# Closed Source Cryptography

Lukas Zobernig

25 March 2025

# Intro

- Open Source Cryptography Workshop?
- Let's talk about closed source cryptography then.
- Where? Virtually all proprietary systems.
- Why? Corporate red tape, paranoia, security through obscurity, DRM, IP, etc.
- What? Lots and lots of "homegrown" cryptography. Amazing.

# Reverse Engineering

How does one look at closed source cryptography?

- Primarily through reverse engineering — a spectrum of difficulty.
- JavaScript — easy.
- Compiled Java/.NET — easy.
- Compiled C/C++/Rust/etc. — trickier (decompilers can help).
- Synthesised IP (hardware) — difficult (need expensive lab setup).

Many mistakes reoccur (for and between vendors). Often we can guess what could go wrong once we built a good library of mistakes. We will see examples of that.

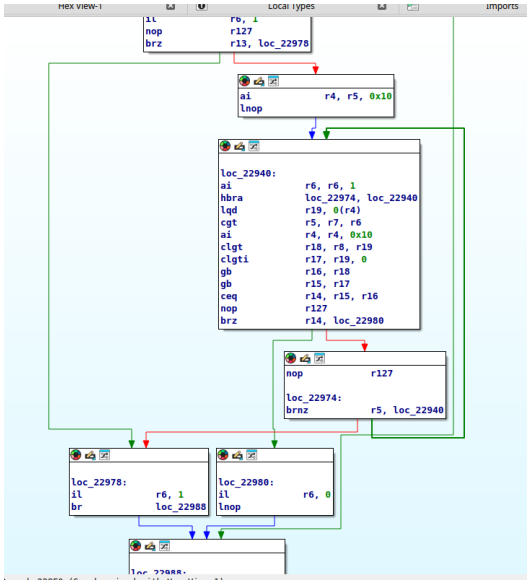
# Sony/IBM/Toshiba - Playstation 3

- First released in 2006 (JP, NA) and 2007 (EU).
- GPU by Nvidia, **CPU** co-developed by Sony, Toshiba, and IBM.
- Cell Broadband Engine Architecture (CELL/B.E.) CPU.
  - PPC main cores.
  - SPU accelerator/compute cores.
  - Secure Boot (boots from **isolated** SPU).
- Lots and lots of custom Sony DRM.
- Security through obscurity.
- "Custom" cryptography.

# Reverse Engineering - PS3

- IBM released public versions of the CELL/B.E. manuals.
- Sony provided a mechanism to run Linux on the PS3, we can look at drivers.
- The CPU was also used in super-computing applications, SPU instruction set publicly documented.
- Can develop tooling like disassemblers and emulators.

# Reverse Engineering - SPU



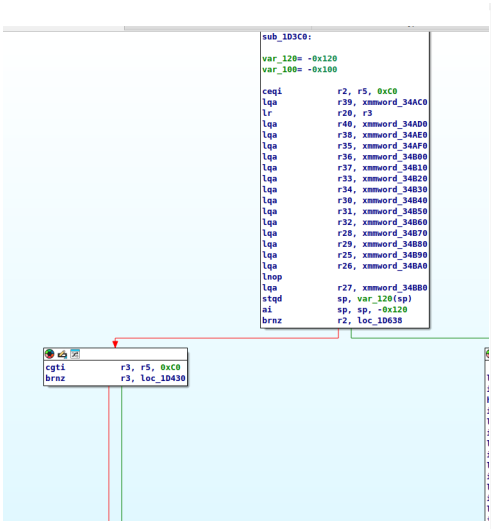

- Tricky since there was no decompiler for the SPU architecture.
- This was also long before Ghidra was published, so one would have had to develop their own.
- Thus, reverse engineering by hand.

# Reverse Engineering Cryptography Algorithms

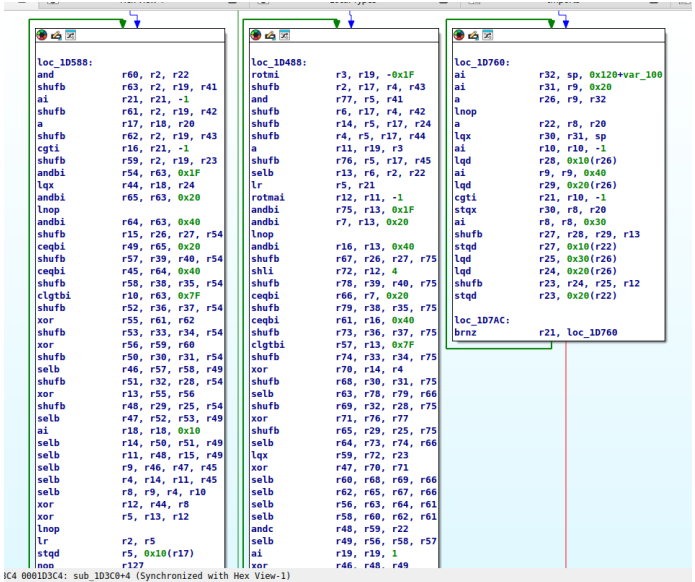
There are a few tricks.

