HACKABLE SECURITY MODULES:

REVERSING AND EXPLOITING A FIPS 140-2 LVL 3 HSM FIRMWARE

FOTIS LOUKOS

<FOTISL@SSL.COM>

SSL CORP

RECON BRUSSELS 2016

WHO AM I?

Fotis Loukos

- Go by the nick fotisl
- Work at SSL.com, a globally trusted Certification Authority focusing on TLS/SSL and Code Signing
- Hold a PhD in Computer Science from the Aristotle University of Thessaloniki
- My work focuses on Public Key Infrastructures, Certification Authorities security, vulnerability research and reverse engineering

OUTLINE

- What is an HSM and HSM security requirements
- The Utimaco HSM and its Firmware
- The TMS320C64x DSP
- Adding a new architecture to Capstone
- Searching for vulnerabilities

WHAT IS AN HSM?

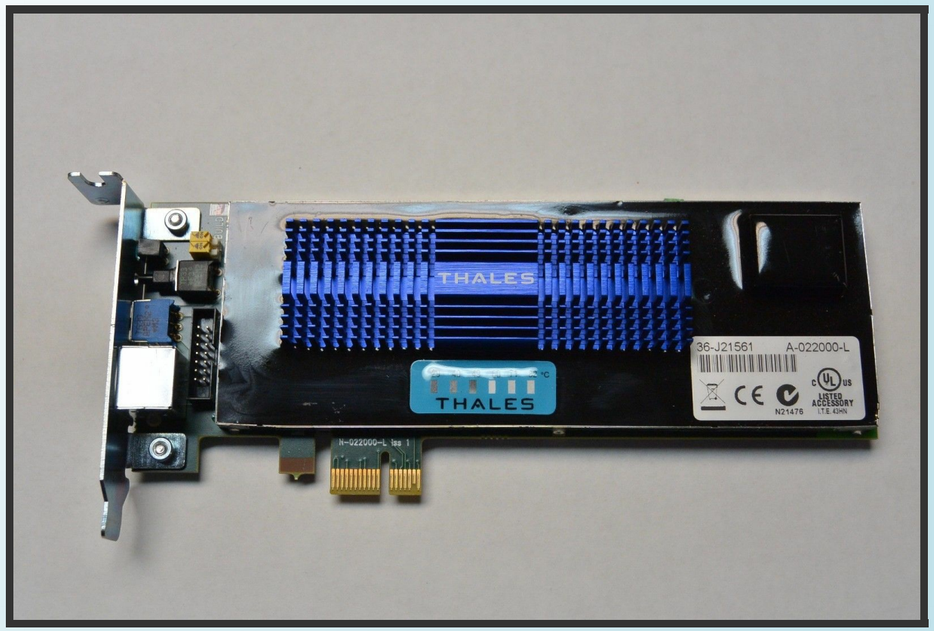
USAGE

- Securely store cryptographic keys
- Manage (generate etc) cryptographic keys
- Perform cryptographic operations (encrypt, decrypt, sign, verify, wrap, unwrap, etc)

PHYSICAL FORM

Network device

PCI card



BIG PLAYERS

- Gemalto
  - Bought Safenet, another big player
  - Famous for the Luna series of products
  - Sample value for the Safenet Network HSM 7000 Model: $29,500.00

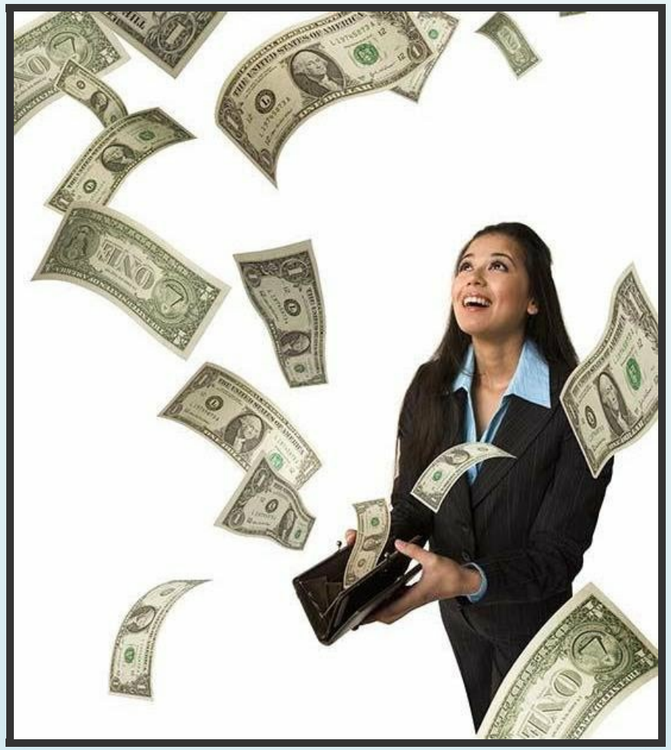
BIG PLAYERS

- Thales
  - Bought nCipher, the second big player
  - Famous for the nShield series of products
  - Sample value for the USB nShield Edge HSM with ECC activation: $9,500.00

BIG PLAYERS

- Utimaco
  - Primary target
  - Famous for the Cryptoserver series of products
  - Part of the EJBCA ready-to-deploy PKI solution by PrimeKey
  - Sample value for the SecurityServer Se52 LAN V4: 15,000.00€

A BUSINESS WITH A LOT OF MONEY



WHERE ARE THEY USED?

PKI

All Certification Authorities (CAs) are required to use HSMs. From the CA/B Forum Baseline requirements:

The CA SHALL protect its Private Key in a system or device that has been validated as meeting at least FIPS 140 level 3 or an appropriate Common Criteria Protection Profile or Security Target, EAL 4 (or higher), which includes requirements to protect the Private Key and other assets against known threats.

DNSSEC

Root Zone operators store keys inside HSMs. From the DNSSEC Practice Statement for the Root Zone KSK Operator:

For RZ KSK generation and RZ KSK private component operations and storage, ICANN uses hardware security modules that are validated at FIPS 140-2 level 4 overall.

ELECTRONIC TRANSACTIONS

PCI DSS in some cases requires HSMs to secure cardholder data throughout the transaction process.

The PCI HSM Security Requirements document describes the minimum security requirements for compliant HSMs.

ONE DOES NOT SIMPLY

Implement serious crypto

without an HSM

SECURITY REQUIREMENTS

GENERAL INFORMATION

- As HSMs are used in critical applications and infrastructures, different standards have been proposed to evaluate their security.
- Major standards:
  - FIPS 140-2
  - Common Criteria

FIPS 140-2

- Federal Information Processing Standards Publication 140-2
- ...or otherwise, the US stuff
- Issued at 2001 by the National Institute of Standards and Technology, updated at 2002
- Most widely used standard
- Supersedes FIPS 140-1
- Will be superseded by FIPS 140-3... when it gets released!

FIPS 140-2

Defines 4 different security levels:

- Level 1: Lowest security level. At least one approved algorithm must be implemented and there are no physical security controls.
- Level 2: Level 1 plus physical security controls. Cryptographic keys and Critical Security Parameters (CSPs) are protected with tamper-evident coatings or seals

FIPS 140-2

Defines 4 different security levels:

- Level 3: Level 2 with harder physical security controls. Keys and CSPs are deleted if potential breach is detected.
- Level 4: Level 3 with more strict physical security controls to make the HSM usable in physically unprotected environments.

FIPS 140-2

FIPS 140-2 validation happens at the Cryptographic Module Testing Laboratories which are accredited by the National Voluntary Laboratory Accreditation Program as part of the Cryptographic Module Validation Program.

Currently 22 laboratories have been accredited to perform FIPS 140-2 validation.

COMMON CRITERIA

- ISO Standard
- The EU stuff
- Has Evaluation Assurance Levels (EALs) similar to FIPS 140-2 levels
- In addition there are Protection Profiles (PP), Security Targets (ST), etc

COMMON CRITERIA

Interesting fact:

- In the PKI world the standard is EAL4+
- EAL4 is Methodically Designed, Tested and Reviewed
- Best security practices should be used during design and test... Lets see how it goes!

