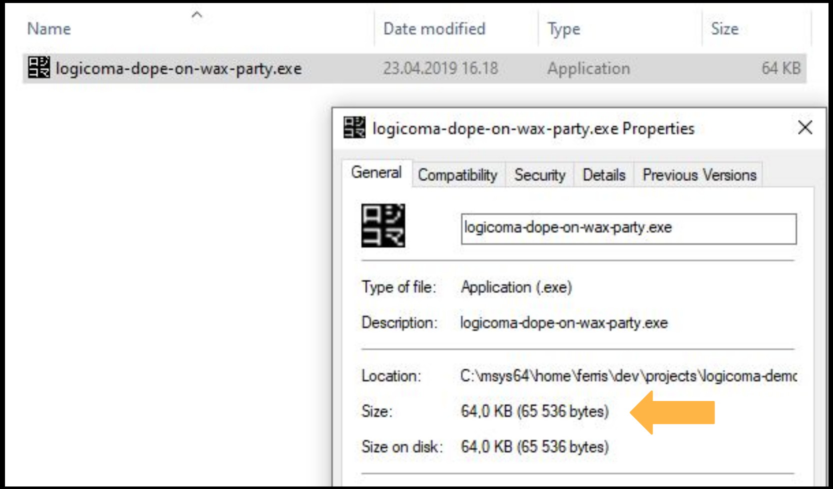
modern(ish) 64k intro compression

jake “ferris” taylor / logicoma

64k intro lightning round

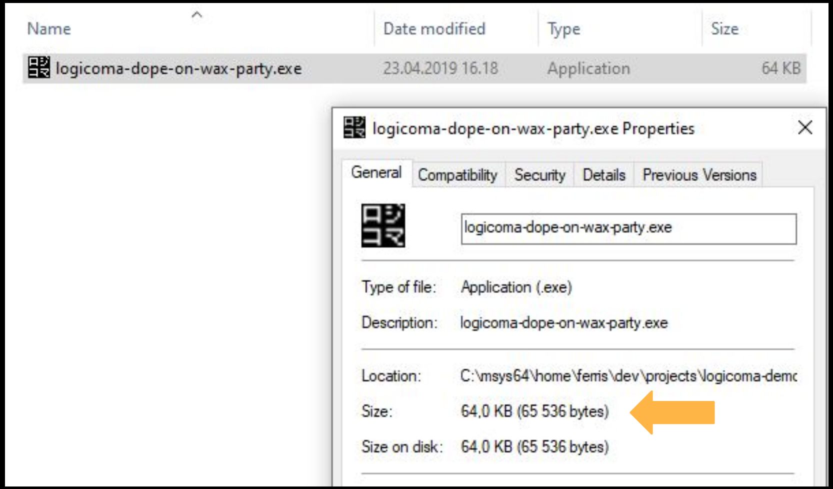
64k intro lightning round

- demo that fits in 65536 bytes (often less because lazy)



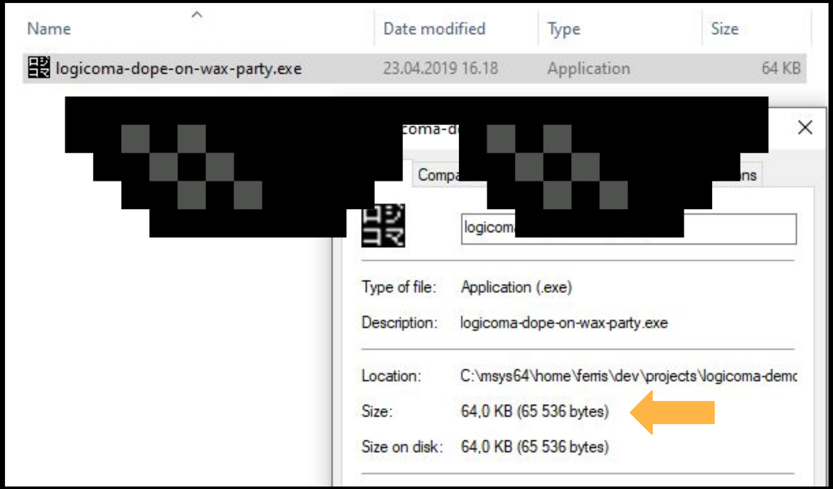
64k intro lightning round

- demo that fits in 65536 bytes (often less because lazy)
- single executable, no external media (except OS/drivers)



64k intro lightning round

- demo that fits in 65536 bytes (often less because lazy)
- single executable, no external media (except OS/drivers)
- coolest demoscene category imo (totally not biased)



here’s what some of them look like

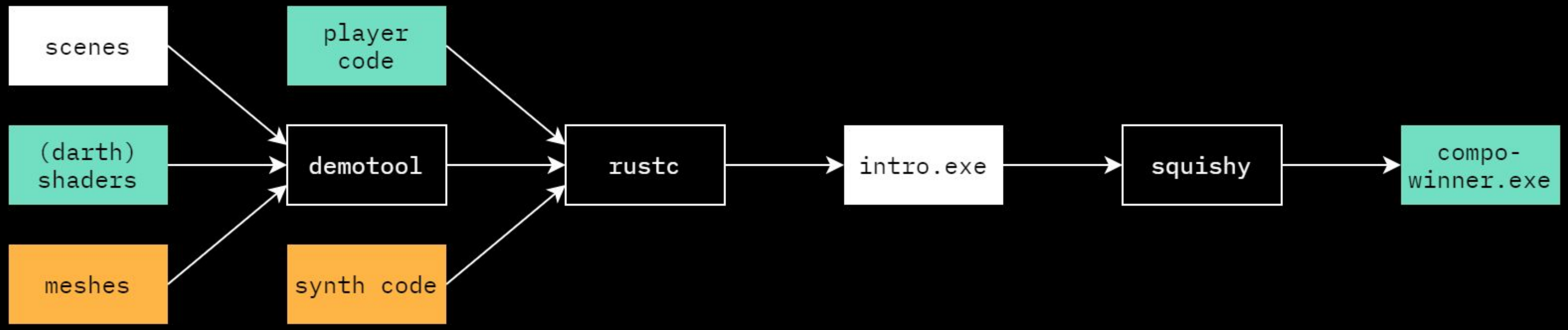
here's what some of them look like

(4 of these used squishy btw)

what’s a squishy?

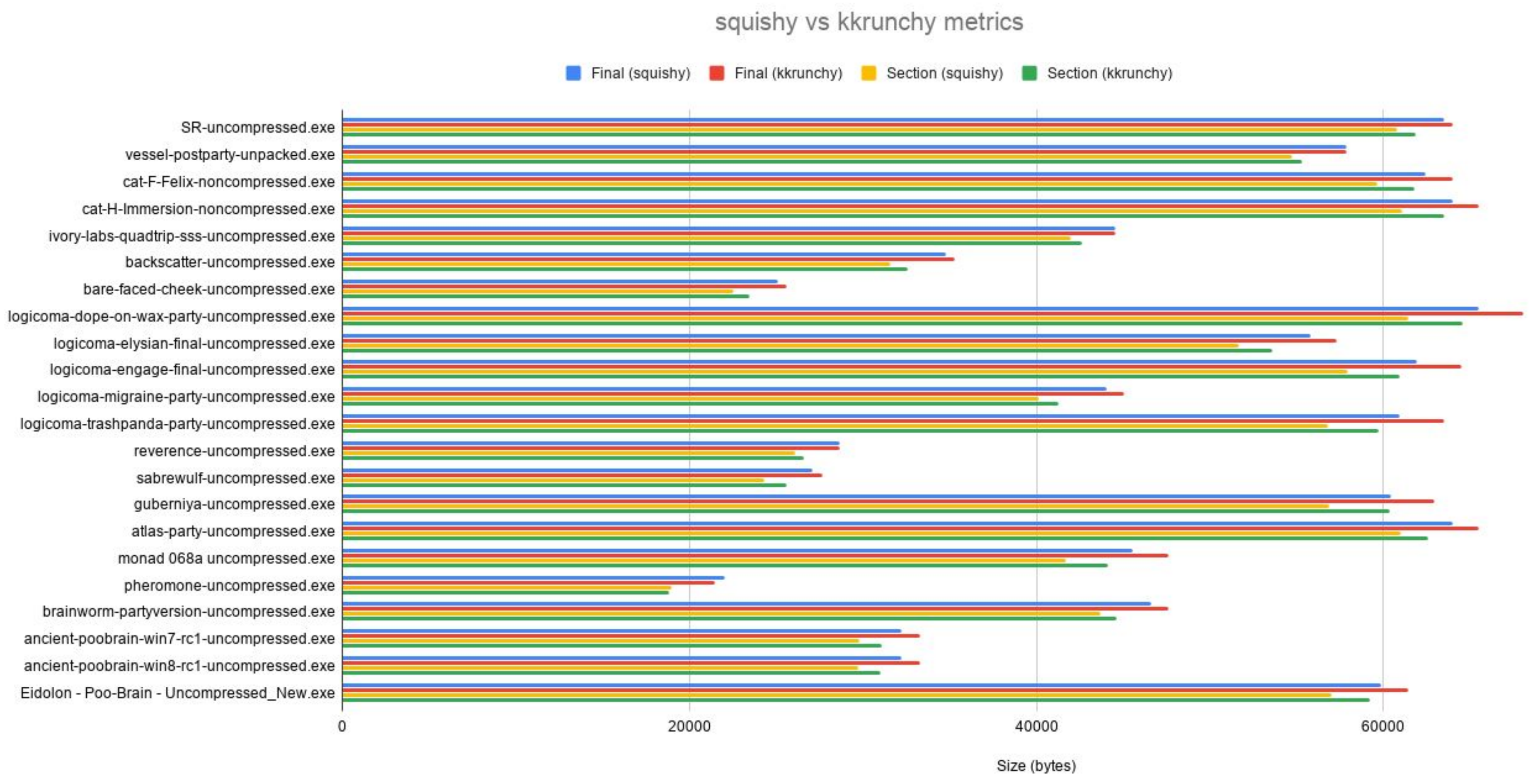
what's a squishy?

- our (logicoma's) executable compressor
- developed since 2016
- specifically built for 64k
  - much heavier compression engine than 1k/4k/8k
- http://logicoma.io/squishy/



what’s a squishy?

- is it good?

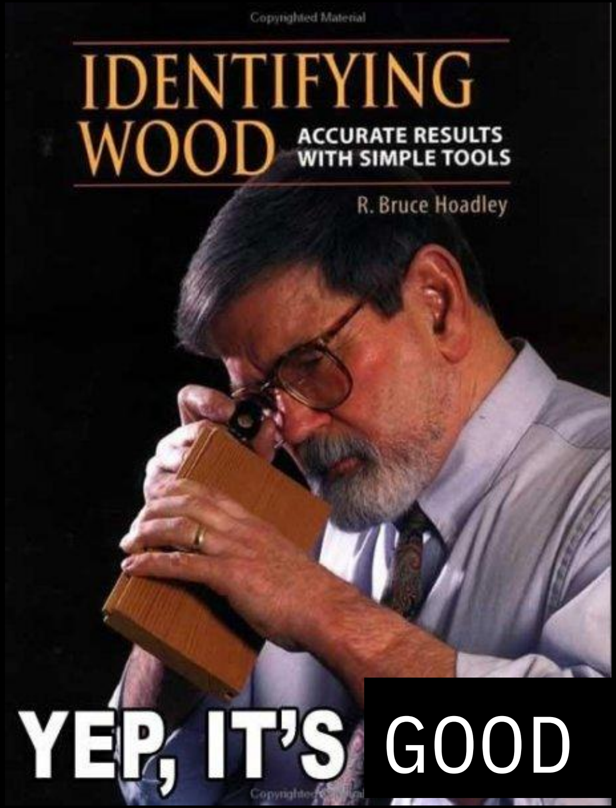

squishy vs kkrunchy metrics
Final (squishy)
Final (kkrunchy)
Section (squishy)
Section (kkrunchy)
SR-uncompressed.exe
vessel-postparty-unpacked.exe
cat-F-Felix-noncompressed.exe
cat-H-Immersion-noncompressed.exe
ivory-labs-quadtrip-sss-uncompressed.exe
backscatter-uncompressed.exe
bare-faced-cheek-uncompressed.exe
logicoma-dope-on-wax-party-uncompressed.exe
logicoma-elysian-final-uncompressed.exe
logicoma-engage-final-uncompressed.exe
logicoma-migraine-party-uncompressed.exe
logicoma-trashpanda-party-uncompressed.exe
reverence-uncompressed.exe
sabrewulf-uncompressed.exe
guberniya-uncompressed.exe
atlas-party-uncompressed.exe
monad 068a uncompressed.exe
pheromone-uncompressed.exe
brainworm-partyversion-uncompressed.exe
ancient-poobrain-win7-rc1-uncompressed.exe
ancient-poobrain-win8-rc1-uncompressed.exe
Eidolon - Poo-Brain - Uncompressed_New.exe
0
20000
40000
60000
Size (bytes)

what’s a squishy?

- is it good?

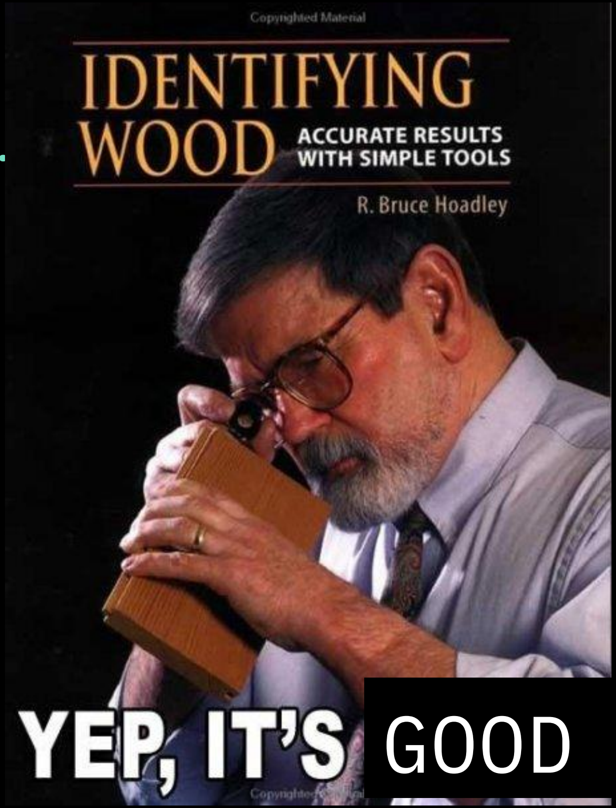
what’s a squishy?

- is it good?



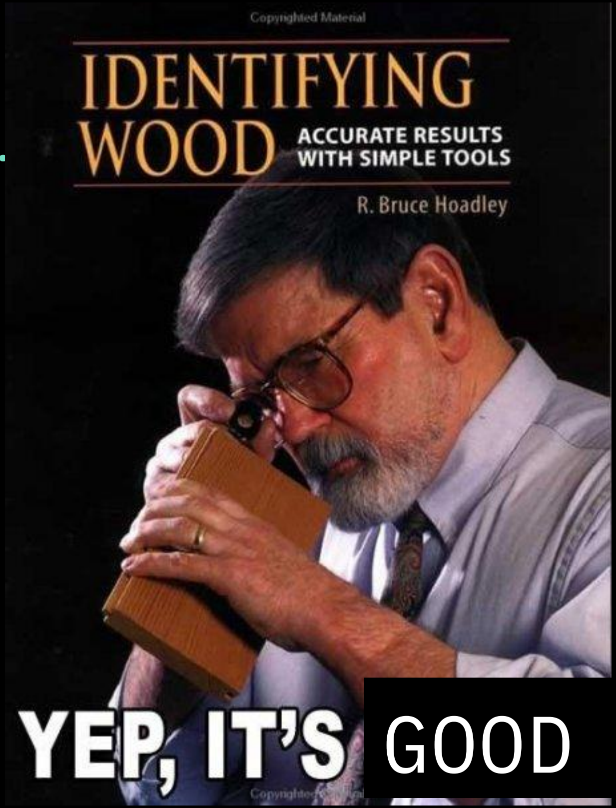
what's a squishy?