- Most often standard algorithms implemented.
- AES, SHAx, HMAC, ECC, RSA, etc. all use characteristic constants.
- Ciphers and hashes have different algorithmic structure than signatures or key exchange.
- Often high-entropy data (keys, IVs, ) referenced through inputs.

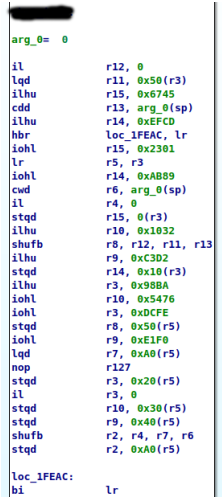
# Reverse Engineering - SPU

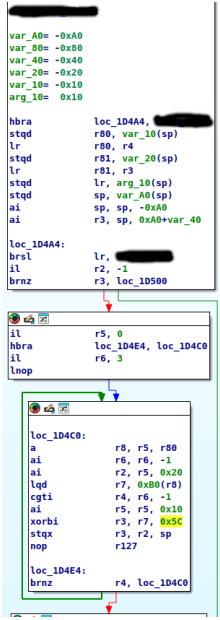



# Reverse Engineering - SPU



# Reverse Engineering - SPU



# Reverse Engineering - SPU

```
sha1_init:

arg_0=  0

il              r12, 0
lqd             r11, 0x50(r3)
ilhu            r15, 0x6745
cdd             r13, arg_0(sp)
ilhu            r14, 0xEFCD
hbr             loc_1FEAC, lr
iohl            r15, 0x2301
lr              r5, r3
iohl            r14, 0xAB89
cwd             r6, arg_0(sp)
il              r4, 0
stqd            r15, 0(r3)
ilhu            r10, 0x1032
shufb           r8, r12, r11, r13
ilhu            r9, 0xC3D2
stqd            r14, 0x10(r3)
ilhu            r3, 0x98BA
iohl            r10, 0x5476
iohl            r3, 0xDCFE
stqd            r8, 0x50(r5)
iohl            r9, 0xE1F0
lqd             r7, 0xA0(r5)
nop             r127
stqd            r3, 0x20(r5)
il              r3, 0
stqd            r10, 0x30(r5)
stqd            r9, 0x40(r5)
shufb           r2, r4, r7, r6
stqd            r2, 0xA0(r5)

loc_1FEAC:
bi              lr
```

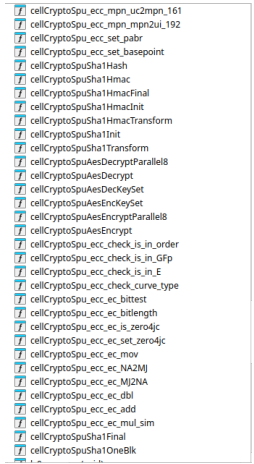
```
sha1_hmac_final:

var_A0= -0xA0
var_80= -0x80
var_40= -0x40
var_20= -0x20
var_10= -0x10
arg_10=  0x10

hbra            loc_1D4A4, sha1_final
stqd            r80, var_10(sp)
lr              r80, r4
stqd            r81, var_20(sp)
lr              r81, r3
stqd            lr, arg_10(sp)
stqd            sp, var_A0(sp)
ai              sp, sp, -0xA0
ai              r3, sp, 0xA0+var_40

loc_1D4A4:
brsl            lr, sha1_final
il              r2, -1
brnz            r3, loc_1D500
```

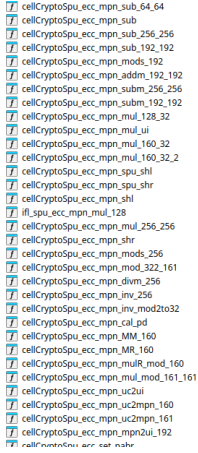
```
il              r5, 0
hbra            loc_1D4E4, loc_1D4C0
il              r6, 3
lnop
```

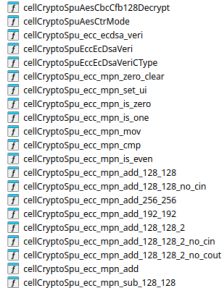
```
loc_1D4C0:
a               r8, r5, r80
ai              r6, r6, -1
ai              r2, r5, 0x20
lqd             r7, 0xB0(r8)
cgti            r4, r6, -1
ai              r5, r5, 0x10
xorbi           r3, r7, 0x5C
stqx            r3, r2, sp
nop             r127

loc_1D4E4:
brnz            r4, loc_1D4C0
```

