Reverse engineering tools for ST DVB chipsets

SRP-2018-01

DISCLAIMER

INFORMATION PROVIDED IN THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW NEITHER SECURITY EXPLORATIONS, ITS LICENSORS OR AFFILIATES, NOR THE COPYRIGHT HOLDERS MAKE ANY REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR THAT THE INFORMATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS, OR OTHER RIGHTS. THERE IS NO WARRANTY BY SECURITY EXPLORATIONS OR BY ANY OTHER PARTY THAT THE INFORMATION CONTAINED IN THE THIS DOCUMENT WILL MEET YOUR REQUIREMENTS OR THAT IT WILL BE ERROR-FREE.

YOU ASSUME ALL RESPONSIBILITY AND RISK FOR THE SELECTION AND USE OF THE INFORMATION TO ACHIEVE YOUR INTENDED RESULTS AND FOR THE INSTALLATION, USE, AND RESULTS OBTAINED FROM IT.

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL SECURITY EXPLORATIONS, ITS EMPLOYEES OR LICENSORS OR AFFILIATES BE LIABLE FOR ANY LOST PROFITS, REVENUE, SALES, DATA, OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, PROPERTY DAMAGE, PERSONAL INJURY, INTERRUPTION OF BUSINESS, LOSS OF BUSINESS INFORMATION, OR FOR ANY SPECIAL, DIRECT, INDIRECT, INCIDENTAL, ECONOMIC, COVER, PUNITIVE, SPECIAL, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND WHETHER ARISING UNDER CONTRACT, TORT, NEGLIGENCE, OR OTHER THEORY OF LIABILITY ARISING OUT OF THE USE OF OR INABILITY TO USE THE INFORMATION CONTAINED IN THIS DOCUMENT, EVEN IF SECURITY EXPLORATIONS OR ITS LICENSORS OR AFFILIATES ARE ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS.

INTRODUCTION

STMicroelectronics' [1] SlimCORE processor is one of the helper cores of STi7111 DVB chipset SoC (Fig. 1) [2]. This SoC is used as a base chipset of PayTV set-top-box devices of many digital TV operators around the world (both satellite and terrestrial).

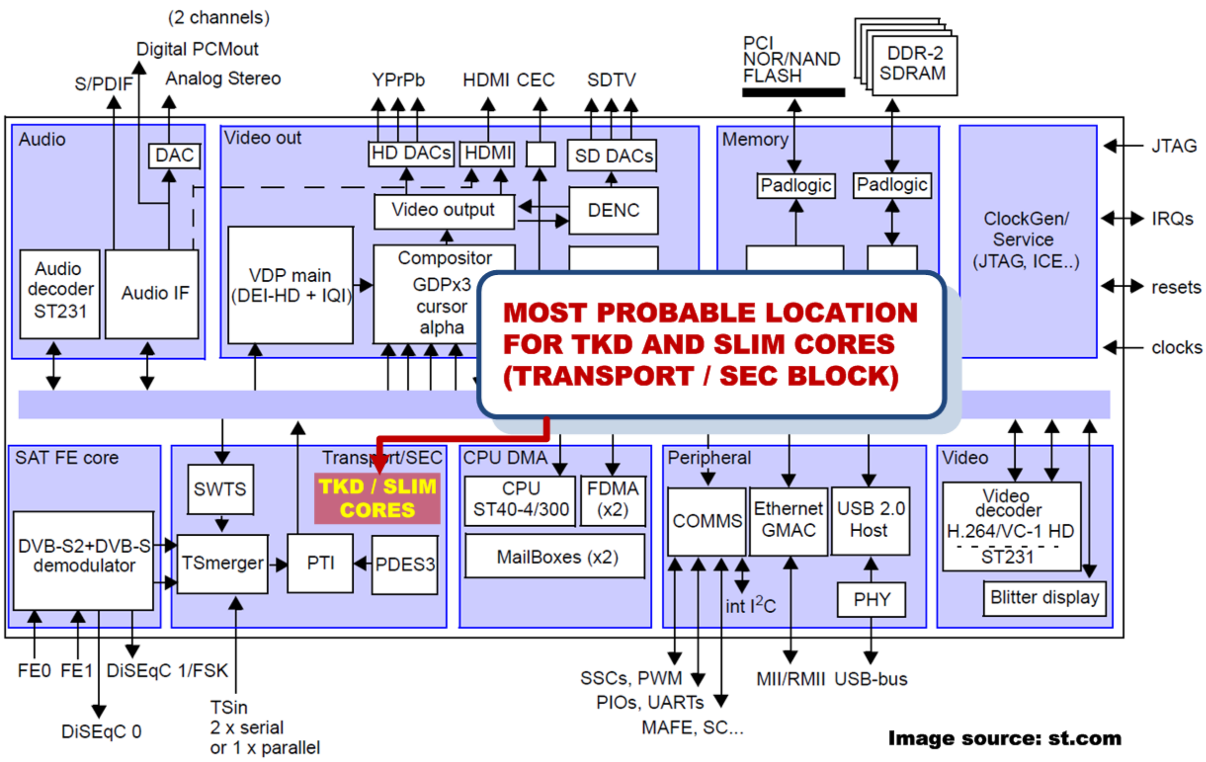


Fig. 1 SlimCORE location in STi7111 SoC.

This document provides a brief description of SlimCORE CPU and its firmware code used by Platform N digital satellite TV provider at the end of 2011 in its Advanced Digital Broadcast (ADB) set-top-boxes (models ITI-2849ST and ITI-2850ST)[1]. This was the base firmware code used by Security Explorations to analyze security of STi7111 chipset as part of SE-2011-01 security research project [3].

All of the information contained in this document are the result of a tedious reverse engineering effort conducted in 2010 and 2011. As such, provided information may not be consistent with original vendor's documentation for SlimCORE processor. It may be incomplete and include many inaccuracies. Regardless of the above, it was sufficient to discover 2 security vulnerabilities (Issue 18 and 19) [4][5] in STi7111 SoC and implement tools facilitating the analysis of a chipset operation (SlimCORE disassembler and tracer).

SlimCORE PROCESSOR

SlimCORE processor came to life as a result of a collaboration between ST UK and OneSpin after the spin-off from Infineon [6]. It is a lightweight processor with 27 instructions and a 4-stage pipeline.

[1] SlimCORE firmware version `STTKDMA-REL_3.1.6`

Processor special features include a coprocessor interface, circular buffer operation, a STOP and RPT instructions.

Register Set

SlimCORE is a 32-bit core. It has 14 general purpose 32-bit registers (R0-R14), a special register corresponding to the instruction pointer (IP) and a special I/O register (R15). This is illustrated on Fig. 2.

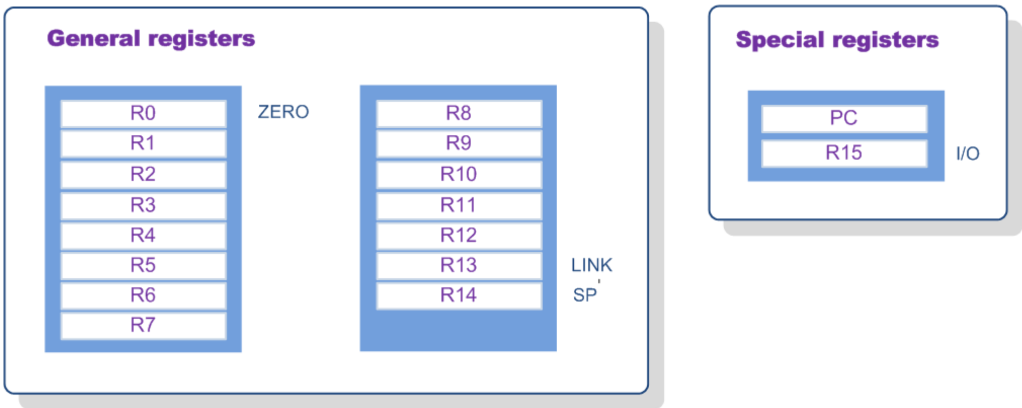


Fig. 2 SlimCore registers.

We figured out that register R0 denotes a zero value due to its use as a base register of certain memory addressing instructions:

```
ld r9,[r0,0020] // 0x4080 = 0x4000+0000+0x20*4
```

Register R13 corresponds to the LINK register due to its frequent use as a holder of a return address from subroutine calls:

```
0039    0x00ed003b   mov r13,#003b   ;subroutine return addr
003a    0x008c04e1   j l_04e1        ;init keys subroutine
003b    0x00e40312   mov r4,#0312
```

Finally, register R14 was concluded to be an equivalent of a stack pointer register upon the construction of instruction sequences denoting prologs of arbitrary subroutine calls:

```
########################
SUB l_050f
MAIN DISPATCH
r14 = 0xd0
########################

l_050f  0x00b01eff   st r1,[r14,00ff]        ;save r1 on stack
  0510  0x00b0aefe   st r10,[r14,00fe]       ;save r10 on stack
  0511  0x003ee002   sub r14,r14,r0,#0002    ;alloc locals
```

IP register denotes an index of a 32bit memory word containing an instruction to execute. The memory location from which an instruction opcode is to be fetched and executed is described by this formula:


```
opcode_addr = IP*4
```

Register R15 indicates that a given register move, memory load or store operation are to be conducted with respect to I/O communication link with one of chipsets' cores (such as TKD Crypto core).

SlimCORE also contains register flags. We neither figured out, nor proceeded with reverse engineering of the flags register location and its access methods (instructions)[2]. It is sufficient to say that sequences of arithmetic and conditional instructions indicate the existence of an equivalent (known from other CPU architectures) of the following flags:

- Z / EQ (zero or equal result),
- S (signed result),
- C (result with carry / borrow).