THE UTIMACO HSM

VARIOUS MODELS OF THE CRYPTOSERVER

- SecurityServer Se Gen2
- SecurityServer CSe
- SecurityServer Se (End of line)
- TimestampServer

SPECIFICATIONS

- FIPS 140-2 Level 3 certification (some have higher levels in specific areas, such as physical security)
- Available as both PCIe cards and Network Attached Appliances
- Support for RSA, DSA, ECDSA (NIST and Brainpool curves), DH, ECDH, AES, DES, 3DES, SHA1, SHA2, SHA3, RIPEMD, etc
- Depending on your license you get more transactions per second

THE HARDWARE

- Network HSMs are Linux boxes with a PCIe HSM.
- There is a physical protection layer and a battery that helps erase the contents of the memory in case of breach when the HSM is powered off
- Sensors can detect changes in temperature, voltage, power supply in general and tampering of the protecting foil

THE HARDWARE

Every PCIe HSM contains:

- A Texas Instruments TMS320C64x DSP that performs all cryptographic operations
- A hardware True Random Number Generator (TRNG) and a Deterministic Random Bit Generator (DRBG)
- A Key-RAM which contains the Device Key and in case of attack gets deleted

DEVICE KEY

- A single key created when the HSM is brought into operation
- Cannot be extracted, exported, imported, or in any way manipulated
- Encrypts all other cryptographic keys and critical security parameters

MASTER BACKUP KEYS

- 256 bit AES or 128 bit (?) 3DES Key
- Used to encrypt backups of cryptographic keys
- Can be split into many shares using an n out of m scheme (XOR for 2 out of 2, Shamir's Secret Sharing otherwise)
- Keys inside the HSM are not encrypted using the MBK but using the Device Key

TOOLS

- csadm: Command line tool to manage both PCIe and Network HSMs
- p11tool2: Command line tool to use the PKCS#11 API
- cxitool: Command line tool to use the CXI API
- cat / p11cat: Java versions of the above

SIMULATOR

- A simulator exists which runs its own firmware too
- As we'll see later on, this will be very useful!

USERS

Every user has:

- An authentication method: RSA signature (in soft form or in smartcards), ECDSA signature (in soft form), HMAC password
- Permissions: The value 0, 1 or 2 at 8 different permission groups
- Flags, attributes, etc

PERMISSIONS

- To complete a task you need all logged in users to add up to a certain permission level at a certain group.
- For example, to add a user you need all logged in users' permissions to add up to 2 at group 7 (no more, no less).

COMMUNICATION PROTOCOL

- Custom communication protocol with the PCIe HSM
- Network HSMs listen at both a TCP and a UDP port and send everything received to the internal PCIe card
- No public specification available
- AES256 encrypted communication with unique session keys and MAC

THE UTIMACO FIRMWARE

FORMAT

- A single blob in a custom .mpkg format
- Both FIPS and non-FIPS versions
- After reversing much of the format I found out that csadm has an option to unpack mpkg files
- Lesson learned: RTFM!
- After unpacking you get a number of .mtc files

SAMPLE FIRMWARE

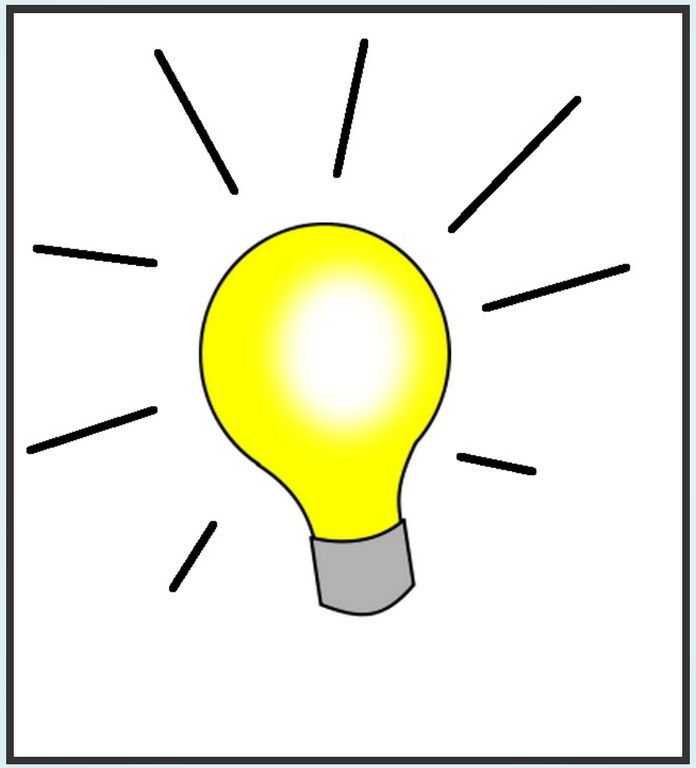
SecurityServer-Se-Series-4.01.0.5.mpkg →

adm_3.0.18.1_c50.mtc
aes_1.3.7.0_c50.mtc
asn1_1.0.3.4_c50.mtc
bcm_1.0.2.0_c50.mtc
cmds_3.5.1.6_c50.mtc
cxi_2.1.11.3_c50.mtc
db_1.3.1.1_c50.mtc
dsa_1.2.2.1_c50.mtc
eca_1.1.7.6_c50.mtc
ecdsa_1.1.8.7_c50.mtc
hash_1.0.10.1_c50.mtc
hce_2.2.2.1_c50.mtc
lna_1.2.3.4_c50.mtc
mbk_2.2.4.4_c50.mtc
ntp_1.2.0.7_c50.mtc
pp_1.2.5.1_c50.mtc
sc_1.2.0.3_c50.mtc
smos_3.3.4.2_c86.mtc
util_3.0.3.0_c50.mtc
vdes_1.0.9.1_c50.mtc
vrsa_1.3.0.6_c50.mtc

MTC FILES

- Individual firmware modules
- Each one of them seems to be associated with a specific function:
  - adm_3.0.18.1_c50.mtc → Administration functions
  - aes_1.3.7.0_c50.mtc → AES implementation
  - asn1_1.0.3.4_c50.mtc → ASN.1 encoding/decoding (this is the X.509 world!)

MTC FILES



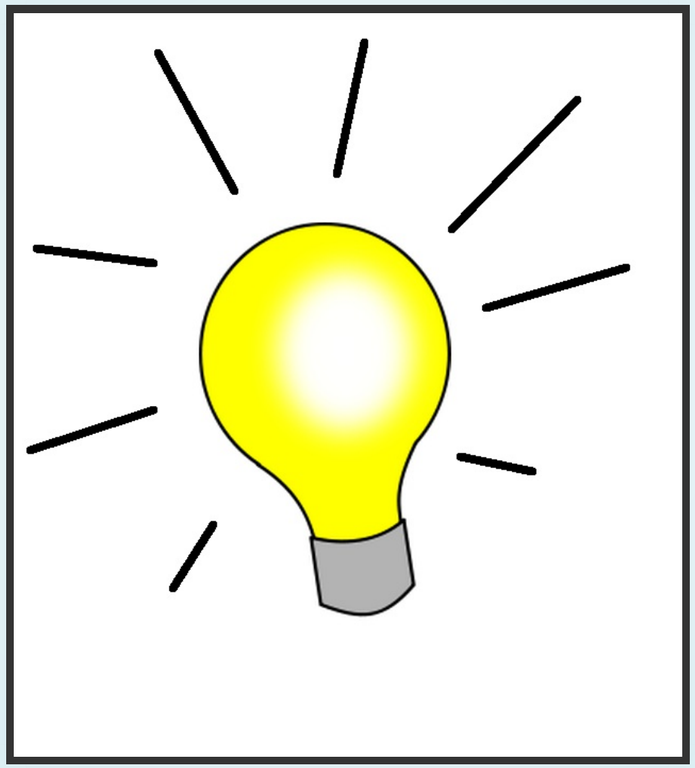
Let's use binwalk!

MTC FILES

```
$ binwalk adm_3.0.18.1_c50.mtc

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
53925         0xD2A5          Copyright string: "Copyright (c) 1996-2014 Texas Instruments Incorporated"
54123         0xD36B          Copyright string: "Copyright (c) 1996-2014 Texas Instruments Incorporated"
55658         0xD96A          Unix path: /tmp/TI_MKLIBqpFZmO/OBJ/memset.asm:$C$L2:1:1398463752
55816         0xDA08          Copyright string: "Copyright (c) 1996-2014 Texas Instruments Incorporated"
55985         0xDAB1          Copyright string: "Copyright (c) 1996-2014 Texas Instruments Incorporated"
```

Four copyright strings inside and one unix path →
WTF guys, use a standard format!

MTC FILES



Hexdump to the rescue!

