Google

# A Hacker's guide to reducing side-channel attack surfaces using deep-learning


Elie Bursztein
Google, @elie

Jean-Michel Picod
Google, @jmichel_p

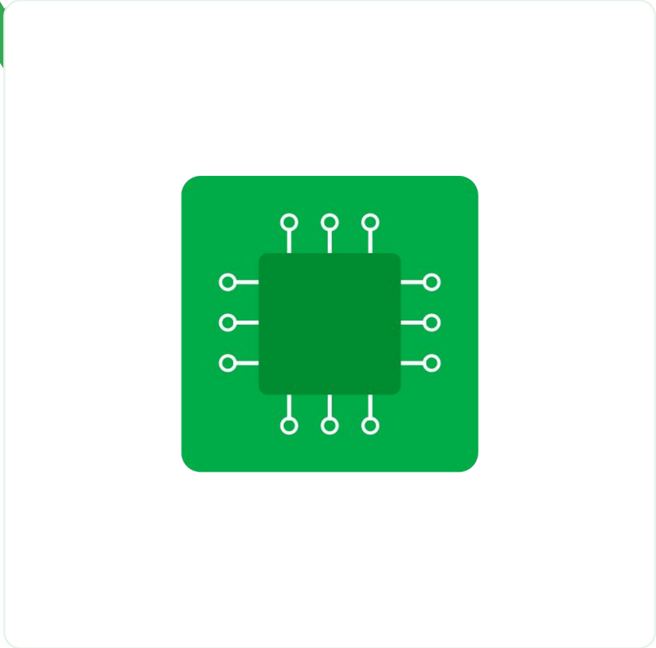

with the help of **many** Googlers and external collaborators

Security and Privacy Group

Talk is based on some of the results of a joint research project with many collaborators on hardening hardware cryptography

# Work in progress

Experimental results and code ahead



Side channel attacks are one of the most efficient ways to attack secure hardware

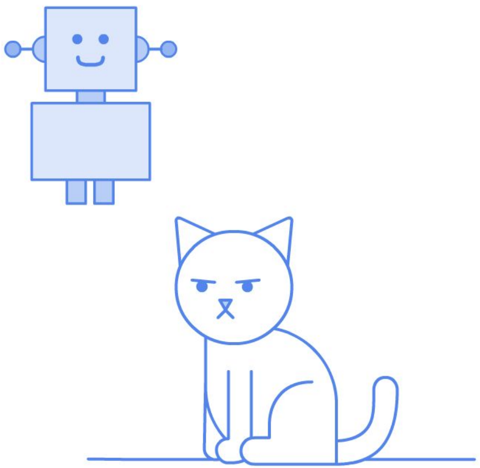
# A side-channel attack was used to recover the Trezor bitcoin wallet private key


https://jochen-hoenicke.de/crypto/trezor-power-analysis/

# Side-channels attacks are notoriously hard to debug and fix

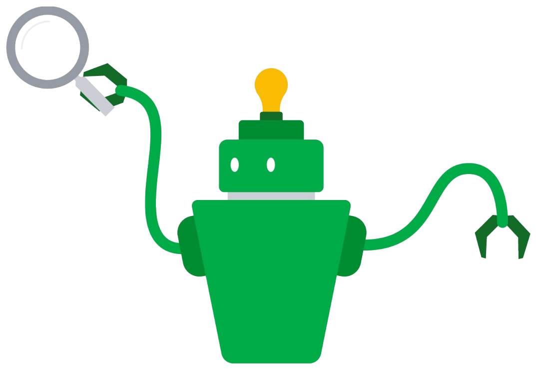

# Can we create a debugger that accurately pinpoints the code vulnerable to side-channel attacks?

# Combine deep-learning and dynamic analysis to pinpoint origin of leakage



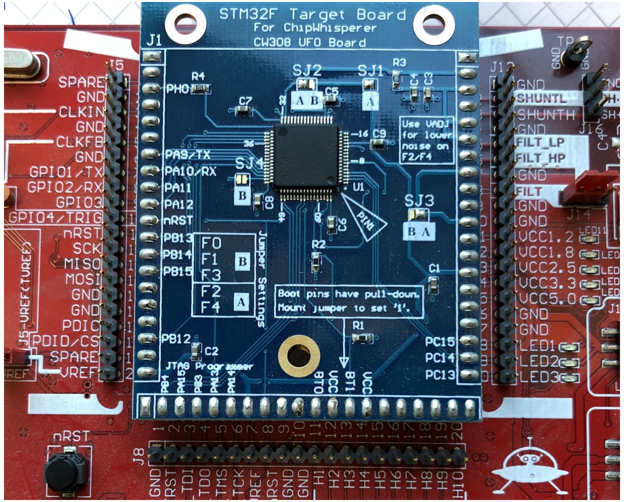
# AI? Really?

# Side Channel Attacks Leak Detector



Google

Security and Privacy Group




# **Today's goal:** use SCALD to debug tinyAES running on STM32F4

# Agenda

- What are side channels?
- AI based side-channel attacks
- AI explainability
- Finding implementation leakage origin with SCALD

# Code and slides

https://elie.net/scald

# Disclaimer

This talk purposely focuses on showcasing a high level overview of how to debug a cryptographic implementation end-to-end using SCALD. For technical details, see the paper

Part 1

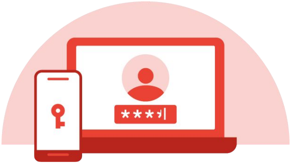
# What are side-channel attacks?

A side-channel attack is **an indirect measurement** of a **computation result** via an **auxiliary mechanism**

