instructions, there are still an order of magnitude more buffers that disassemble than decompress.

If we assume groups of four consecutive bytes have an independent and identically distributed probability of successful disassembly, we expect the likelihood of completely disassembling a buffer to decay exponentially relative to the

[32]In practice, it's not exactly an enum. Otherwise we would be able to use it with `std.EnumArray` instead of rolling our own `counts` array indexing.

[33]https://ziglang.org/documentation/0.14.1/std/#std.EnumArray

[34]https://github.com/jstrieb/random-instructions/blob/master/src/main.zig

[35]https://developer.arm.com/documentation/ddi0403/d/Application-Level-Architecture/The-Thumb-Instruction-Set-Encoding/32-bit-Thumb-instruction-encoding?lang=en

[36]https://en.wikipedia.org/wiki/Bernoulli_distribution

Total	1,000,000,000
Disassembled	44,726,130
Decompressed	5,036,173
Decompressed and disassembled	1,577,559

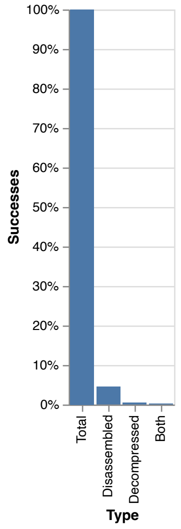

Fig. 3. Statistics for 1,000,000,000 random 128-byte buffers, counting instances that disassemble 100% of the bytes.

size of the buffer. In other words, the probability of finding a sequence that does *not* disassemble should quickly approach 1 as the size of the buffer increases. The experimental results validate that theoretical prediction.

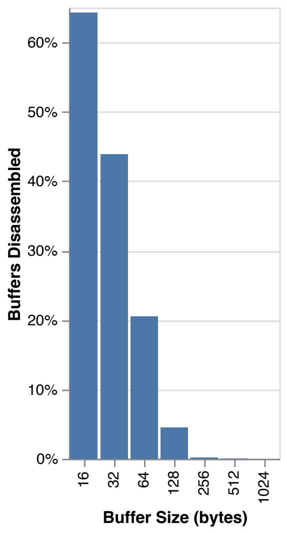

Fig. 4. Statistics for 1,000,000,000 random buffers of each size (in bytes), counting instances that disassemble 100% of the bytes.

Also, unsurprisingly, as the minimum threshold for successful disassembly decreases, the proportion of successes increases for each buffer size.

Decompression Results

Based on our experiments, we know why disassembly is likely to succeed. But, independent of successful disassembly, why is decompression likely to fail?

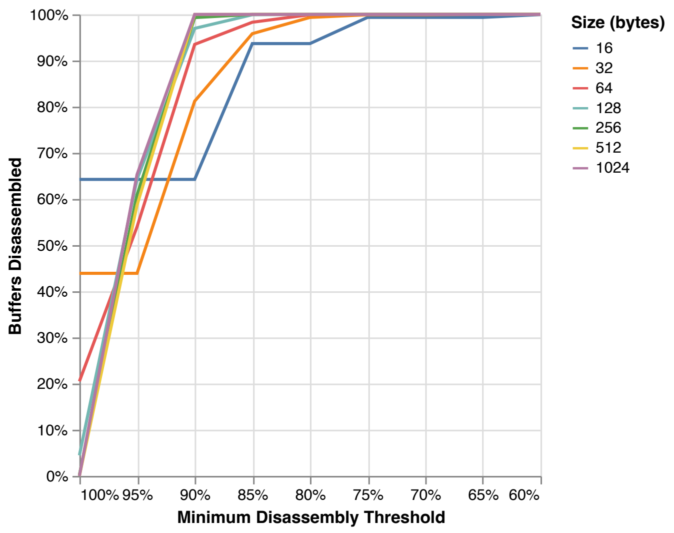

Fig. 5. Statistics for 1,000,000,000 random buffers of each size (in bytes) and threshold for disassembly, counting instances that disassemble at least the threshold percent of the bytes.

The rate of successful decompression is fairly constant for buffers above 32 bytes in size.