Memory Addressing

SlimCORE implements all memory addressing with the use of a word number - an index to an array of 32bit data items.

Arbitrary memory accesses are implemented with the use of load (LD) and store (SR) instructions. These instruction make use of the following addressing modes to indicate either source (LD) or destination (ST) memory operand:

1) a register based addressing with an immediate index:

```
[register+index]
```

2) a register based addressing and an immediate value incrementing the base register

```
[register],register+=imm
```

Taking into account that the immediate index denotes a word number, for case 1 the target memory address to access is computed as following:

```
addr = register_content+4*index
```

SlimCORE processor operates in a little endian mode. As a result, 32-bit memory words for both code (instruction opcodes) and data are stored starting from the least significant byte. Thus, a 32-bit wide integer value of 0x11223344 is stored in memory as a sequence of 0x44, 0x33, 0x22 and 0x11 bytes.

Memory spaces

SLIMCore instructions can access either DATA or I/O memory spaces. In our environment, the beginning of a DATA memory region was set at 0x4000 offset relative to the chip base address[3]. I/O memory space began at 0x5e00 offset. All load / store instructions with `reg2` opcode equal to 0 (register 0) referenced these areas solely with the use of an immediate index as indicated below:

[2] this wasn't necessary from a point of view of completing our security analysis of the chip.
[3] the value of 0xFE248000 for ADB set-top-boxes.

```
0x00b03085   st r3,[r0,0085] // store r3 to 0x5e14
0x00b0002c   st r0,[r0,002c] // store r0 to 0x40b0
```

Additionally, arbitrary communication I/O operations (such as data exchange with TKD core) are implemented with the use of special load, store and move instructions. This is illustrated in Table 1.

OPERATION TYPE	INSTRUCTION	DESCRIPTION
Store data (OUT operation)	`mov r15, reg`	Store the contents of register `reg` to TKD core
	`ld r15,[r0,imm]`	Store the contents of a memory location indicated by `imm` index `reg` to TKD core
Load data (IN operation)	`mov reg, r15`	load the contents of register `reg` with the value read from TKD core
	`st r15,[reg,imm]`	Load the contents of a memory location indicated by `imm` index `reg` with the value read from TKD core

Table 1 Instructions for data exchange with Crypto TKD core.

It's worth to mention that IN and OUT channels linked to the I/O register seem to be associated with different IN and OUT buffers (or a single buffer with different IN and OUT positions). We reason this upon the following code implementing byte swap operation during DMA crypto transfer:

```
l_03c2  0x00030f3c   mov r3,r15        ; r3 <- IN
  03c3  0x000330c0   swap r3,r3
  03c4  0x000f033c   mov r15,r3        ; r3 -> OUT
  03c5  0x00030f3c   mov r3,r15        ; r3 <- IN
  03c6  0x000330c0   swap r3,r3
  03c7  0x000f033c   mov r15,r3        ; r3 -> OUT
  03c8  0x00030f3c   mov r3,r15        ; r3 <- IN
  03c9  0x000330c0   swap r3,r3
  03ca  0x000f033c   mov r15,r3        ; r3 -> OUT
  03cb  0x00030f3c   mov r3,r15        ; r3 <- IN
  03cc  0x000330c0   swap r3,r3
  03cd  0x000f033c   mov r15,r3        ; r3 -> OUT
```

If the I/O register was connected to the same buffer (or position), consecutive IN and OUT operations would be able to change only 1 word, not 4 of them.

Reverse engineering approach

Reverse engineering of the format of all instructions described below was started from a format of a single unconditional instruction jump (JMP), which was leaked by a GNU source code for SLIM Core Generic driver [7]:

```
// Init imem so every instruction is a jump to itself
for (n = 0; n < core->imem_size/ 4; n++)
      SLIM_IMEM(core, n) =  0x00d00010 | (n & 0xf)
                            | ((n & 0xfff0) << 4);
```


The above code sequence carries the following generic information about SlimCORE instructions:

- instruction opcode is 32-bit wide - hint A,
- memory addressing is conducted by a word index (n denotes an address of an instruction itself, although the instruction opcode width is 4 bytes, n is incremented by 1) - hint B.

In the next step, the format of a memory store (ST) instruction was discovered. This was achieved by the means of matching the pattern of the result provided by the `GetPublicID` command format with a sequence of instruction opcodes embedded in SlimCORE firmware.

The result of `GetPublicID` command was provided as a sequence of four 32-bit words as indicated by Fig. 3.

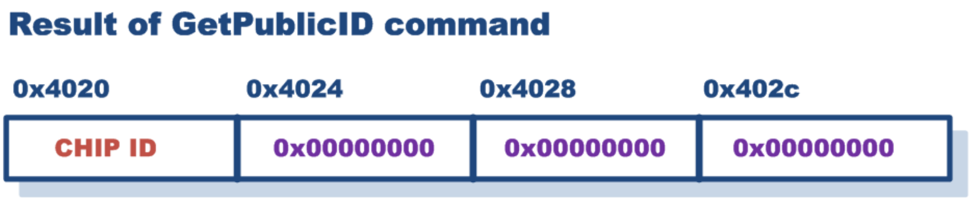


Fig. 3 Output buffer of a GetPublicID command.

The only 32-bit words opcode sequence (hint A) available in firmware code that exploited the filling of an output buffer in a form of accesses to consecutive memory indexes (hint B) was conducted in only one memory location as shown on Fig. 4.

SLIM Core firmware sequence

```
01a1   0x00a5008b    ld r5,[r0,008b]  // 0x5e2c
01a2   0x00a9001f    ld r9,[r0,001f]  // 0x407c
01a3   0x00b05008    st r5,[r0,0008]  // 0x4020
01a4   0x00b00009    st r0,[r0,0009]  // 0x4024
01a5   0x00b0000a    st r0,[r0,000a]  // 0x4028
01a6   0x00b0000b    st r0,[r0,000b]  // 0x402c
01a7   0x00d01c1a    jmp l_01ca
```

Fig. 4 SlimCORE instruction sequence corresponding to GetPublicID result.

Our guess was confirmed by the means of a manual change of the located sequence and observation of the result of `GetPublicID` command. Most importantly, a change of a ST R5 instruction with ST R0 instruction resulted in a first word of the output buffer to be set to 0. This and other experiments with the located opcode sequence such as those changing the index word and source register in particular confirmed that these are indeed memory store instructions. As a result, its initial format could be discovered:

ST - Store reg1 to memory location pointed by reg2 and memory index imm

23	22	21	20	19	18	17	16	15–12	11–8	7–0
1	0	1	1	0	0	0	0	reg1	reg2	imm

The format of a memory load instruction opcode (LD) was discovered building on the format of ST opcode and by the means of changing the LD R5 instruction from the located opcode sequence and observation of the output buffer obtained. More specifically, changing the source register field to given index of the output buffer filled with a particular value resulted in that value being returned as the first word of the output buffer (chip ID location). This was sufficient to confirm an initial format of a memory load (LD) instruction:

LD - Load reg1 from memory location pointed by reg2 and memory index imm

23	22	21	20	19	18	17	16	15-12	11-8	7-0
1	0	1	0	0	0	0	0	reg1	reg2	imm

Knowledge about the format of JMP, LD and ST instructions was sufficient to discover all other SlimCORE instruction opcodes.

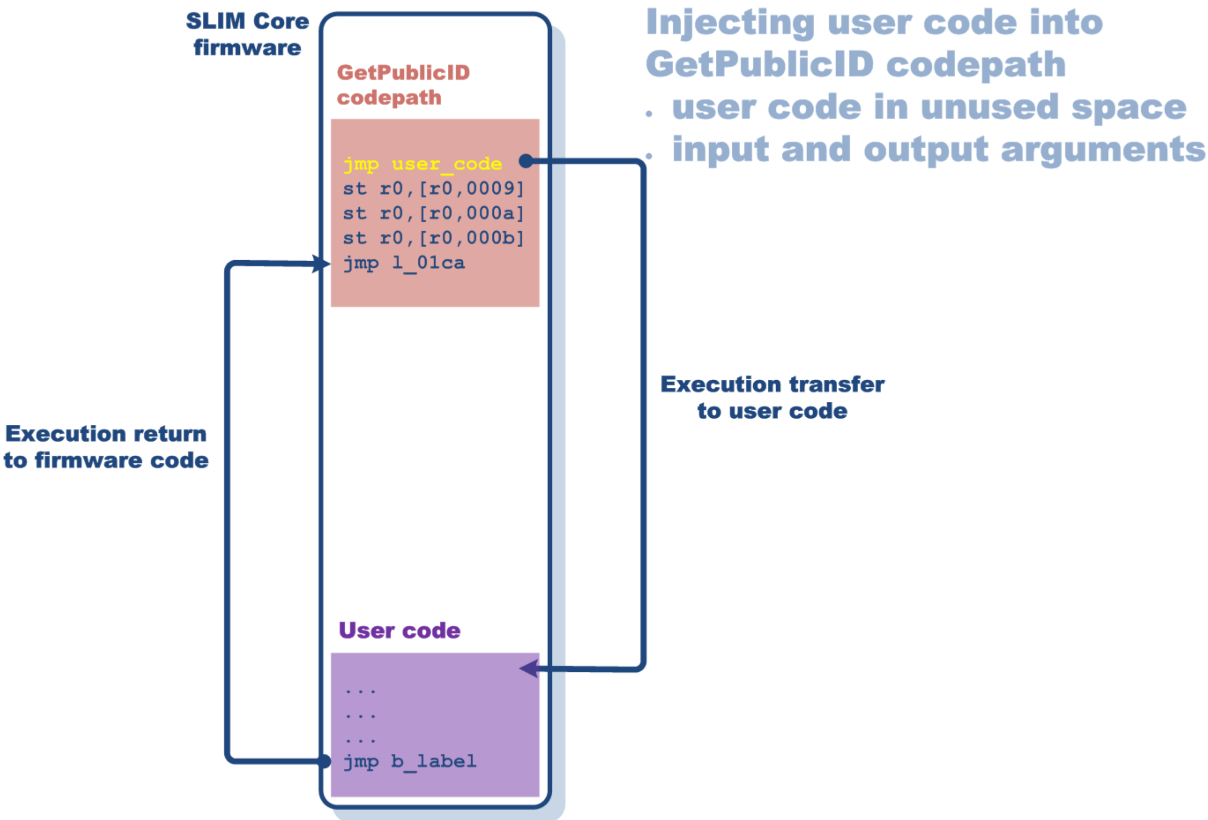


Fig. 5 Running user provided code as part of GetPublicID code path.