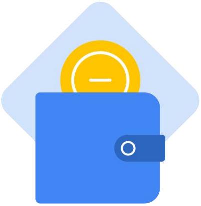
# Real-world side-channel applications

**Recover encryption keys**

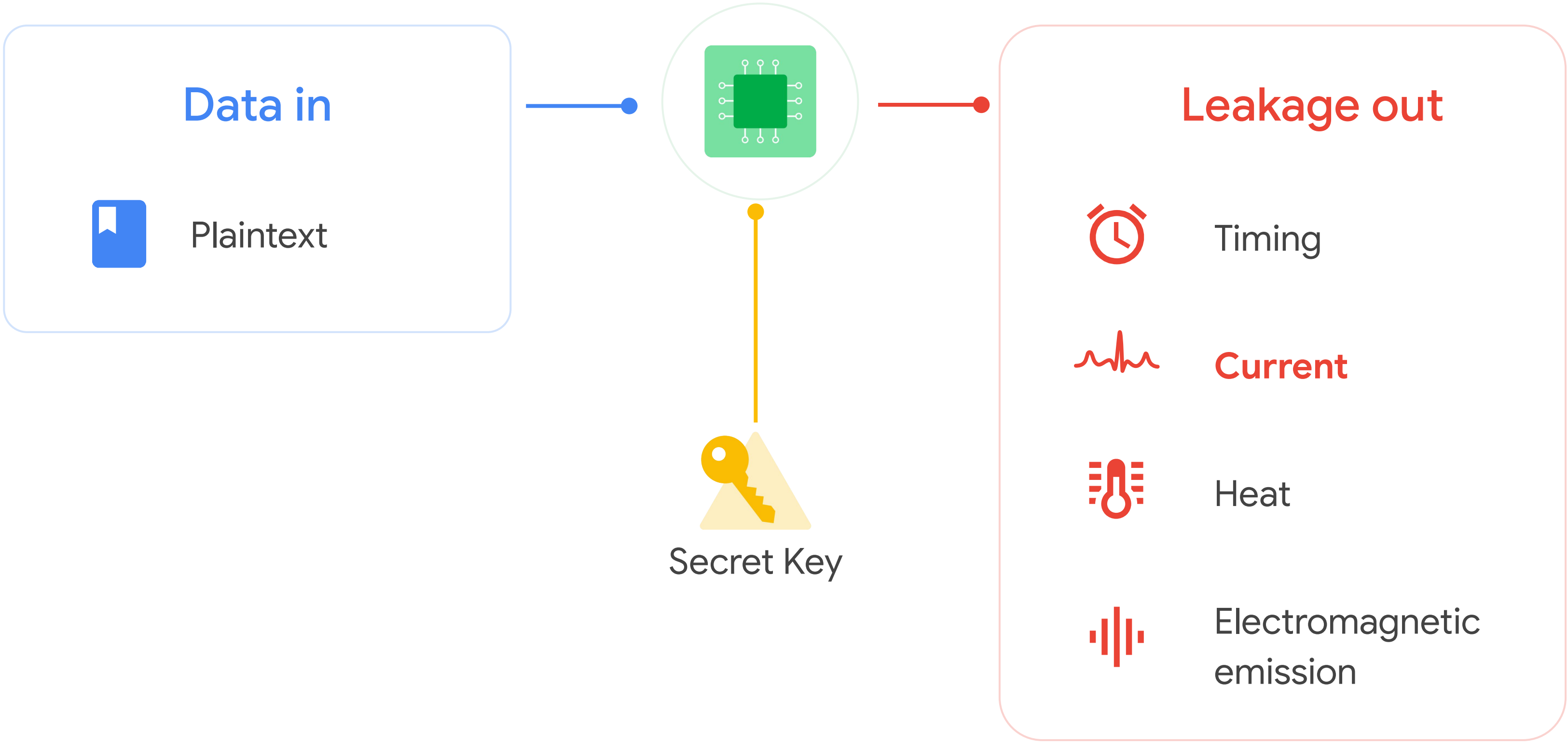
Perform blind SQL injections

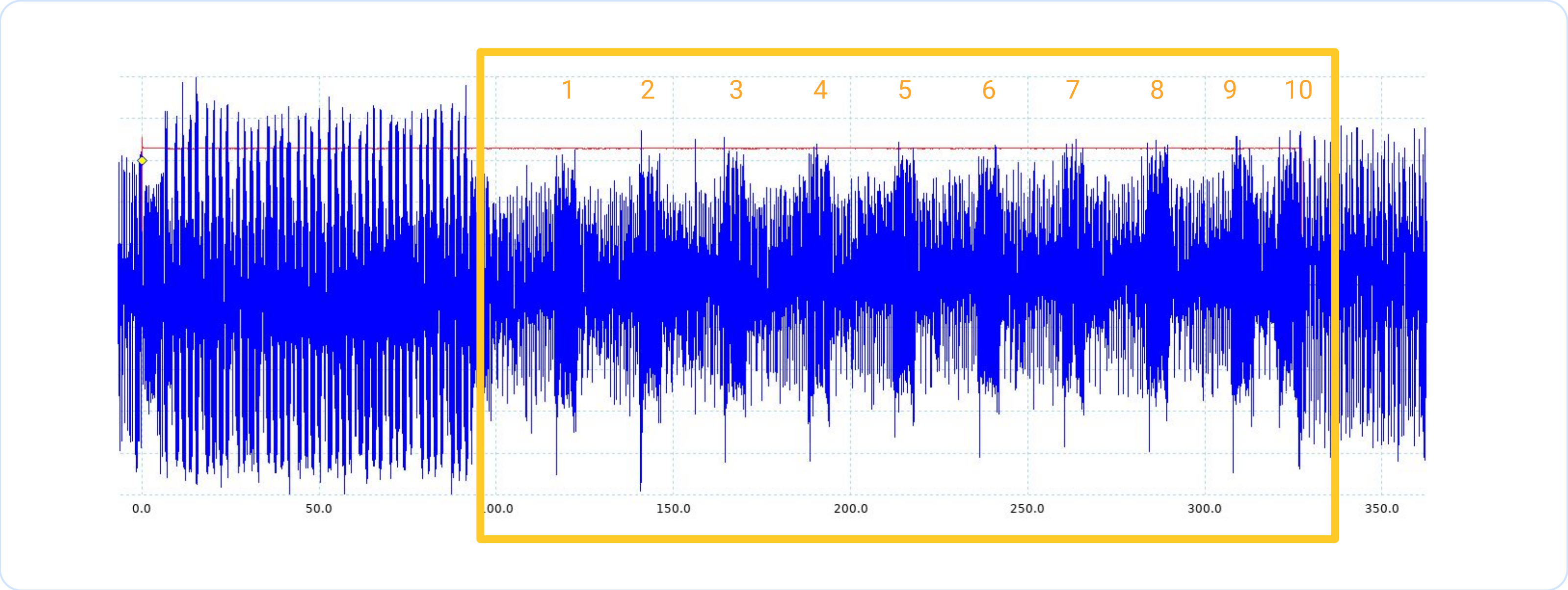


Steal passwords and pins



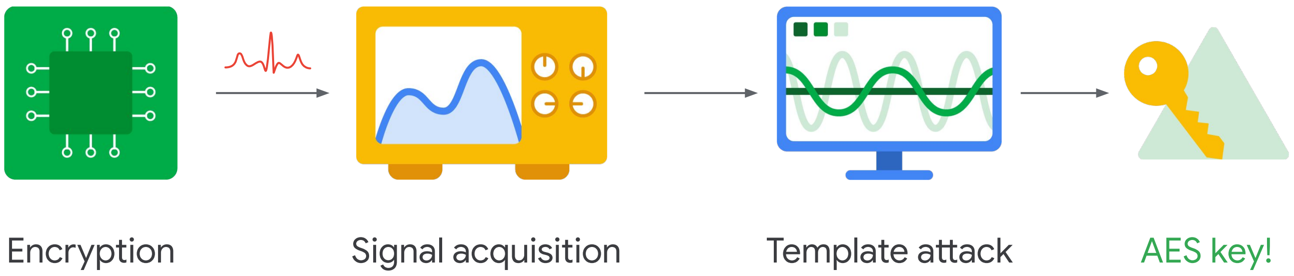
Extract crypto wallets


Data in
Plaintext
Secret Key
Leakage out
Timing
Current
Heat
Electromagnetic emission



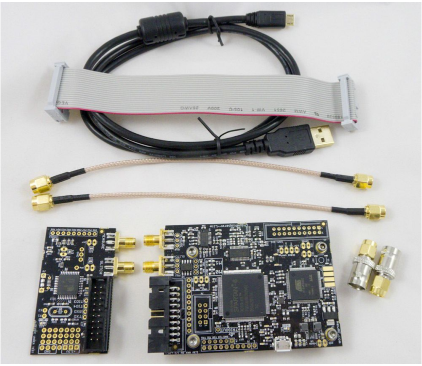

# AES round are visible in lightly protected AES implementation power traces

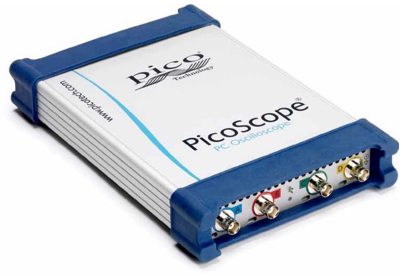
# SCA in a nutshell



# NewAE Chipwhisperer Pro + Picoscope 6000 for fast sampling rate is what we use for our research

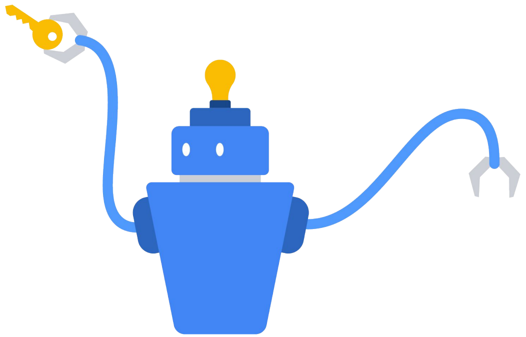
This is not an ad :) it is a recommendation based on what we use





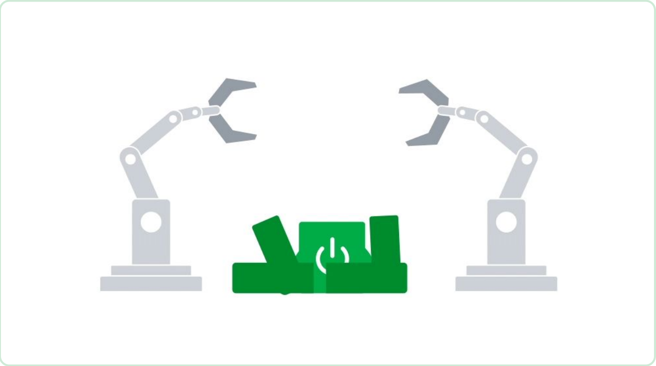
Section 2

# AI based side-channel attacks



# Side Channel Attacks Automated with Machine Learning

# How do SCAAML attacks work in practice?



Google

Security and Privacy Group

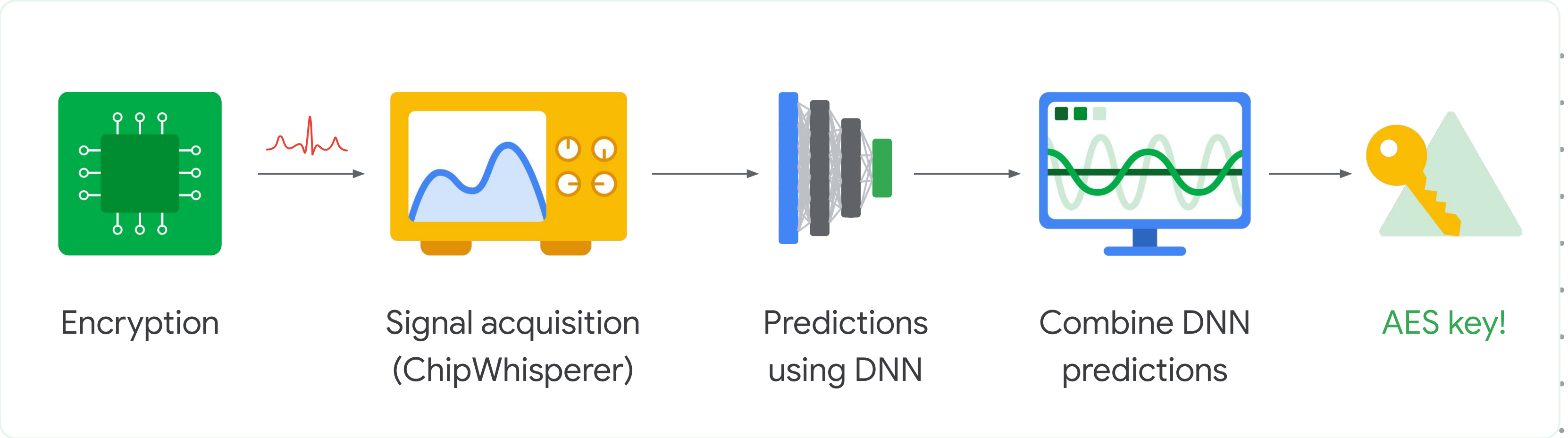
Check out last year talk for in-depth explanation https://elie.net/scaaml

# Threat model

## whitebox attack

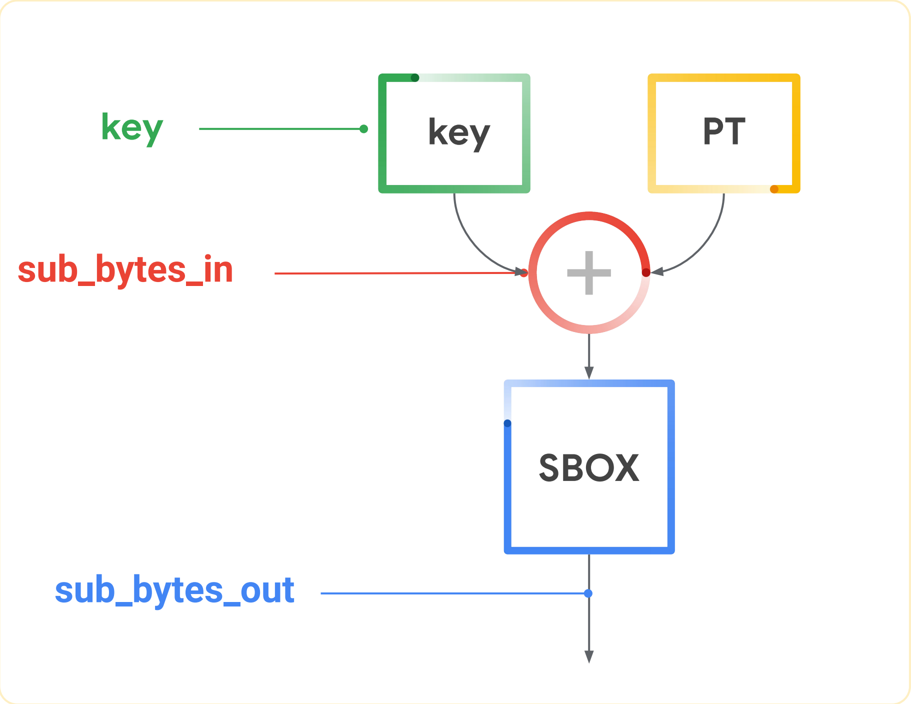
**Contrary to our previous work** that focused on black box attacks, **the traces used in this talk are truncated and collected synchronously** to improve debugging quality. This is **consistent with the white-box attack model** used during chip development. Additionally, the model architecture is also optimized for debugging, not pure performance.