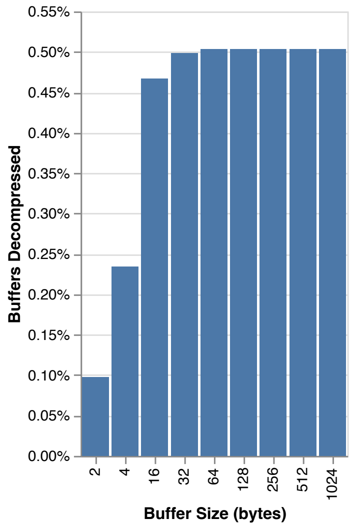

Fig. 6. Statistics for 1,000,000,000 random buffers of each size (in bytes), counting instances that decompress successfully.

This suggests that the majority of errors from unsuccessful decompression happen in the first 32 bytes of the buffer, which makes sense, since that is where the small DEFLATE header would be found for valid compressed streams. In other words, it's unlikely that random bytes make a valid DEFLATE header. But if the header is valid, the rest is likely to successfully decompress, regardless of buffer size.

The errors returned when decompression fails corroborate the hypothesis that failures are primarily caused by invalid DEFLATE metadata.

Total	1,000,000,000
Success	5,030,965
Wrong Stored Block Nlen	251,255,219
Invalid Block Type	251,254,609
Invalid Match	214,744,118
Oversubscribed Huffman Tree	146,011,334
Incomplete Huffman Tree	74,024,330
Invalid Dynamic Block Header	31,141,638
Invalid Code	26,438,043
Missing End of Block Code	95,816
End of Stream	3,928

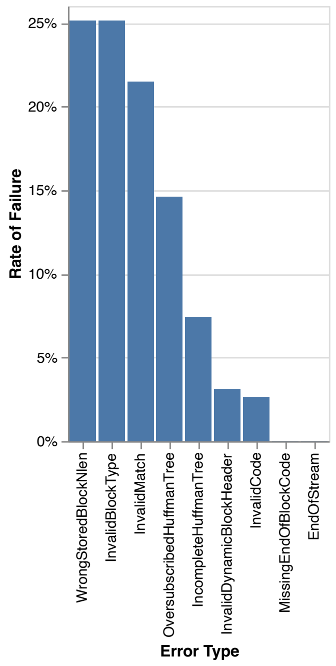

Fig. 7. Statistics for 1,000,000,000 random 128-byte buffers, counting the errors that are returned when decompression fails.

To understand these errors completely, we must explore the structure of the DEFLATE stream.

DEFLATE Structure: A DEFLATE stream consists of one or more blocks. The first bit of every block denotes whether it is the final block in the stream.[37] If the first bit is `1`, the decompressor terminates when it sees an "end of block" token. On the other hand, if the first bit is `0`, the decompressor will look for *another* valid DEFLATE header after an end of block token in the stream. Thus, in that case, there is an additional opportunity for any of the errors that could have caused decompression to fail on the first block header.

As a result, we can reduce the failure rate for most classes of error by fixing the first bit of the stream to be `1`, so that only one valid header is required, instead of two or more.

The second and third bits of the DEFLATE block header denote how the data is encoded.[38] [39] [40] There are four possible options for these two bits, and three of the four options are considered valid.

A value of `11` for the second and third bits is considered invalid, and the decompressor will return an "invalid block type" error in that case. Because this pattern occurs for one in four random streams (in expectation), it accounts for approximately 25% of the attempts that end in decompression failure. Setting the second and third bits to `11` makes 100% of decompression attempts fail.

If the second and third bits are `00`, the data is included "raw" – it is not encoded or compressed before being included in the stream. In this case, the second and third bytes of the stream represent the length of the raw data, and the two bytes after that are their ones' complement (logical negation). According to the DEFLATE specification,[41] the data after the first byte in a raw DEFLATE stream looks like this:

```
  2   3   4   5   6...
+---+---+---+---+=========================+
|  LEN  | NLEN  |... LEN bytes of data ...|
+---+---+---+---+=========================+
```

The likelihood of the fourth and fifth bytes being the exact ones' complement of the second and third bytes is:

$$1\ /\ 2^{16} \approx 0.0015258789\%$$

The extremely low probability of success for the raw block type explains another ~25% of decompression attempts that end in failure with the "wrong stored block nlen" error. Setting the second and third bits to `00` makes almost all of the decompression attempts fail.