- is it good?
- could be better...



what’s a squishy?

- is it good?
- could be better..
- still wip :)



executable compression

full disclosure..

- squishy was most focused on improving the compression engine
- all other techniques are icing on the cake
- as long as the main engine kicks butt!

full disclosure..

- to me, this is the boring part
  - and often covered in other talks!
- but overview is necessary
- so I’ll describe some stuff

executable compression
(for reals this time)

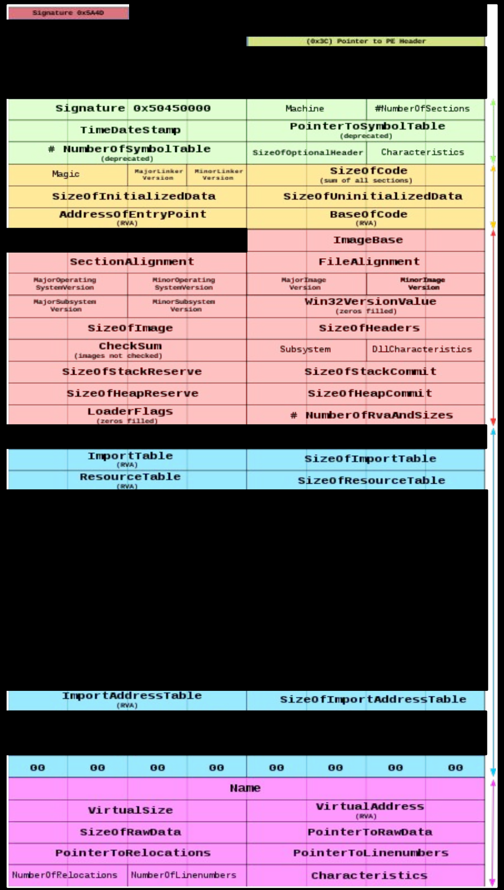
executable compression

- .exe -> .exe (instead of .whatever -> .ziphead)
- OS needs to be able to execute it!
- most metadata not needed
- some things can be folded/overlapped
- function imports?
- resources need to be uncompressed (but not all!)

executable compression

https://en.wikipedia.org/wiki/Portable_Executable#/media/File:Portable_Executable_32_bit_Structure_in_SVG_fixed.svg

executable compression



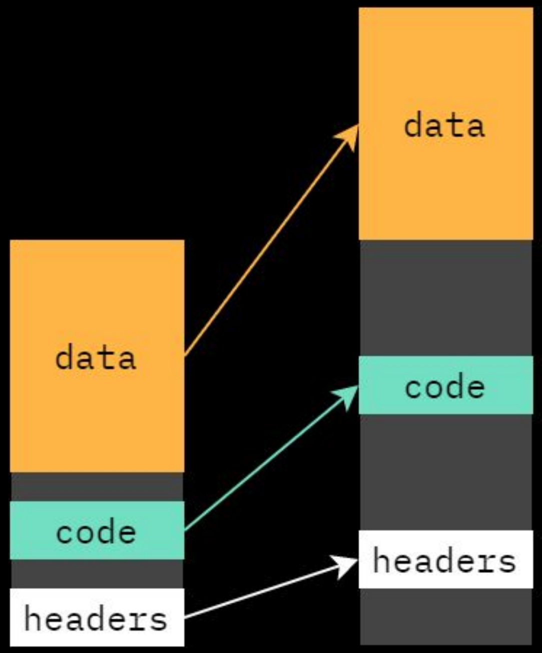
https://en.wikipedia.org/wiki/Portable_Executable#/media/File:Portable_Executable_32_bit_Structure_in_SVG_fixed.svg

executable compression

https://en.wikipedia.org/wiki/Portable_Executable#/media/File:Portable_Executable_32_bit_Structure_in_SVG_fixed.svg

executable compression

executable compression

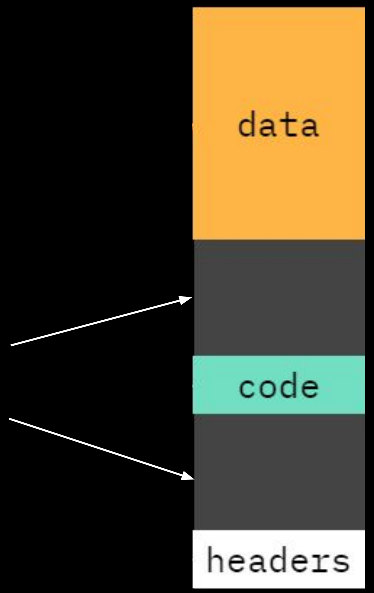


executable compression

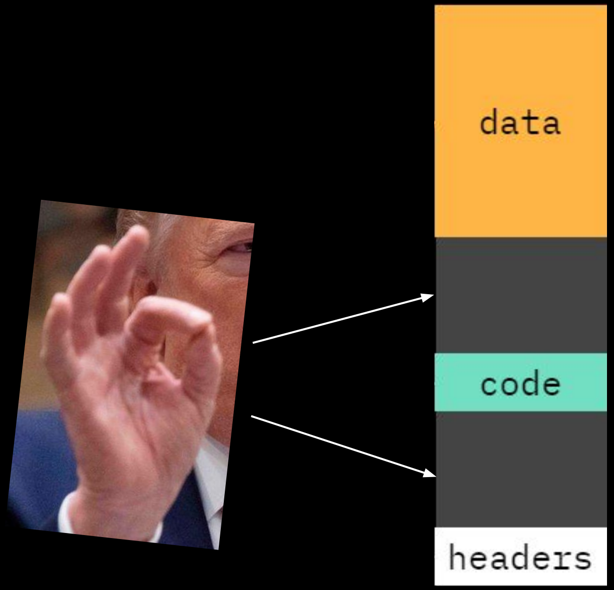
https://en.wikipedia.org/wiki/Portable_Executable#/media/File:Portable_Executable_32_bit_Structure_in_SVG_fixed.svg

executable compression

executable compression



executable compression



executable compression

- .exe’s need .dll’s to do all the fun stuff

executable compression

- .exe’s need .dll’s to do all the fun stuff
  - system APIs to make a window (into the void)
  - graphics APIs (for that cool shader you stole from shadertoy)
  - sound APIs (to spill blood from the ears of your enemies)

executable compression

- .exe’s need .dll’s to do all the fun stuff
  - system APIs to make a window (into the void)
  - graphics APIs (for that cool shader you stole from shadertoy)
  - sound APIs (to spill blood from the ears of your enemies)
  - network APIs to download and show the real demo

executable compression

- .exe’s need .dll’s to do all the fun stuff
  - system APIs to make a window (into the void)
  - graphics APIs (for that cool shader you stole from shadertoy)
  - sound APIs (to spill blood from the ears of your enemies)
  - network APIs to download and show the real demo
- called imports
  - dll names (eg. D3DX_SpikeBall_1337.dll) + function names (eg. glUtenFree)

executable compression

- .exe’s need .dll’s to do all the fun stuff
  - system APIs to make a window (into the void)
  - graphics APIs (for that cool shader you stole from shadertoy)
  - sound APIs (to spill blood from the ears of your enemies)
  - network APIs to download and show the real demo
- called imports
  - dll names (eg. D3DX_SpikeBall_1337.dll) + function names (eg. glUtenFree)
- strings can be big!

executable compression

- common method is import-by-hash

executable compression

- common method is import-by-hash
    - don’t store function names as strings, hash them, and store hashes
    - walk system .dll’s manually and hash strings, find matches

executable compression

- common method is import-by-hash
  - don’t store function names as strings, hash them, and store hashes
  - walk system .dll’s manually and hash strings, find matches
- chicken-and-egg problem:
  - how can we walk system .dll’s without loading them?
  - walk opaque PEB structures to get kernel32.lib (YES, these CAN AND DO change between Windows versions!)
  - import LoadLibraryA and go from there

executable compression

- common method is import-by-hash
  - don’t store function names as strings, hash them, and store hashes
  - walk system .dll’s manually and hash strings, find matches
- chicken-and-egg problem:
  - how can we walk system .dll’s without loading them?
  - walk opaque PEB structures to get kernel32.lib (YES, these CAN AND DO change between Windows versions!)
  - import LoadLibraryA and go from there
- what about hash collisions?

executable compression

- for compatibility, we do it more safely

executable compression

- for compatibility, we do it more safely
  - no hash collision risk
  - more importantly, no opaque structure walking

executable compression

- for compatibility, we do it more safely
  - no hash collision risk
  - more importantly, no opaque structure walking
- use standard import mechanism to import LoadLibraryA and GetProcAddress from kernel32.dll

executable compression

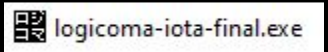
- for compatibility, we do it more safely
  - no hash collision risk
  - more importantly, no opaque structure walking
- use standard import mechanism to import LoadLibraryA and GetProcAddress from kernel32.dll
- compress library/function strings(/ordinals)

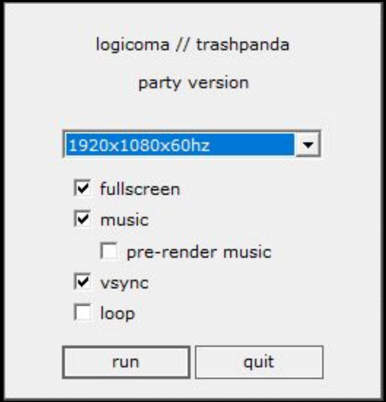
executable compression

- for compatibility, we do it more safely
  - no hash collision risk
  - more importantly, no opaque structure walking
- use standard import mechanism to import LoadLibraryA and GetProcAddress from kernel32.dll
- compress library/function strings(/ordinals)
- resolve at runtime using above fn’s