# SCAAML process overview

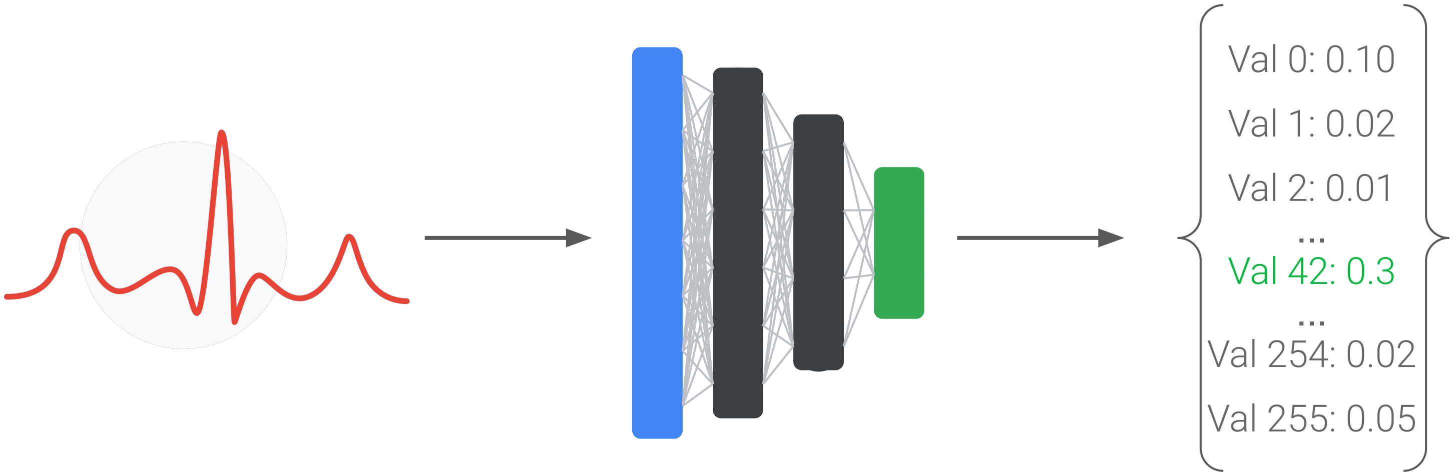


TinyAES has multiples attack points that can be targeted by SCAAML.

Today we focus on sub_bytes_in



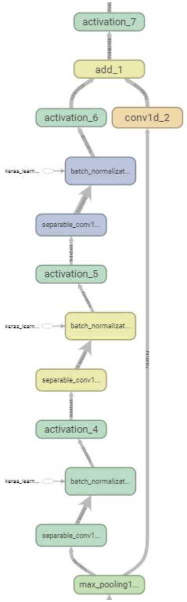
# Probabilistic attack: single trace



# Probabilistic attack: summing predictions*

{ Val 0: 0.10, Val 1: 0.02, Val 2: 0.01, ..., Val 42: 0.3, ..., Val 254: 0.02, Val 255: 0.05 } + . . . + { Val 0: 0.08, Val 1: 0.04, Val 2: 0.05, ..., Val 42: 0.12, ..., Val 254: 0.03, Val 255: 0.10 } → { Val 0: 4.4, Val 1: 5.3, Val 2: 3.2, ..., Val 42: 21.4, ..., Val 254: 2.9, Val 255: 4.2 }

*sum uses log10 + ε

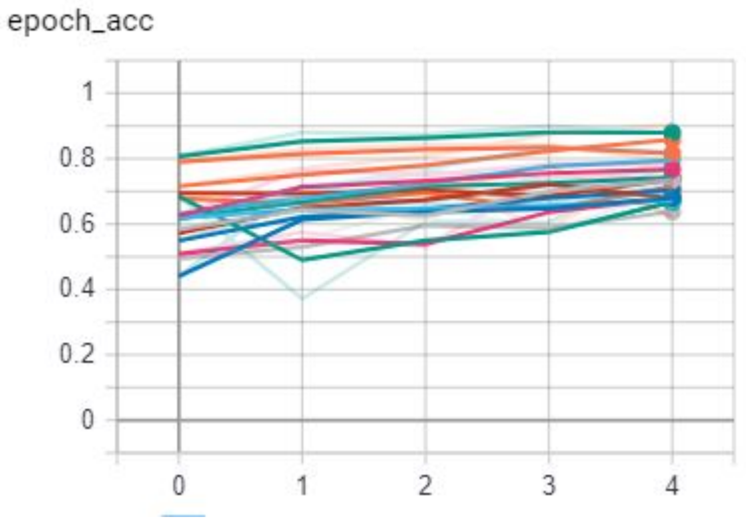



Custom residual block used

# Model architecture

Hypertuned residual separated 1D convolution network

# Tensorboards - 1 model per byte




| | Name | Smoothed | Value | Step |
|---|---|---|---|---|
| | tinyaes_sync-cnn-v3-ap_sub_bytes_in-byte_0-len_8000\validation | 0.8795 | 0.8787 | 4 |
| | tinyaes_sync-cnn-v3-ap_sub_bytes_in-byte_1-len_8000\validation | 0.8165 | 0.7926 | 4 |
| | tinyaes_sync-cnn-v3-ap_sub_bytes_in-byte_10-len_8000\validation | 0.7671 | 0.7822 | 4 |
| | tinyaes_sync-cnn-v3-ap_sub_bytes_in-byte_11-len_8000\validation | 0.7345 | 0.7798 | 4 |
| | tinyaes_sync-cnn-v3-ap_sub_bytes_in-byte_12-len_8000\validation | 0.6796 | 0.7205 | 4 |
| | tinyaes_sync-cnn-v3-ap_sub_bytes_in-byte_13-len_8000\validation | 0.6722 | 0.6948 | 4 |
| | tinyaes_sync-cnn-v3-ap_sub_bytes_in-byte_14-len_8000\validation | 0.6673 | 0.787 | 4 |
| | tinyaes_sync-cnn-v3-ap_sub_bytes_in-byte_15-len_8000\validation | 0.8582 | 0.9032 | 4 |
| | tinyaes_sync-cnn-v3-ap_sub_bytes_in-byte_2-len_8000\validation | 0.6791 | 0.6245 | 4 |
| | tinyaes_sync-cnn-v3-ap_sub_bytes_in-byte_3-len_8000\validation | 0.6799 | 0.7369 | 4 |
| | tinyaes_sync-cnn-v3-ap_sub_bytes_in-byte_4-len_8000\validation | 0.6377 | 0.702 | 4 |
| | tinyaes_sync-cnn-v3-ap_sub_bytes_in-byte_5-len_8000\validation | 0.7029 | 0.7336 | 4 |
| | tinyaes_sync-cnn-v3-ap_sub_bytes_in-byte_6-len_8000\validation | 0.7951 | 0.8205 | 4 |
| | tinyaes_sync-cnn-v3-ap_sub_bytes_in-byte_7-len_8000\validation | 0.7423 | 0.7649 | 4 |
| | tinyaes_sync-cnn-v3-ap_sub_bytes_in-byte_8-len_8000\validation | 0.7139 | 0.8047 | 4 |
| | tinyaes_sync-cnn-v3-ap_sub_bytes_in-byte_9-len_8000\validation | 0.7366 | 0.803 | 4 |

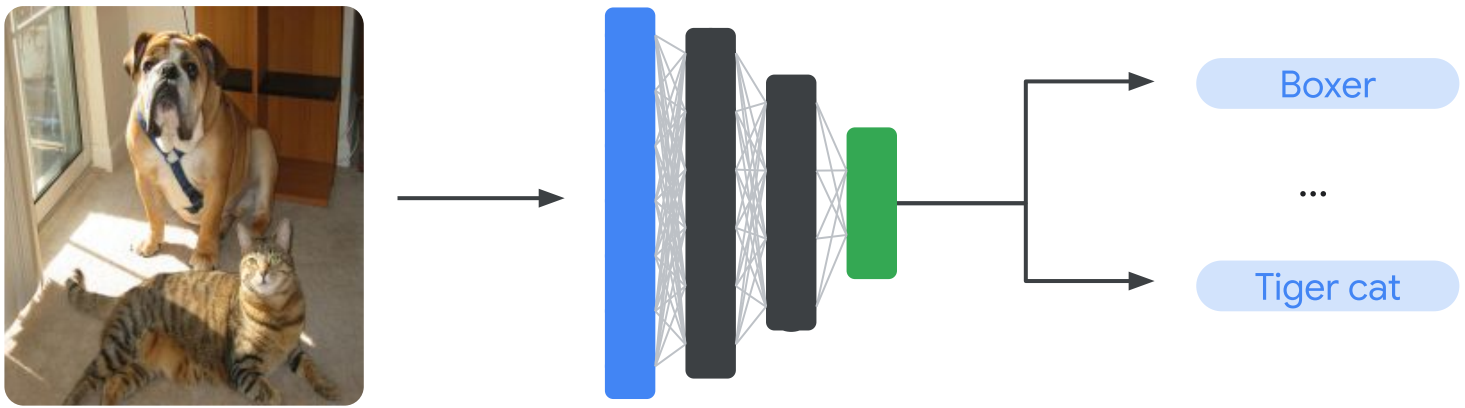
Our side-channel optimized model architecture yield 16 high accuracy model in 5 epoch as expect on this easy use-case