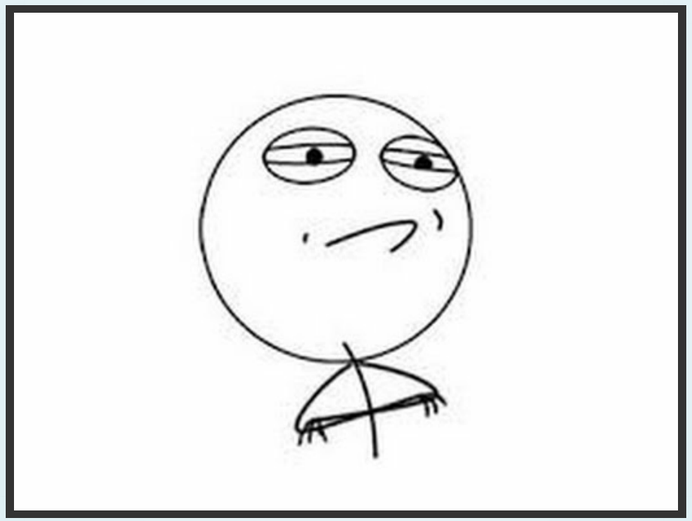
MTC FILES

```
$ hexdump -C adm_3.0.18.1_c50.mtc
00000000  4d 54 43 48 00 01 2d e0  01 00 00 00 01 02 01 00  |MTCH..-.........|
00000010  00 00 00 18 00 00 00 00  4d 4d 43 48 00 01 2d c8  |........MMCH..-.|
00000020  01 01 00 00 a3 07 55 2e  d3 44 d5 ba 41 44 4d 00  |......U..D..ADM.|
00000030  00 00 00 00 00 00 00 00  00 00 00 87 03 00 12 01  |................|
00000040  41 64 6d 69 6e 69 73 74  72 61 74 69 6f 6e 20 4d  |Administration M|
00000050  6f 64 75 6c 65 00 00 00  00 00 00 00 00 00 00 00  |odule...........|
....
00000110  00 00 2e 74 65 78 74 00  00 00 80 0f 00 00 80 0f  |...text.........|
00000120  00 00 e0 bf 00 00 8e 11  00 00 5e e2 00 00 00 00  |..........^.....|
00000130  00 00 b6 03 00 00 00 00  00 00 20 05 00 00 00 00  |.......... .....|
00000140  00 00 2e 64 61 74 61 00  00 00 60 cf 00 00 60 cf  |...data...`...`.|
00000150  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000160  00 00 00 00 00 00 00 00  00 00 80 00 00 00 00 00  |................|
```

- A header at the begining with a description!
- ".text" and ".data" seem to be names of sections
- Maybe some modified standard format???

MTC FILES

Challenge accepted!



Let's reverse the administration tool that extracted the files!

MTC FILES

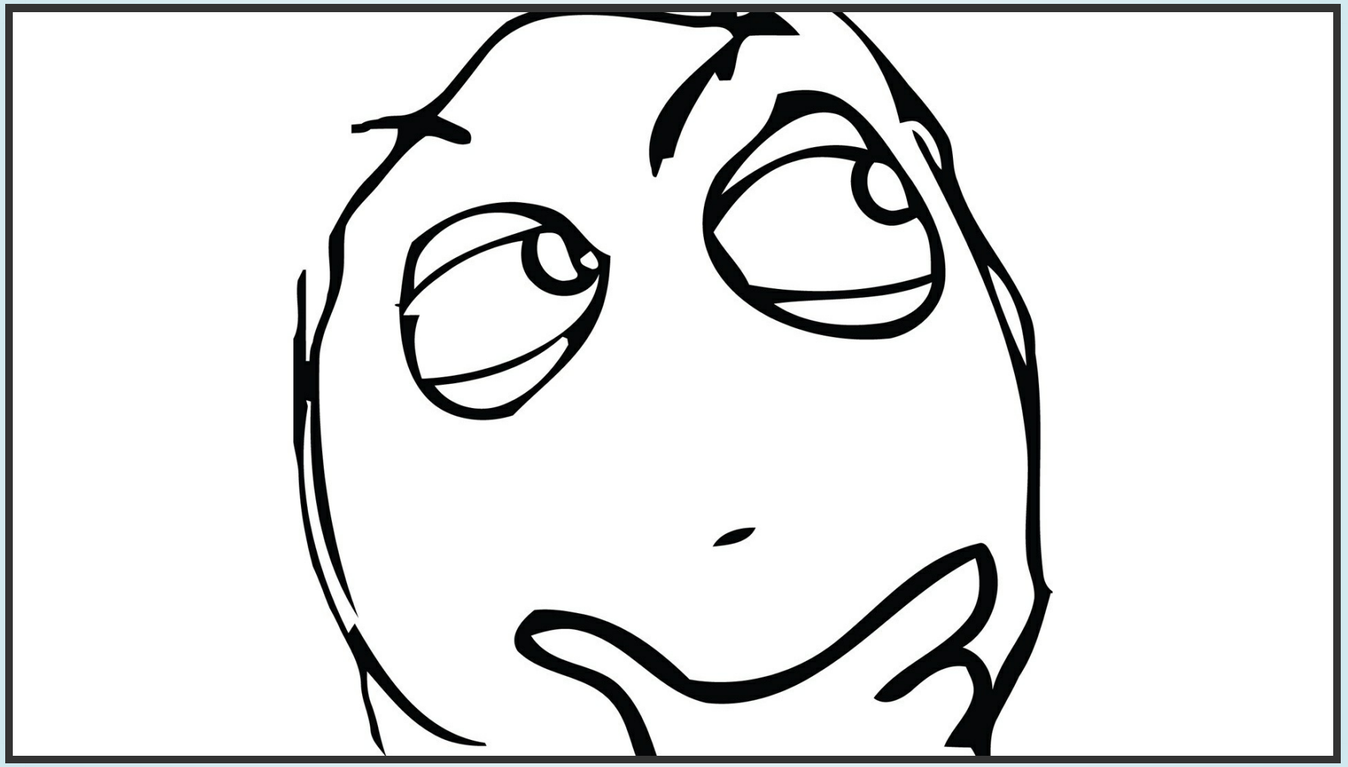
Fast forward many hours...

```
$ strings csadm | grep COFF
CS2-COFF
COFF section
```

MTC FILES

- COFF files!
- But why didn't binwalk recognize this?
- The answer is simple: someone thought it would be nice to change the standard COFF format to add their own fields over there!

MTC FILES



Now what???

MTC FILES

- We first identify the length of the section header by calculating differences between names
- We then fit the records we found to the standard section header
- Next step is finding out how much the COFF file header was messed up (you thought they would mess with this one?)
- And we fit this header to the standard COFF file header