executable compression

- some special handling for resources

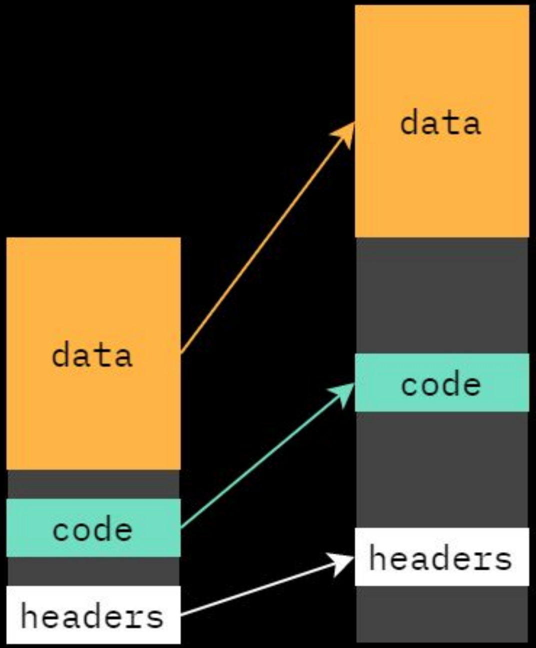




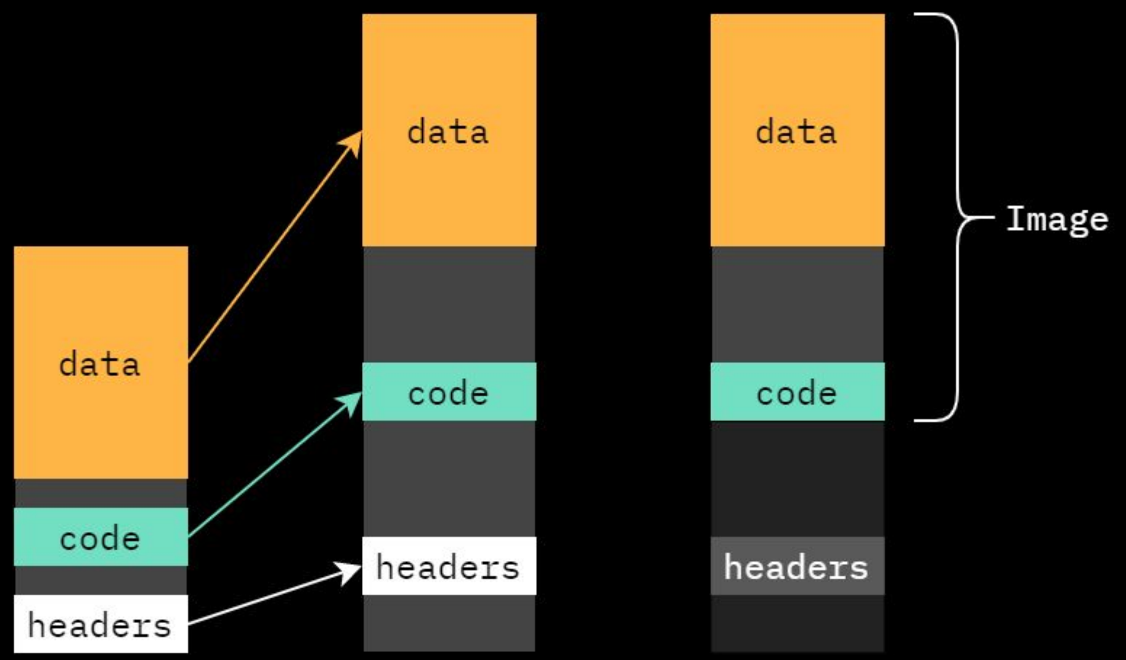
packing flow

packing flow

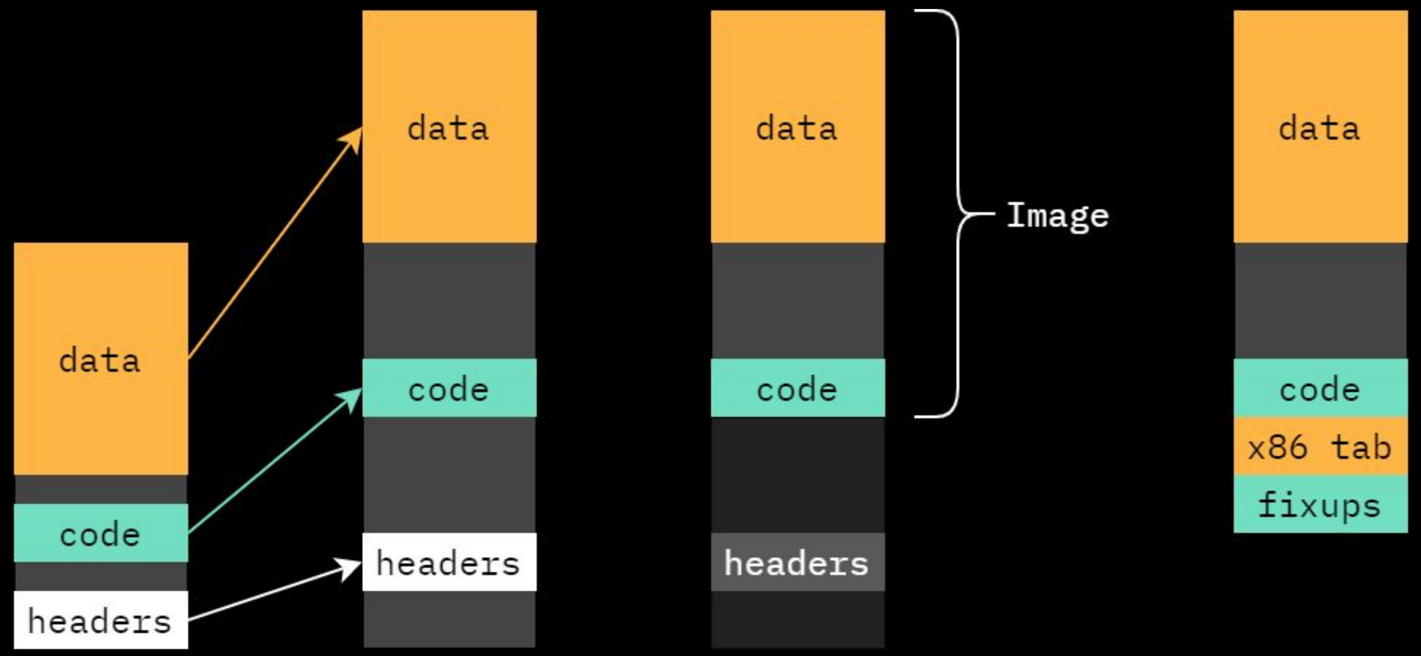
packing flow



packing flow

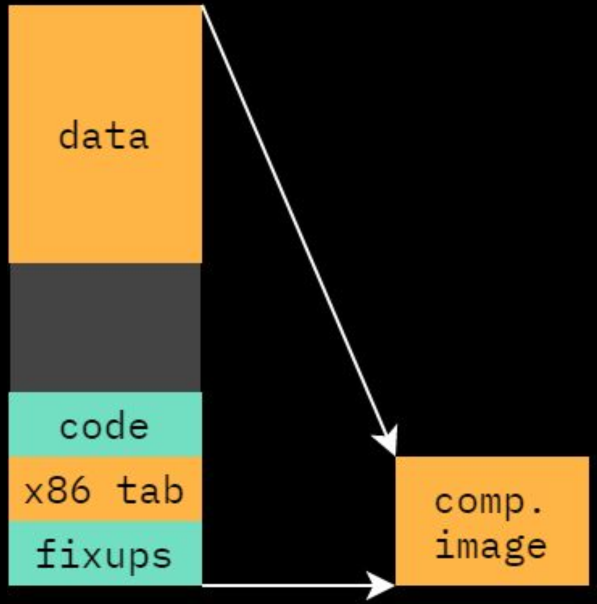


packing flow



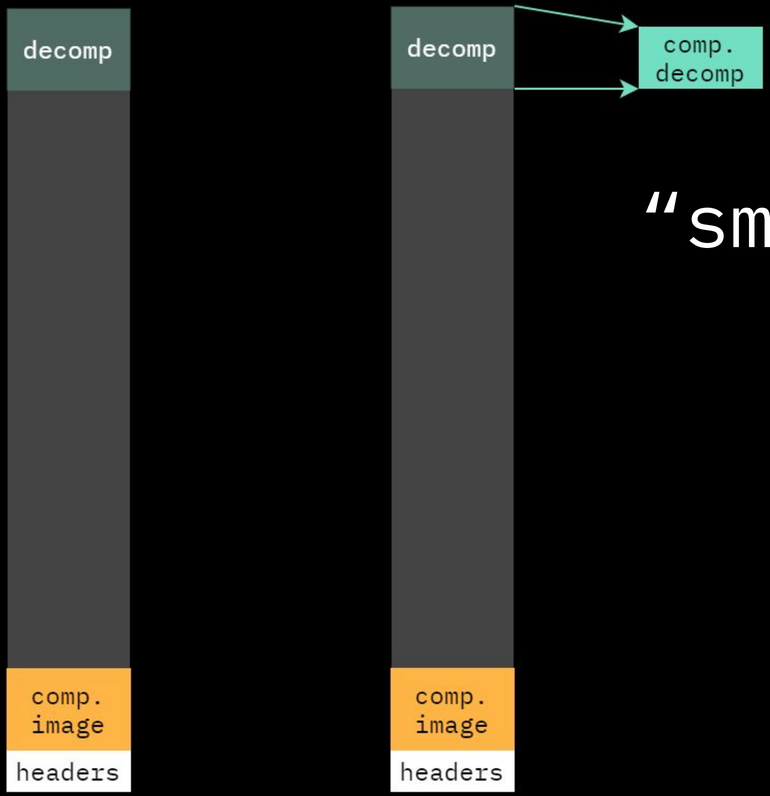
packing flow

“big squish”



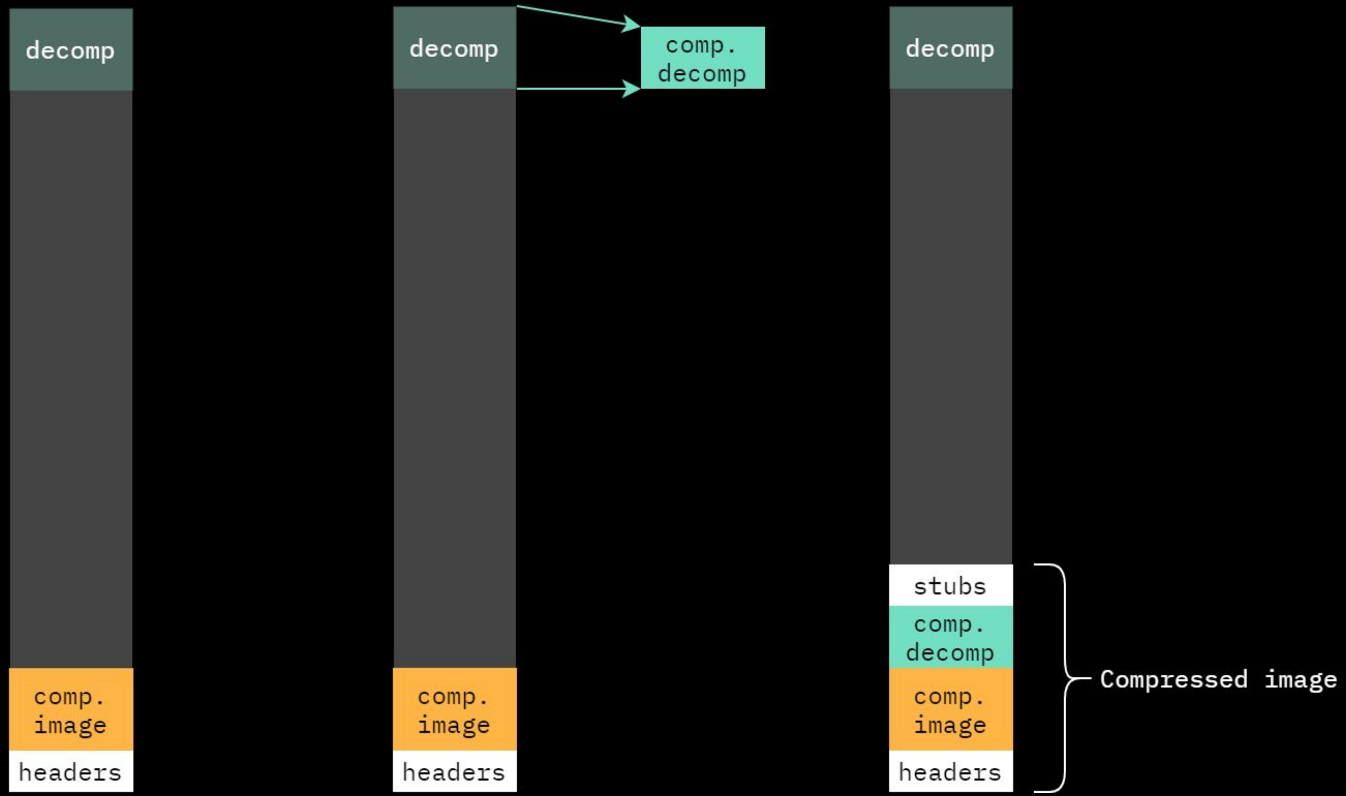
packing flow

packing flow

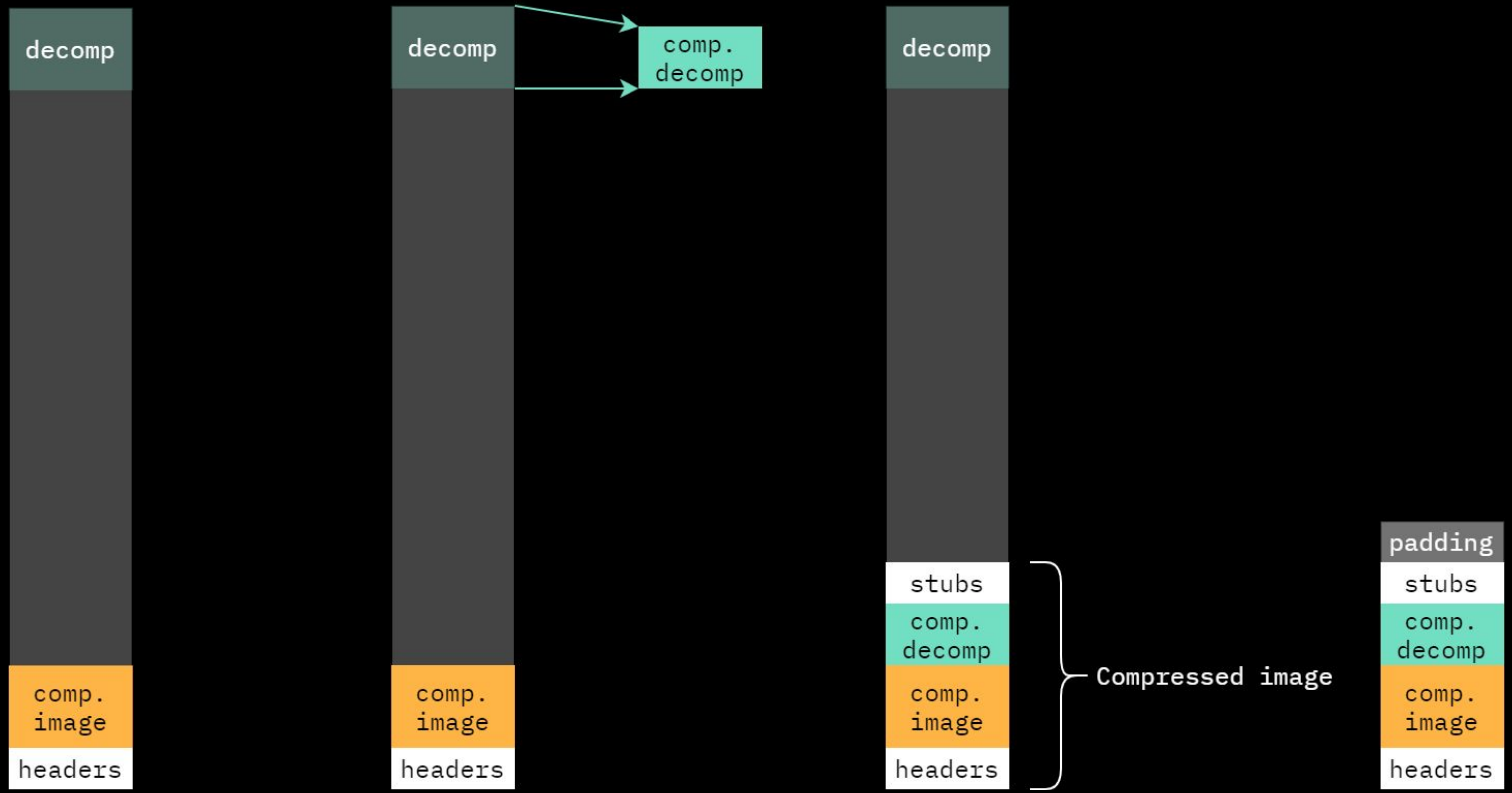


“smol squish”