In the other two block modes, raw data is first LZ77-encoded,[42] then Huffman-encoded[43] to compress it. In Huffman coding, an efficient bit representation of each symbol is generated from the frequency distribution of symbols in the uncompressed data. Symbols that appear more frequently will be represented using fewer bits. The rules for decoding symbols from bits are stored as a "Huffman tree" data structure.

If the second and third bits of the DEFLATE stream are `01`, the data is decoded using a hard-coded ("static") Huffman tree whose structure is specified in the DEFLATE specification. If the second and third bits are `10` instead, the Huffman tree was built dynamically, and is included in the stream itself. Using the hard-coded Huffman tree typically leads to worse compression ratios than using a dynamic Huffman tree, but the static tree was more useful

[37]When looking at the bytes themselves, the "first bit" here actually refers to the least significant bit of the first byte. In the RFC, the bytes are arranged from right to left, so the first bit of the stream is the rightmost bit.

[38]Similar to the first bit, the second and third bits here are actually the second and third least significant bits of the first byte. They are also reversed when serialized into the stream. Blame the authors of the DEFLATE RFC or maybe Phil Katz of PKZIP fame for this confusing situation.

[39]https://en.wikipedia.org/wiki/Phil_Katz

[40]https://en.wikipedia.org/wiki/PKZIP

[41]https://datatracker.ietf.org/doc/html/rfc1951#page-11

[42]https://en.wikipedia.org/wiki/LZ77_and_LZ78

[43]https://en.wikipedia.org/wiki/Huffman_coding

	First bit 0	First bit 1
Total	1,000,000,000	1,000,000,000
Success	56,372	10,010,648
Wrong Stored Block Nlen	252,530,381	250,005,035
Invalid Block Type	252,495,113	249,999,800
Invalid Match	215,816,449	213,698,259
Oversubscribed Huffman Tree	146,737,762	145,281,110
Incomplete Huffman Tree	74,398,039	73,635,423
Invalid Dynamic Block Header	31,290,580	30,987,133
Invalid Code	26,574,767	26,283,441
Missing End of Block Code	96,424	95,216
End of Stream	4,113	3,935