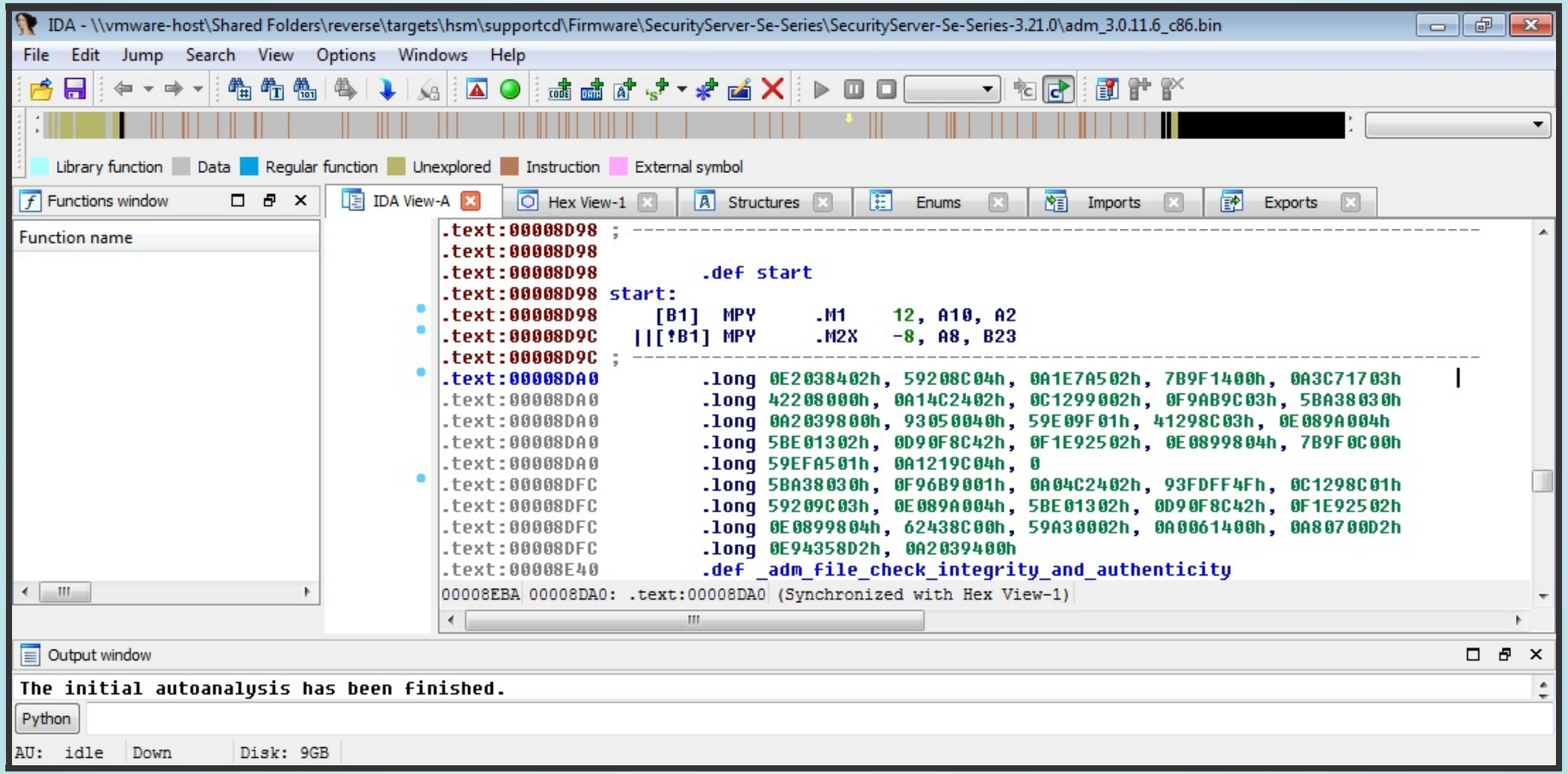
MTC FILES

We can now extract the COFF file!

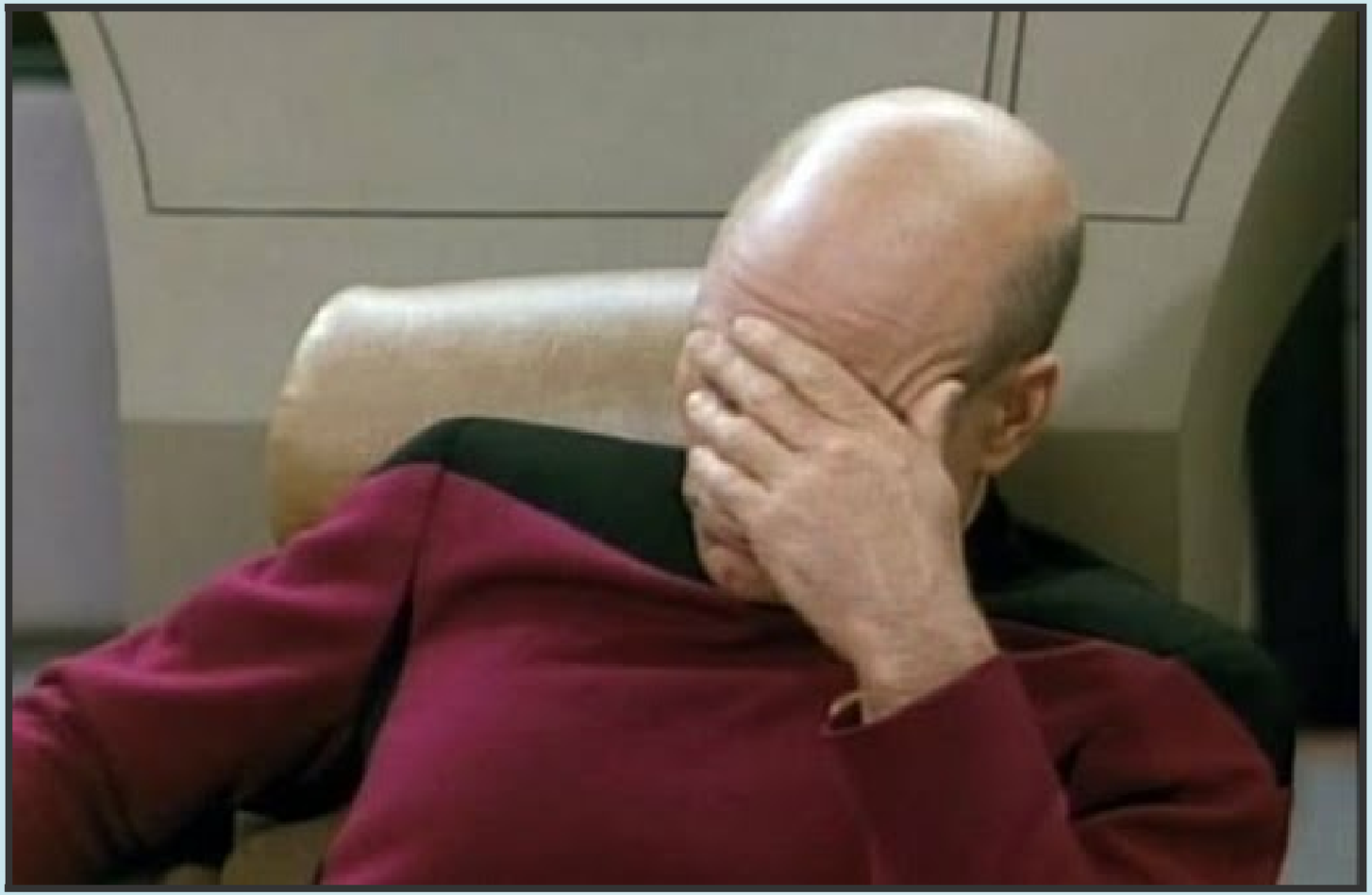
Now what?

MTC FILES

MTC FILES


IDA - \\vmware-host\Shared Folders\reverse\targets\hsm\supportcd\Firmware\SecurityServer-Se-Series\SecurityServer-Se-Series-3.21.0\adm_3.0.11.6_c86.bin
File Edit Jump Search View Options Windows Help
Library function Data Regular function Unexplored Instruction External symbol
Functions window
Function name
IDA View-A
Hex View-1
Structures
Enums
Imports
Exports
.text:00008D98 ; ---
.text:00008D98
.text:00008D98 .def start
.text:00008D98 start:
.text:00008D98 [B1] MPY .M1 12, A10, A2
.text:00008D9C ||[!B1] MPY .M2X -8, A8, B23
.text:00008D9C ; ---
.text:00008DA0 .long 0E2038402h, 59208C04h, 0A1E7A502h, 7B9F1400h, 0A3C71703h
.text:00008DA0 .long 42208000h, 0A14C2402h, 0C1299002h, 0F9AB9C03h, 5BA38030h
.text:00008DA0 .long 0A2039800h, 93050040h, 59E09F01h, 41298C03h, 0E089A004h
.text:00008DA0 .long 5BE01302h, 0D90F8C42h, 0F1E92502h, 0E0899804h, 7B9F0C00h
.text:00008DA0 .long 59EFA501h, 0A1219C04h, 0
.text:00008DFC .long 5BA38030h, 0F96B9001h, 0A04C2402h, 93FDFF4Fh, 0C1298C01h
.text:00008DFC .long 59209C03h, 0E089A004h, 5BE01302h, 0D90F8C42h, 0F1E92502h
.text:00008DFC .long 0E0899804h, 62438C00h, 59A30002h, 0A0061400h, 0A80700D2h
.text:00008DFC .long 0E94358D2h, 0A2039400h
.text:00008E40 .def _adm_file_check_integrity_and_authenticity
00008EBA 00008DA0: .text:00008DA0 (Synchronized with Hex View-1)
Output window
The initial autoanalysis has been finished.
Python
AU: idle Down Disk: 9GB

MTC FILES



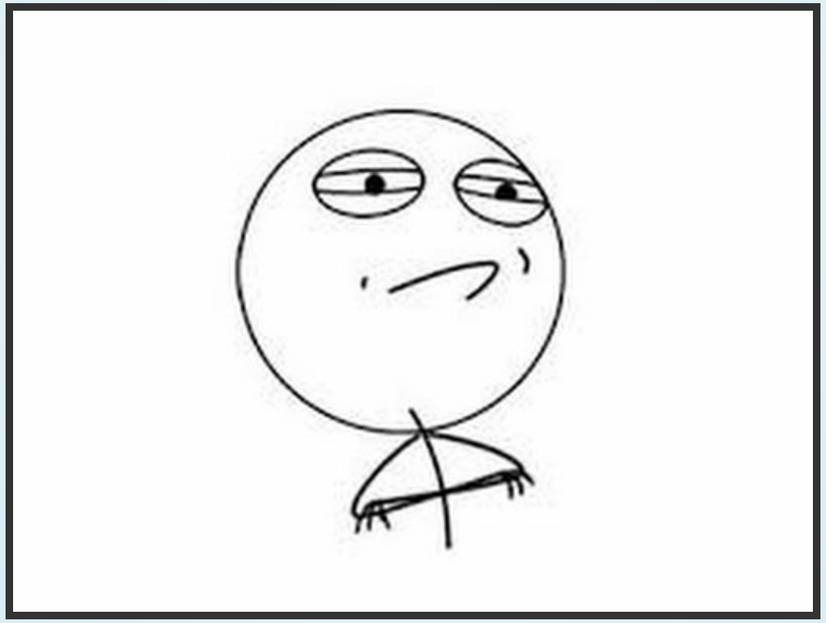
MTC FILES

(Insert multiple slides with Picard facepalms and radare2, binutils, etc)

(Ok, actually TI has a disassembler but it's unusable)

MTC FILES

Challenge accepted!