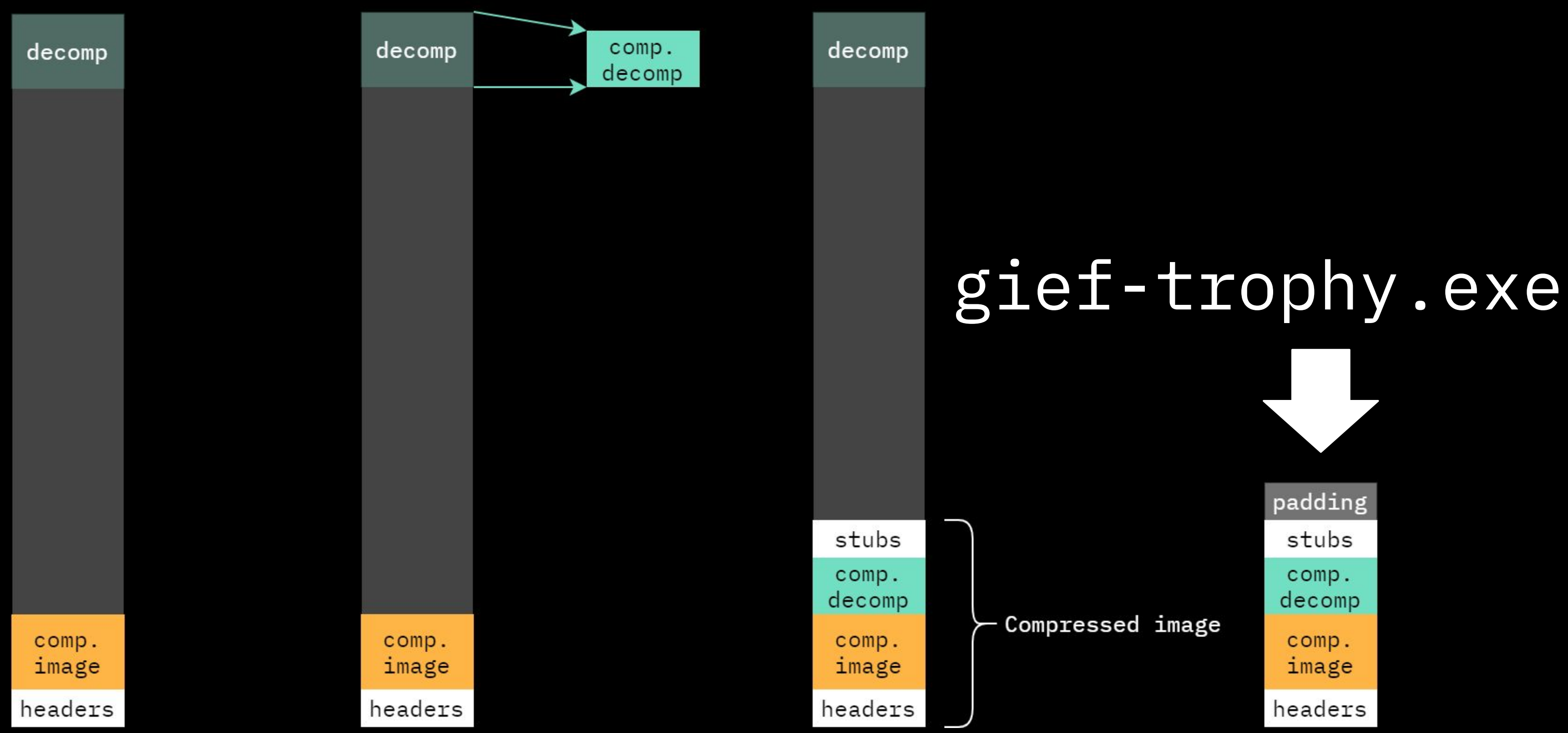
packing flow



packing flow

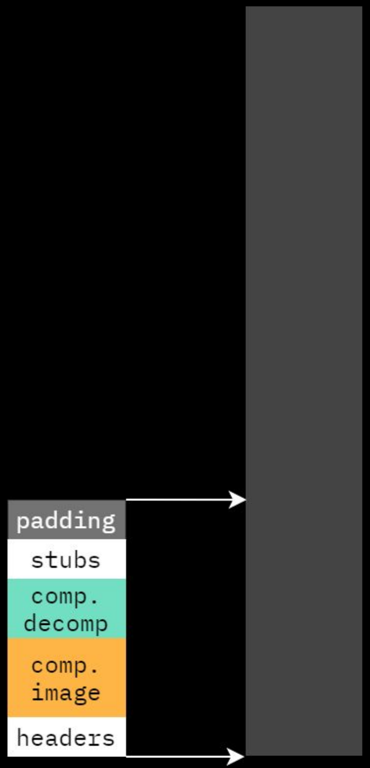


packing flow

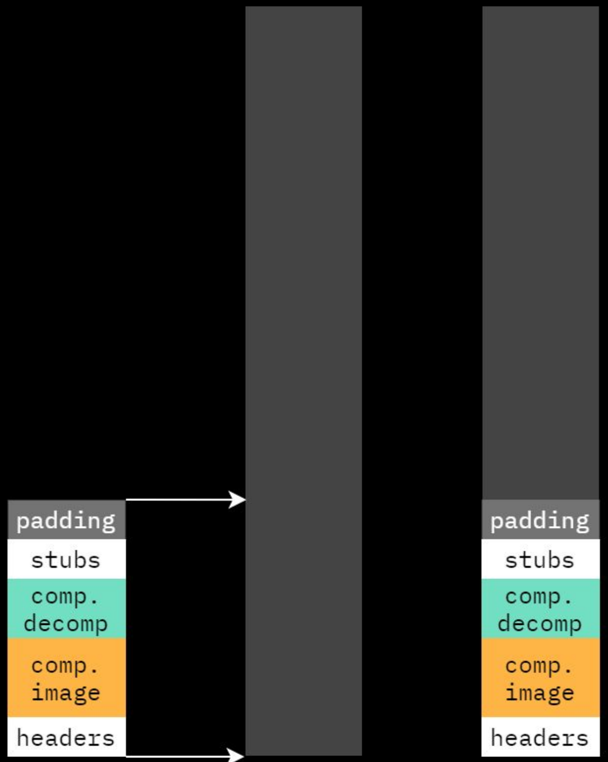


unpacking flow

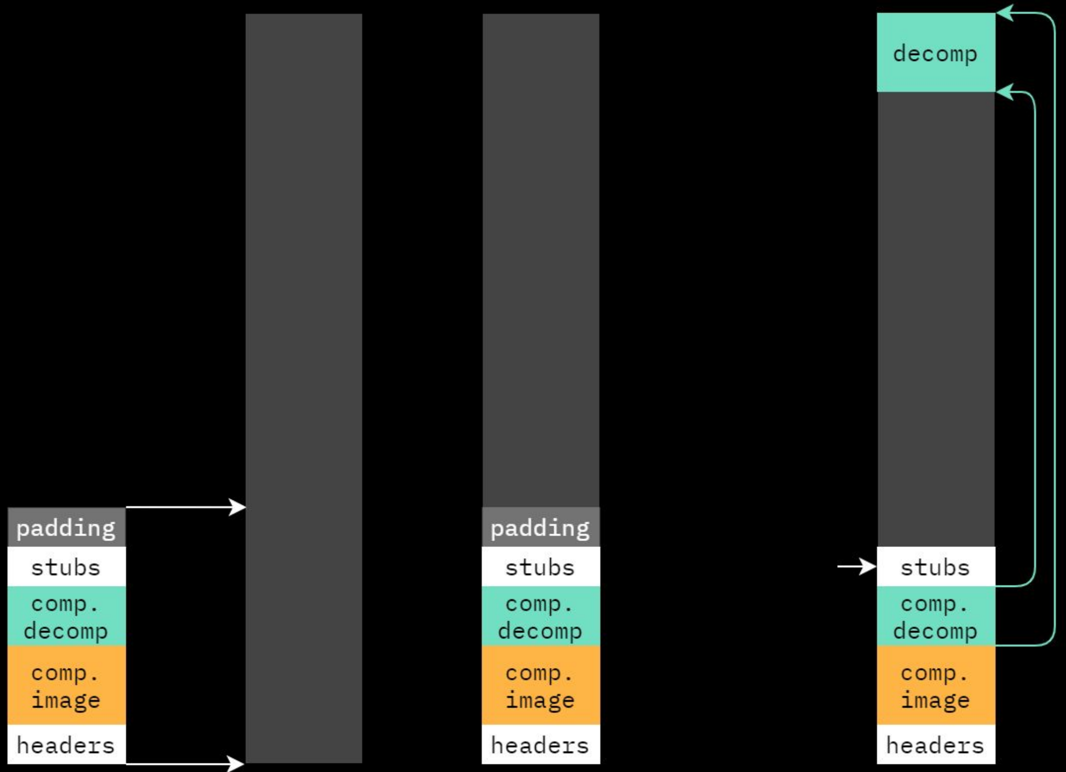
unpacking flow



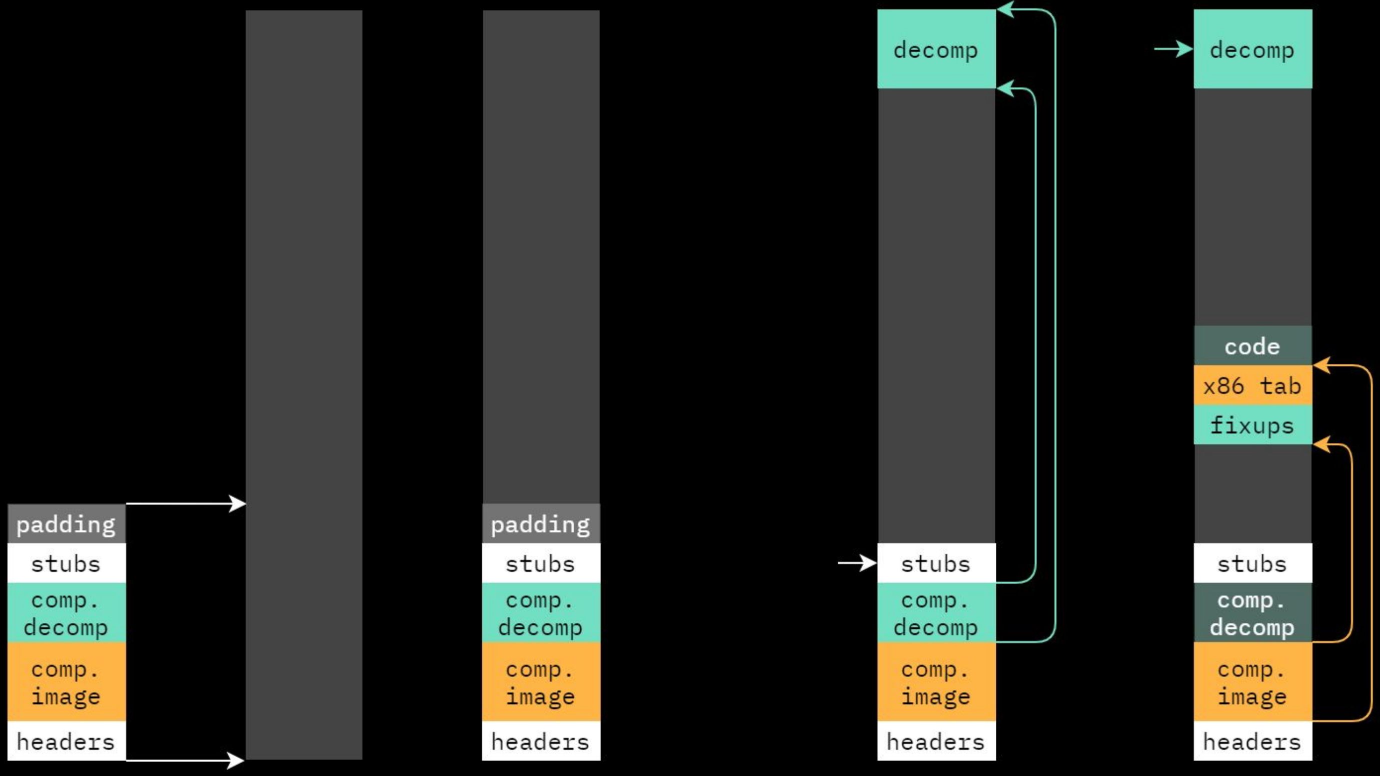
unpacking flow



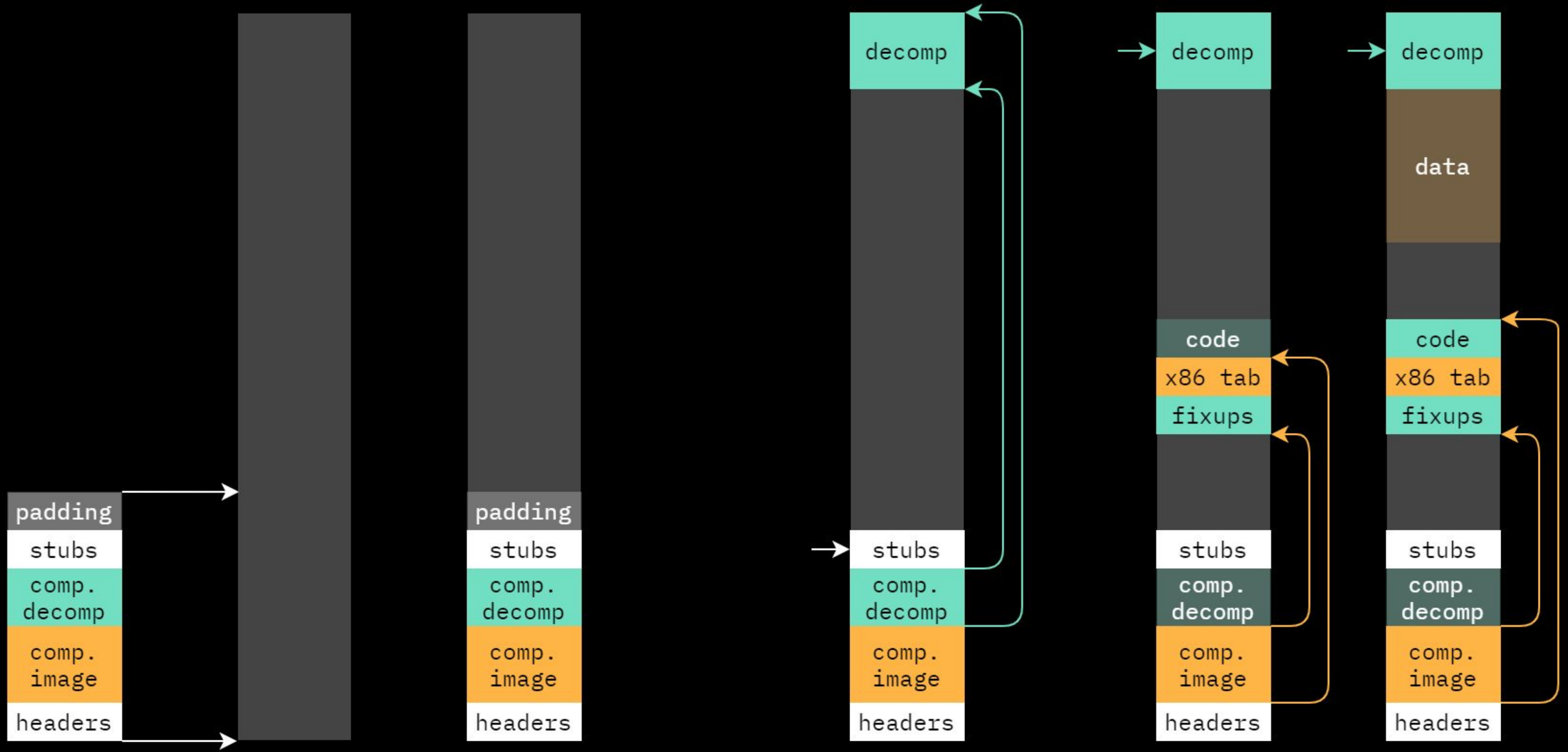
unpacking flow



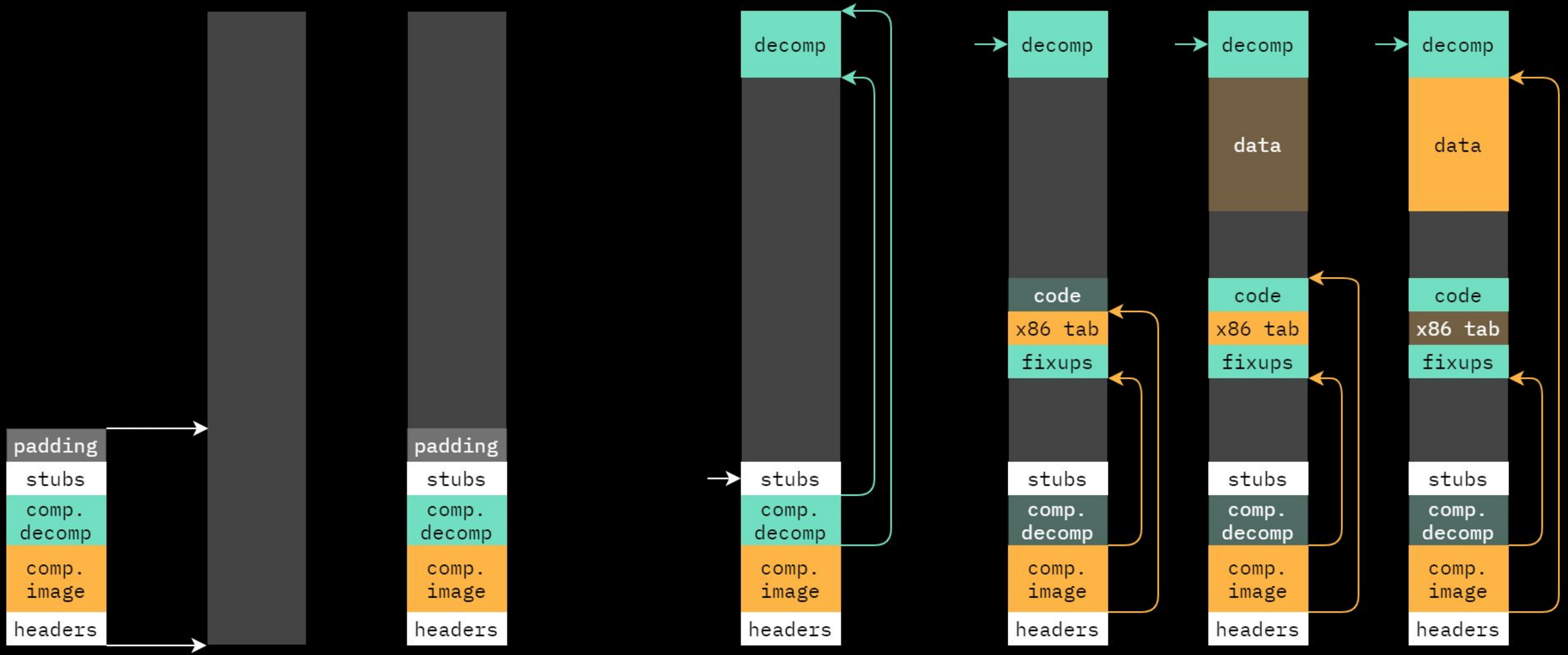
unpacking flow



unpacking flow

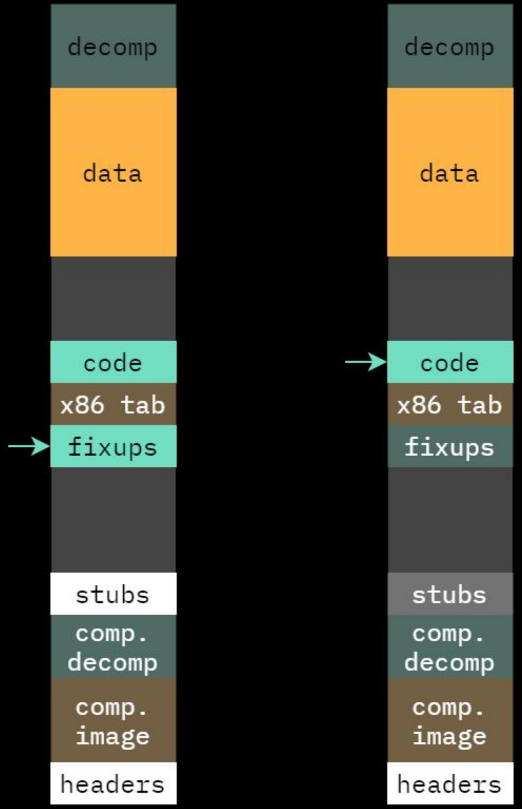


unpacking flow

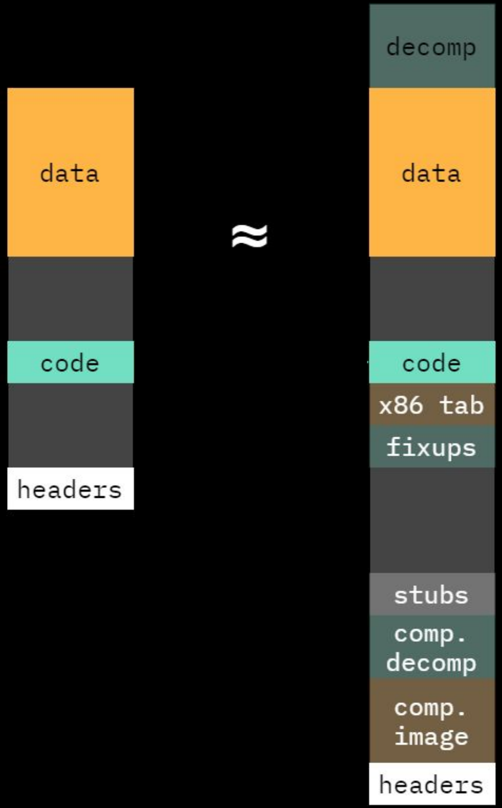


unpacking flow

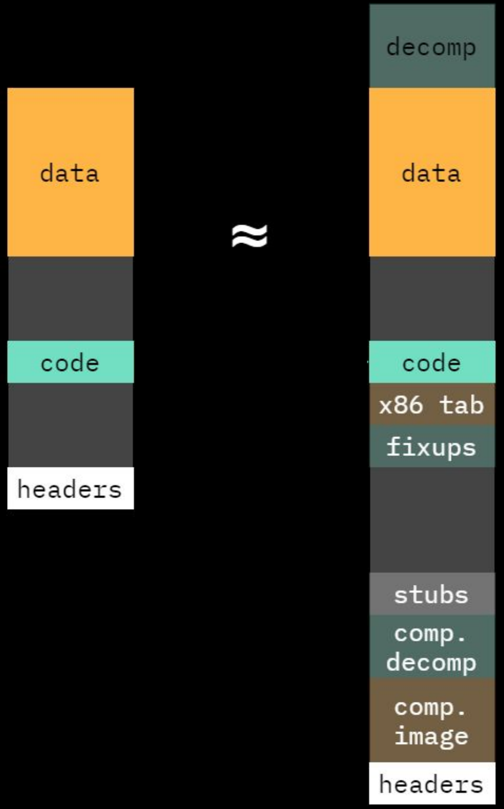
unpacking flow



unpacking flow



unpacking flow



executable compression
(final notes)

executable compression

- we don’t do very advanced slicing/header packing
- no hash import, minimal overlapping
  - largely compatibility measures
  - we can afford this in 64k
  - compatible with future Windows loaders = better user experience
- compression engine pulls most of the weight
- maybe I’ll cave and add some hackier stuff optionally anyways eventually :)

compression 101

quick disclaimer

- two compressors in squishy
  - both have statistical components
- only have time to talk about the big one
  - more “pure” in the theoretical sense anyways
  - everything in this section still applies to both

compression 101

- start with a symbol alphabet
  - could be any length >= 1
  - eg. {0, 1} for bits, {0, 1, .. 255} for bytes, {A, B, C}, etc.

compression 101

- start with a symbol alphabet
  - could be any length >= 1
  - eg. {0, 1} for bits, {0, 1, .. 255} for bytes, {A, B, C}, etc.
- construct a string of symbols
  - eg. AABC, which, assuming each symbol is 2 bits, is 8 bits in this representation

compression 101

- start with a symbol alphabet
  - could be any length >= 1
  - eg. {0, 1} for bits, {0, 1, .. 255} for bytes, {A, B, C}, etc.
- construct a string of symbols
  - eg. AABC, which, assuming each symbol is 2 bits, is 8 bits in this representation

- find a new representation which requires fewer bits but conveys the same information

compression 101

-
-


s, {A, B, C}, etc.

its, is 8 bits i

- find a new rep es fewer bits but conveys t
- ok ow?

some dang stats, ya dingus!

compression 101

- most real-world data contains statistical redundancy
  - put simply: some symbols are more common than others

- we can exploit this by making a statistical model of the string we want to compress

compression 101

- let’s take an example, our string from earlier: AABC
- now we’ll determine the frequency of each symbol
  - this just means count them!

- how many A’s are there in our string?
  - 2, so f(A) = 2
- how many B’s are there in our string?
  - 1, so f(B) = 1
- how many C’s are there in our string?
  - 1, so f(C) = 1

compression 101

- let’s take an example, our string from earlier: AABC
- now we’ll determine the probability of each symbol
    - this just means divide the frequencies by the total string length!

- len = 4 (chars)
- p(A) = f(A) / len = 1/2
- p(B) = f(B) / len = 1/4
- p(C) = f(C) / len = 1/4
    - note how these are normalized, i.e. they sum to 1

compression 101

- let’s take an example, our string from earlier: AABC
- now we have a model: p = { A: 1/2, B: 1/4, C: 1/4 }

compression 101

- let's take an example, our string from earlier: AABC
- now we have a model: p = { A: 1/2, B: 1/4, C: 1/4 }

- what next?

compress 1

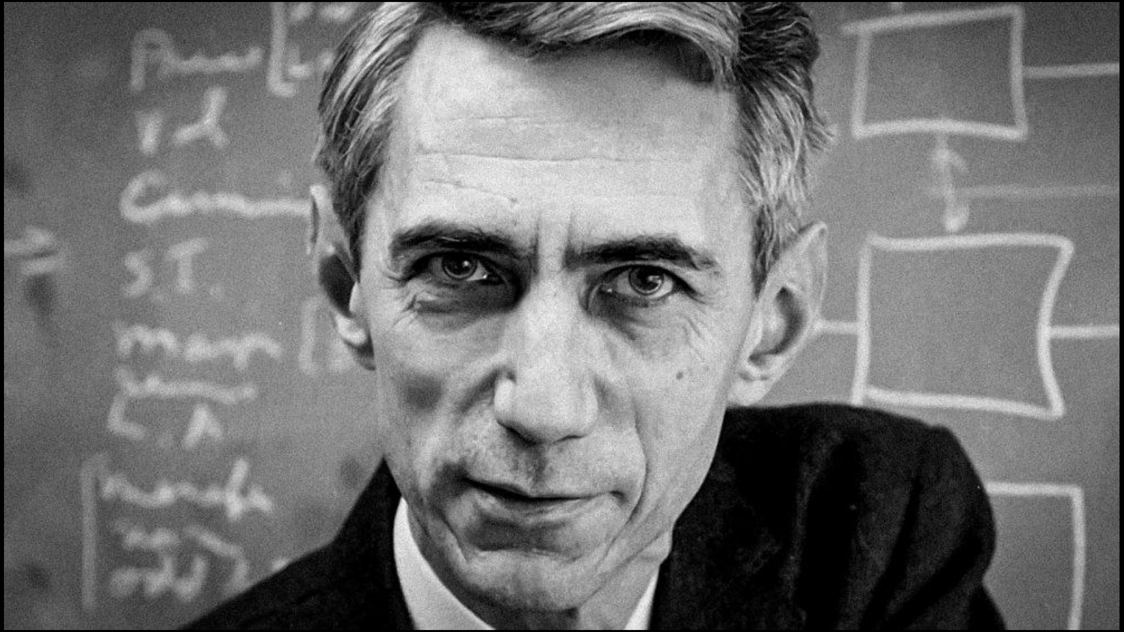
- let's tak le, our str rom earl ^ABC
- now we ha B: 1/4,
- nex
- we r!

BRUK VUKLAN FFS

no jake you fool this talk sucks

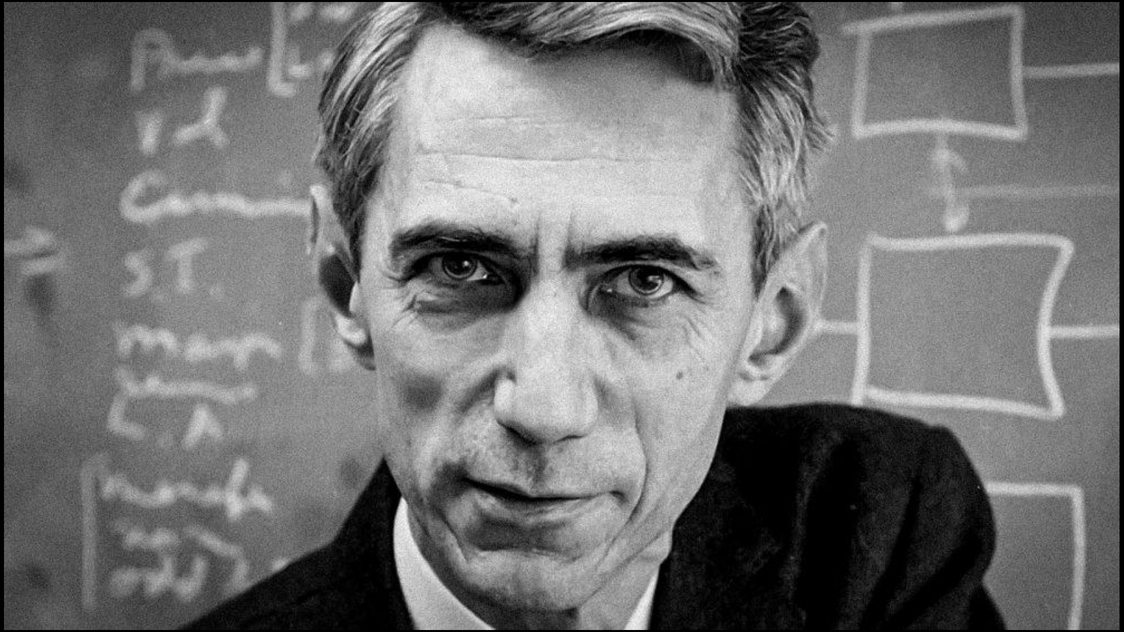
quick tangent: source coding theorem

- meet claude shannon



quick tangent: source coding theorem

- meet claude shannon
- look at that hot piece of man(tropy)



- ok this pic is haunting af but hear me out

quick tangent: source coding theorem

- claudy with a shans of meatballs over here did this really cool thing
    - he actually did a butt ton of awesome stuff!!!

quick tangent: source coding theorem

- claudy with a shans of meatballs over here did this really cool thing
  - he actually did a butt ton of awesome stuff!!!
- he came up with Shannon’s source coding theorem, which states that:

quick tangent: source coding theorem

- claudy with a shans of meatballs over here did this really cool thing
  - he actually did a butt ton of awesome stuff!!!
- he came up with Shannon's source coding theorem, which states that:

- the optimal code length for a symbol is -log2(p) bits
  - where p is the probability of the symbol, as discussed earlier

compression 101

- the optimal code length for a symbol is -log2(p) bits

compression 101

- the optimal code length for a symbol is -log2(p) bits
- our earlier model model: p = { A: 1/2, B: 1/4, C: 1/4 }

compression 101