# How to find where TinyAES is leaking using our model?

Section 3

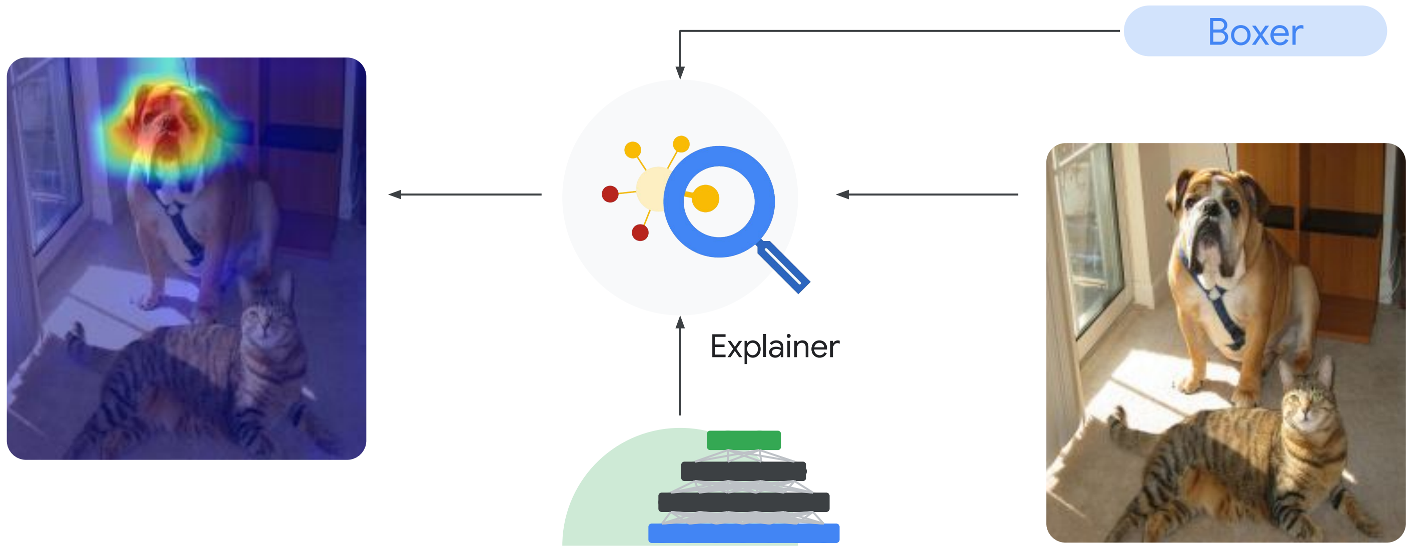
# Deep-learning explainability

# A classic vision model prediction

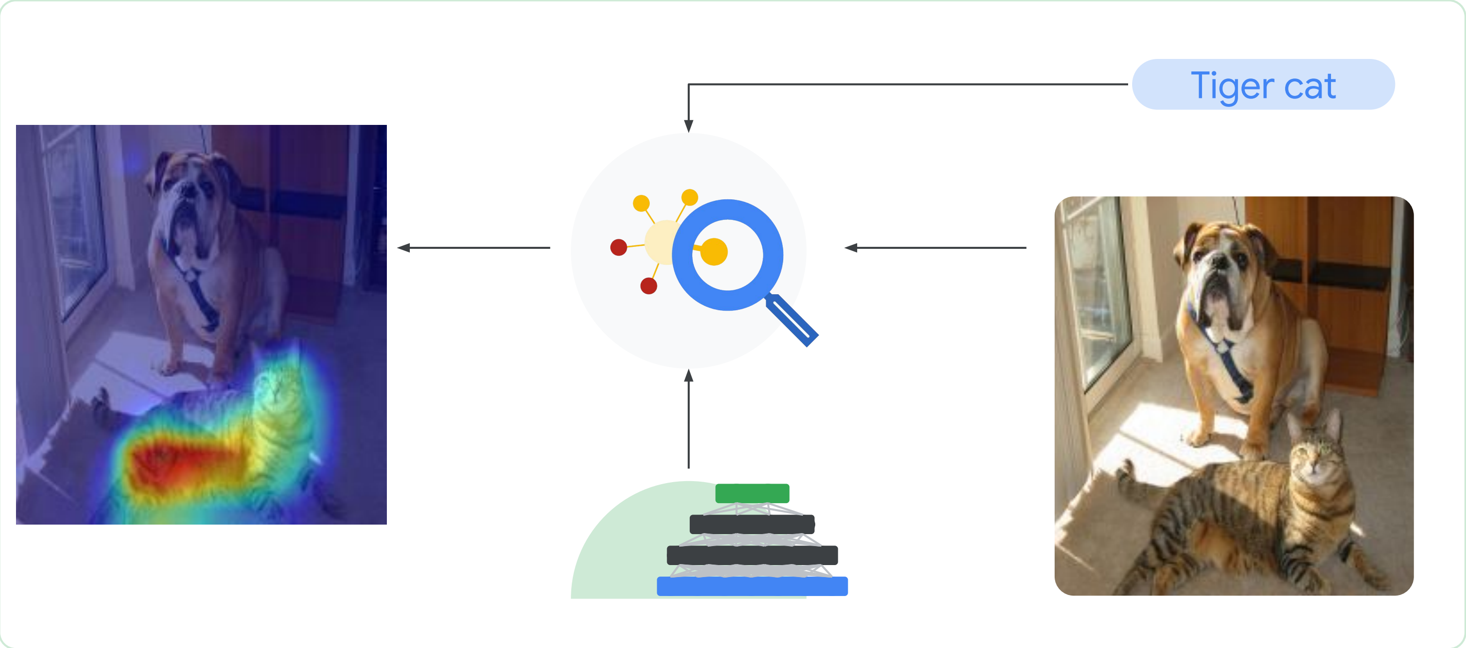


# Why did the model predict a tiger cat and a boxer?

# Explainability to the rescue: boxer prediction

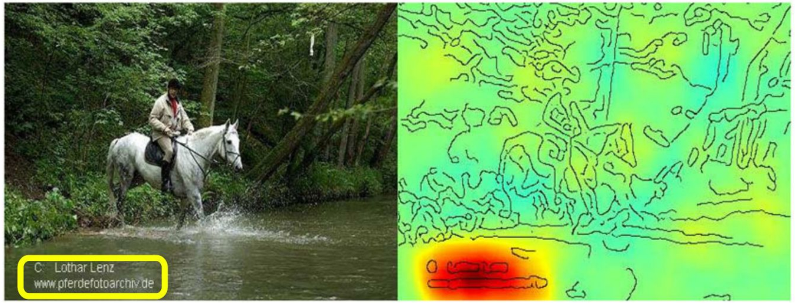


# Explainability to the rescue: **cat prediction**



# Identifying errors and biases

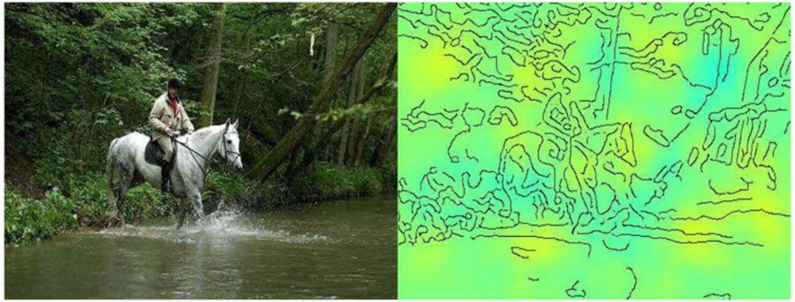
Horse-picture from Pascal VOC data set




Source tag present

↓

Classified as horse



No source tag present

↓

Not classified as horse


How do I use explainability and combine it with dynamic analysis to debug leakages?