Let's write our own disassembler!

THE TMS320C64X DSP

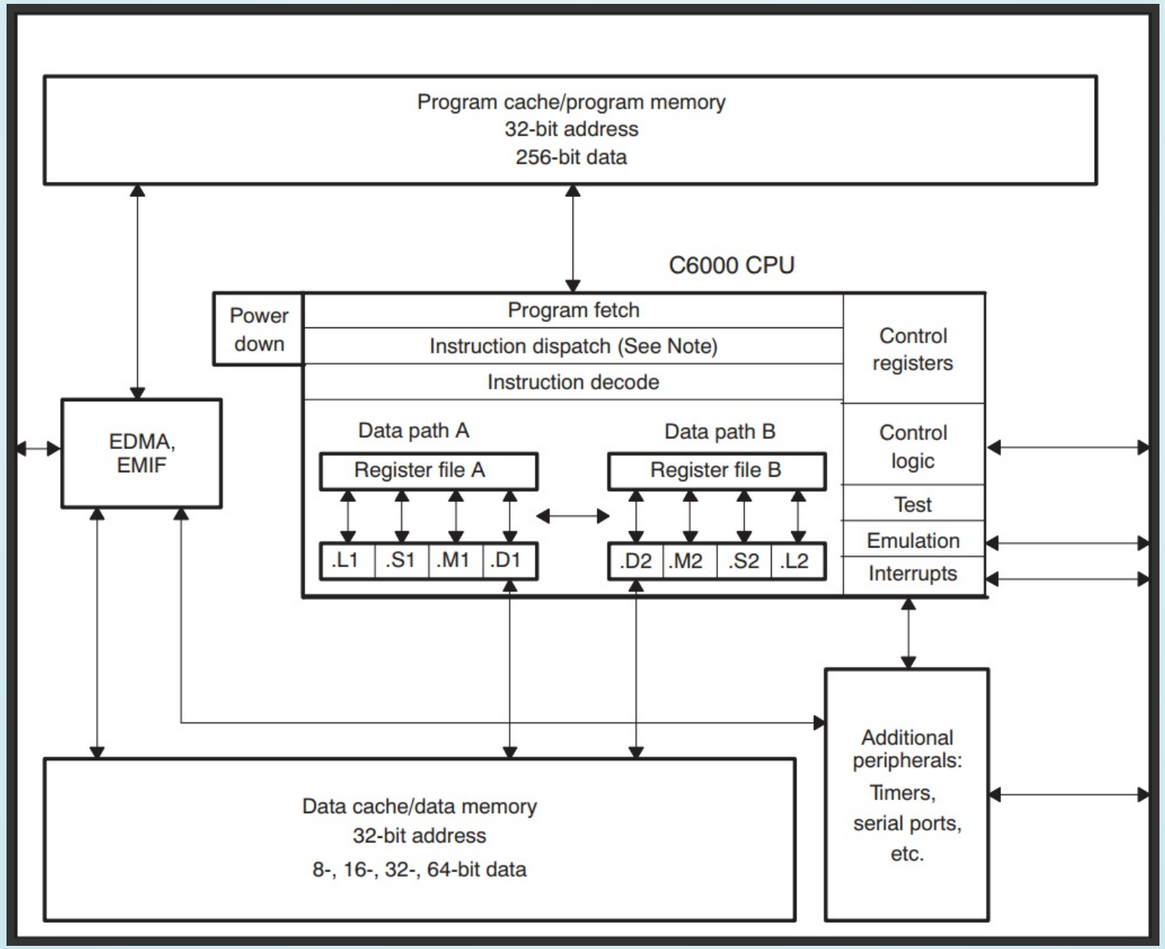
Steps to write our own disassembler:

- Study the architecture of the DSP
- Study the memory organization
- Study the ABI
- Find potential frameworks we can use
- Write the actual disassembler

THE ARCHITECTURE

- An 'exotic' architecture compared to x86, x86_64, ARM, MIPS, etc
- 16 bit Very Long Instruction Word (VLIW) DSP
- 2x4 Functional Units (.L1, .S1, .M1, .D1, .L2, .S2, .M2, .D2), each one with it's own assembly instructions
- Ability to execute multiple instructions in parallel at different Functional Units
- 2 Register Files (A and B)
- Crosspath to transfer data between A and B

BLOCK DIAGRAM



DATA PATHS

Each data path contains:

- 1 ALU (the .L functional unit)
- 1 Shifter (the .S functional unit)
- 1 Multiplier (the .M functional unit)
- 1 Adder/Subtractor, also used for address generation (the .D functional unit)
- 1 Register File with 32 32-bit registers

There is also one cross path for transfers between data path A and B

PARALLEL EXECUTION

- 8 execution units can execute up to 8 commands in parallel
- At any single time only one command can use the cross path
- Instructions are fetched in fetch packets (FP) of 8 words
- You cannot execute instructions in two different FPs in parallel

REGISTERS

- General purpose registers A0-A31 and B0-B31
- Instructions operate on 8, 16, 32 or 40 bit data
- For 40 bit operations one register from the even register file is used, together with the relevant register from the odd register file, e.g. A11:A10 or B25:B24
- There are instructions that operate on packed data (e.g. 4 8-bit quantities at a single register)
- 64 bit loads and stores can be performed in a single operation

REGISTERS

- A0-A2,B0-B2 are also conditional registers
- All instructions can take a conditional prefix, those instructions will be executed only if the corresponding register is zero or non-zero
- Control Register File: Control registers such as:
  - PCE1: Program Counter, E1 phase
  - ICR: Interrupt Clear Register
  - CSR: Control Status Register

INSTRUCTION FORMAT

Sample instructions:

```
        SHR .S1   A1, 10, A2

        MV  .S1X B0, A0

        AND .D1   A0, A1, A2
||      AND .D2   B0, B1, B2

   [ A0] ADD .L1   A1, A2, A3
|| [!A0] ADD .L2   B1, B2, B3
```