- the optimal code length for a symbol is -log2(p) bits
- our earlier model model: p = { A: 1/2, B: 1/4, C: 1/4 }
  - for A: -log2(1/2) = 1 bit
  - for B: -log2(1/4) = 2 bits
  - for C: -log2(1/4) = 2 bits

compression 101

- the optimal code length for a symbol is −log2(p) bits
- our earlier model model: p = { A: 1/2, B: 1/4, C: 1/4 }
  - for A: -log2(1/2) = 1 bit
  - for B: -log2(1/4) = 2 bits
  - for C: -log2(1/4) = 2 bits

- an [entropy] coder codes symbols using probabilities, such that each symbol is represented using the optimal number of bits as calculated above
  - ^ THAT's the kind of coder we need!

compression 101

- back to our example string from earlier: AABC
- and our model: p = { A: 1/2, B: 1/4, C: 1/4 }
- and a coder c with the following interface:

```
class coder:
    output_string: string
    coder():
        output_string = empty;
    method encode(symbol, model):
        /* ignore impl for now */
```

compression 101

- back to our example string from earlier: AABC
- and our model: p = { A: 1/2, B: 1/4, C: 1/4 }
- and a coder c
- we simply invoke our coder for each symbol, and it handles the rest! (assuming our model is correct)

```
fn encode(string, model):
    c = new coder();
    for s in string:
        c.encode(s, model);
    return c.output_string;
```

compression 101

- back to our example string from earlier: AABC
- and our model: p = { A: 1/2, B: 1/4, C: 1/4 }
- and a prefix code coder c:

string: AABC

output_string: empty

compression 101

- back to our example string from earlier: AABC
- and our model: p = { A: 1/2, B: 1/4, C: 1/4 }
- and a prefix code coder c:

```
string: AABC
        ^
output_string: 0
```

compression 101

- back to our example string from earlier: AABC
- and our model: p = { A: 1/2, B: 1/4, C: 1/4 }
- and a prefix code coder c:

```
string: AABC
         ^
output_string: 0 0
```

compression 101

- back to our example string from earlier: AABC
- and our model: p = { A: 1/2, B: 1/4, C: 1/4 }
- and a prefix code coder c:

```
string: AABC
            ^
output_string: 0 0 10
```

compression 101

- back to our example string from earlier: AABC
- and our model: p = { A: 1/2, B: 1/4, C: 1/4 }
- and a prefix code coder c:

```
string: AABC
           ^
output_string: 0 0 10 11
```

compression 101

- back to our example string from earlier: AABC
- and our model: p = { A: 1/2, B: 1/4, C: 1/4 }
- and a prefix code coder c:

string: AABC

output_string: 0 0 10 11 = 001011 (6 bits!)

compression 101

```
fn encode(string, model):
    c = new coder();
    for s in string:
        c.encode(s, model);
    return c.output_string;

fn decode(output_string, original_string_len, model):
    original_string = empty;
    d = new decoder(output_string);
    for i in 0..original_string_len:
        original_string.append(d.decode(model));
    return original_string;
```

compression 101

- contrived example, but it illustrates key things:
- general statistical coding algorithm for compression
  - model determines per-symbol probabilities
  - coder faithfully encodes symbols with # of bits determined by the model probabilities
- model/coder are completely decoupled!

compression 101

- many kinds of coders
- we saw an example of a prefix code “implementation”
  - huffman coding belong to this family
  - limited to fixed bit widths due to direct symbol replacement
  - real-world probabilities are rarely powers of 2!
  - eg. the alphabet { A, B, C } with string ABC gives us 1/3 prob for each symbol, and a total length of 4.75 bits (approx.)

compression 101

- there exist coders that output fractional bits
- you won’t BELIEVE this ONE SIMPLE TRICK!
  - coder keeps some internal state representing fractional bits
  - consider a symbol that should be represented with 1/10 bits
  - for every 100 of these symbols, the coder keeps track of fractional bits and only outputs 10 bits
  - it’s just averages!
  - hidden behind (de)coder per-symbol interface

- I wish I had more time to talk :(
  - arithmetic/range coders
  - asymmetric numeral systems family

compression 101

- tl;dr: coding is a well-understood, largely solved problem
- squishy uses a simple binary range coder
  - good fractional precision, fast (for binary models)
  - rABS was also experimented with, no compelling advantages

- modeling, however, is the hard part!

modeling deep-dive

modeling deep-dive

- getting a model right is really hard
  - often requires intimate knowledge of the data
- 64k's can contain all kinds of different data
- we need a good general-purpose model
  - needs to basically handle anything
  - perhaps tuned to shaders/text a bit these days
- one thing in common: x86 code
  - specialized modeling for this in squishy
  - I tried not to do this for a _long_ time, eventually caved!
  - it's that important!