We exploited the ability to change the operation of SLIM Core firmware in runtime and overwrote SlimCORE firmware memory in a way that made it possible to inject a custom code sequence into the `GetPublicID` code path. This is illustrated on Fig. 5.

Custom code sequence was implemented by the means of embedding an unknown instruction or their sequence around the sequence of JMP, LD and STORE instructions only. The custom sequence was formatted as following:

- JMP from firmware to user's code path
    - STORE the contents of registers (firmware context)
        - LOAD user's environment (contents of registers)

        - EXECUTE unknown SLIMCore instruction opcode
        - STORE user's environment (contents of registers)
    - LOAD the contents of registers (firmware context)
- JMP back to firmware code path.

An effect of the execution of an unknown instruction opcode to memory and registers was observed. In our case, the custom SlimCore code sequence injected into the `GetPublicID` code path had the following implementation:

```
int code[]={
  0x00b01050,//   st r1,[r0,0050] offset 0x05b7
  0x00b02051,//   st r2,[r0,0051] offset 0x05b8
  0x00b03052,//   st r3,[r0,0052] offset 0x05b9
  0x00b04053,//   st r4,[r0,0053] offset 0x05ba
  0x00b05054,//   st r5,[r0,0054] offset 0x05bb
  0x00b06055,//   st r6,[r0,0055] offset 0x05bc
  0x00b07056,//   st r7,[r0,0056] offset 0x05bd
  0x00b08057,//   st r8,[r0,0057] offset 0x05be
  0x00b09058,//   st r9,[r0,0058] offset 0x05bf
  0x00b0a059,//   st r10,[r0,0059] offset 0x05c0
  0x00b0b05a,//   st r11,[r0,005a] offset 0x05c1
  0x00b0c05b,//   st r12,[r0,005b] offset 0x05c2
  0x00b0d05c,//   st r13,[r0,005c] offset 0x05c3
  0x00b0e05d,//   st r14,[r0,005d] offset 0x05c4

  0x00000000,//   SLOT FOR AN UNKNOWN INSTRUCTION
                  OPCODE TO TEST

  0x00a10050,//   ld r1,[r0,0050] offset 0x05d6
  0x00a20051,//   ld r2,[r0,0051] offset 0x05d7
  0x00a30052,//   ld r3,[r0,0052] offset 0x05d8
  0x00a40053,//   ld r4,[r0,0053] offset 0x05d9
  0x00a50054,//   ld r5,[r0,0054] offset 0x05da
  0x00a60055,//   ld r6,[r0,0055] offset 0x05db
  0x00a70056,//   ld r7,[r0,0056] offset 0x05dc
  0x00a80057,//   ld r8,[r0,0057] offset 0x05dd
  0x00a90058,//   ld r9,[r0,0058] offset 0x05de
  0x00aa0059,//   ld r10,[r0,0059] offset 0x05df
  0x00ab005a,//   ld r11,[r0,005a] offset 0x05e0
  0x00ac005b,//   ld r12,[r0,005b] offset 0x05e1
  0x00ad005c,//   ld r13,[r0,005c] offset 0x05e2
  0x00ae005d,//   ld r14,[r0,005d] offset 0x05e3

  0x00d01c1a //   jmp l_01ca       offset 0x05e4
};
```

The abovementioned approach was used for a systemic discovery of SlimCORE instructions' format. Instruction opcodes were discovered one by one. The scope of a discovery process was limited to unknown opcodes from firmware code.

Beside the approach outlined above, some code patterns that started to become visible along instructions' discovery process were also exploited. This in particular includes, but is not limited to the patterns of MOV instructions (Fig. 6) along with CMP and conditional jump instructions (Fig. 7).


```
0003   0x00e30374    mov r3,#0374
0004   ....
0005   0x00e4ffff    mov r4,#ffff
0006   0x00e3ffff    mov r3,#ffff
0007   ....
0008   0x00e30001    mov r3,#0001
```

Fig. 6 MOV instructions patterns.

Finally, for proper conditional jump handling, the custom code needed to be extended to include more than one instruction (a sequence of MOV, CMP and an unknown conditional jump).

```
0014   0x00981026   je l_0026
0015   0x00c03002   cmp r3,#02
0016   0x00981028   je l_0028
0017   0x00c03006   cmp r3,#06
0018   0x0098102a   je l_002a
0019   0x00c0300b   cmp r3,#0b
001a   0x0098102c   je l_002c
001b   0x00c0300f   cmp r3,#0f
001c   0x0098102e   je l_002e
001d   0x00c03003   cmp r3,#03
001e   0x00981030   je l_0030
001f   0x00c03007   cmp r3,#07
```

Fig. 7 CMP and conditional jump instructions patterns.

Instruction set

SlimCORE uses a RISC-style fixed length instruction opcodes. All processor opcodes are 32-bit wide. Only lower 24 bits of each opcode seem to be used though (bits 24-31 of instruction opcode are set to a value of 0).

The processor implements memory access, branching, arithmetic, logical, shift and coprocessor instructions among others. Below, a more detailed information regarding the opcode format and operation of specific instructions is given. All instruction are listed according to their opcode value (bits 20-23).

Please, note that in some cases little or no generalization of discovered instruction opcodes was performed as reverse engineering process was focused on discovering instructions' functionality needed for a successful analysis of firmware code, not to obtain a complete and accurate information regarding SlimCORE instruction set.

0x00 opcodes (MOV, SWAP)

MOV - Move to register

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	reg1				0	0	0	0	reg2				0	0	1	1	1	1	0	0

Notation:

MOV *reg1, reg2*

Description:

Move the contents of register reg2 to register reg1.

SWAP - Swap registers

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	reg1				reg2				0	0	0	0	1	1	0	0	0	0	0	0

Notation:

SWAP *reg1, reg2*

Description:

Swaps contents of registers reg1 and reg2.

0x01 opcodes (SHL, SHR)

SHL - Logical shift left

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	reg1				reg2				0	0	0	0	0	0	0	imm				

Notation:

SHL *reg1,reg2,#imm*

Description:

Shift the contents of registers reg2 to the left by the number of bits denoted by an immediate operand and store result to register reg1.

SHR - Logical shift right

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	reg1				reg2				0	0	0	0	0	0	1	imm				

Notation:

SHR *reg1,reg2,#imm*

Description:

Shift the contents of registers reg2 to the right by the number of bits denoted by an immediate operand and store result to register reg1.

0x02 opcodes (ADD)

ADD - Arithmetic Add

23	22	21	20	19-16	15-12	11-8	7-0
0	0	1	0	reg1	reg2	reg3	imm

Notation:

ADD *reg1, reg2, reg3, #imm*

Description:

Add the contents of reg3 register and an immediate operand to the contents of reg2 register and store result to register reg1.

0x03 opcodes (SUB)

SUB - Arithmetic Sub

23	22	21	20	19-16	15-12	11-8	7-0
0	0	1	1	reg1	reg2	reg3	imm

Notation:

SUB *reg1, reg2, reg3, #imm*

Description:

Substract the contents of reg3 register and an immediate operand from the contents of reg2 register and store result to register reg1.

0x04 opcodes (AND, TST)

AND - Logical AND

23	22	21	20	19-16	15-12	11-8	7	6	5	4	3	2	1	0
0	1	0	0	reg1	reg2	reg3	0	0	0	0	0	0	0	0

Notation:

AND *reg1,reg2,reg3*

Description:

Perform logical AND of the contents of registers reg2 and reg3 and store result to register reg1.

AND - Logical AND

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	reg1				reg2				0	0	0	0	imm							

Notation:

AND *reg1,reg2,#imm*

Description:

Perform logical AND of the contents of register reg2 and an immediate operand and store result to register reg1.

TST - Test register value

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	reg				0	0	0	0	imm							

Notation:

TST *reg,#imm*

Description:

Conduct logical AND of a register content with an immediate operand value without modifying the register. The operation sets register flags accordingly (i.e. indicating zero / non-zero result).

TST - Test register value

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	reg1				reg2				0	0	0	0	0	0	0	0

Notation:

TST *reg1,reg2*

Description:

Conduct logical AND of the contents of registers reg1 and reg2 without modifying the registers. The operation sets register flags accordingly (i.e. indicating zero / non-zero result).

0x05 opcodes (OR, TST)

OR - Logical OR

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	reg1				reg2				reg3				0	0	0	0	0	0	0	0

Notation:

OR *reg1,reg2,reg3*

Description:

Perform logical OR of the contents of registers reg2 and reg3 and store result to register reg1.

OR - Logical OR

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	reg1				reg2				0	0	0	0	imm							

Notation:

OR *reg1,reg2,#imm*

Description:

Perform logical OR of the contents of register reg2 and an immediate operand and store result to register reg1.