DELAY SLOTS

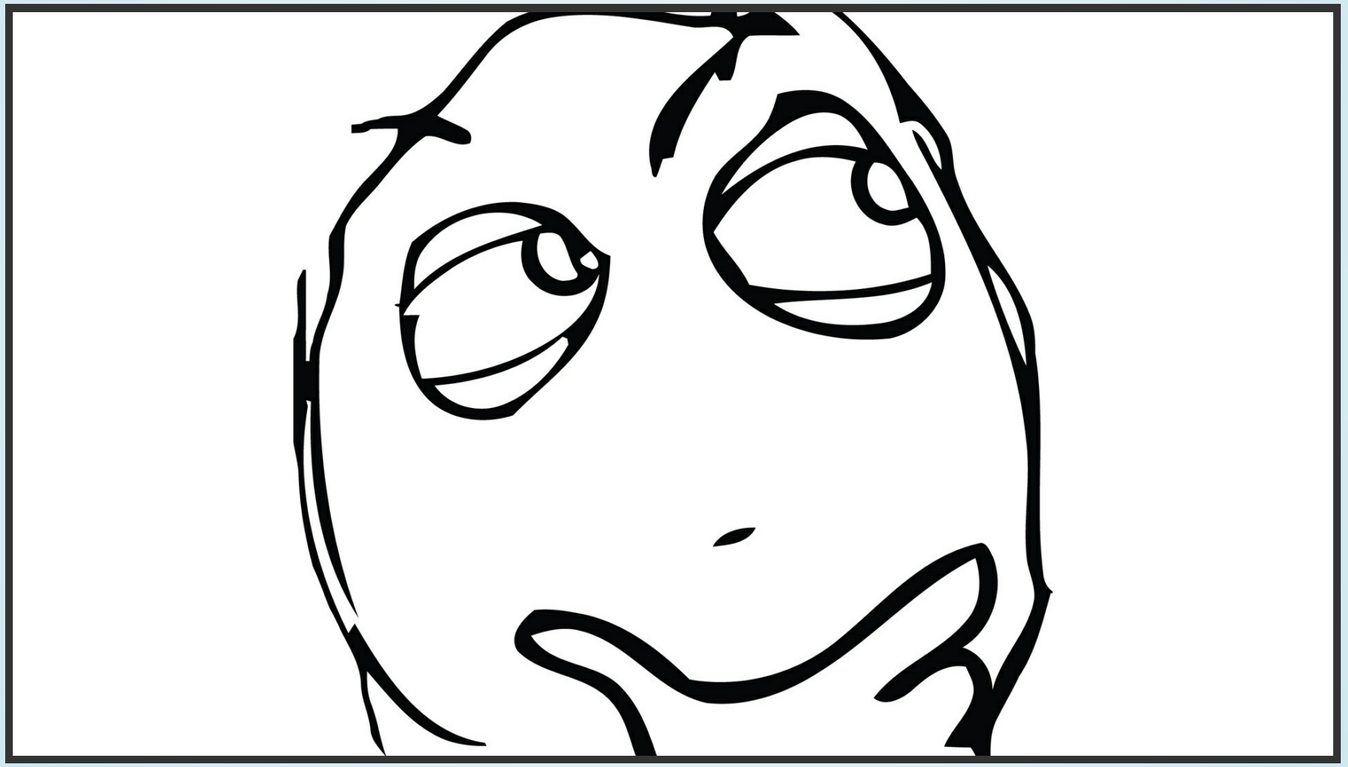
- Most instructions are executed at a single cycle
- ...but not all of them!
- Some multi-cycle instructions read the source operands at one cycle and write the result at a different one
- Branches read the jump target at cycle i and jump to it at cycle i+5
- In the mean time, if the jump target was a register, its value may have changed

DELAY SLOTS

Instruction Type	Delay Slots	Functional Unit Latency	Read Cycles (1)	Write Cycles (1)	Branch Taken (1)
NOP (no operation)	0	1			
Store	0	1	i	i	
Single cycle	0	1	i	i	
Two cycle	1	1	i	i + 1	
Multiply (16 × 16)	1	1	i	i + 1	
Four cycle	3	1	i	i + 3	
Load	4	1	i	i, i + 4 (2)	
Branch	5	1	i (3)		i + 5

(1) Cycle i is in the E1 pipeline phase.
(2) For loads, any address modification happens in cycle i. The loaded data is written into the register file in cycle i + 4.
(3) The branch to label, branch to IRP, and branch to NRP instructions do not read any general-purpose registers.

REVERSING TOOLS



Still wondering why no tools are available???

MEMORY ORGANIZATION

- 32 bit, byte addressable address space
- On-chip memory:
  - Organized in data and program spaces
  - Two 64 bit internal ports to access data memory
  - One 256 bit internal port to access program memory
- Off-chip memory:
  - Data and program spaces are unified via the External Memory InterFace (EMIF)
  - EMIF is 32 bit

ABI - CALLING CONVENTIONS

- No stack exists (stack as in x86 stack)
- To call a function you need to calculate the return address by adding PC to the relative offset of the return address using a special assembly instruction (ADDKPC)
- To return you need to jump to the register where the previous value has been saved
- ABI specifies this register should be B3
- The displacement at a branch instruction is a 21 bit word offset. If the destination is unreachable, the linker must generate a trampoline

ABI - CALLING CONVENTIONS

```
        B       .S1  func
        AND     .D1  A0, A1, A2
||      AND     .D2  B0, B1, B2
        ADDKPC  .S2  returnhere, B3
        ADD     .D1  A2, 1, A3
        MV      .D2  B3, B0
||      MV      .D1  A3, A0
        OR      .S2X B0, A0, B4
returnhere:
        MVK     .S1  10, A0
...

func:
...
        B       .S1  B3
```

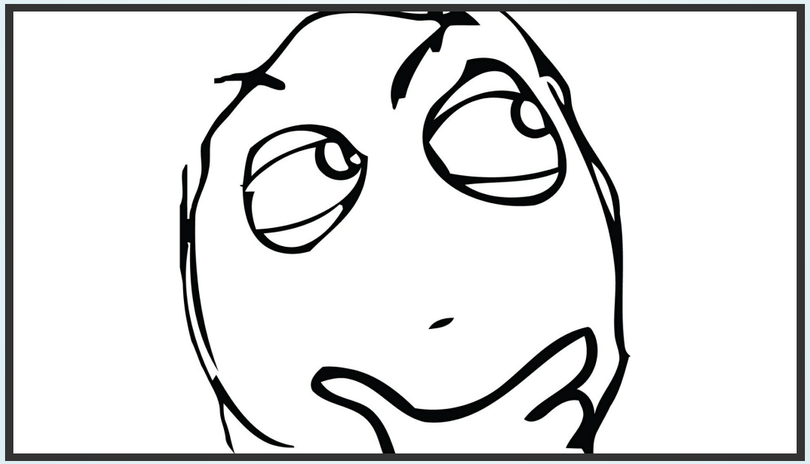
ABI - REGISTERS

- 12 callee-saved registers (A10-A15, B10-B15)
- A15 acts as Frame Pointer
- B14 acts as Data Page Pointer
- B15 acts as Stack Pointer
- B3 keeps the return address

ABI - ARGUMENT PASSING

- First 10 arguments are passed in registers
- If an argument is 64 bits, a register pair is used
- That makes us a total of 20 registers for argument passing
- 32 bit values are returned in A4, 64 bit values in A5:A4, and bigger values are returned by reference

IMPLEMENTING THE DISASSEMBLER



We know the architecture, the memory layout and the ABI. Now what???

FRAMEWORK CHOICES

- Binutils: too much work
- LLVM: still too much work
- Radare2: Potential choice
- Capstone: Potential choice, much more attractive for implementing new architectures

THE CAPSTONE DISASSEMBLY FRAMEWORK

CAPSTONE
ENGINE

THE CAPSTONE DISASSEMBLY FRAMEWORK

- Written in pure C
- Open source
- Based on LLVM
- Bindings for many other programming languages
- No documentation exists on adding a new architecture, but it should be fairly easy

WHERE DO WE START?

- Clone the project
- See how other architectures are implemented
- Try to implement our own

ARCHITECTURE IMPLEMENTATION

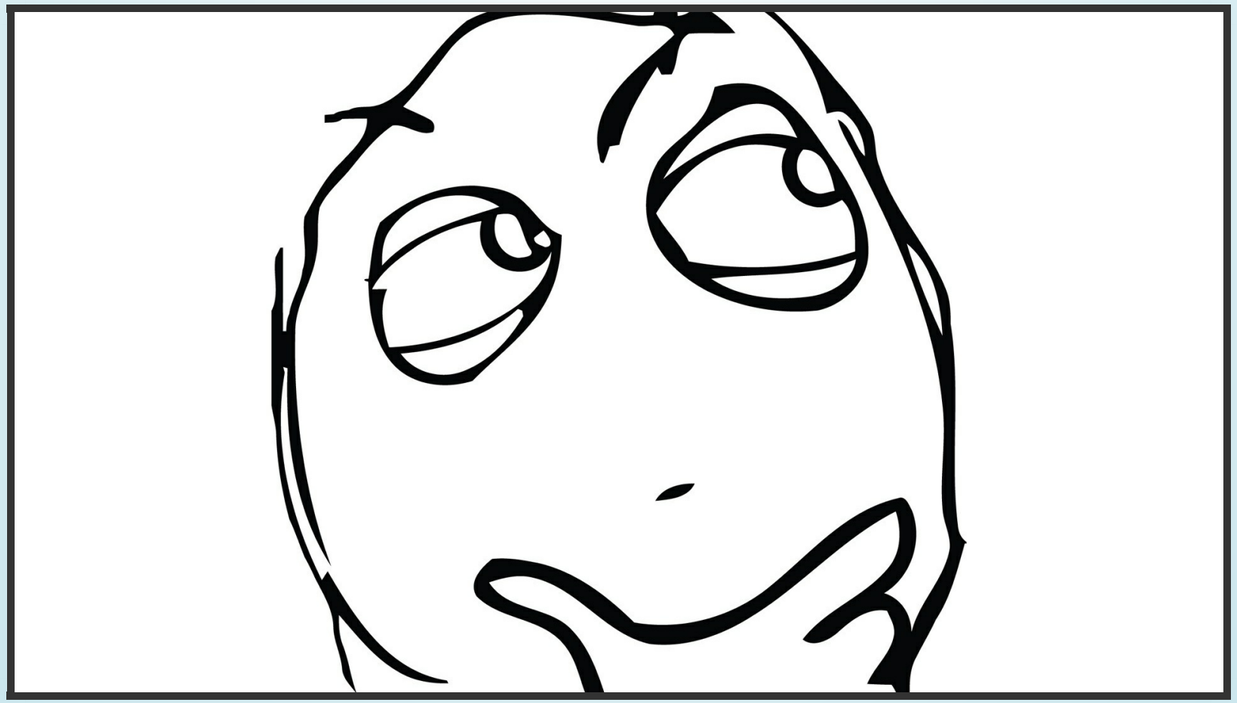
A TableGen (.td) file with the architecture and instructions description helps generate:

- Register/Instruction/Subtarget Info Files
- Disassembler Tables
- Assembly Writer

But there's more needed!