# Reverse Engineering - SPU

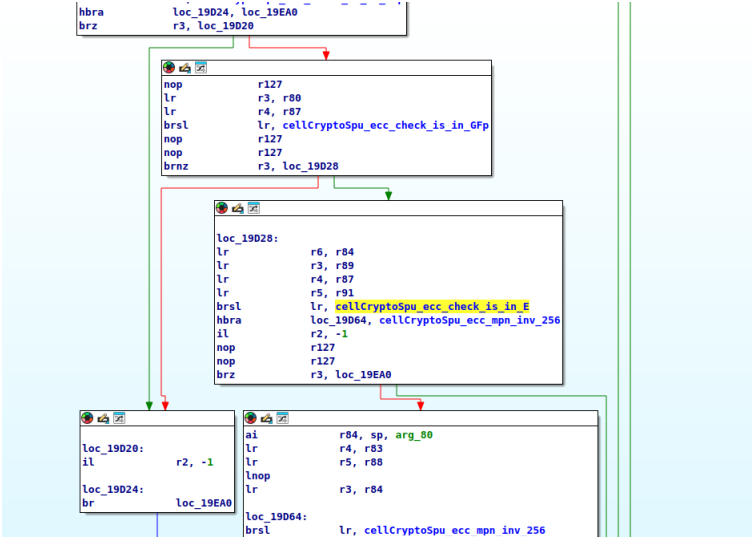
And of course sometimes you get lucky and developers forget to strip symbols.



# Reverse Engineering - SPU



# PS3 DRM - ECDSA Signatures

ECDSA signing in a nutshell:

- Given elliptic curve $E$, point $G \in E$ of order $n$, signer private key $k \in [1, n-1]$.
- For signing a message $m$:
  - Sample **random** $t \in [1, n-1]$,
  - compute $(x, y) = tG$,
  - compute $r = x \mod n$
  - compute $s = t^{-1}(H(m) + rk) \mod n$,
  - output $(r, s)$ as the signature.

What did Sony do? Keep $t$ **fixed**.

- For different $m, m'$ we then have signatures $(r, s)$ and $(r, s')$, respectively.
- Solve for $t = (s - s')^{-1}(H(m) - H(m')) \mod n$.
- Solve for **private key** $k = r^{-1}(st - H(m)) \mod n$.

# Nintendo - DS/DSi/3DS

- DS first released in 2004, DSi in 2008, and 3DS in 2011.
- Mostly custom designed hardware by Nintendo Research & Engineering.
- Lots and lots of custom Nintendo DRM.
- Security through obscurity.
- “Custom” cryptography.

# Nintendo - DS/DSi/3DS

- These use a hardware AES engine for encryption/decryption of sensitive data.
- Keys are written to **keyslots** by secure boot stages, keyslots are write-only.
- Some keyslots are flushed by the hardware AES engine whenever **any** part of the key is written to.
  - Encrypt some plaintext $M$, and record corresponding ciphertext $C^*$.
  - Overwrite one byte of the key with $b \in [0, 255]$, encrypt same $M$ to obtain $C_b$.
  - Compare $C^*$ with $C_b$, if they match we discovered one byte of the key.
  - Otherwise, chose different $b \in [0, 255]$, repeat until all bytes known.

# Nintendo - DS/DSi/3DS

- Other keyslots are derived from a hardware key scrambler $K_i = F(X_i, Y_i)$ for inputs slots $X_i$, $Y_i$, and some "secret" function $F$.
- Encrypting a chosen plaintext for carefully chosen $X, Y$ pairs recovers $F(X, Y) = (((X \lll 2) \oplus Y) + C) \lll 87$, for some constant $C$.
- A SAT solver can find a solution to such a system quickly.
- This was also used to recover secret $X_i$ values set by the bootrom, some keys $K_i$ did leak from another device (used for interoperability).

# Nintendo/Nvidia - Switch

- First released in 2017.
- SoC (CPU & GPU) by Nvidia.
  - ARM main CPU cores.
  - Secure Boot from BPMP (Boot and Power Management Processor).
  - ARM TrustZone runs tiny “hypervisor” mostly responsible for cryptography.
- Lots and lots of custom Nintendo DRM.
- Security through obscurity.
- “Custom” cryptography.

# Nintendo/Nvidia - Switch

- SoC contains hardware AES engine for encryption/decryption.
- Keys are written to keyslots by secure boot stages, keyslots are write-only.
- Keyslots are flushed by the hardware AES engine whenever any word (32bit) of the key is written.

We have already seen this problem earlier. Partial key overwrites is a flaw many hardware crypto engines have.

# Nvidia - TSEC

- Nvidia makes graphics cards.
- These cards need to talk HDCP (i.e. HDMI DRM).
- TSEC is a security co-processor handling **cryptography**.
- It supports secure boot for programs that can access sensitive hardware keys.
- To load such a program, one needs to write a signature to a hardware register before jumping to the program.
- The signature is essentially a Davies-Meyer hash of the input program.
- The signature is computed into another hardware register, but **not** cleared on verification failure.

# Outro

- All of the points and bugs mentioned can be found documented in various places on the internet.
- We have glossed over many details, please go read up on them.
- Try to keep cryptography open and documented, hackers will figure it out in any case.
- Questions?