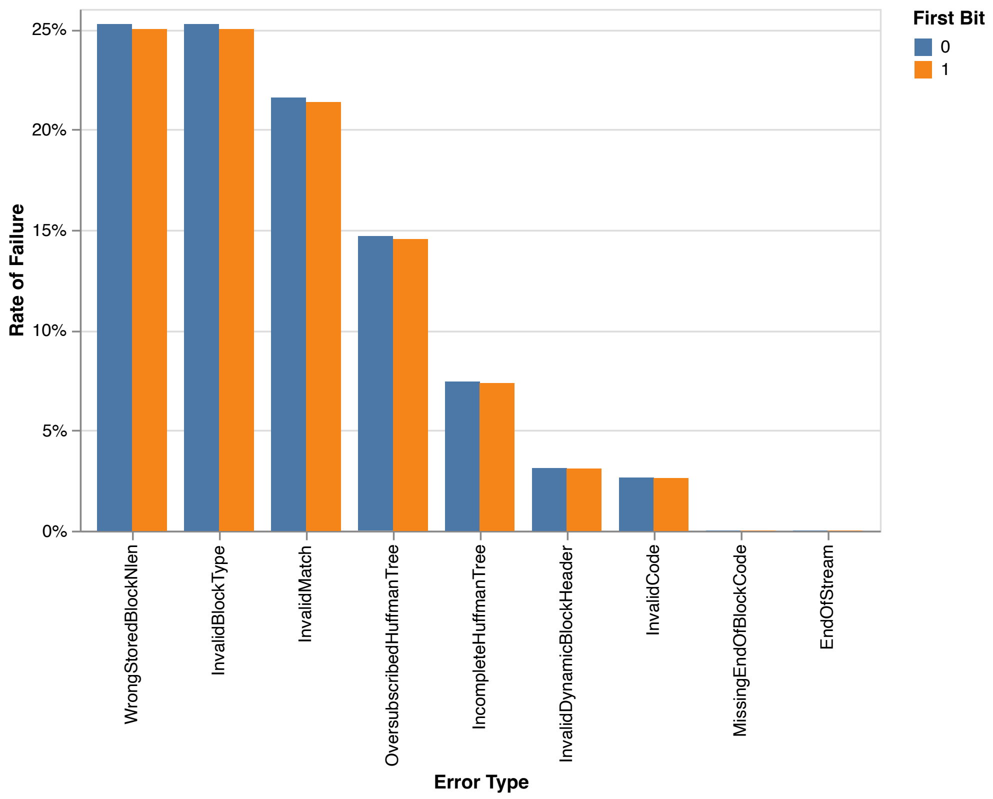

Fig. 8. Statistics for 1,000,000,000 random 128-byte buffers with the first bit fixed to either `0` or `1`, counting the errors that are returned when decompression fails.

Total	1,000,000,000
Success	0
Invalid Block Type	1,000,000,000

Fig. 9. Statistics for 1,000,000,000 random 128-byte buffers with the first three bits fixed to `111`, counting the errors that are returned when decompression fails.

Total	1,000,000,000
Success	24
Wrong Stored Block Nlen	999,984,702
End of Stream	15,274

Fig. 10. Statistics for 1,000,000,000 random 128-byte buffers with the first three bits fixed to `100`, counting the errors that are returned when decompression fails.

for our purpose of decompressing random bytes since it presents fewer opportunities for erroneous header data.

In order to refute the strongest version of the claim that decompressing and then disassembling random bytes is more likely to succeed than just disassembling random bytes, we re-run our initial test after fixing the first three bits of the random buffer to `110`. That way the otherwise random bytes are always considered the final block in the stream, and always use static Huffman codes to avoid header errors.

Though the rates of success are much closer, even in this most generous case, disassembly is more likely than decompression, and once again much more likely than

	Dynamic Huffman Code	Static Huffman Code
Total	1,000,000,000	1,000,000,000
Success	0	40,055,267
End of Stream	233	570
Invalid Code	15,254	105,125,696
Oversubscribed Huffman Tree	581,086,654	0
Incomplete Huffman Tree	294,586,455	0
Missing End of Block Code	380,197	0
Invalid Match	0	854,818,467
Invalid Dynamic Block Header	123,931,207	0