TST - Test

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	0	0	0	0	0	reg				0	0	0	0	0	0	0	0

Notation:

TST *reg,reg*

Description:

Test the value of register operand for zero and set register flags accordingly.

0x06 opcodes (XOR)

XOR - Logical XOR

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	reg1				reg2				reg3				0	0	0	0	0	0	0	0

Notation:

XOR *reg1,reg2,reg3*

Description:

Perform logical XOR of the contents of registers reg2 and reg3 and store result to register reg1.

XOR - Logical OR

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	reg1				reg2				0	0	0	0	imm							

Notation:

XOR *reg1,reg2,#imm*

Description:

Perform logical XOR of the contents of register reg2 and an immediate operand and store result to register reg1.

0x07 opcodes (AND, MOV, MOVZX, MOVHI, BITSET, BITCLR, BITVAL, BITTST)

AND - Logical AND

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	reg1				reg2				1	1	bitnum					0	0	0	0	0

Notation:

AND *reg1, reg2, (1^bitnum-1)*

Description:

Perform logical AND of the contents of registers reg2 and a bitmask denoted by a bitnum operand to register reg1.

MOV - Move to register

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	reg1				reg2				1	1	bitnum					shift				

Notation:

MOV *reg1, (reg2>>shift)&(1^bitnum-1)*

Description:

Shift the contents of registers reg2 to the right by shift bits, and store result number of bits denoted by a bitnum operand to register reg1.

MOV - Move to register

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	reg1				reg2				0	0	bitnum					shift				

Notation:

MOV *reg1, (reg2&(1^bitnum-1))<<shift*

Description:

Shift the lower number of bits denoted by a bitnum operand of register reg2 to the left by shift bits and store the result to register reg1.

MOV - Move to register

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	reg1				reg2				0	1	bitnum					0	0	0	0	0

Notation:

MOV *reg1, reg2&(1^bitnum-1)*

Description:

Move the lower number of bits denoted by a bitnum operand of register reg2 to register reg1.

MOV - Move to register

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	reg1				reg2				0	0	bitnum					0	0	0	0	0

Notation:

MOV *reg1, reg2&(1^bitnum-1)*

Description:

Move the lower number of bits denoted by a bitnum operand of register reg2 to register reg1.

MOV - Move to register

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	reg1				reg2				0	1	0	0	0	0	1	shift				

Notation:

MOV *reg1, reg2&0x01<<shift*

Description:

Shift the lower bit of register reg2 to the left by a shift operand and store the result to register reg1.

MOVZX - Move to register and zero extend

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	reg1				reg2				0	1	0	1	0	0	0	0	0	0	0	0

Notation:

MOVZX *reg1, reg2 &0xff*

Description:

Move the contents of the lower 8 bits of register reg2 to register reg1 and set the remaining bits (bits 8-31) of reg1 to 0.

MOVHI - Move to register high

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	reg1				reg2				0	0	bitnum					1	0	0	0	0

Notation:

MOVHI *reg1, (reg2&(1^bitnum-1))<<16*

Description:

Move bitnum number of lower bits of register reg2 to high 16 bits of register reg1.

MOVHI - Move to register high

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	reg1				reg2				0	0	1	0	0	0	0	1	0	0	0	0

Notation:

MOVHI *reg1, reg2<<16*

Description:

Move 16 lower bits of register reg2 to high 16 bits of register reg1.

BITSET - Bit set

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	reg1				reg2				0	0	0	0	0	0	1	shift				

Notation:

BITSET *reg1, reg2&0x01<<shift*

Description:

Set bit number of register reg1 denoted by a shift operand to the value of bit 0 of register reg2.

BITCLR - Bit clear

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	reg1				0	0	0	0	0	0	0	0	0	0	1	shift				

Notation:

BITCLR *reg1, 0x01<<shift*

Description:

Set bit number of register reg1 denoted by a shift operand to the value of 0.

BITVAL - Get bit value

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	reg1				reg2				1	1	0	0	0	0	1	shift				

Notation:

BITVAL *reg1, reg2, #1<<shift*

Description:

Get the value of a bit number denoted by n operand from register reg2 and store it in register reg1.

BITTST - Bit test

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0	reg				1	1	0	0	0	0	1	shift				

Notation:

BITTST *reg, #1<<shift*

Description:

Test the value of a bit number denoted by a shift operand in register reg.

0x08 opcodes (JMP, J, JZ, JNE, JS, JNS, WAIT)

JMP - Jump register

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	0	0	0	0	0	reg				0	0	0	0	0	0	0	0

Notation:

JMP *reg*

Description:

Unconditionally jump to target location given by the contents of register operand.

J - Always jump to target location

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	0	0	0	0	0	0	target											

Notation:

J *target*

Description:

Unconditionally jump to target location given by an operand.

JZ - Jump if zero

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	0	0	0	0	1	target											

Notation:

JZ *target*

Description:

Jump to target location given by an operand register flags indicate zero result.

JNE - Jump if not equal

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	0	0	0	0	0	1	target											

Notation:

JNE *target*

Description:

Jump to target location given by an operand if register flags indicate non-equal result.

JS - Jump if signed

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	0	0	0	1	0	target											

Notation:

JS *target*

Description:

Jump to target location given by an operand register flags indicate signed result.

JNS - Jump if not signed

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	0	0	0	0	1	0	target											

Notation:

JNS *target*

Description:

Jump to target location given by an operand if register flags indicate non-signed result.

JXX1 - Unknown conditional jump

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	1	1	0	0	1	0	target											

Notation:

JXX1 *target*

Description:

Perform conditional jump based on some unknown condition.

WAIT1 - Wait / perform coprocessor op

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	0	1	0	0	0	1	target											

Notation:

WAIT1

Description:

Wait for some coprocessor result ?

WAIT2 - Wait / perform coprocessor op

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

1	0	0	0	1	1	0	1	1	0	0	0	target

Notation:

WAIT2

Description:

Wait for some coprocessor result ?

0x09 opcodes (JE, JB, JAE, JBE, JNE, JNS, JS, JZS, WAIT)

JE - Jump if equal

23	22	21	20	19	18	17	16	15	14	13	12	11 10 9 8 7 6 5 4 3 2 1 0
1	0	0	1	1	0	0	0	0	0	0	1	target

Notation:

JE *target*

Description:

Jump to target location given by an operand register flags indicate equal result.

JB - Jump if below

23	22	21	20	19	18	17	16	15	14	13	12	11 10 9 8 7 6 5 4 3 2 1 0
1	0	0	1	1	0	0	0	0	1	0	0	target

Notation:

JB *target*

Description:

Jump to target location given by an operand if register flags indicate below result.

JAE - Jump if above or equal

23	22	21	20	19	18	17	16	15	14	13	12	11 10 9 8 7 6 5 4 3 2 1 0
1	0	0	1	1	1	0	0	0	1	0	0	target

Notation:

JAE *target*

Description:

Jump to target location given by an operand if register flags indicate above or equal result.

JBE - Jump if below or equal

<table>
<tr><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td colspan="12">target</td></tr>
</table>

Notation:

JBE *target*

Description:

Jump to target location given by an operand if register flags indicate below or equal result.

JNE - Jump if not equal

<table>
<tr><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td colspan="12">target</td></tr>
</table>

Notation:

JNE *target*

Description:

Jump to target location given by an operand if register flags indicate a non-equal result.

JNS - Jump if not signed

<table>
<tr><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td colspan="12">target</td></tr>
</table>

Notation:

JNS *target*

Description:

Jump to target location given by an operand if register flags indicate a non-signed result.

JS - Jump if signed

<table>
<tr><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td colspan="12">target</td></tr>
</table>

Notation:

JS *target*

Description:

Jump to target location given by an operand if register flags indicate a signed result.

JZS - Jump if zero or signed

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	0	0	0	0	1	1	target											

Notation:

JZS *target*

Description:

Jump to target location given by an operand if register flags indicate a zero or signed result.

JXX2 - Unknown conditional jump

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	1	0	1	0	0	target											

Notation:

JXX2 *target*

Description:

Perform conditional jump based on some unknown condition.

JXX3 - Unknown conditional jump

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	0	1	0	0	0	target											

Notation:

JXX3 *target*

Description:

Perform conditional jump based on some unknown condition.

WAIT3 - Wait / perform coprocessor op

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	1	1	0	0	0	0	1	target											

Notation:

WAIT3

Description:

Wait for some coprocessor result ?

WAIT4 - Wait / perform coprocessor op

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	1	0	1	1	0	0	0	target											

Notation:

WAIT4

Description:

Wait for some coprocessor result ?

0x0a opcodes (LD)

LD - Load from memory

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	reg1				0	0	0	0	reg2				m	imm						

Notation:

LD *reg1,[reg2+imm]*

Description:

Load register operand reg1 with the content of a memory location denoted by register reg2 and an imm index.

If reg2 field equals 0, bit m denotes whether access to DATA (bit value 0) or I/O space (bit value 1) memory region is made.

LD - Load from memory

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	reg1				0	0	0	1	reg2				Imm							

Notation:

LD *reg1,[reg2],reg2+=#imm*

Description:

Load register operand reg1 with the content of a memory location denoted by register reg2 and increment the content of reg2 by an immediate operand.

0x0b opcodes (ST)

ST - Store to memory

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	reg1				reg2				m	Imm						

Notation:

ST *reg1,[reg2+imm]*

Description:

Store the content of register operand reg1 to memory location denoted by register reg2 and an imm index.

If reg2 field equals 0, bit m denotes whether access to DATA (bit value 0) or I/O space (bit value 1) memory region is made.