THE TABLEGEN FILE

Let's first generate this and we'll see about the rest...



THE TABLEGEN FILE FORMAT

A file defining:

- Targets/Subtargets
- Processors
- Instruction Set
- Registers
- Calling Conventions

WHAT'S ACTUALLY NEEDED?

In our case we don't need:

- Calling conventions - LLVM needs them for the compilation part but we'll just create a disassembler
- The instruction selection pattern - LLVM needs it to select the appropriate instruction when compiling
- Many flags

WHAT'S ACTUALLY NEEDED?

What we will need is:

- Register description
- Instruction description with:
  - Hardware encoding
  - Input/output registers
  - Assembly string

HOW DO WE DEFINE REGISTERS?

Simple, we define a register class and then instances for all registers!

```
class TMS320C64xReg<string n, bits<5> num, bit file,
        bits<3> condition = 7> : Register<n> {
    let HWEncoding{15-5} = 0;
    let HWEncoding{4-0} = num;
    field bit File = file;
    field bits<3> Condition = condition;
    let Namespace = "TMS320C64x";
}
def A0  : TMS320C64xReg<"A0",  0,  0, 6>, DwarfRegNum<[0]>;
...
def B31 : TMS320C64xReg<"B31", 31, 1>, DwarfRegNum<[63]>;
```

HOW DO WE DEFINE INSTRUCTIONS?

- More difficult
- We have to define operands
- Then instruction classes
- And if needed multiclasses

HOW DO WE DEFINE OPERANDS?

```
def memop : Operand<i32>, PatLeaf<(imm),
        [{ return isInt<15>(N->getZExtValue()); }]>
{
    let DecoderMethod = "DecodeMemOperand";
    let EncoderMethod = "EncodeMemOperand";
    let PrintMethod = "printMemOperand";
}
```

For every operand we need to create a decoder, an encoder and maybe a print method!

HOW DO WE DEFINE INSTRUCTION CLASSES?

```
// Superclass
class TMS320C64xInst<dag outops, dag inops, string asmstr,
        list<dag> pattern> : Instruction {
    field bits<32> Inst;
    field bits<32> SoftFail = 0;
    bits<3> cond;
    bit condzero;
    bit side;
    bit parallel;

    let OutOperandList = outops;
    let InOperandList = !con(inops, (ins condreg:$cond,
        condregzero:$condzero, sideop:$side,
        parallelop:$parallel));
    let AsmString = asmstr;
    let Pattern = pattern;
```

HOW DO WE DEFINE INSTRUCTION CLASSES?

```
    let Inst{31-29} = cond;
    let Inst{28} = condzero;
    let Inst{1} = side;
    let Inst{0} = parallel;
    let Size = 4;
    let isPredicable = 1;
    let hasSideEffects = 0;
    let Namespace = "TMS320C64x";
}
```

HOW DO WE DEFINE INSTRUCTION CLASSES?

```
class TMS320C64xInstD1<bits<6> opVal, dag outops, dag inops,
        string asmstr, list<dag> pattern> :
        TMS320C64xInst<outops, inops, asmstr, pattern> {
    bits<5> dst;
    bits<5> src2;
    bits<5> src1;
    bits<6> op = opVal;

    let Inst{27-23} = dst;
    let Inst{22-18} = src2;
    let Inst{17-13} = src1;
    let Inst{12-7} = op;
    let Inst{6-2} = 0b10000;
}
```

HOW DO WE DEFINE INSTRUCTION CLASSES?

```
class TMS320C64xInstD2<bits<4> opVal, dag outops, dag inops,
      string asmstr, list<dag> pattern> :
      TMS320C64xInst<outops, !con(inops,
      (ins crosspathopx2:$crosspath)), asmstr, pattern> {
   bits<5> dst;
   bits<5> src2;
   bits<5> src1;
   bit crosspath;
   bits<4> op = opVal;

   let Inst{27-23} = dst;
   let Inst{22-18} = src2;
   let Inst{17-13} = src1;
   let Inst{12} = crosspath;
   let Inst{9-6} = op;
   let Inst{11-10} = 0b10;
   let Inst{5-2} = 0b1100;
}
```

HOW DO WE DEFINE INSTRUCTION MULTICLASSES?

```
multiclass TMS320C64xInstD1_ri<bits<6> opVal1, bits<6> opVal2,
        string asmstr1, string asmstr2> {
    def _d1_rrr : TMS320C64xInstD1<opVal1, ( outs GPRegs:$dst ),
        ( ins GPRegs:$src2, GPRegs:$src1 ),
        asmstr1, []>;
    def _d1_rir : TMS320C64xInstD1<opVal2, ( outs GPRegs:$dst ),
        ( ins GPRegs:$src2, ucst5:$src1 ),
        asmstr2, []>;
}
```

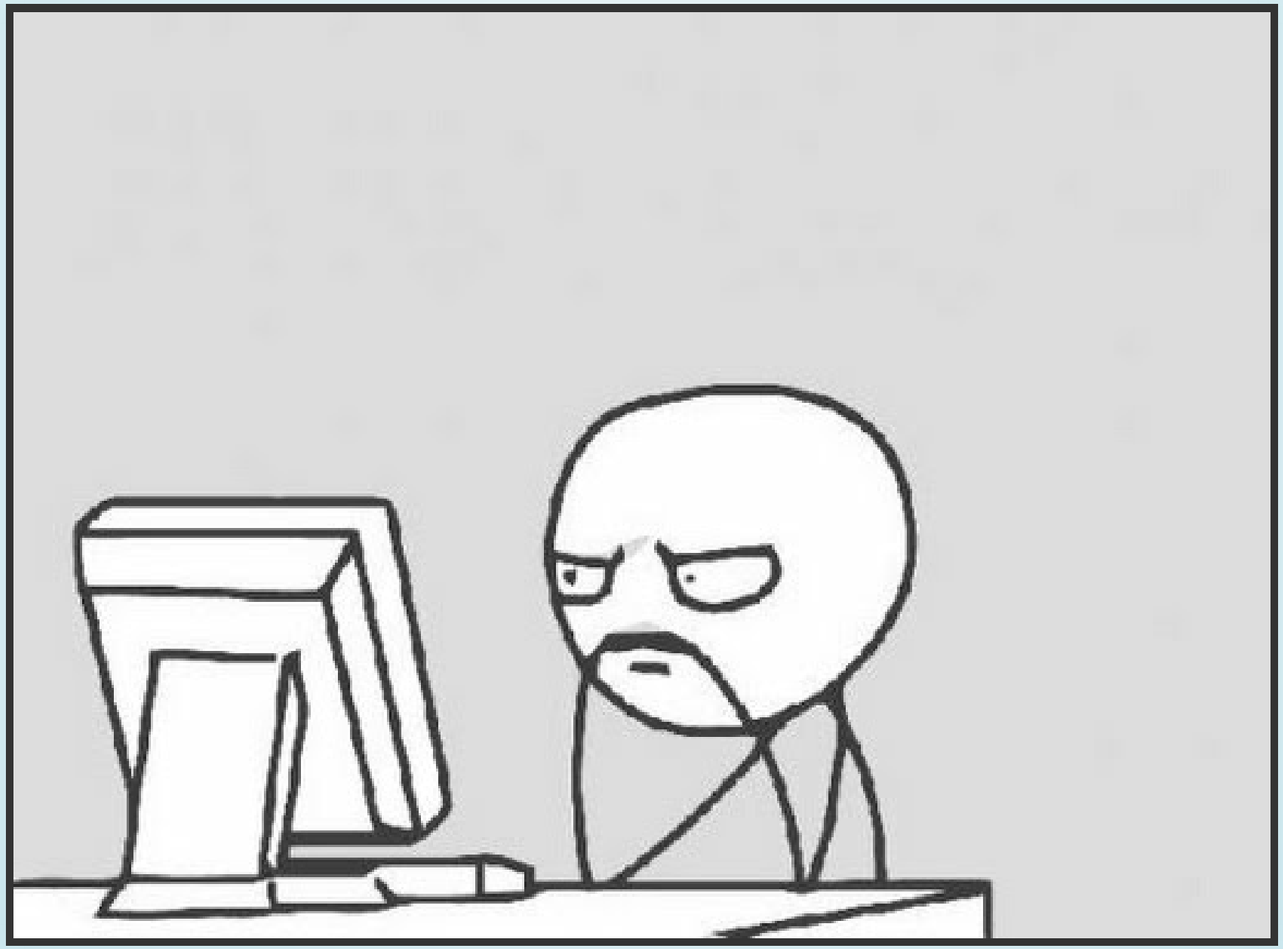
HOW DO WE DEFINE INSTRUCTIONS?

```
defm ADD : TMS320C64xInstD1_ri<0b010000, 0b010010,
    "ADD\t$src2, $src1, $dst", "ADD\t$src2, $src1, $dst">;

def ADD_d2_rrr : TMS320C64xInstD2<0b1010,
    ( outs GPRegs:$dst ),
    ( ins GPRegs:$src2, GPRegs:$src1 ),
    "ADD\t$src1, $src2, $dst",
    []>;
```