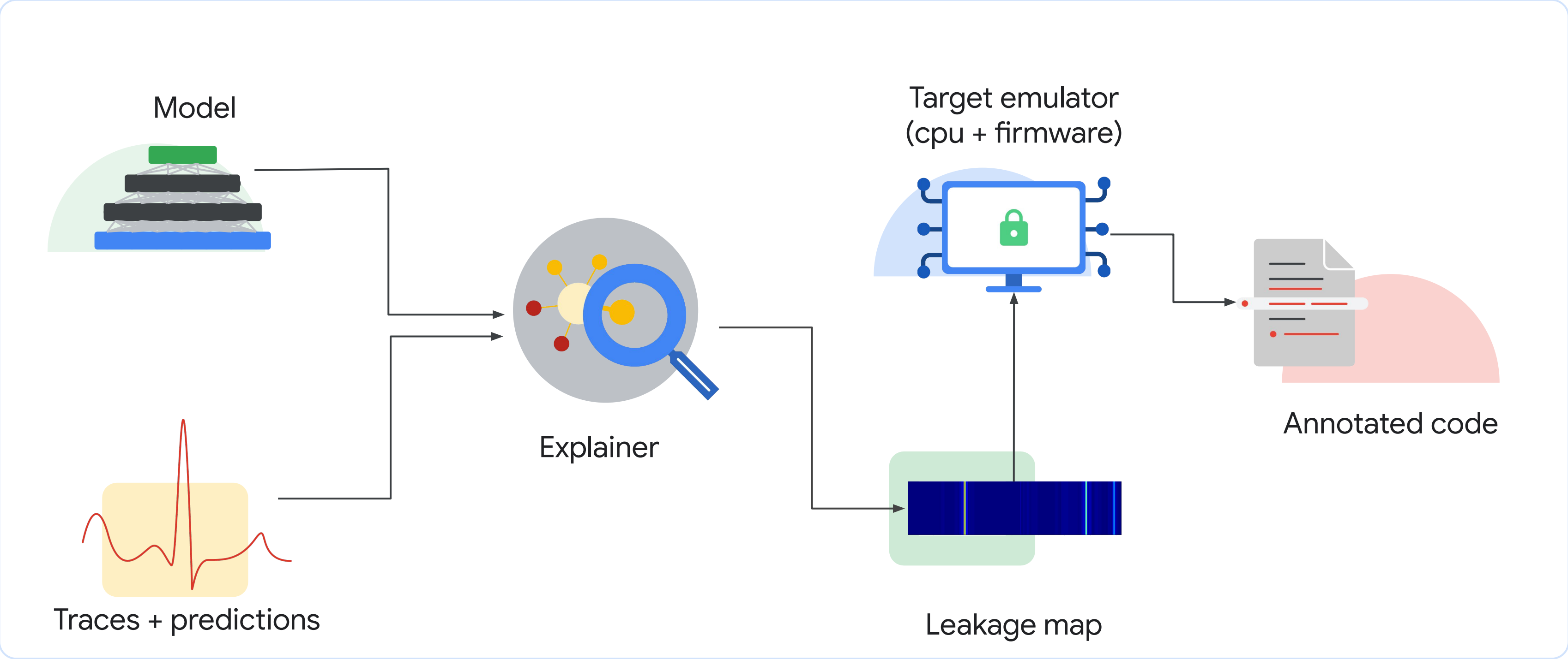
Section 4

# Finding leakage origin with SCALD

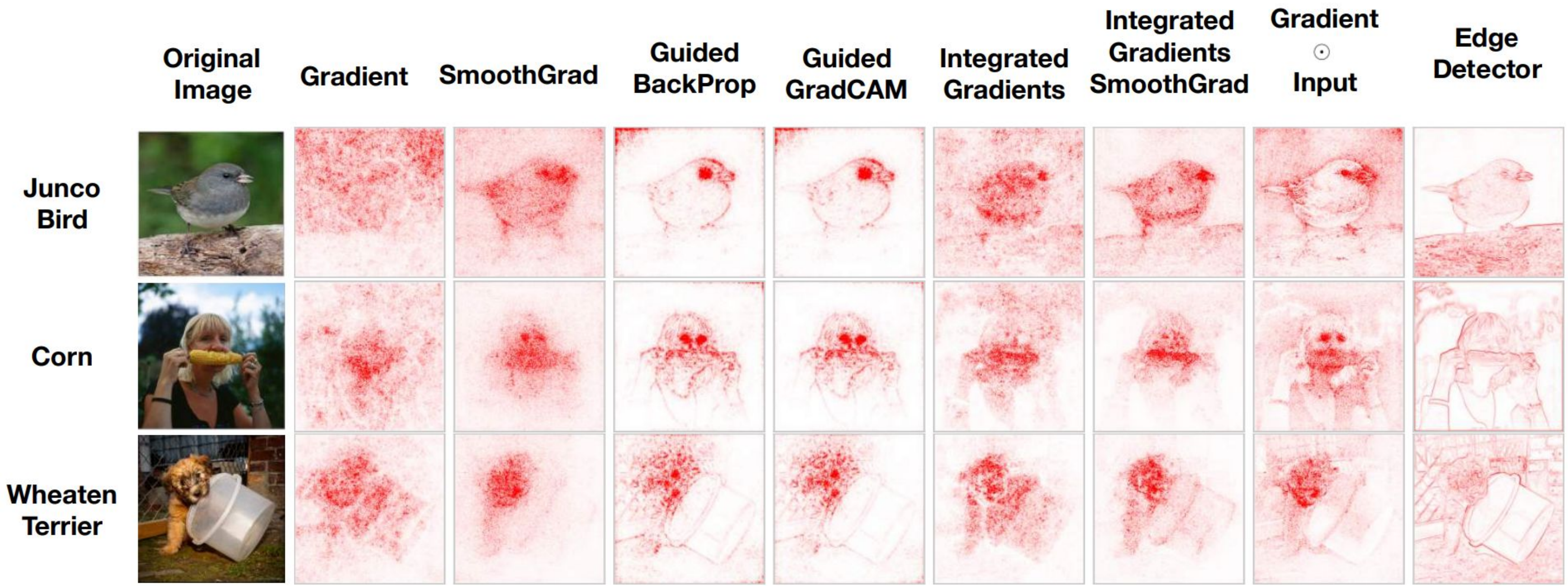
Google

Security and Privacy Group

# SCALD: Game plan

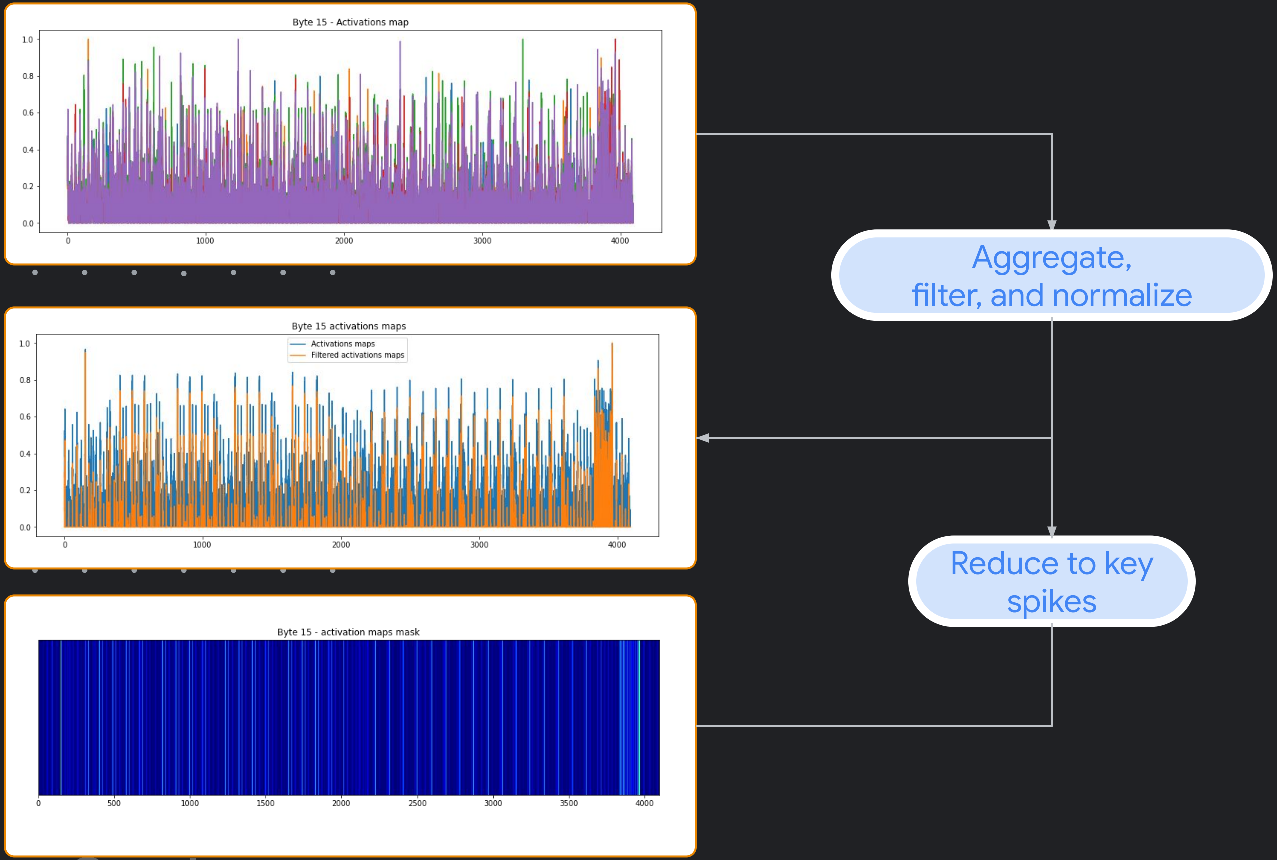


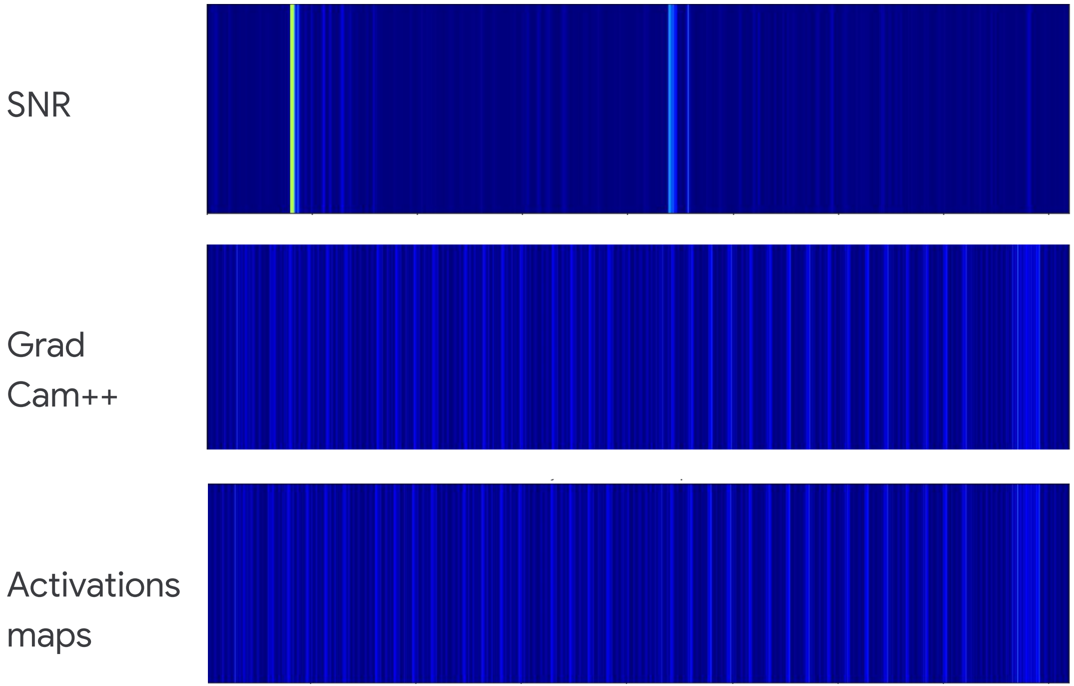
# Many explainability techniques exists




# Which explainability techniques work best?

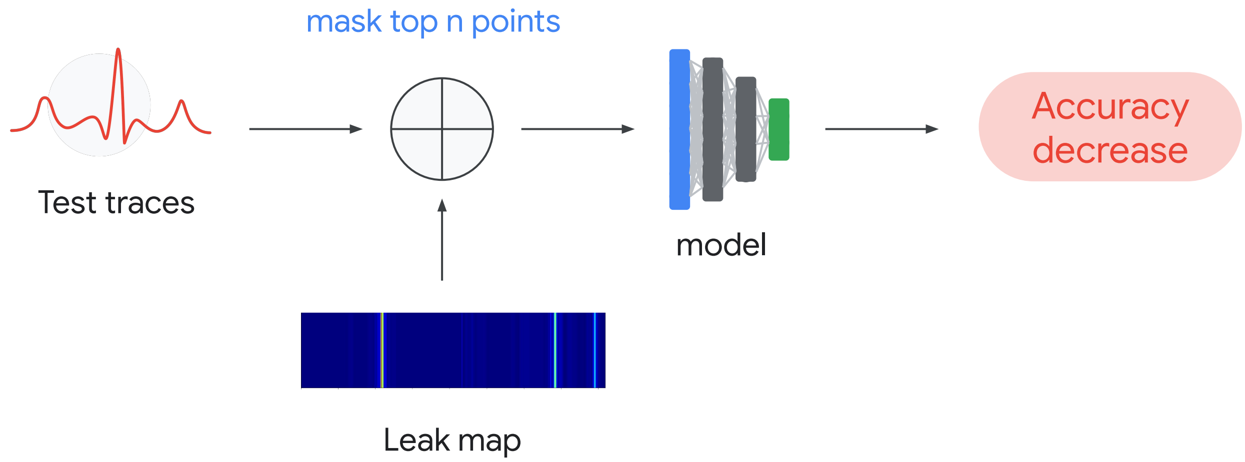
# Leak maps


Byte 15 - Activations map
Aggregate, filter, and normalize
Byte 15 activations maps