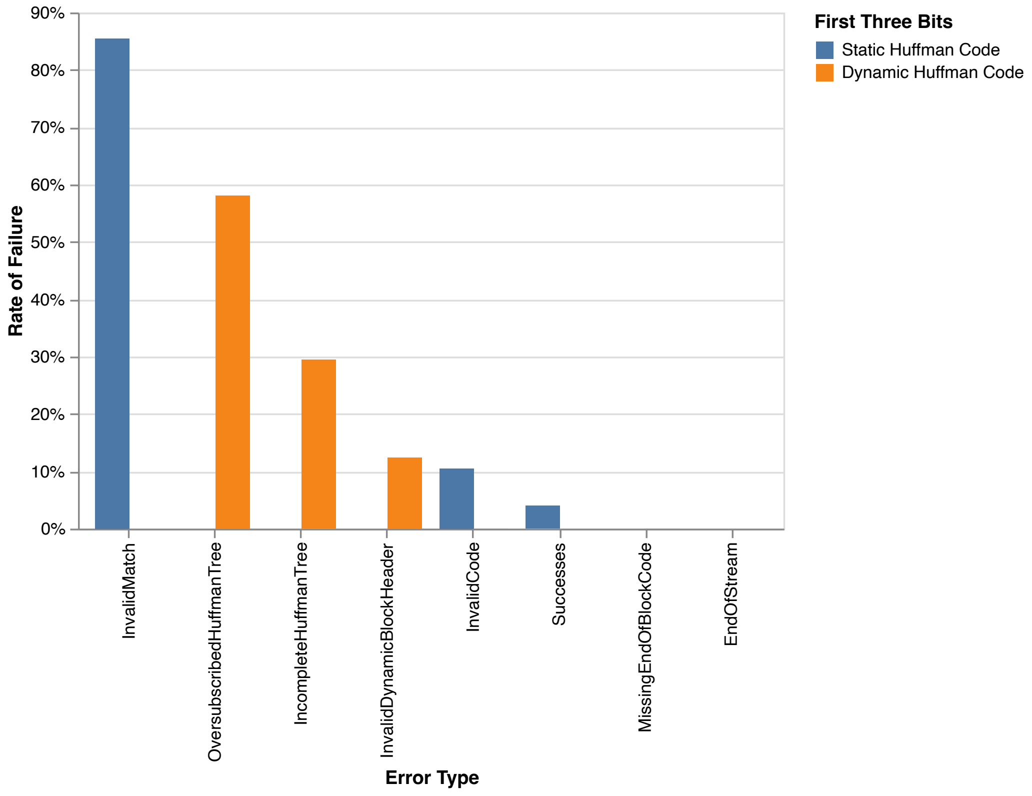

Fig. 11. Statistics for 1,000,000,000 random 128-byte buffers with the first three bits fixed to either '101' or '110', counting the errors that are returned when decompression fails.

decompression *and* disassembly.

Other Architectures

Besides ARM in Thumb mode, Capstone supports disassembling a wide variety of other architectures and modes.[44] Using C macro reflection in Zig,[45] we can re-run our randomized disassembly across all available architectures and modes.

In many architectures, instruction disassembly is stateful, and is conditioned on previous instructions (especially when assembled instructions have variable length, such as in x86). As a result, many other architectures do not admit the assumption that aligned, fixed-sized buffer subsections have an independent probability of being disassembled successfully.

Performance

Overall, the Monte Carlo code that powers these "experiments" is very performant. Zig is a fast language, and even without any deliberate optimization, the code can disassemble and decompress one billion 128-byte buffers in about a half hour on the 10-year-old, 48-core server we used for testing.

```
$ time ./zig-out/bin/random_instructions \
    --total-iterations 1_000_000_000 \
    --disassembly-threshold 100
[ ... ]
```

[44]https://github.com/capstone-engine/capstone/blob/5.0.6/include/capstone/capstone.h#L132-L231

[45]https://jstrieb.github.io/posts/c-reflection-zig/

Total	1,000,000,000
Disassembled	44,724,849
Decompressed	40,056,259
Decompressed and disassembled	12,543,383

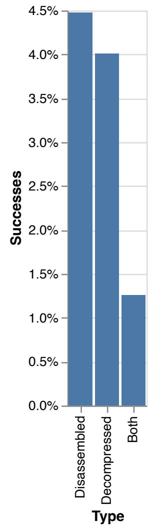

Fig. 12. Statistics for 1,000,000,000 random 128-byte buffers with the first three bits fixed to `110` before decompressing (for static Huffman trees), counting instances that disassemble 100% of the bytes.

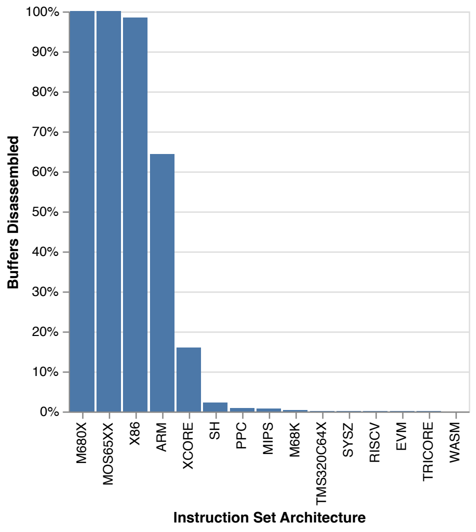

Fig. 13. Statistics for 10,000,000 random 128-byte buffers, counting instances that disassemble at least 95% of the bytes.

```
real    30m35.158s
user    1396m34.236s
sys     1m0.426s
```

Most of the computation time was spent in Capstone. Without profiling in detail, we suspect that overhead from dynamic memory allocation performed inside of Capstone during disassembly accounts for a significant portion of that time. The code is fast enough for testing that it wasn't worth experimenting with different allocator implementations in Capstone. It likewise wasn't worth doing any detailed profiling.