ARE WE DONE?

- Not even close
- We need to implement code for encoding/decoding/printing operands (real and 'virtual' ones such as parallel and crosspath)
- We need to implement instruction mappings
- We need to add supporting code for the architecture
- We need to add at least python bindings

LETS FINISH THIS THING!



PROBLEMS FACED

- The converter was not the LLVM one but a not-disclosed version that produces C code instead of C++
  - No problem, converted the original LLVM converter to produce output compatible with Capstone
- Decoding of all fields must be done manually
  - Still no problem, we have all the documentation so it can be done

PROBLEMS FACED

- Instructions that use PC relative addressing must be handled specially
  - We'll just have to keep the PC and make all calculations correctly
  - During coding the TMS320C64x architecture support, PC relative addressing bugs were found at other architectures

PROBLEMS FACED

- Branches must be handled with care since they are relative to the first instruction in the fetch packet
  - Again, we'll have to keep the PC and based on the implementation of the instruction dispatch unit find out the first instruction in each fetch packet
- The parallel bit is not set at the same instruction but at the previous one
  - Difficult, but can be done with some post-processing

CONGRATULATIONS!

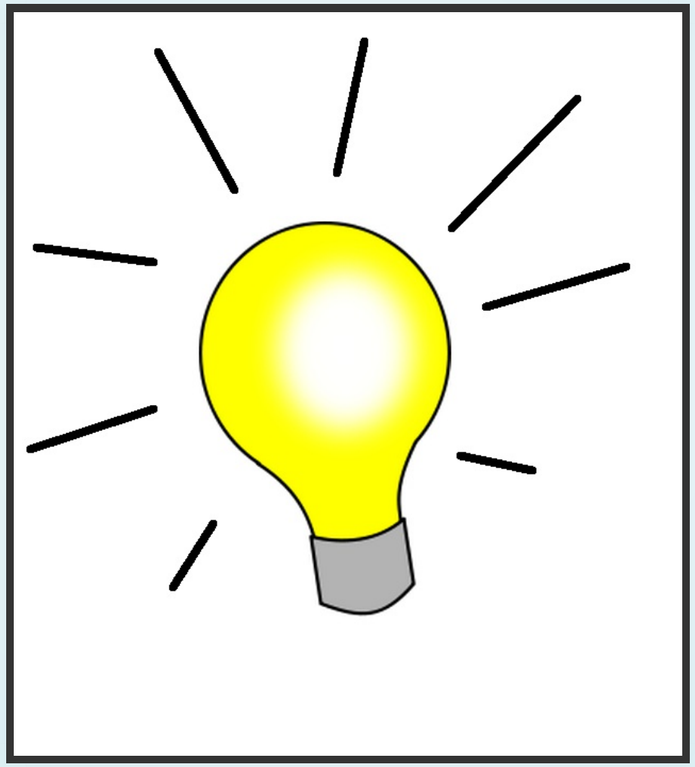
We have a disassembler we can use!

REVERSING THE FIRMWARE

How do we proceed?

- We can disassemble everything
- But we have no symbols
- And we have thousands of lines of strange assembly code

REVERSING THE FIRMWARE



Let's check the simulator firmware!

THE SIMULATOR FIRMWARE

- Extract the MTC files and then the internal binary
- At the actual firmware this was a modified COFF
- At the simulator firmware it's... a DLL!

REVERSING THE FIRMWARE

We've got symbols!

WHAT NOW?

- Correlate string references in both firmwares, same strings at same functions (easy)
- Check out branches from the same functions (harder)
- Fix the rest by hand (much harder)
- Read the code and profit!

CAN WE PROFIT?

- These products are methodically tested and the best security practices are used during design
- Bugs should not exist
- At least, simple and easy to identify bugs should be eliminated
- Let's move on to an example of security checks...

EXTRACTION OF MBK DATABASE BACKUP

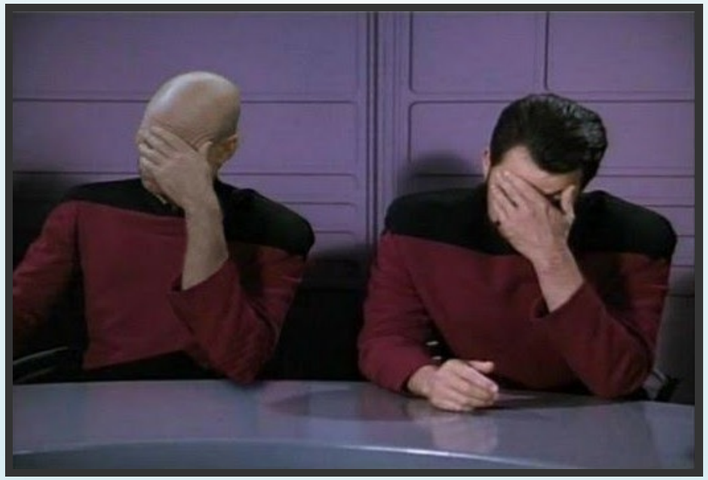
```
000009d8   00564d42          .word 0x00564d42  // '\x00VMB'
000009dc   4b310073          .word 0x4b310073  // 'K1\x00s'

_adm_ext_list_db_search_keys:
...
||        MVK.S1        0x09d9,A5
||        ADD.L1X       A4,B4,A3
||        ADD.D2        B15,0x17,B4
          MVKH.S1       0x0000,A5
          SUB.L1        A5,0x1,A4
          LDB.D1T1      *++A4[1],A0
          LDBU.D2T2     *++B4[1],B5
          MVK.L1        0,A1
          NOP           2
          EXTU.S1       A0,24,24,A5
|| [ A0]  SUB.L1        A3,0x1,A1
   [ A0]  SUB.L1        A3,0x1,A3
          CMPEQ.L2X     B5,A5,B0
  [!B0]   MVK.L1        0,A1
|| [!B0]  B.S2          0x003010
...
```

EXTRACTION OF MBK DATABASE BACKUP

- Security controls: check if string starts with 'VMBK1', 'MBK' or 'session_obj' to protect sensitive databases
- If tests pass, then open the database with the common open function
- What's the argument to this function?
  - The name of the database, such as CXIKEY.db
  - Possibly prefixed with the location, such as FLASH\CXIKEY.db

EXTRACTION OF MBK DATABASE BACKUP

What if we extract FLASH\VMBK1.db?



Sometimes one facepalm is not enough...

WHAT ABOUT OTHER BUGS?

- All the code has the same quality
- Similar logical errors can be found
- Haven't searched for buffer overflows yet

DISCLOSURE

- Contacted Utimaco at 10/03/2016
- Managed to receive PGP keys and certificate at 19/5/2016
- Sent details at 20/5/2016
- Received final official answer at 17/6/2016
- Got access to an HSM with updated firmware for testing at 20/10/2016

DOWNLOADS

- Reversing tools: https://github.com/fotisl/utimaco
- Capstone: https://github.com/aquynh/capstone/tree/tms320c64x

QUESTIONS???