Activations maps
Filtered activations maps
Reduce to key spikes
Byte 15 - activation maps mask

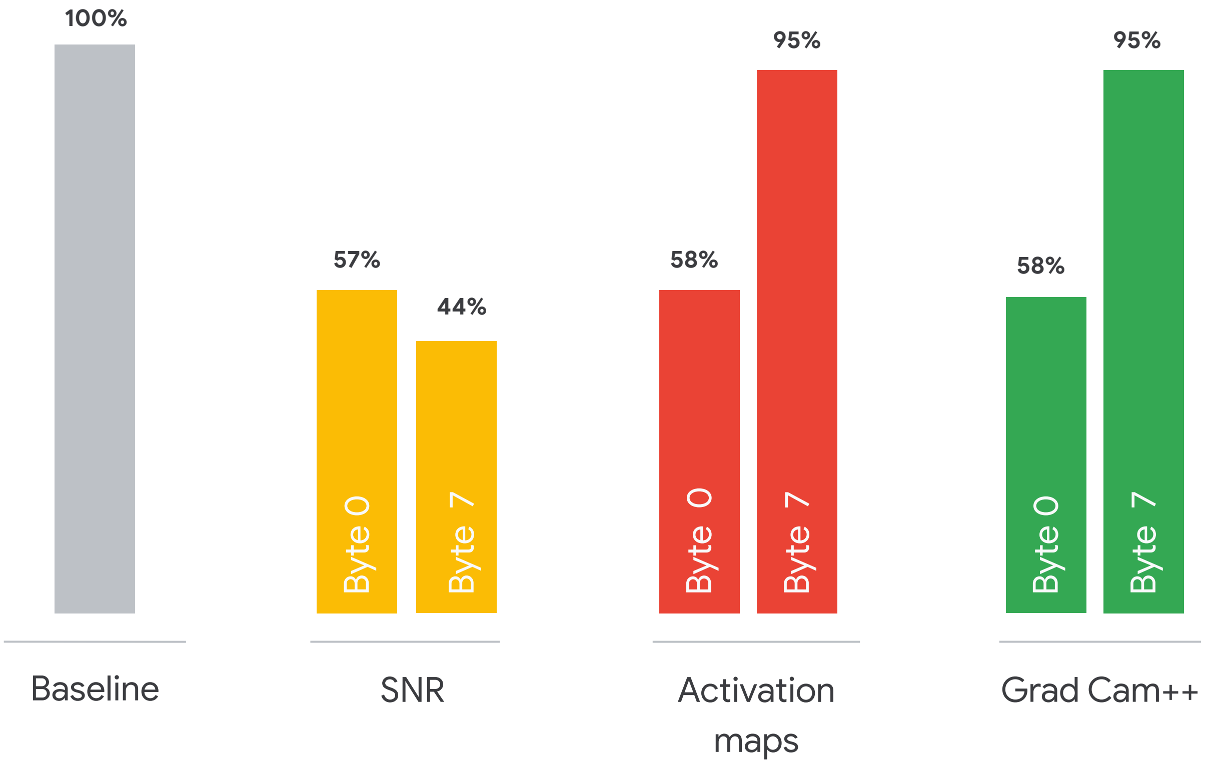



# Byte 0 leak map visualization for various techniques

# Benchmarking key explainability techniques



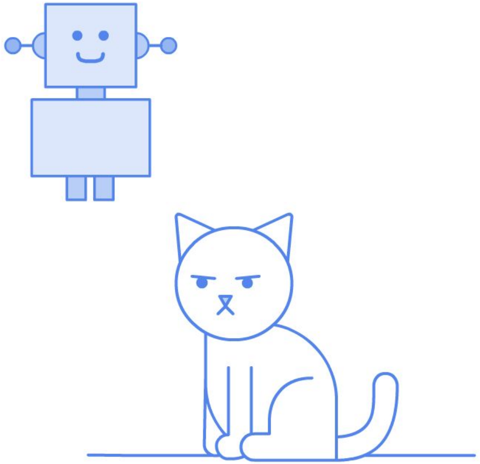
# Benchmark results: lower is better




Preliminary results - 4 points masked

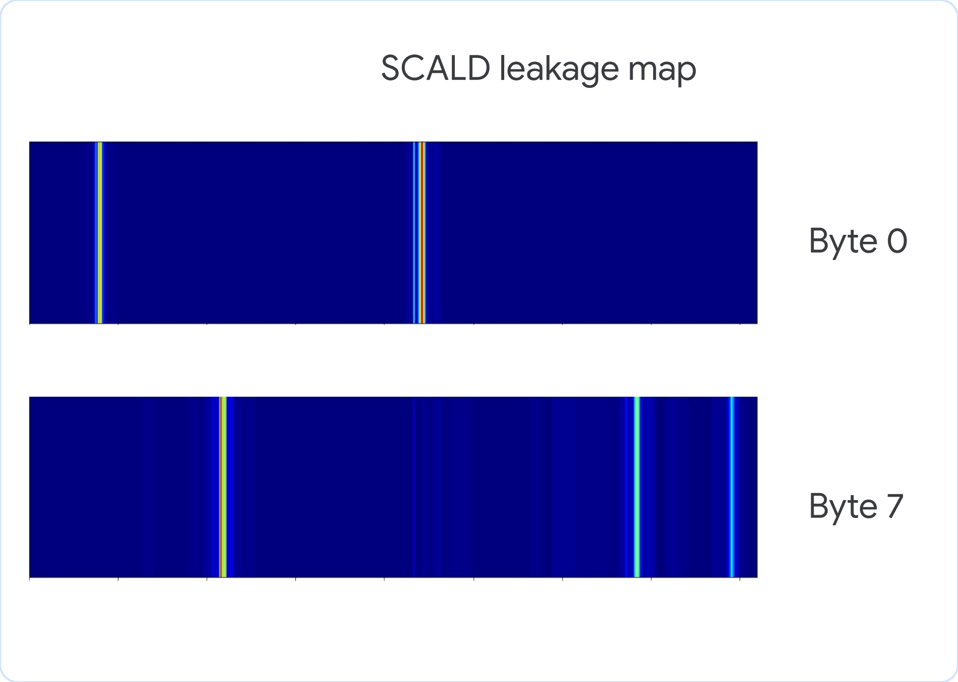
# Explainability techniques don’t work better than SNR and have very noisy leak maps