ST - Store to memory location

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	1	reg1				reg2				Imm							

Notation:

ST *reg1,[reg2],reg2+=#imm*

Description:

Store the content of register operand reg1 to memory location denoted by register reg2 and increment the content of reg2 by an immediate operand.

0x0c opcodes (CMP)

CMP - Compare register value

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	reg				0	0	0	0	imm							

Notation:

CMP *reg,#imm*

Description:

Compare register content with an immediate operand value. The operation sets register flags accordingly.

0x0d opcodes (JMP, BITSRCH, SYNC, RPT)

JMP - Jump to address

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	target hi								0	0	0	1	target lo			

Notation:

JMP *target*

Description:

Unconditionally jump to target location given by an operand.

BITSRCH - Search for bits

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	reg1				reg2				0	0	0	0	0	1	0	0	0	0	0	0

Notation:

BITSRCH TOPMOST *reg1, reg2*

Description:

Search for the first bit set to value 1 in reg2 starting from the topmost bit and store the found bit number in reg1.

SYNC1 - Sync on / perform some coprocessor op

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Notation:

SYNC1

Description:

Synchronize on some coprocessor operation ?

SYNC2 - Sync on / perform some coprocessor op

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0

Notation:

SYNC2

Description:

Synchronize on some coprocessor operation ?

RPT - Repeat

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	n			

Notation:

RPT n

Description:

Repeat execution of a next instruction n times.

RPT - Repeat

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

Notation:

RPT 16

Description:

Repeat execution of a next instruction 16 times.

0x0e opcodes (MOV)

MOV - Move value to register

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	reg				imm															

Notation:

MOV *reg,#imm*

Description:

Move 16-bit immediate operand to given register.

0x0f opcodes (copAES, copTDES)

copAES - AES operation

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Notation:

copAES

Description:

Perform AES crypto coprocessor operation.

copTDES - TDES operation

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Notation:

copTDES

Description:

Perform TDES crypto coprocessor operation.

Further work

The conditional instructions are one of the first candidates for any further work aimed at the improvement of the correctness of the presented instruction's set and SlimCORE disassembler tool's operation.

The reverse engineering of these instructions requires acquiring information about actual conditions (register flags) that are used for a decision about a given conditional jump. This in particular includes information about S (signed result) and C (borrow / carry) conditions.

The operation of the conditional jump instructions based on the above conditions can be reverse engineered with the use of instruction sequences influencing these conditions (register flags). This in particular includes CMP, ADD and SUB instructions:

- JS - jump if signed
```
MOV r1,#0001
MOV r2,#0002
SUB r1,r1,r2,#0000
JS label
```
- JNS - jump if not signed
```
MOV r1,#0001
MOV r2,#0002
SUB r1,r2,r1,#0000
```

```
JNS label
```

- JC - jump if carry

```
MOV r1,#0001
MOV r2,#0002
SUB r1,r1,r2,#0000
JC label

MOV r1,#ffff
MOV r2,#ffff
movhi r1,r2<<16
add r1,r1,r0,#0001
JC label
```

- JNC - jump if not carry

```
MOV r1,#0001
MOV r2,#0002
SUB r1,r2,r1,#0000
JNC label

MOV r1,#0001
MOV r2,#0002
add r1,r1,r0,#0000
JNC label
```

For all of the above instruction sequence, the jump should be taken only if a given target condition is met. The problem with such an approach is that one needs to be careful about conditional jumps that may take multiple conditions into account (carry and zero, signed and zero). For this reason, an observation for a signed an zero results need to be done as well (similarly to the case of signed and carry result, for which the conditions can be distinguished as jump for carry will take place also for non-signed results).

Additionally, opcodes of conditional jump instructions should be also inspected as in most CPU architectures, a target condition is encoded with the use of a dedicated bit field within the instruction opcode.

SlimCORE FIRMWARE

In STi7111 environment, SlimCore runs firmware code implementing access to all crypto related functionality of TKD core (main crypto core of the SoC).

Prior to running the firmware code it is loaded into the memory space of the SlimCORE processor.

The loading process is implemented by the `sttkdma_core_user.ko` device driver and its `st_tkdma_loader` subroutine in particular (Fig. 8).

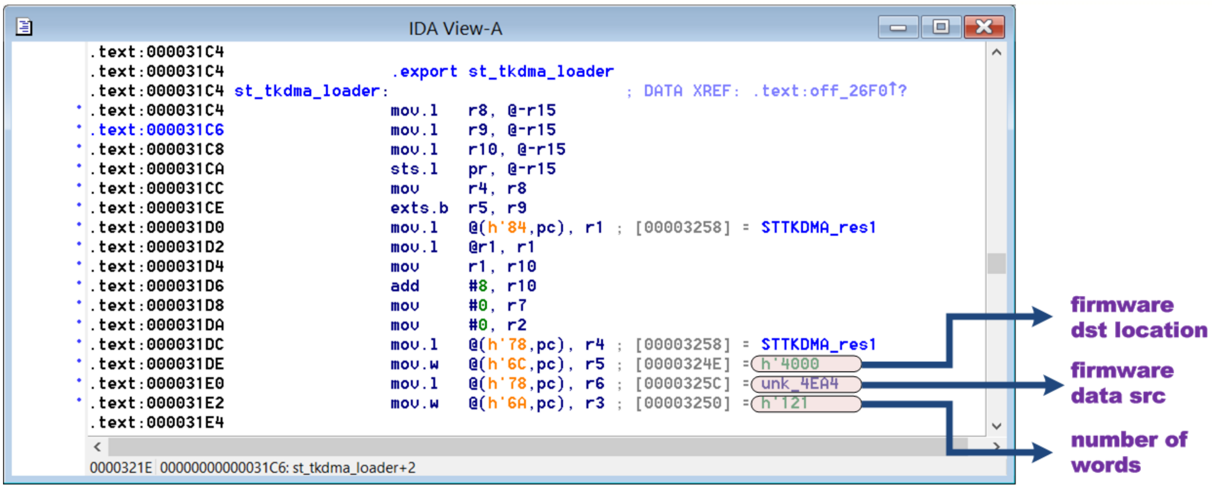


Fig. 8 SlimCORE firmware loading code.

Both data and code sections for the firmware are loaded. The data section usually starts at 0x4000 offset relative to the chip base address. Instruction opcodes start at offset 0x6000. These offsets can be obtained from the implementation of `st_tkdma_check_fw` subroutine (Fig. 9).

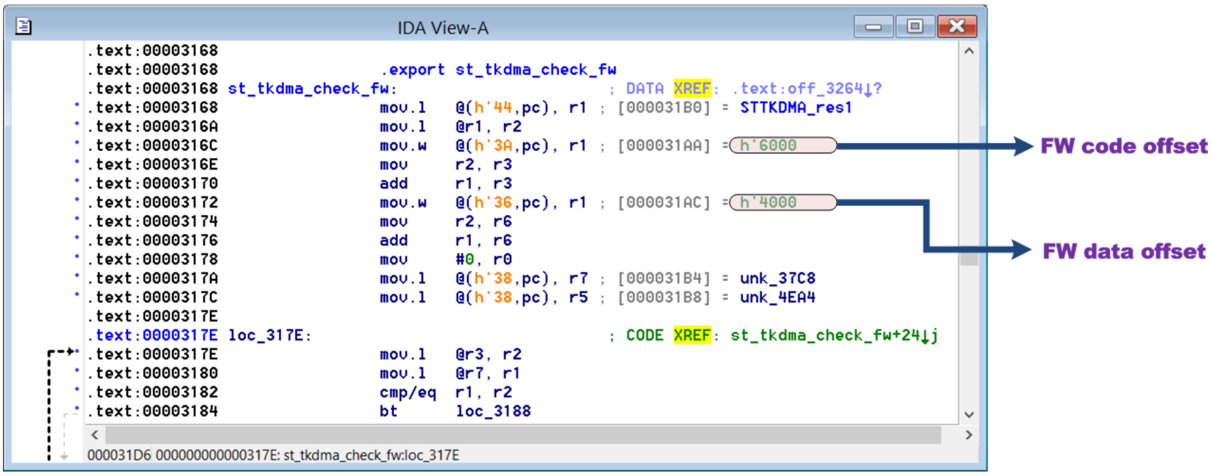


Fig. 9 SlimCORE firmware offsets in chipset memory space.

Locating firmware code and data sections

Inspection of the `sttkdma_core_user.ko` device driver and its `st_tkdma_loader` subroutine is only one of the ways to locate[4] data corresponding to SlimCORE firmware sections.

Data bytes corresponding to firmware code and data sections can be also successfully located by inspecting the code of the following firmware checking subroutines:

- `st_tkdma_check_fw` (information about code start, code end and data start)
- `st_tkdma_loader_checksum` (information about code start, code size, data start and data size)

Below, two other ways are described to achieve this.

[4] and dump the contents of both data and code sections of the firmware.

Magic string and NOP instruction

We have observed that a code section for the firmware starts just behind the `HAL_INT_NAME` symbol of `sttkdma_core_user.ko` device driver (Fig. 10).