modeling deep-dive

- we saw a ternary, static, order 0 model previously

modeling deep-dive

- we saw a ternary, static, order 0 model previously
          ^ 3 symbols (A, B, C)

modeling deep-dive

- we saw a ternary, static, order 0 model previously
  ^ same model used to code whole string

modeling deep-dive

- we saw a ternary, static, order 0 model previously

  ...we’ll get to this part :) ^

modeling deep-dive

- we saw a ternary, static, order 0 model previously
- squishy has a binary, adaptive, context mixing model

modeling deep-dive

- we saw a ternary, static, order 0 model previously
- squishy has a binary, adaptive, context mixing model

  ^ 2 symbols, specifically, 0 and 1 (bits!)

modeling deep-dive

- we saw a ternary, static, order 0 model previously
- squishy has a binary, adaptive, context mixing model

```
                ^ 2 symbols, specifically, 0 and 1 (bits!)
                  for speed and simplicity, as we'll see
```

modeling deep-dive

- we saw a ternary, static, order 0 model previously
- squishy has a binary, adaptive, context mixing model

  ^ this requires some explanation

modeling deep-dive

- we saw a ternary, static, order 0 model previously
- squishy has a binary, adaptive, context mixing model
  ^ this requires some explanation
- many files, especially intros, contain many kinds of data

modeling deep-dive

- we saw a ternary, static, order 0 model previously
- squishy has a binary, adaptive, context mixing model
  ^ this requires some explanation
- many files, especially intros, contain many kinds of data
  - shader data
  - music data
  - dialog boxes
  - animation curves
  - greetings text
  - who even knows anymore

modeling deep-dive

- we saw a ternary, static, order 0 model previously
- squishy has a binary, adaptive, context mixing model

  ^ this requires some explanation
- many files, especially intros, contain many kinds of data
  - shader data
  - music data
  - dialog boxes
  - animation curves
  - greetings text
  - who even knows anymore

modeling deep-dive

- we saw a ternary, static, order 0 model previously
- squishy has a binary, adaptive, context mixing model
                                  ^ this requires some explanation
- our model should adapt to changing statistics throughout the data for better compression
    - this also means we don’t have to store a model in the compressed file

modeling deep-dive

- we saw a ternary, static, order 0 model previously
- squishy has a binary, adaptive, context mixing model

  we actually mix many models’ probs together ^

modeling deep-dive

- we saw a ternary, static, order 0 model previously
- squishy has a binary, adaptive, context mixing model
  we actually mix many models’ probs together ^
- different models are good for different patterns in the data

modeling deep-dive

- we saw a ternary, static, order 0 model previously
- squishy has a binary, adaptive, context mixing model
  we actually mix many models' probs together ^
- different models are good for different patterns in the data
- so let's run several models in parallel and mix results somehow

modeling deep-dive

- we saw a ternary, static, order 0 model previously
- squishy has a binary, adaptive, context mixing model
- this kind of model is used in several top-performing compressors:
    - PAQ
    - CCM
    - CMIX
    - kkrunchy
    - ...many more!

modeling deep-dive

- we saw a ternary, static, order 0 model previously
- squishy has a binary, adaptive, context mixing model
- this kind of model is used in several top-performing compressors:
  - PAQ
  - CCM
  - CMIX
  - kkrunchy
  - ...many more!
- it’s a really good strat, but very slow

modeling deep-dive

- we saw a ternary, static, order 0 model previously
- squishy has a binary, adaptive, context mixing model
- this kind of model is used in several top-performing compressors:
  - PAQ
  - CCM
  - CMIX
  - kkrunchy
  - ...many more!
- it’s a really good strat, but very slow
- luckily we only compress a few hundred kb’s so whatevs

modeling deep-dive

- we saw a ternary, static, order 0 model previously
- squishy has a binary, adaptive, context mixing model

- so what is context?

modeling deep-dive

- context represents local stuff “around” a symbol

modeling deep-dive

- context represents local stuff “around” a symbol
- our earlier example didn’t use any
  - it modeled each symbol the same for the entire string
  - hence “order 0”

modeling deep-dive

- context represents local stuff “around” a symbol
- our earlier example didn’t use any
  - it modeled each symbol the same for the entire string
  - hence “order 0”
- consider an english sentence:

modeling deep-dive

- context represents local stuff “around” a symbol
- our earlier example didn’t use any
    - it modeled each symbol the same for the entire string
    - hence “order 0”
- consider an english sentence:
    - the cat kicked the dog in the face

modeling deep-dive

- context represents local stuff “around” a symbol
- our earlier example didn’t use any
  - it modeled each symbol the same for the entire string
  - hence “order 0”
- consider an english sentence:
  - the cat kicked the dog in the face
- an order 0 model doesn’t care about placement of words
  - might as well have been cat dog face kicked in the the the
  - same per-word (symbol) count for the whole string as above

modeling deep-dive

- but we know that context matters!

modeling deep-dive

- but we know that context matters!
- a better model would give nouns (cat, dog, face) higher probabilities after articles (the)
    - and loads of other rules like this

modeling deep-dive

- but we know that context matters!
- a better model would give nouns (cat, dog, face) higher probabilities after articles (the)
  - and loads of other rules like this
- a context that includes one symbol before the current symbol is an order 1 context
  - just like markov chains
  - can have order 2, 3, … 100 if you want
  - contexts become more sparse as order increases
  - they also get YUGE!

modeling deep-dive

- how might we represent a context?

modeling deep-dive

- how might we represent a context?
- we effectively want to capture substrings of symbols

modeling deep-dive

- how might we represent a context?
- we effectively want to capture substrings of symbols
- let’s do a hash!

modeling deep-dive

- how might we represent a context?
- we effectively want to capture substrings of symbols
- let’s do a hash!
  - eg. hash last n bytes for order n context

modeling deep-dive

- how might we represent a context?
- we effectively want to capture substrings of symbols
- let’s do a hash!
  - eg. hash last n bytes for order n context
  - numerical representation of “after these n bytes, then ...”

modeling deep-dive

- how might we represent a context?
- we effectively want to capture substrings of symbols
- let's do a hash!
  - eg. hash last n bytes for order n context
  - numerical representation of "after these n bytes, then ..."
- look up hash in a table of model states

modeling deep-dive

- how might we represent a context?
- we effectively want to capture substrings of symbols
- let’s do a hash!
  - eg. hash last n bytes for order n context
  - numerical representation of “after these n bytes, then ...”
- look up hash in a table of model states
  - eg. `model_state = states[context_hash]`

modeling deep-dive

- how might we represent a context?
- we effectively want to capture substrings of symbols
- let’s do a hash!
  - eg. hash last n bytes for order n context
  - numerical representation of “after these n bytes, then ...”
- look up hash in a table of model states
  - eg. model_state = states[context_hash]
  - fetch prediction from model_state, update after symbol is seen

modeling deep-dive

- many possible context hashes!

modeling deep-dive

- many possible context hashes!
  - esp. with high-order contexts

modeling deep-dive

- many possible context hashes!
  - esp. with high-order contexts
- we want an upper-bound for memory usage

modeling deep-dive

- many possible context hashes!
  - esp. with high-order contexts
- we want an upper-bound for memory usage
  - we could use direct mapping from context -> table index in a big table

modeling deep-dive

- many possible context hashes!
  - esp. with high-order contexts
- we want an upper-bound for memory usage
  - we could use direct mapping from context -> table index in a big table
  - lots of collisions (= bad predictions = worse compression)

modeling deep-dive

- many possible context hashes!
  - esp. with high-order contexts
- we want an upper-bound for memory usage
  - we could use direct mapping from context -> table index in a big table
  - lots of collisions (= bad predictions = worse compression)
  - crinkler does this and uses a very large table to avoid collisions :)

modeling deep-dive

- many possible context hashes!
  - esp. with high-order contexts
- we want an upper-bound for memory usage
  - we could use direct mapping from context -> table index in a big table
  - lots of collisions (= bad predictions = worse compression)
  - crinkler does this and uses a very large table to avoid collisions :)
- alternative: use an n-way associative cache table

modeling deep-dive

- many possible context hashes!
  - esp. with high-order contexts
- we want an upper-bound for memory usage
  - we could use direct mapping from context -> table index in a big table
  - lots of collisions (= bad predictions = worse compression)
  - crinkler does this and uses a very large table to avoid collisions :)
- alternative: use an n-way associative cache table
  - each hash maps to n tagged entries in a fixed size table

modeling deep-dive

- many possible context hashes!
  - esp. with high-order contexts
- we want an upper-bound for memory usage
  - we could use direct mapping from context -> table index in a big table
  - lots of collisions (= bad predictions = worse compression)
  - crinkler does this and uses a very large table to avoid collisions :)
- alternative: use an n-way associative cache table
  - each hash maps to n tagged entries in a fixed size table
  - if no matching entry for a hash is found, replace with new one

modeling deep-dive

- many possible context hashes!
  - esp. with high-order contexts
- we want an upper-bound for memory usage
  - we could use direct mapping from context -> table index in a big table
  - lots of collisions (= bad predictions = worse compression)
  - crinkler does this and uses a very large table to avoid collisions :)
- alternative: use an n-way associative cache table
  - each hash maps to n tagged entries in a fixed size table
  - if no matching entry for a hash is found, replace with new one
  - different eviction strategies possible (eg. LRU)

modeling deep-dive

- many possible context hashes!
  - esp. with high-order contexts
- we want an upper-bound for memory usage
  - we could use direct mapping from context -> table index in a big table
  - lots of collisions (= bad predictions = worse compression)
  - crinkler does this and uses a very large table to avoid collisions :)
- alternative: use an n-way associative cache table
  - each hash maps to n tagged entries in a fixed size table
  - if no matching entry for a hash is found, replace with new one
  - different eviction strategies possible (eg. LRU)
  - squishy uses a 32MB 4-way cache table with LRU
  - optimized for cache line alignment and compression efficiency

modeling deep-dive

- we typically want to model many different contexts with many different models

modeling deep-dive

- we typically want to model many different contexts with many different models
- our cache table entries contain multiple model states

modeling deep-dive

- we typically want to model many different contexts with many different models
- our cache table entries contain multiple model states
- some models work alongside the cache table
  - if a better representation is available

modeling deep-dive

- we typically want to model many different contexts with many different models
- our cache table entries contain multiple model states
- some models work alongside the cache table
  - if a better representation is available
- this is fine since models can be arbitrary history functions

modeling deep-dive

- we typically want to model many different contexts with many different models
- our cache table entries contain multiple model states
- some models work alongside the cache table
  - if a better representation is available
- this is fine since models can be arbitrary history functions
- note that many useful contexts are sparse!
  - eg. order 2 context with the byte that was 4 bytes ago and the one that was 8 bytes ago, instead of the 2 previous

modeling deep-dive

- we saw a ternary, static, order 0 model previously
- squishy has a binary, adaptive, context mixing model

- so yeah, our job is to build models that predict bits, and use them to model our data in different contexts

modeling deep-dive

- we saw a ternary, static, order 0 model previously
- squishy has a binary, adaptive, context mixing model

- so yeah, our job is to build models that predict bits, and use them to model our data in different contexts
- what kind of binary, adaptive models do we have?

quick note on representing probs

- we’ve been using real-number probabilities

quick note on representing probs

- we’ve been using real-number probabilities
- real numbers constipate computers and make them sad :(

quick note on representing probs

- we’ve been using real-number probabilities
- real numbers constipate computers and make them sad :(
- can use fixed-point instead!

quick note on representing probs

- we’ve been using real-number probabilities
- real numbers constipate computers and make them sad :(
- can use fixed-point instead!
- works real good for binary probabilities

quick note on representing probs

- we've been using real-number probabilities
- real numbers constipate computers and make them sad :(
- can use fixed-point instead!
- works real good for binary probabilities
  - only two probabilities:
  - p0 (probability for 0 bit)
  - p1 (probability for 1 bit)

quick note on representing probs

- we've been using real-number probabilities
- real numbers constipate computers and make them sad :(
- can use fixed-point instead!
- works real good for binary probabilities
  - only two probabilities:
  - p0 (probability for 0 bit)
  - p1 (probability for 1 bit)
  - actually only need to represent one, since the other is implicit
  - eg. only represent p1, since p0 = 1 - p1

quick note on representing probs

- we've been using real-number probabilities
- real numbers constipate computers and make them sad :(
- can use fixed-point instead!
- works real good for binary probabilities
  - only two probabilities:
  - p0 (probability for 0 bit)
  - p1 (probability for 1 bit)
  - actually only need to represent one, since the other is implicit
  - eg. only represent p1, since p0 = 1 - p1
- scale by const power of 2, eg. 4096
  - (0, 4096) instead of (0, 1)
  - this is what's used in squishy/PAQ/kkrunchy
  - for final probs at least, most models have more bits Internally; I want to increase this at some point!

modeling deep-dive

- baby’s first model:

modeling deep-dive

- baby's first model: const model!

modeling deep-dive

- baby’s first model: const model!
- outputs a fixed p1, no adaptation

modeling deep-dive

- baby's first model: const model!
- outputs a fixed p1, no adaptation
- eg. 2048 (50/50 p0/p1)
    - results in no compression/expansion
    - good litmus test to see if your coder works!

modeling deep-dive

- baby's first model: const model!
- outputs a fixed p1, no adaptation
- eg. 2048 (50/50 p0/p1)
    - results in no compression/expansion
    - good litmus test to see if your coder works!
- not useful for much else except mixer bias (later)
    - unless it happens to match your data :)

modeling deep-dive

- baby's first model: const model!
- outputs a fixed p1, no adaptation
- eg. 2048 (50/50 p0/p1)
  - results in no compression/expansion
  - good litmus test to see if your coder works!
- not useful for much else except mixer bias (later)
  - unless it happens to match your data :)

```
fn prob():
    return 1337;
```

modeling deep-dive

- stationary context model

modeling deep-dive

- stationary context model
- two variables: prob, bit_count

modeling deep-dive

- stationary context model
- two variables: prob, bit_count

```
                  ^ current prediction, init to 1/2 (usually)
```

modeling deep-dive

- stationary context model
- two variables: prob, bit_count

  ^ # of bits seen so far, init to 0

modeling deep-dive

- stationary context model
- two variables: prob, bit_count
- update rule biases prob in inverse proportion to bit_count

modeling deep-dive

- stationary context model

```
fn init():
    prob = 2048;
    num_bits = 0;

fn prob():
    return prob;

fn update(bit):
    prob += (bit * 4096 - prob) / (num_bits + delta);
    if num_bits < limit: num_bits++;
```

modeling deep-dive

- stationary context model
- two variables: prob, bit_count
- update rule biases prob in inverse proportion to bit_count
- delta and limit are tunable parameters
  - best values depend on data
  - static, hand-tuned in squishy based on test corpus performance
  - maybe exposed in future versions

modeling deep-dive

- stationary context model
- two variables: prob, bit_count
- update rule biases prob in inverse proportion to bit_count
- delta and limit are tunable parameters
  - best values depend on data
  - static, hand-tuned in squishy based on test corpus performance
  - maybe exposed in future versions
- initially learns quickly, then becomes static (hence “stationary”)
  - eventually becomes adaptive again as bit_count saturates, but learns slowly at that point

modeling deep-dive

- stationary context model
- this was an example of a direct model

modeling deep-dive

- stationary context model
- this was an example of a direct model
    - a context maps directly to a prediction

modeling deep-dive

- stationary context model
- this was an example of a direct model
  - a context maps directly to a prediction
- what might an indirect model look like?

modeling deep-dive

- stationary context model
- this was an example of a direct model
  - a context maps directly to a prediction
- what might an indirect model look like?
- what kind of data might it model?