# Develop a technique tailored to leakage explanation

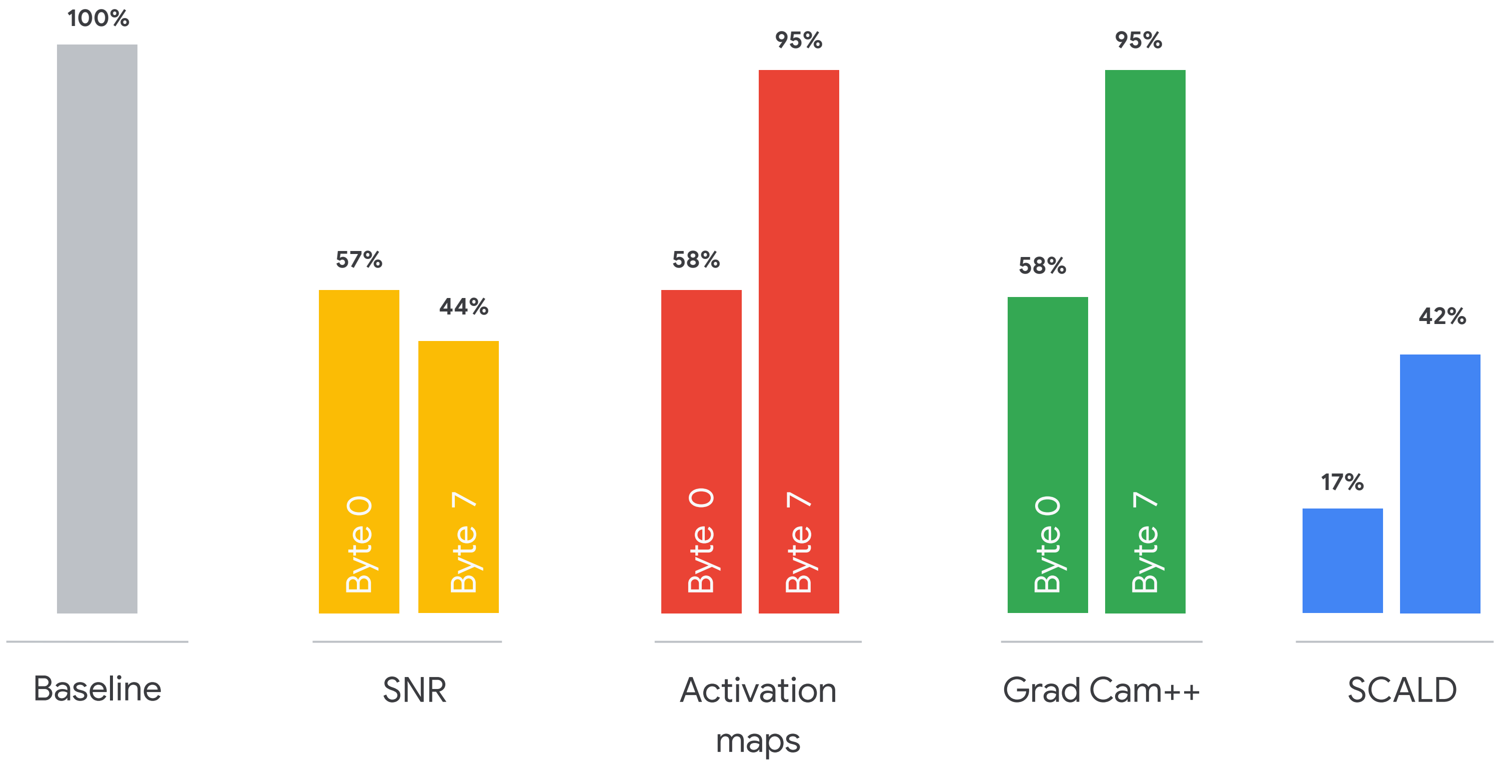


# Custom code? Really?

SCALD explainer combines partitioned and convolutive occlusion for speed and precise leakage pinpointing



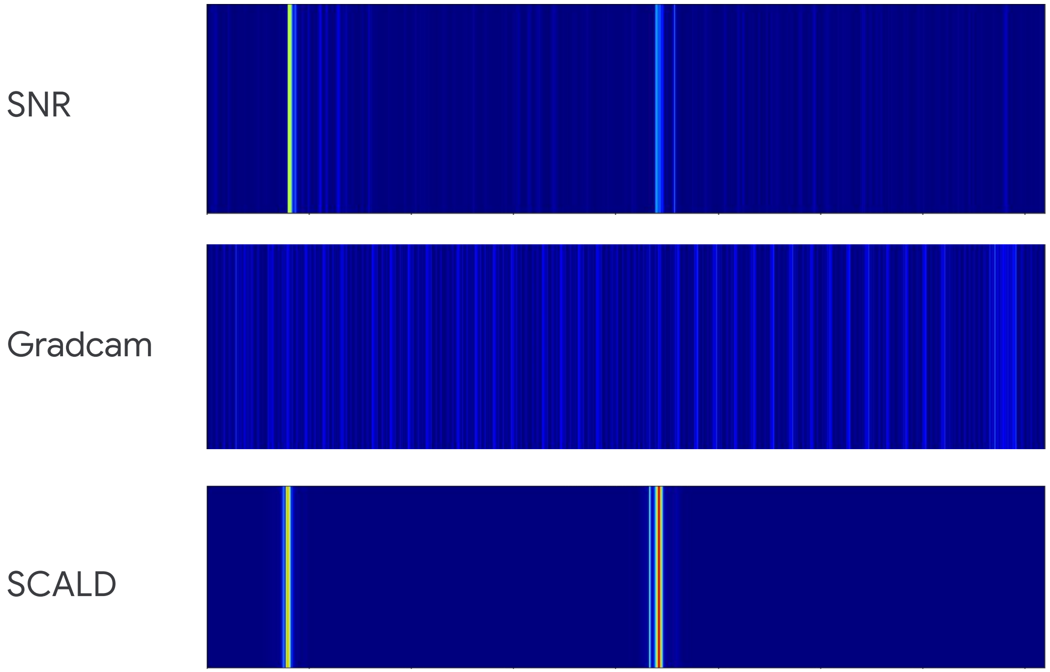
# Benchmark results: lower is better




Google

Preliminary results - 4 points masked

Security and Privacy Group

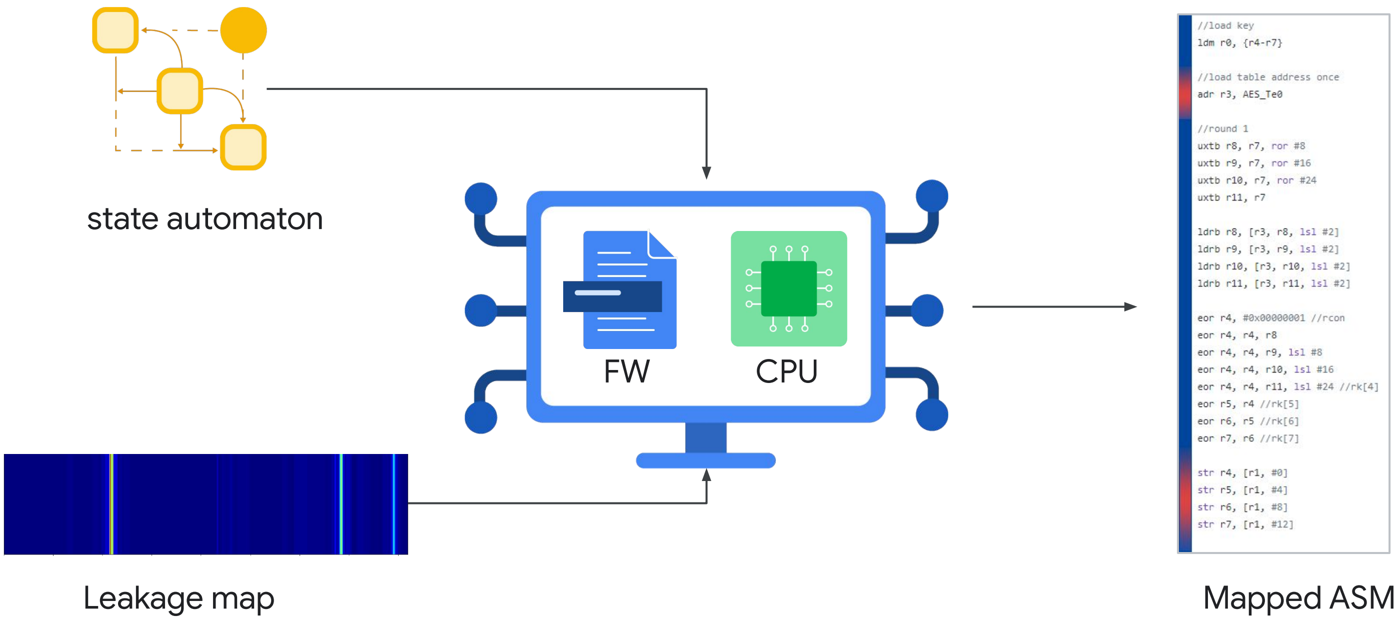



byte 0 leak maps comparaison: the SCALD map is visibly cleaner

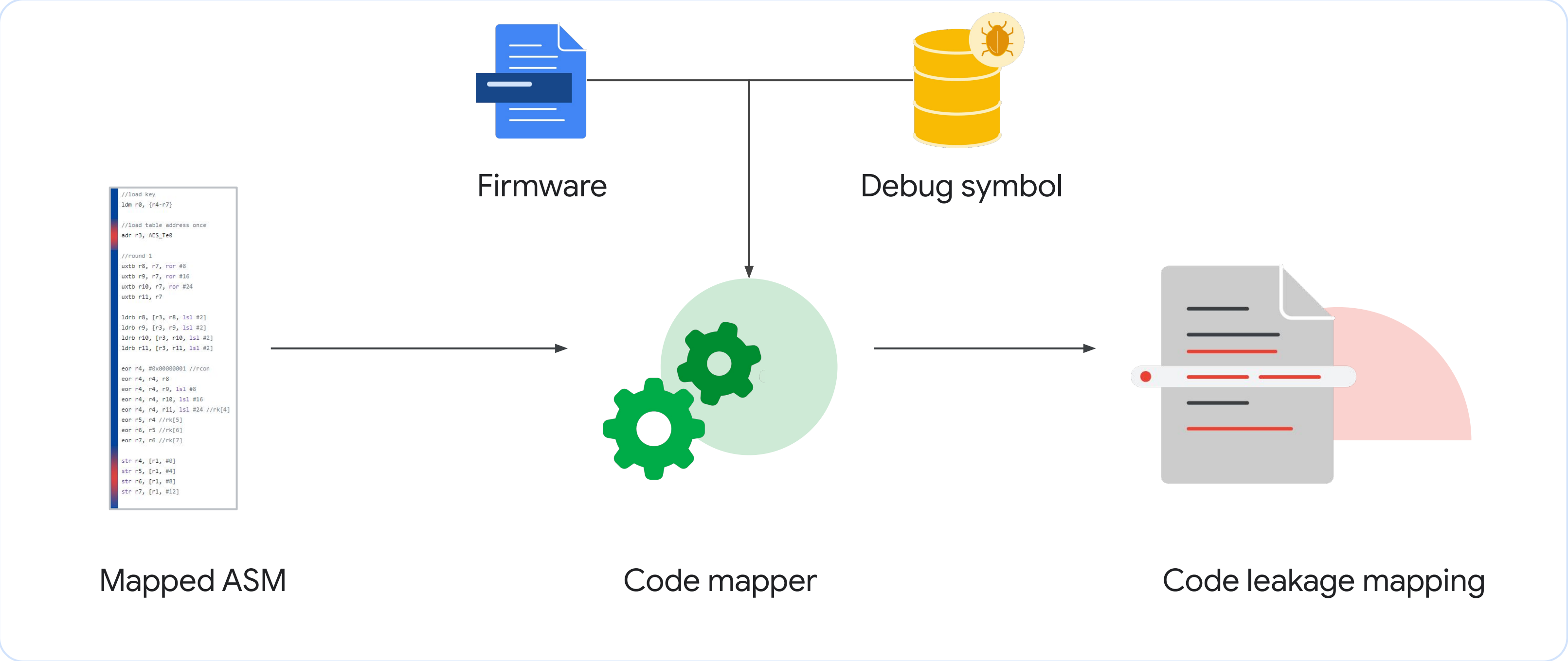
# SCALD custom explainability technique decreases accuracy the most and generate low noise leak map