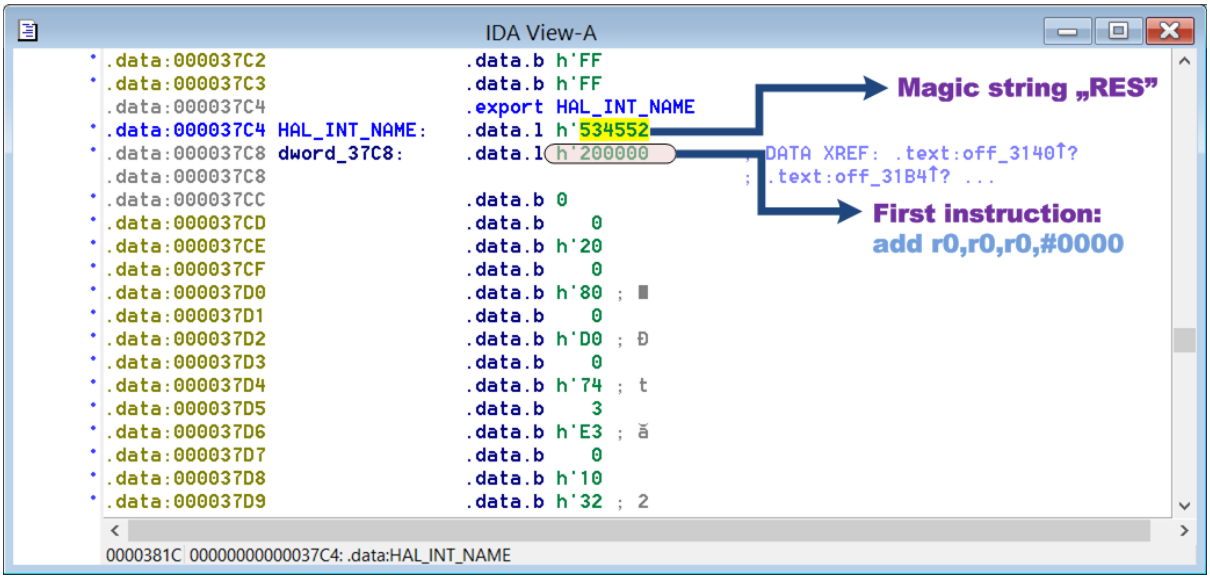


Fig. 10 Firmware code location and a magic string.

This symbol holds a constant 32-bit value of 0x00534552, which corresponds to "RES" string.

Additionally, we have observed that for both old (STTKDMA-REL_3.1.6) and new (STTKDMA-REL_3.9.2) firmware versions, firmware code sequence started with the following instructions:

```
l_0000   0x00200000   add r0,r0,r0,#0000
  0001   0x00200000   add r0,r0,r0,#0000
  0002   0x00d00080   sync
```

The above observations can be used to easily locate the start of a firmware code section in `sttkdma_core_user.ko` device driver. All that is needed to accomplish that is to find a first occurrence of two 32-bit integer values in it (a magic string 0x00534552 and nop[5] instruction 0x00200000).

The end of a code section can be located by exploiting an observation that it always ends with a return from a subroutine instruction and is immediately followed by a firmware data section (its first word equal to 0):

```
0x00840d00   jmp r13        ;jmp to link register (subroutine
                                                return address)
0x00000000                  ;firmware data section start
```

The method described above to locate SlimCORE firmware code and data sections in `sttkdma_core_user.ko` device driver file is implemented in our SCDisasm tool.

[5] `add r0,r0,r0,#0000` can be considered as an equivalent of a nop instruction taking into account that `r0` is a zero register.

Kernel symbols

In some cases, the file of `sttkdma_core_user.ko` device driver might not be immediately available as part of the main root FS file system distribution[6].

For such cases, the image of a device driver along the code of all subroutines necessary to locate SlimCORE firmware need to be obtained from kernel memory by the means of a `/proc` filesystem.

The `/proc/modules` file contains information about dynamically loaded kernel modules, their addresses and sizes. It can be used to obtain the kernel address where `sttkdma_core_user.ko` device driver was loaded:

```
sttkdma_core_user                                        34384                             6
stdrmcrypto_ioctl,stdrmcrypto_core_user,sttkdma_ioctl_local,rfs_sec,nand_crypt,adb_tkdma_ioctl
, Live 0x81931280 (P)
```

The `/proc/kallsyms` file contains information about kernel symbols such as those of dynamically loaded kernel modules:

```
81937494 d HAL_INT_NAME              [sttkdma_core_user]
81932c60 T STTKDMA_Term              [sttkdma_core_user]
81932b00 T STTKDMA_ConfigureTK       [sttkdma_core_user]
819346c0 t sttkdmaHal_GetNonce       [sttkdma_core_user]
8192b380 u STAPLER_InterruptMake     [sttkdma_core_user]
81934080 t sttkdmaHal_ProcessCommand [sttkdma_core_user]
819392cc b sttkdma_ControlBlock_p    [sttkdma_core_user]
81933600 T STTKDMA_DecryptKey        [sttkdma_core_user]
81933480 T STTKDMA_GetCounter        [sttkdma_core_user]
81934e20 t sttkdmaHal_configuretk    [sttkdma_core_user]
81933360 T STTKDMA_ReadPublicID      [sttkdma_core_user]
...
819357a0 t st_tkdma_loader_checksum  [sttkdma_core_user]
...
```

The above information can be used to dynamically extract firmware data and code sections directly from the kernel memory.

Firmware architecture

SlimCORE firmware is responsible for direct access to and interaction with a TKD Crypto core component of STi7111 SoC. The firmware operation is controlled from within the `sttkdma_core_user.ko` device driver through an API interface (Fig. 11).

The API interface is implemented by the means of STK commands and their arguments. They are written to dedicated firmware data locations (0x401c STK cmd, 0x4020-0x402c STK cmd arguments) to trigger proper command dispatch.

[6] this was the case for ITI-2849ST and ITI-2850ST set-top-boxes. The `sttkdma_core_user.ko` device driver file was available as part of ADB loader partition (ADB Loader v7 SSU image, which was successfully decrypted by the means of a custom Hitachi SH4 emulator with I/O proxy [11]).

SlimCore firmware processes STK commands and issues corresponding TKD commands directly to TKD Crypto core. The results of STK commands (if any) are written back to the arguments buffer.

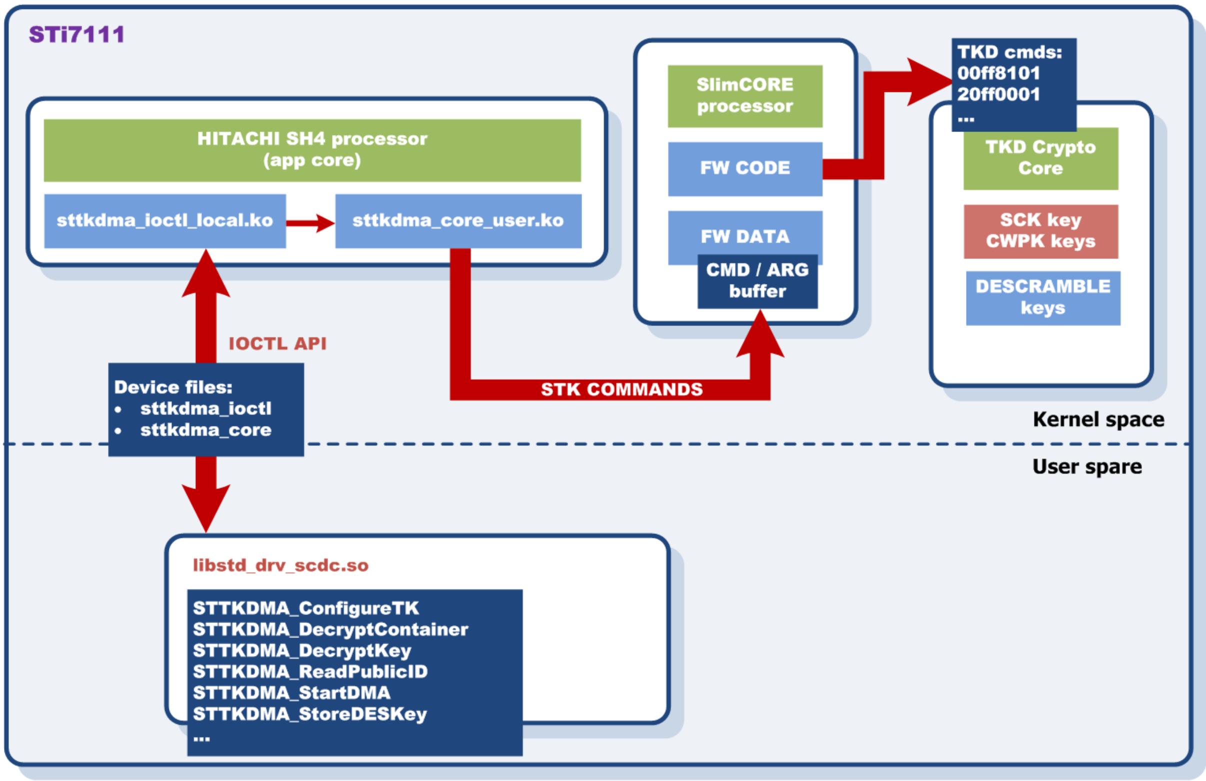


Fig. 11 SlimCORE firmware architecture (associated components and APIs).

STK commands are issued as a response to IOCTL calls received by `sttkdma_ioctl_local.ko` device driver from user space library by the means of special device files[7].

TKD Crypto core

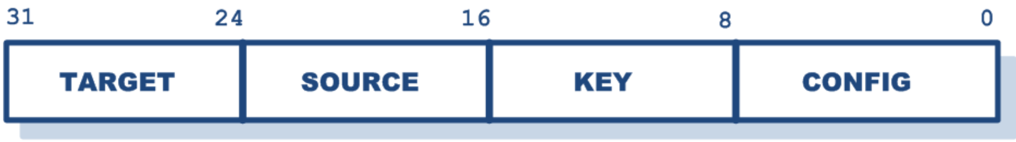
TKD Crypto core is the main core of STi7111 SoC responsible for all cryptographic and key storage related operations. The core is controlled by the means of 32-bit TKD commands and associated arguments being sent to an I/O port.