modeling deep-dive

- indirect context model

modeling deep-dive

- indirect context model
- maps a context to a history, which is then mapped again to a prediction (hence indirect)

modeling deep-dive

- indirect context model
- maps a context to a history, which is then mapped again to a prediction (hence indirect)
- 3 variables: counts, last_n_bits, prediction_table

modeling deep-dive

- indirect context model
- maps a context to a history, which is then mapped again to a prediction (hence indirect)
- 3 variables: counts, last_n_bits, prediction_table
    - prediction_table might be shared between many models
    - depends on data it’s designed to model

modeling deep-dive

- indirect context model
- maps a context to a history, which is then mapped again to a prediction (hence indirect)
- 3 variables: counts, last_n_bits, prediction_table
  - prediction_table might be shared between many models
  - depends on data it's designed to model
- construct table index by combining counts, last_n_bits

modeling deep-dive

- indirect context model
- maps a context to a history, which is then mapped again to a prediction (hence indirect)
- 3 variables: counts, last_n_bits, prediction_table
  - prediction_table might be shared between many models
  - depends on data it’s designed to model
- construct table index by combining counts, last_n_bits
- look up prediction in prediction_table

modeling deep-dive

- indirect context model
- maps a context to a history, which is then mapped again to a prediction (hence indirect)
- 3 variables: counts, last_n_bits, prediction_table
    - prediction_table might be shared between many models
    - depends on data it’s designed to model
- construct table index by combining counts, last_n_bits
- look up prediction in prediction_table
    - typically with nonlinear mapping

modeling deep-dive

- indirect context model
- maps a context to a history, which is then mapped again to a prediction (hence indirect)
- 3 variables: counts, last_n_bits, prediction_table
  - prediction_table might be shared between many models
  - depends on data it’s designed to model
- construct table index by combining counts, last_n_bits
- look up prediction in prediction_table
  - typically with nonlinear mapping
- update both table entry and model state after symbol
  - order matters here!

modeling deep-dive

- indirect context model
- maps a context to a history, which is then mapped again to a prediction (hence indirect)
- 3 variables: counts, last_n_bits, prediction_table
  - prediction_table might be shared between many models
  - depends on data it's designed to model
- construct table index by combining counts, last_n_bits
- look up prediction in prediction_table
  - typically with nonlinear mapping
- update both table entry and model state after symbol
  - order matters here!
- possibly interpolate/update adjacent entries

modeling deep-dive

- indirect context model

```
fn init():
    counts = 0 | 0;
    last_n_bits = 0;

fn prob():
    return prediction_table[counts | last_n_bits];

fn update(bit):
    prediction_table[counts | last_n_bits] += …;
    counts += …;
    last_n_bits = (last_n_bits << 1) | bit;
```

modeling deep-dive

- indirect context model
- models repeating patterns in contexts well

modeling deep-dive

- indirect context model
- models repeating patterns in contexts well
    - eg. 000001000001

modeling deep-dive

- indirect context model
- models repeating patterns in contexts well
  - eg. 000001000001
  - 000 001 000 001

modeling deep-dive

- indirect context model
- models repeating patterns in contexts well
    - eg. 000001000001
    - 000 001 000 001

      ^ consider the end of each 3-symbol block

modeling deep-dive

- indirect context model
- models repeating patterns in contexts well
  - eg. 000001000001
  - 000 001 000 001

    ^ assume an order-2 context, so the last two symbols are context

modeling deep-dive

- indirect context model
- models repeating patterns in contexts well
    - eg. 000001000001
    - 000 001 000 001

      ^ “after we see 00, there’s a repeating 0, 1, 0, 1 pattern”

modeling deep-dive

- indirect context model
- models repeating patterns in contexts well
  - eg. 000001000001
  - 000 001 000 001
  - an indirect model with 1 history bit will learn that if we saw a 0 in this context last time, a 1 is highly likely, and vice versa

modeling deep-dive

- indirect context model
- models repeating patterns in contexts well
  - eg. 000001000001
  - 000 001 000 001
  - an indirect model with 1 history bit will learn that if we saw a 0 in this context last time, a 1 is highly likely, and vice versa
- this kind of pattern is very common!

modeling deep-dive

- indirect context model
- models repeating patterns in contexts well
  - eg. 000001000001
  - 000 001 000 001
  - an indirect model with 1 history bit will learn that if we saw a 0 in this context last time, a 1 is highly likely, and vice versa
- this kind of pattern is very common!
  - these models pull a _lot_ of weight in squishy

modeling deep-dive

- indirect context model
- models repeating patterns in contexts well
  - eg. 000001000001
  - 000 001 000 001
  - an indirect model with 1 history bit will learn that if we saw a 0 in this context last time, a 1 is highly likely, and vice versa
- this kind of pattern is very common!
  - these models pull a _lot_ of weight in squishy
- number of history bits to track, prediction table size, and update parameters are all tunable

modeling deep-dive

- indirect context model
- models repeating patterns in contexts well
  - eg. 000001000001
  - 000 001 000 001
  - an indirect model with 1 history bit will learn that if we saw a 0 in this context last time, a 1 is highly likely, and vice versa
- this kind of pattern is very common!
  - these models pull a _lot_ of weight in squishy
- number of history bits to track, prediction table size, and update parameters are all tunable
  - again, static in squishy, tuned on corpus
  - maybe exposed someday

modeling deep-dive

- many more primitive model types available!

modeling deep-dive

- many more primitive model types available!
  - run models for long strings of the same symbol
  - match models for higher-order contexts
  - variable-order models (eg. PPM, DMC, CTW)
  - this is a fun place to be creative!

model mixing

wow is this guy really still talking

model mixing

- so we have loads of predictions now from our models

model mixing

- so we have loads of predictions now from our models
- how might we combine them to form a better prediction?

model mixing

- so we have loads of predictions now from our models
- how might we combine them to form a better prediction?
- short answer: however we want!

model mixing

- so we have loads of predictions now from our models
- how might we combine them to form a better prediction?
- short answer: however we want!
- let’s look at some options!

model mixing

- linear mixing with fixed weights

model mixing

- linear mixing with fixed weights
- exactly what it sounds like

model mixing

- linear mixing with fixed weights
- exactly what it sounds like
  - p(X) * a + p(Y) * (1 - a), a in [0, 1]

model mixing

- linear mixing with fixed weights
- exactly what it sounds like
  - p(X) * a + p(Y) * (1 - a), a in [0, 1]
- a is tunable

model mixing

- linear mixing with fixed weights
- exactly what it sounds like
  - p(X) * a + p(Y) * (1 - a), a in [0, 1]
- a is tunable
- not particularly useful

model mixing

- linear mixing with fixed weights
- exactly what it sounds like
  - p(X) * a + p(Y) * (1 - a), a in [0, 1]
- a is tunable
- not particularly useful
  - basically saying one model is always better than the other

model mixing

- linear mixing with fixed weights
- exactly what it sounds like
  - p(X) * a + p(Y) * (1 - a), a in [0, 1]
- a is tunable
- not particularly useful
  - basically saying one model is always better than the other
- but we can do loads of stuff here!

model mixing

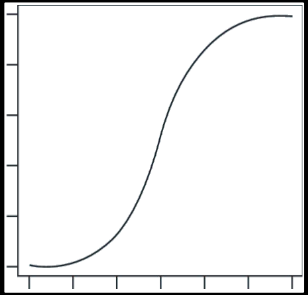
- linear mixing with fixed weights
- exactly what it sounds like
  - p(X) * a + p(Y) * (1 - a), a in [0, 1]
- a is tunable
- not particularly useful
  - basically saying one model is always better than the other
- but we can do loads of stuff here!
  - average?
  - weighted average?
  - be creative!

model mixing

- logistic mixing

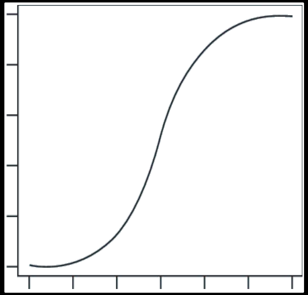
model mixing

- logistic mixing
- remap input predictions on a logistic curve

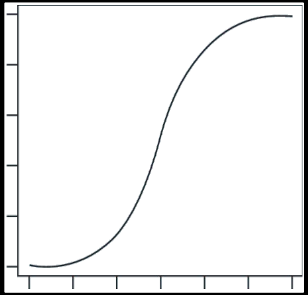


model mixing

- logistic mixing
- remap input predictions on a logistic curve
  - more precision at ends
  - better scaling for logarithmic source coding equation

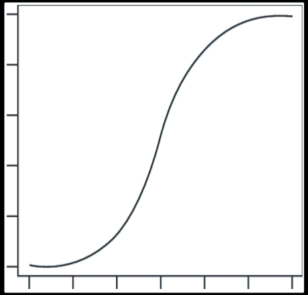


model mixing



- logistic mixing
- remap input predictions on a logistic curve
  - more precision at ends
  - better scaling for logarithmic source coding equation
- weighted sum remapped predictions

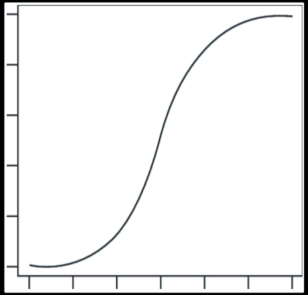
model mixing

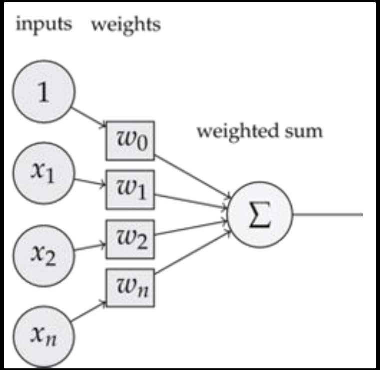


- logistic mixing
- remap input predictions on a logistic curve
  - more precision at ends
  - better scaling for logarithmic source coding equation
- weighted sum remapped predictions
- remap sum back to linear domain

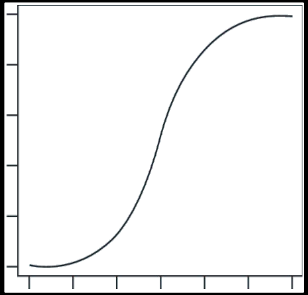
model mixing

- logistic mixing
- remap input predictions on a logistic curve
  - more precision at ends
  - better scaling for logarithmic source coding equation
- weighted sum remapped predictions
- remap sum back to linear domain
- this is a perceptron

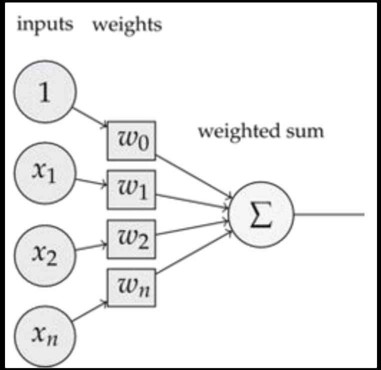




model mixing

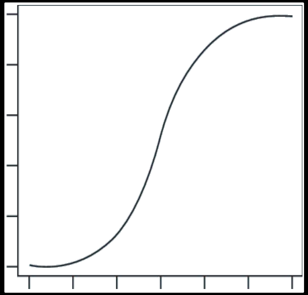


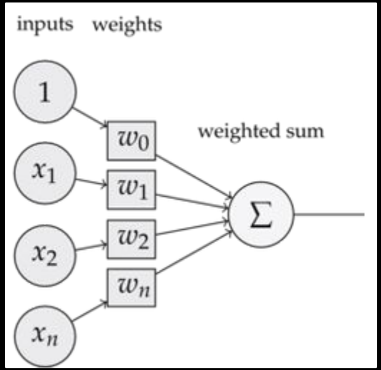
- logistic mixing
- remap input predictions on a logistic curve
  - more precision at ends
  - better scaling for logarithmic source coding equation
- weighted sum remapped predictions
- remap sum back to linear domain
- this is a perceptron
- performs very well!



model mixing

- logistic mixing
- remap input predictions on a logistic curve
  - more precision at ends
  - better scaling for logarithmic source coding equation
- weighted sum remapped predictions
- remap sum back to linear domain
- this is a perceptron
- performs very well!
- can select different weights with a context
  - squishy uses several!





loose ends

wow yeah he’s seriously still talking

loose ends

- we have loads of freedom to mix/match/adjust predictions

loose ends

- we have loads of freedom to mix/match/adjust predictions
- how about postprocessing?
  - SSE/APM
  - ISSE

loose ends

- we have loads of freedom to mix/match/adjust predictions
- how about postprocessing?
  - SSE/APM
  - ISSE
- multiple mixer stages?
  - you betcha!

loose ends

- we have loads of freedom to mix/match/adjust predictions
- how about postprocessing?
  - SSE/APM
  - ISSE
- multiple mixer stages?
  - you betcha!
- enabling multiple models for different data?
  - sure, why not?
  - this is part of how we model x86 in squishy actually

squishy model architecture

squishy model architecture

- based on PAQ7 with hand-tuned models
  - increased precision everywhere
  - several improvement ideas from PAQ8/ZPAQ/kkrunchy/etc
  - some special x86 stuff (later)
- ~2017/2018 focused on a ZPAQ-like VM
  - genetic algorithm used to “grow” architectures
  - never reached suitable performance :(
  - I didn’t quite understand multiple mixer weight contexts at the time
  - models weren’t as good as they are now
  - will probably try this again sometime!
- not set-in-stone
  - I want to experiment more :)

squishy model architecture

- context models
  - single 32MB cache table
  - each entry (16 bytes) contains:
    - cache tag (4 bytes)
    - a stationary/direct model (4 bytes)
    - an indirect model (4 bytes)
    - a run model (4 bytes)
  - 4-way associative cache
    - each bucket is 16 bytes * 4 = 64 bytes
    - same size as cache lines, aligned mem alloc

squishy model architecture

- context models
  - 21 contexts used (in data sections)
  - hand-picked, static descriptions
  - combined previous bytes (and bits of those bytes) selected by masks
  - each context modeled by one cache table entry (3 models/predictions)
  - 21 * 3 = 63 context model predictions (in data sections)
- single const model
  - also hand-tuned
  - 63 + 1 = 64 predictions (in data sections)
- 8 match models
  - each with increasing context orders
  - 64 + 8 = 72 total predictions (in data sections)

squishy model architecture

- logistic mixing
- first stage
  - all 72 predictions are mixed 8 different times
  - each time with weights selected from different contexts
    - 1 static context (order 0), same weights every time
    - 5 byte history contexts with increasing order (orders 1-4 and 8)
    - 1 bit history context
    - 1 weird, custom context (some match model state and other stuff)
  - key here is to mix/match stuff!
  - pulls a LOT of weight!
- second stage
  - 8 mixed outputs mixed again by second stage with static context

squishy model architecture

- logistic mixing
- both stages mixed with 16 bit * 8 lane SIMD
- not super fancy, mostly SSE2 with SSSE3 horizontal sums
- basically the only SIMD in the whole thing

squishy model architecture

- APM stages
  - final mixer output adjusted by 3 APM stages in serial
  - each with increasing context orders
  - static, linear weights
  - not a huge difference after heavy modeling, but pays for itself

squishy model architecture

- final output clamped to [1, 4095] and sent to coder
- lots more possibilities
- this is what worked so far
- future squishy versions will likely do different stuff
    - or not, who knows
    - we have enough tooling/demos to make already as it is!

x86 modeling in squishy

x86 modeling in squishy

- tried to avoid, couldn’t to be competitive!