# How do you go from the leakage map to code?

# From traces to CPU instructions



# From CPU instructions to code




# Theory looks great but how hard is it in practice?

# Requirements

## An explanation technique that have single point precision

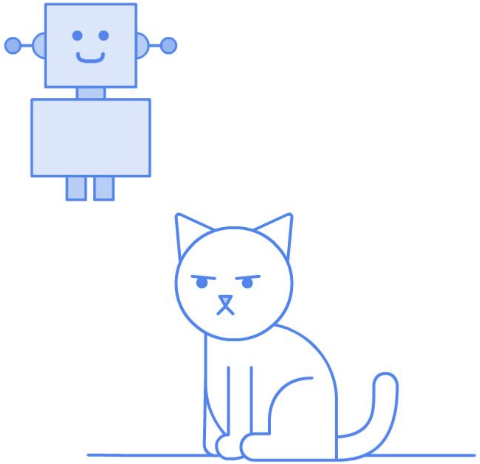
We need to isolate the exact few points of the traces that cause most of the leakage as some instruction only take one cycle or two (4 or 8 traces points)

## An emulator that have single cycle precision

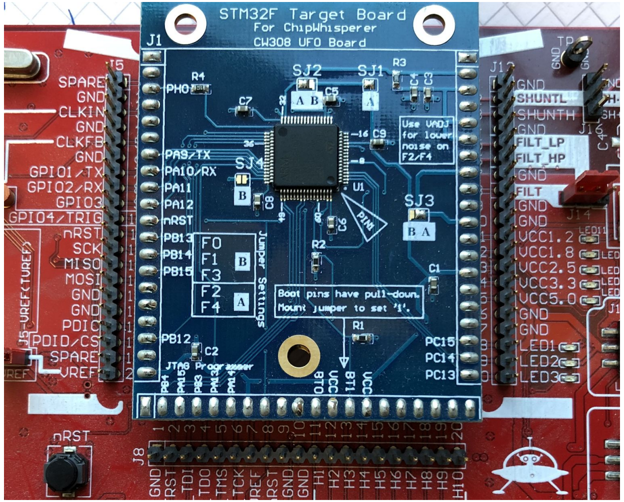
We need to map each instruction to its exact cycle to be able to map them to the trace. A single error and the entire analysis is wrong as all instruction will be shifted.

## A bit of computation

You need a 1M data point dataset, 16 models, 16 explanations, 1 full target execution and 1 mapping. With all our optimization this is requires a few days of computation that are parallelizable.



This level of explainability and emulation precision seems out-of reach




STM32F4 - TinyAES

Model targeting **sub_bytes_in** are expected to mostly **exploit leakage in the AddRoundKey() function**

```
stm32f415-tinyaes_sync
├──
│   ├── AES128_ECB_indp_crypto()
│   ├── AddRoundKey()
│   │   └── 0 - residual leakage (leak score:80)
│   ├── Cipher()
│   ├── ShiftRows()
│   ├── SubBytes()
│   ├── aes_indep_enc()
│   └── xtime()
├── aes.c
│   ├── AES128_ECB_indp_crypto()
│   ├── AddRoundKey()
│   │   ├── 207 - residual leakage (leak score:112)
│   │   └── 213 - Main leakage (leak score:240)
│   ├── Cipher()
│   │   ├── 276 - potential leakage (leak score:144)
│   │   ├── 277 - residual leakage (leak score:96)
│   │   ├── 278 - residual leakage (leak score:112)
│   │   ├── 279 - residual leakage (leak score:112)
│   │   ├── 280 - potential leakage (leak score:128)
│   │   ├── 371 - Secondary Leakage (leak score:176)
│   │   ├── 380 - residual leakage (leak score:96)
│   │   ├── 383 - residual leakage (leak score:112)
│   │   └── 393 - Secondary Leakage (leak score:176)
│   ├── ShiftRows()
│   │   └── 240 - residual leakage (leak score:96)
│   ├── SubBytes()
│   │   ├── 130 - potential leakage (leak score:128)
│   │   ├── 221 - residual leakage (leak score:80)
│   │   └── 227 - residual leakage (leak score:80)
│   └── xtime()
│       └── 265 - residual leakage (leak score:80)
├── simpleserial-aes.c
│   └── get_pt()
└── stm32f4_hal_lowlevel.c
    └── HAL_GPIO_WritePin()
```

Scald analysis result output

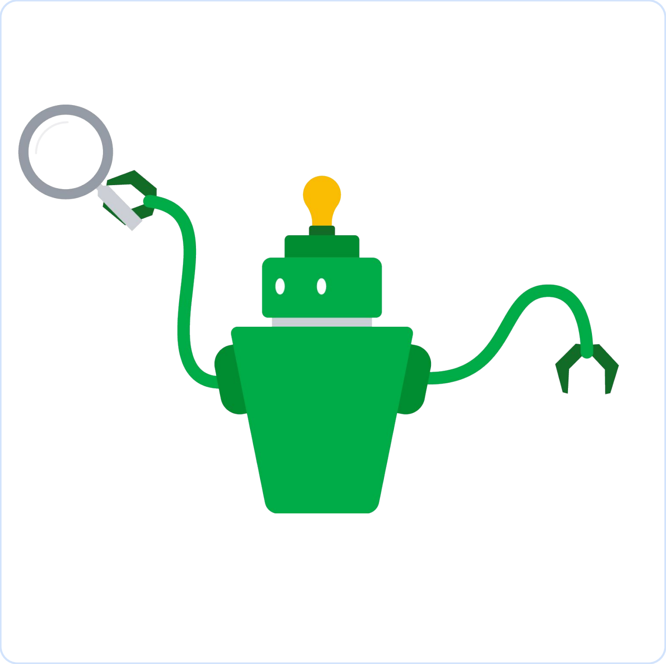
aes.c ×

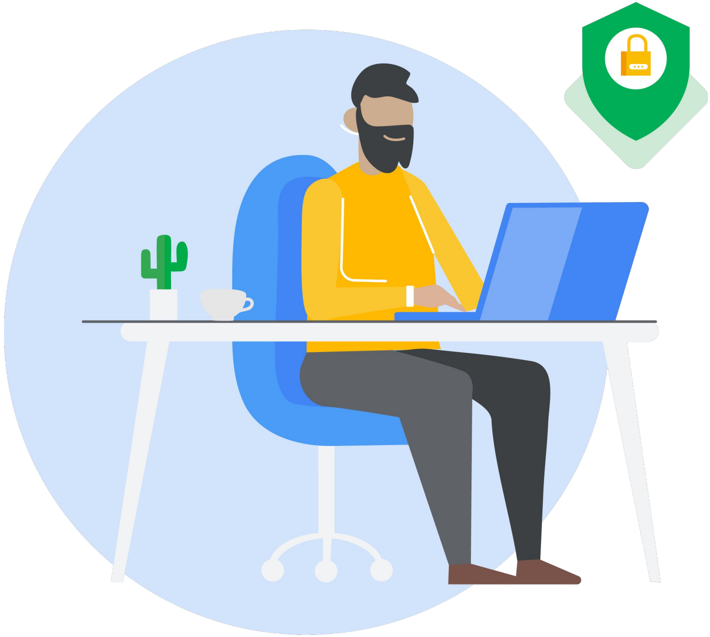
scald > firmwares > tinyaes_src > aes.c > AddRoundKey(uint8_t)

```
203
204 // This function adds the round key to state.
205 // The round key is added to the state by an XOR function.
206 static void AddRoundKey(uint8_t round)
207 {
208   uint8_t i,j;
209   for(i=0;i<4;++i)
210   {
211     for(j = 0; j < 4; ++j)
212     {
213       (*state)[i][j] ^= RoundKey[round * Nb * 4 + i * Nb + j];
214     }
215   }
216 }
217
```

TinyAES aes.c line 213 is **exactly** the sub_byte_in operation! SCALD perfectly identify the main source of leakage.

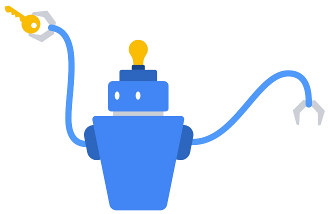
SCALD is able to automatically isolate the exact code vulnerable to a given SCAAML side-channel attack





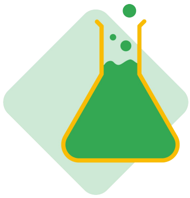
SCALD annotated code empowers developers to quickly figure out what to patch and focus on developing stronger crypto

# Takeaways



SCAAML attacks allows to perform SOTA SCA attacks automatically

SCALD use AI to find automatically leakage origin - reducing development cost



AI for side-channel is still a nascent field with a lot of exciting opportunities

Keep up with our research on deep-learning for side-channel attacks: https://elie.net/scaaml