TKD Crypto Core supports the following ciphers:

- TDES_ECB_128
- AES_ECB_128
- AES_CBC_128
- AES_CTR_128

Generic format of a TKD command is presented on Fig. 12 .

[7] `/tmp/sttkdma_ioctl`, `/tmp/sttkdma_core` for ITI-2849ST and ITI-2850ST set-top-boxes.



TARGET:
- **target, where the result of the operation should be stored**
- **a key slot number or 0xff for chip registers**

SOURCE:
- **source, from which data for the operation should be fetched**
- **a key slot number or 0xff for chip registers,**

KEY:
- **key slot number, which holds the key used for the crypto operation**
- **value 0x00 usually identifies SCK key (unique key for each chip)**

CONFIG:
- **configuration bits**
- **bit 0 usually denotes encryption (0) or decryption (1) operation**

Fig. 12 Generic TKD command format.

TKD commands make it possible to store a given source value to a given target key memory location. Depending on the chip configuration, the source value can be encrypted or decrypted[8] with the use of a given key. As a result, TKD commands provide means for a secure loading of secret key values into the chip.

The following TKD commands are usually at the base of an implementation of an arbitrary PayTV CAS with chipset pairing functionality:

- Setting encrypted Control Word Pairing Key (CWPK)
  - TKD CMD 0x00ff0000
  - Interpreted as decryption (always) of register input (0xff) with SCK key (0x00) and storing the result at a key slot 0x00
- Setting encrypted Control Word (CW)
  - TKD CMD 0x20ff0001
  - Interpreted as decryption (0x01) of register input (0xff) with CWPK key (key slot 0x00) and storing the result at a key slot 0x20.

It's worth to mention that for targets in the range of 0x00-0x04, there is no output provided as a result of a given TKD command execution (secret pairing key locations). Such an output is however provided for targets 0x05-0x0f.

Beside making it possible to load encrypted key values to the chip, TKD crypto core also implements commands facilitating crypto DMA operations. Their generic format for standard DMA (making use of user provided crypto keys) is presented on Fig. 13.

[8] more details pertaining to decryption / encryption bit of TKD command and observed peculiarities can be found in APPENDIX A.

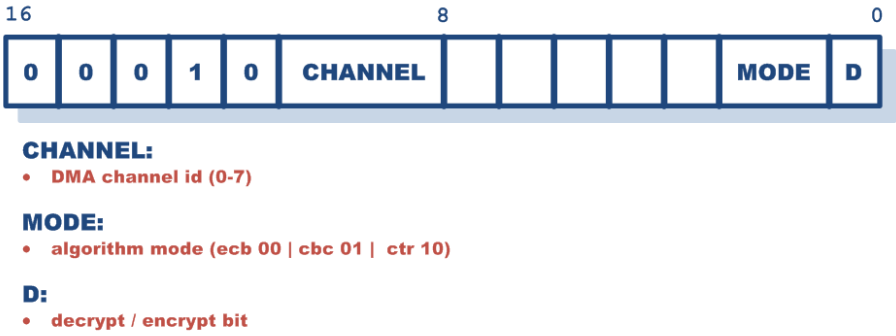


Fig. 13 Lower 16 bits of a TKD command for standard DMA operation.

TKD command corresponding to DMA crypto transfer making use of the SCK key is presented on Fig. 14.

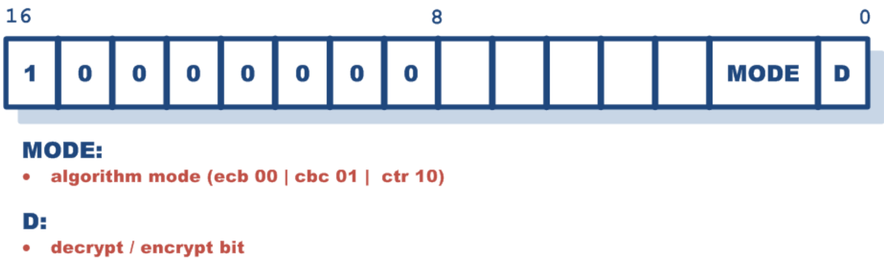


Fig. 14 Lower 16 bits of a TKD command for SCK DMA operation.

The higher 16 bits (bits 16-31) of the above crypto DMA commands are set to the value 0xffff.

Finally, TKD Crypto core maintains dedicated memory locations for arbitrary key storage:

- 0x3100 - descrambling keys (keys 0-31, key size 0x10)
- 0x3420 - crypto DMA / custom user keys (keys 0-7 corresponding to given DMA channel id, key size 0x10).

Memory locations corresponding to descrambler keys are not readable, while the area corresponding to DMA / custom user keys can be read by user code. Key at index *N* corresponds to DMA channel *N*.

For CBC and CTR based ciphers, the following I/O register locations are also used:

- 0x3004-0x3010 - CBC IV vector
- 0x3014-0x3020 - CTR IV vector

Commands and configuration variables

There are more than a dozen of STK commands implemented by the `sttkdma_core_user.ko` device driver, which correspond to different TKD commands issued to TKD Crypto core. The mapping of STK commands to their TKD counterparts is shown in Table 2.

ASSOCIATED NAME[9]	STK CMD LOCATION[10]	STK CMD	TKD COMMAND

[9] these names do not necessarily correspond to the `sttkdma_core_user.ko` device driver symbols, but are all the symbols that could be associated with given STK commands through other device drivers and user space libraries.

STTKDMA_reset	0x4068	0x00	
	0x406c	0x01	01ff8101
setCWPK / set_descrambling_internalkeys	0x4070	0x02	00ff8101
STTKDMA_DecryptKey / scdc_ImplModifyKeyIndex / set_protected_descramblingkey	0x4074	idx[11]<<8 | 0x03	20ff0001 + idx<<24
	0x4078	idx<<8 | 0x04	10ff0101 + idx<<24
getPublicID	0x407c	0x05	
	0x4080	idx<<8 | 0x06	20ff0010 + idx<<24
	0x4084	idx<<8 | 0x80	10ff8001 + idx<<24
	0x4088	0x10	03ff0001
	0x408c	0x11	04000001
sttkdmaHal_GetNonce	0x4090	0x12	ffff0401
resetAES_NOT_TDES		0x13	
	0x4094	0x20	02ff8101
	0x4098	0x21	80ff0203
	0x409c	0x22	81ff0203
	0x40a0	0x23	82ff0203
sttkdmaHal_GetSWReg	0x40a4	0x24	83ff0203
STTKDMA_GetCounter	0x4068	0x40	
STTKDMA_NOP	0x4068	0x41	

Table 2 The mapping of STK commands to TKD commands.

SlimCORE firmware reads STK commands and their optional arguments from the following SlimCORE data section locations:

```
0x401C          STK CMD ID
0x4020-0x402c   STK CMD buffer for arguments and output result
```

Additionally, SlimCORE firmware makes an active use of several other data section locations for storage of various configuration and state settings. This is illustrated in Table 3.

FIRMWARE DATA SECTION OFFSET	VARIABLE DESCRIPTION
0x4004	DMA CONFIG ▪ 0x01 container DMA ▪ 0x02 decrypt ▪ 0x04-0x10 channel id (0-7) ▪ 0x20 AES algorithm ▪ 0x40 SCK dma ▪ 0x80 custom DMA cmd ▪ 0x100 CBC mode ▪ 0x200 CTR mode ▪ 0x400 IV seed ▪ 0x800 swap_halves ▪ 0x1000 IV init? ▪ 0x2000 swap_bytes
0x4008	DMA source (aligned to 0x20)

[10] in SlimCORE firmware.
[11] idx denotes key index.

0x400c	DMA destination (aligned to 0x20)
0x4010	DMA size (in 32-bit words)
0x4014	part of STK command
0x4018	TK CONFIG
0x401C	STK cmd
0x4020-0x402c	STK cmd buffer (arguments / result)
0x4030	Customer mode
0x4040	state flag indicating STK cmd 0x01 was executed (checked by STK cmd 0x04)
0x4044	state flag indicating STK cmd 0x02 was executed (checked by STK cmds 0x03, 0x10 and 0x11)
0x4048	state flag indicating STK cmd 0x05 was executed (checked by STK cmds 0x01, 0x02, 0x04 and 0x80) TKD operation mode: ▪ 0x01 tkd is active ▪ 0x02 dma is active
0x404c	state flag indicating STK cmd 0x10 was executed (checked by STK cmd 0x03)
0x4050	state flag indicating STK cmd 0x11 was executed (checked by STK cmd 0x12)
0x4054	state flag indicating STK cmd 0x20 was executed (checked by STK cmds 0x21, 0x22, 0x23 and 0x24)
0x40b0	SW counter
0x40b4	number of packets for DMA transfer
0x4120	bit idx of current stack frame
0x4124	bit idx of next stack frame

Table 3 SlimCORE firmware configuration / state variables.

Firmware operation

Generic schema of a SlimCORE firmware operation is illustrated on Fig. 15.