x86 modeling in squishy

- tried to avoid, couldn't to be competitive!
- e8/e9 filter
  - this will be replaced shortly due to false positives
  - all experiments with fancier cache schemes so far help code compression, hurt in total, needs further work
- need something more comprehensive

x86 modeling in squishy

- main idea: leave code in-place
    - don't reorder like kkrunchy in case there are useful correlations
- use same models as in data sections
- on-the-fly state machine disasm

```
00602161 0fb6460e     movzx eax, byte ptr [esi + 0xe]
00602165 db45f8       fild dword ptr [ebp - 8]
00602168 dd05b0997800 fld qword ptr [0x7899b0]
0060216e 8945f8       mov dword ptr [ebp - 8], eax
00602171 8b450c       mov eax, dword ptr [ebp + 0xc]
00602174 dcf9         fdiv st(1), st(0)
00602176 03c0         add eax, eax
00602178 33db         xor ebx, ebx
0060217a 8945f0       mov dword ptr [ebp - 0x10], eax
0060217d d9c9         fxch st(1)
0060217f d95dfc       fstp dword ptr [ebp - 4]
00602182 db45f8       fild dword ptr [ebp - 8]
00602185 def1         fdivrp st(1)
00602187 dc0d189a7800 fmul qword ptr [0x789a18]
0060218d d95df4       fstp dword ptr [ebp - 0xc]
00602190 85c0         test eax, eax
00602192 7e7a         jle 0x60220e
00602194 57           push edi
00602195 8b4508       mov eax, dword ptr [ebp + 8]
00602198 8d3c98       lea edi, dword ptr [eax + ebx*4]
0060219b d907         fld dword ptr [edi]
0060219d 51           push ecx
0060219e d87510       fdiv dword ptr [ebp + 0x10]
006021a1 d95df8       fstp dword ptr [ebp - 8]
006021a4 d945f8       fld dword ptr [ebp - 8]
006021a7 d91c24       fstp dword ptr [esp]
006021aa e8b6feffff   call 0x602065
006021af d95d0c       fstp dword ptr [ebp + 0xc]
006021b2 d9450c       fld dword ptr [ebp + 0xc]
```

x86 modeling in squishy

- use disasm state as additional context
  - select different mixer weights, indirect probs, etc
- use disasm state to maintain different history buffers
  - one per state mostly, some for multiple states
  - represent “last opcode bytes”, “last displacement bytes”, etc
- double the number of context models in code section
  - second set only looks at history for current disasm state
  - 21 * 2 * 3 + 1 + 8 = 135 predictions!
  - single wasted SIMD lane during mixing (136 / 8 = 17)
- best of both worlds:
  - model can find correlations in in-place code
  - model can find correlations in “reordered” code (history buffers)

x86 modeling in squishy

- possibility to mix/match histories in more arbitrary ways
  - main motivation behind (so far failed) genetic algorithm idea
  - might still be interesting, needs more experimentation
- leads to larger model due to x86-specific stuff
- much can be folded into the main compressed data
  - as long as it's decompressed before code section
- additional model logic/history buffer code still somewhat big
- this is why a second compressor stage makes sense :)

wrap-up

wrap-up

- squishy is good

wrap-up

- squishy is good
- lots of stuff, this is a big project!

wrap-up

- squishy is good
- lots of stuff, this is a big project!
- very important part of the 64k tool flow

wrap-up

- squishy is good
- lots of stuff, this is a big project!
- very important part of the 64k tool flow
  - most intro authors seem to forget how much work goes into these
  - this is actually a sign of good tooling
  - have a reminder anyways :)

wrap-up

- squishy is good
- lots of stuff, this is a big project!
- very important part of the 64k tool flow
  - most intro authors seem to forget how much work goes into these
  - this is actually a sign of good tooling
  - have a reminder anyways :)
- it’s big, but not rocket science

wrap-up

- squishy is good
- lots of stuff, this is a big project!
- very important part of the 64k tool flow
  - most intro authors seem to forget how much work goes into these
  - this is actually a sign of good tooling
  - have a reminder anyways :)
- it’s big, but not rocket science
- packers are fun!

wrap-up

- squishy is good
- lots of stuff, this is a big project!
- very important part of the 64k tool flow
  - most intro authors seem to forget how much work goes into these
  - this is actually a sign of good tooling
  - have a reminder anyways :)
- it’s big, but not rocket science
- packers are fun!
- MAKE MORE INTROS!!!!!!!!!!!

wrap-up

- squishy is good
- lots of stuff, this is a big project!
- very important part of the 64k tool flow
  - most intro authors seem to forget how much work goes into these
  - this is actually a sign of good tooling
  - have a reminder anyways :)
- it’s big, but not rocket science
- packers are fun!
- MAKE MORE INTROS!!!!!!!!!!!
  - we have a synth that can help too... :)

```
    Finished release [optimized] target(s) in 0.01s
     Running `target\release\squishy.exe 'C:/Program Files (x86)/Steam/steamapps/common/DOOMEternal/DOOMEternalx64vk.exe' ./out.exe`
squishy 0.1.0 | made with <3 by Jake "ferris" Taylor / logicoma 2016-2020
 - big squish:  510674660 -> 16213766 (96.83%) in 888.75s (~561.13kb/s)
thread 'main' panicked at 'Compressed size too large; can't adjust image base to make room for compressed image.', src\main.rs:357:9
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
error: process didn't exit successfully: `target\release\squishy.exe 'C:/Program Files (x86)/Steam/steamapps/common/DOOMEternal/DOOM
Eternalx64vk.exe' ./out.exe` (exit code: 101)
```

many thank, wow

jake “ferris” taylor / logicoma

@ferristweetsnow

yupferris at gee-mail

turn back, here be dragons

FORK ENDS HERE BRUH

- Asdfasdfasdfasdfasfasdf
- Asdf
- Asd
- Fa
- Sdf
- Asdf
- Asdf
- Asd
- Fasd
- f

compression 101

- let's take an example, our string from earlier: AABC
- armed with a model: p = { A: 50%, B: 25%, C: 25% }
- create a new encoding by assigning new bit strings to the original symbols
- intuitively, make more common symbols use fewer bits than less common symbols

- eg. e = { A: 0, B: 10, C: 11 }

compression 101

- let’s take an example, our string from earlier: AABC
- armed with a model: p = { A: 50%, B: 25%, C: 25% }
- and an encoding: { A: 0, B: 10, C: 11 }
- encode our string with our encoding via per-symbol substitution

- A A B C
- 0 0 10 11 -> 001011
- the same string is 6 bits in our new representation!

compression 101

- let's try decoding now, using our encoding:
  - { A: 0, B: 10, C: 11 }
- decode our string with our encoding via per-symbol substitution

- 001011 -> 0 0 10 11
- A A B C
- it works!

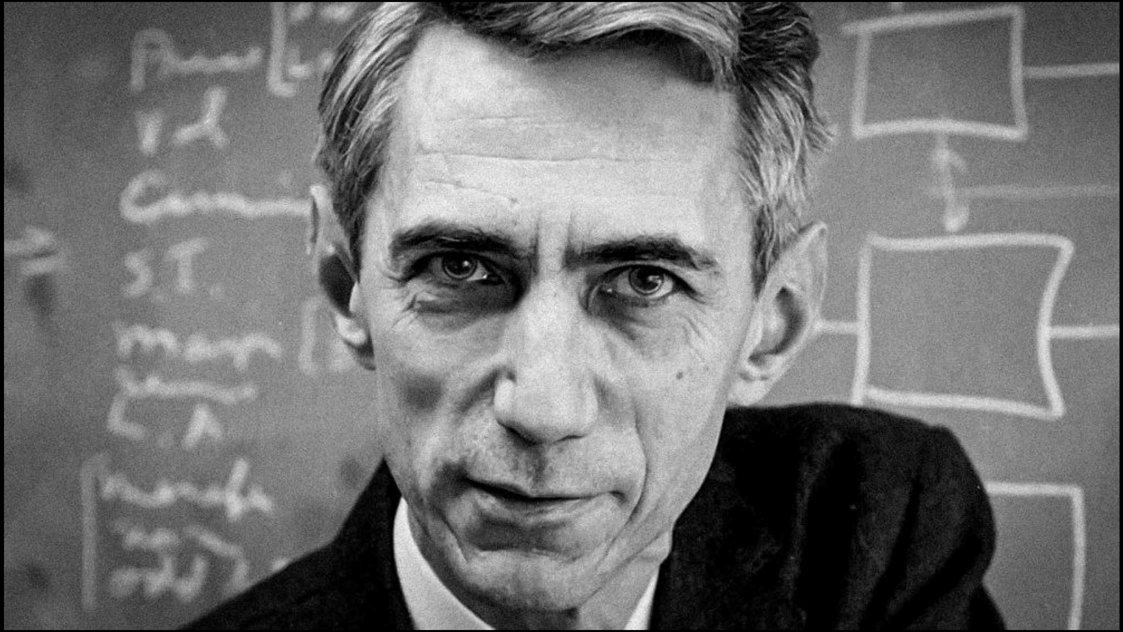
compression 101

- this was a contrived example, but you just learned a lot!
- our encoding was an example of a prefix code (just like huffman)
- it was also an optimal code for our model
  - this means that given the same model, we can not make a representation that would code the string using fewer bits than this!

- wait what how

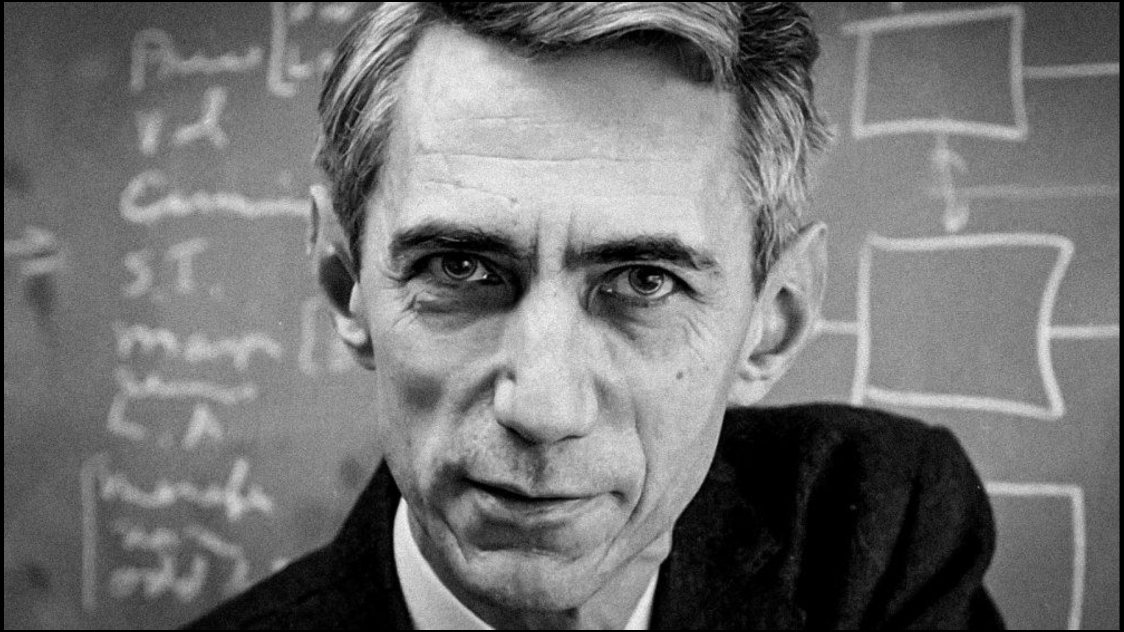
compression 101

- meet claude shannon



compression 101

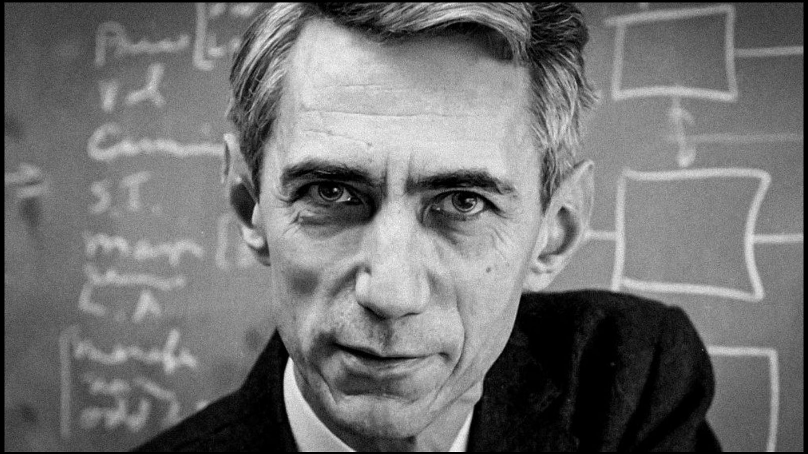
- meet claude shannon
- look at that hot piece of man(tropy)
- ok this pic is haunting af but hear me out



compression 101

- claudy with a shans of meatballs over here did this really cool thing
    - he actually did a butt ton of awesome stuff!!!
- he came up with a way to quantify the average information content of a string
- he called it entropy
- and it goes a little somethin like this:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i$$



compression 101

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

compression 101

log2 of symbol probability

negative
for some reason

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

entropy

for every symbol, sum the following

probability of symbol i

compression 101

$$-\sum_{i=1}^{n} p_i \log_2 p_i$$

compression 101

$$\sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i}$$

compression 101

$$\sum_{i=1}^{n} \mathtt{f(i)} \log_2 \frac{1}{\mathtt{p(i)}}$$

not exactly
equivalent but it
scales the same
which is the
important part so
pls ignore

compression 101

pretend this is linear

$$\sum_{i=1}^{n} \mathtt{f(i)} \log_2 \frac{1}{\mathtt{p(i)}}$$

compression 101

improbable symbols
(p(i) near zero)
have high
information content
(entropy BIG)

$$\sum_{i=1}^{n} f(i) \log_2 \frac{1}{p(i)}$$

compression 101

$$\sum_{i=1}^{n} f(i) \log_2 \frac{1}{p(i)}$$

improbable symbols (p(i) near zero) have high information content (entropy BIG)

probable symbols (p(i) near one) have low information content (entropy SMOL)

compression 101

improbable symbols
(p(i) near zero)
have high
information content
(entropy BIG)

$$\sum_{i=1}^{n} f(i) \log_2 \frac{1}{p(i)}$$

probable symbols
(p(i) near one) have
low information
content (entropy
SMOL)

we pay for a
symbol's information
content for each
occurrance of the
symbol!

compression 101

$$\sum_{i=1}^{n} \mathtt{f(i)} \log_2 \frac{1}{\mathtt{p(i)}}$$

compression 101

$$\sum_{i=1}^{n} \mathtt{f(i)} \log_2 \frac{1}{\mathtt{p(i)}}$$

compression 101

$$\sum_{i=1}^{n} \mathtt{f(i)} \log_2 \frac{1}{\mathtt{p(i)}}$$

this is actually how
many bits symbol i
should be coded with
optimally!

compression 101

multiply by the
frequency of symbol i

$$\sum_{i=1}^{n} f(i) \log_2 \frac{1}{p(i)}$$

this is actually how
many bits symbol i
should be coded with
optimally!

compression 101

multiply by the
frequency of symbol i

$$\sum_{i=1}^{n} f(i)\log_2 \frac{1}{p(i)}$$

for every symbol in
the string

this is actually how
many bits symbol i
should be coded with
optimally!

compression 101

$$\sum_{i=1}^{n} f(i) \log_2 \frac{1}{p(i)}$$

this equation gives us the
optimal number of bits we can
use to code our string!!!

compression 101

$$\sum_{i=1}^{n} f(i) \log_2 \frac{1}{p(i)}$$

this equation gives us the optimal number of bits we can use to code our string!!!

compression 101

$$-\sum_{i=1}^{n} p_i \log_2 p_i$$

the real entropy equation is just
a normalized version of that :)

compression 101

- let’s apply this to our example

- optimal_bits(AABC) =
  optimal_bits(AA) +
  optimal_bits(B) +
  optimal_bits(C)

compression 101

- let's apply this to our example

- optimal_bits(AABC) =
  ```
  2 * log2(1 / 1/2) +
  1 * log2(1 / 1/4) +
  1 * log2(1 / 1/4)
  ```

compression 101

- let's apply this to our example

- optimal_bits(AABC) =
  2 * log2(2) +
  1 * log2(4) +
  1 * log2(4)

compression 101

- let’s apply this to our example

- optimal_bits(AABC) =
  2 * 1 +
  1 * 2 +
  1 * 2

compression 101

- let's apply this to our example

- optimal_bits(AABC) =
  2 +
  2 +
  2

compression 101

- let’s apply this to our example

- optimal_bits(AABC) = 6

compression 101

- let’s apply this to our example
- optimal_bits(AABC) = 6
- neato burrito