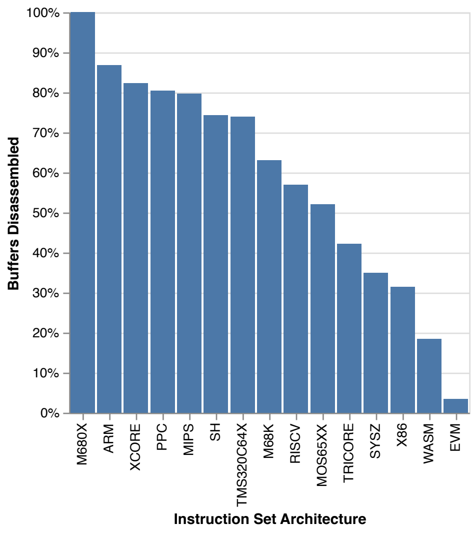

Fig. 14. Statistics for 10,000,000 random 4-byte buffers, counting instances that disassemble 100% of the bytes.

The program's running times scaled approximately linearly with the size of the input buffer. There was no evidence of the disassembly threshold impacting the running time, nor was there an impact on decompression time resulting from fixing particular starting bits.

Adding evidence to the theory that Capstone was the slowest part of the code, the throughput of a different script that only decompressed without attempting to disassemble[46] was much higher. In a randomly chosen run, that script decompressed one billion 128-byte buffers in 5 minutes and 46 seconds.

```
$ time ./zig-out/bin/random_inflate \
    --total-iterations 1_000_000_000
[ ... ]

real    5m46.619s
user    261m22.503s
sys     1m6.833s

$ time ./zig-out/bin/random_inflate \
    --total-iterations 1_000_000_000 \
    --first-bits 0b011 \
    --num-bits 3
[ ... ]

real    5m47.705s
user    262m18.896s
sys     1m6.702s
```

In other words, without any particular optimizations, that program had a decompression throughput upper-bounded by:[47]

[46]https://github.com/jstrieb/random-instructions/blob/master/src/inflate.zig

[47]This is an upper bound because the decompressor returned early if it hit an error. It would only process the full ~120GB of data if all of the buffers were valid. Instead, it processed *at most* 120GB.

$$\frac{128 \times 1000000000 \text{ bytes}}{5 \times 60 + 46 \text{ seconds}} = 369942197 \text{ bytes / second}$$
$$\approx \boxed{345 \text{ MB/s}}$$

The code and raw data used to generate the graphs are on GitHub.[48]

Conclusion

Our theoretical and experimental results point clearly to one conclusion: Thumb instructions have very high code density, and are much more likely to occur in sequences of random bytes than compressed data (especially more likely than compressed data containing valid Thumb instructions).

Even in cases where we artificially modify the random data before decompressing it to make it as likely as possible to decompress, it is still less likely to decompress than disassemble.

Zig makes it fun to write fast code, and was pretty awesome to use for this exploratory analysis. If you've not heard of it, you should definitely check it out.

Further Research

Other ISAs supported by Capstone show varying code density, and it would be worth researching better candidates than Thumb for maximizing the likelihood of successful disassembly of random byte sequences. For example, does M680X actually have absurdly high code density, or was there a methodological error or code bug artificially inflating its disassembly success rate?

We have done no work to assess whether the instruction sequences appearing in our random buffers are of particular interest. It would be unsurprising if an instruction set with relatively low code density (and therefore low rate of random disassembly success) might have a higher likelihood of useful instruction sequences appearing in random byte streams, compared with a dense instruction set with a high likelihood of disassembly.

There might be other ways to cook up the header of a DEFLATE stream to maximize the decompression likelihood of an otherwise random stream. For example, is it possible to craft a "dynamic" Huffman tree (that we insert at the beginning of every random sequence) that maximizes the likelihood of successful decompression?

If anyone continues this research, please email me or use my contact form to notify me of your results.[49]

References

Nice try, but this isn't actually a real research paper. I hope you weren't hoping for APA-style citations or any such nonsense. The important references are linked as footnotes.

[48]https://github.com/jstrieb/random-instructions/blob/master/graphs/

[49]https://jstrieb.github.io/about#contact

Acknowledgments

Thanks to Logan Snow[50] and Amy Liu[51] for reading early versions of this paper.

Shout out to my anonymous friend whose incorrect—but strongly-held—convictions inspired this work in the first place.

[50]https://github.com/lsnow99

[51]https://www.youtube.com/@Liuamyjane