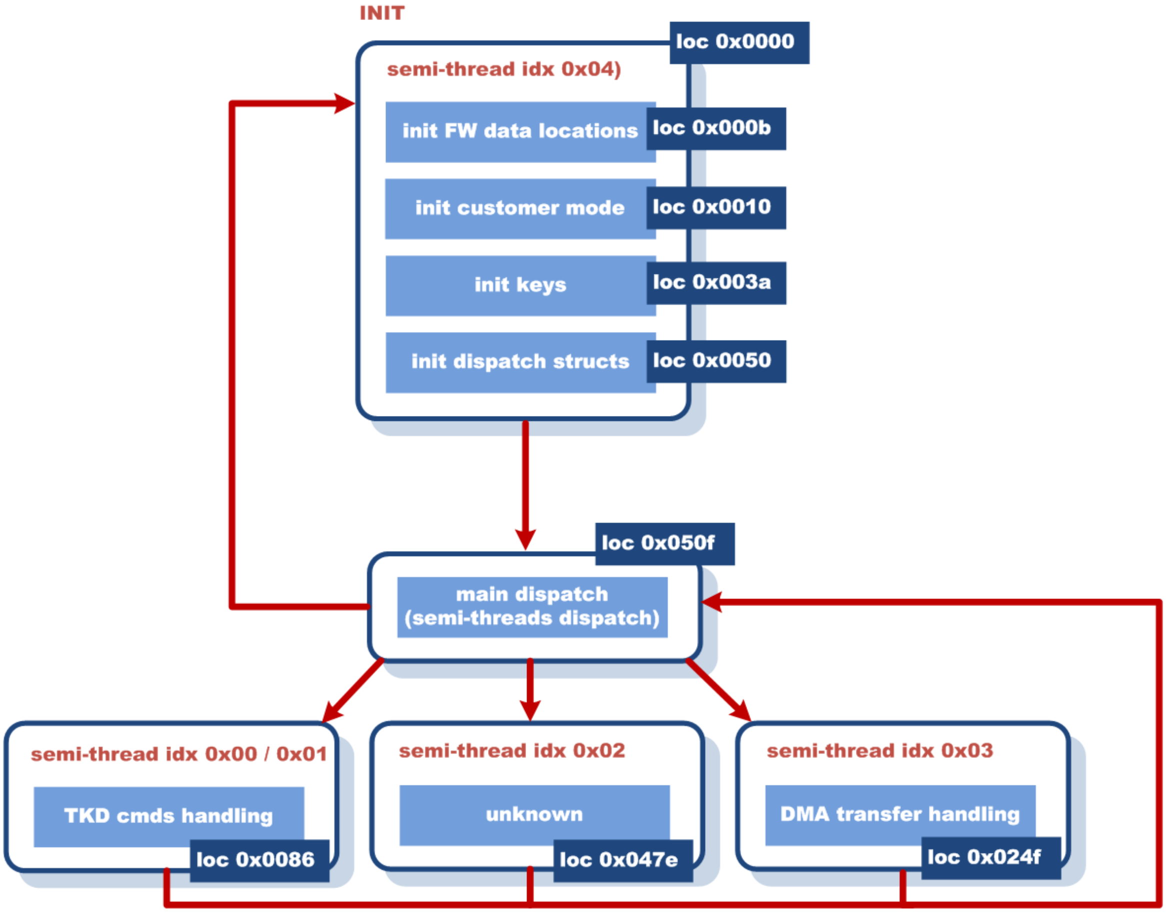


Fig. 15 SlimCORE firmware operation (STTKDMA-REL_3.1.6).

Execution of a SlimCORE firmware starts at instruction idx 0. First, some FW data locations are initialized to 0 such as a counter variable:

```
  000b  0x00b0002c   st r0,[r0,002c]         ;counter = 0
  000c  0x00e60010   mov r6,#0010            ;memory idx of 0x4040 addr
  000d  0x00

d00090   sync
  000e  0x00d00009   rpt 9                   ;loop counter=9
  000f  0x00b10601   st r0,[r6],r6+=#0001    ;store 0 to [0x4040-0x4060]
```

After that, chip customer mode register is read and a corresponding data section variable is initialized with a new value:

```
  0010  0x00a5008a   ld r5,[r0,008a] // 0x5e28 ;chip customer mode register
  0011  0x00e40040   mov r4,#0040
  0012  0x00735c80   and r3,r5,0x0f          ;low nibble of chip customer mode
  0013  0x00c03005   cmp r3,#05
  0014  0x00981026   je l_0026                ;-> chip customer mode == 0x05
  0015  0x00c03002   cmp r3,#02
  0016  0x00981028   je l_0028                ;-> chip customer mode == 0x02
  0017  0x00c03006   cmp r3,#06
  0018  0x0098102a   je l_002a                ;-> chip customer mode == 0x06
  0019  0x00c0300b   cmp r3,#0b
```



```
  001a   0x0098102c   je l_002c                  ;-> chip customer mode == 0x0b
  001b   0x00c0300f   cmp r3,#0f
  001c   0x0098102e   je l_002e                  ;-> chip customer mode == 0x0f
  001d   0x00c03003   cmp r3,#03
  001e   0x00981030   je l_0030                  ;-> chip customer mode == 0x03
  001f   0x00c03007   cmp r3,#07
  0020   0x00981032   je l_0032                  ;-> chip customer mode == 0x07
  0021   0x00c03008   cmp r3,#08
  0022   0x00981034   je l_0034                  ;-> chip customer mode == 0x08
  0023   0x00c0300c   cmp r3,#0c
  0024   0x00981036   je l_0036                  ;-> chip customer mode == 0x0c
  0025   0x00d00318   jmp l_0038

l_0026   0x00e40002   mov r4,#0002               ;05 -> 0x02 as customer mode
  0027   0x00d00318   jmp l_0038

l_0028   0x00e40004   mov r4,#0004               ;02 -> 0x04 as customer mode
  0029   0x00d00318   jmp l_0038

l_002a   0x00e40005   mov r4,#0005               ;06 -> 0x05 as customer mode
  002b   0x00d00318   jmp l_0038
l_002c   0x00e40008   mov r4,#0008               ;0b -> 0x08 as customer mode
  002d   0x00d00318   jmp l_0038
l_002e   0x00e40009   mov r4,#0009               ;0f -> 0x09 as customer mode
  002f   0x00d00318   jmp l_0038
l_0030   0x00e40010   mov r4,#0010               ;03 -> 0x10 as customer mode
  0031   0x00d00318   jmp l_0038
l_0032   0x00e40011   mov r4,#0011               ;07 -> 0x11 as customer mode
  0033   0x00d00318   jmp l_0038
l_0034   0x00e40020   mov r4,#0020               ;08 -> 0x20 as customer mode
  0035   0x00d00318   jmp l_0038
l_0036   0x00e40021   mov r4,#0021               ;0c -> 0x21 as customer mode
  0037   0x00d00090   sync
l_0038   0x00b0400c   st r4,[r0,000c] // 0x4030 ;store customer mode
```

Next, a subroutine call is made to initialize TKD Crypto key storage to default key values[12]:

```
  0039   0x00ed003b   mov r13,#003b             ;subroutine return addr
  003a   0x008c04e1   j l_04e1                  ;init all of the keys (CWPK, CWs)
  003b   0x00e40312   mov r4,#0312
```

The call above is made with the use of a `J` (jump to target location) instruction. Prior to it, the LINK register (r13) is loaded with a subroutine return value indicating the instruction following the jump.

Finally, dispatch structures corresponding to several semi-threads implemented by the firmware code are initialized. As part of the initialization procedure, memory for threads' saved context gets allocated and an address of a dispatch address for a given thread is placed into it:

```
  0050   0x00e60080   mov r6,#0080           ;base addr of temp stack frames
  0051   0x00e700d0   mov r7,#00d0
  0052   0x00e4024f   mov r4,#024f           ;thread code location (dispatch handler)
  0053   0x00e50008   mov r5,#0008           ;thread dispatch bitmask=0x08
  0054   0x00d55040   bitsrch topmost,r5,r5 ;thread dispatch idx=3 (bit# of 0x08)
  0055   0x00155004   shl r5,r5,#0004        ;thread dispatch idx*16=0x30
  0056   0x002e5700   add r14,r5,r7,#0000    ;0xd0+0x30=0x100
```

[12] implementation details of a key initialization subroutine (`04e1`) are presented in the following subparagraph of this paper.

```
  0057   0x00255600   add r5,r5,r6,#0000    ;0x80+0x30=0xb0
  0058   0x00d00090   sync
  0059   0x00b0450d   st r4,[r5,000d]       ;[0xbd] = 0x024f (thread handler)
  005a   0x00b0e50e   st r14,[r5,000e]      ;[0xbe] = 0x100  (thread stack frame)
  005b   0x00d0000d   rpt d                 ;init saved registers (r2-r14) to 0
  005c   0x00b10501   st r0,[r5],r5+=#0001  ;[0xb0-0xbc] = 0
```

In SlimCORE firmware, different threads are frequently represented by consecutive bits of a bitmask (thread idx 0 is represented by bit value 0x01, thread idx 1 is denoted by bit value 0x02 and so on). This is also the case for the above (thread dispatch bitmask 0x08 indicates thread dispatch idx 0x03).

Semi-threads dispatching

SlimCORE firmware makes use of 4 semi-threads (dispatch indices 0-3) dedicated for the handling of TKD commands, crypto DMA and firmware initialization procedure among others.

There are two data section variables that indicate current's thread to execute (dispatch):

- 0x4120 - current thread bitmask idx
- 0x4124 - next thread bitmask idx

Upon completing the initialization code, main dispatch subroutine responsible for semi-threads execution is invoked. This subroutine first stores execution context of a currently executing semi-thread:

```
  0512   0x00a10048   ld r1,[r0,0048] // 0x4120   ;current thread's bitmask idx
  0513   0x00d11040   bitsrch topmost,r1,r1       ;current thread's dispatch idx
  0514   0x00111004   shl r1,r1,#0004             ;thread dispatch idx*16
  0515   0x00ea0080   mov r10,#0080               ;base addr of temp stack frames
  0516   0x00211a00   add r1,r1,r10,#0000         ;r1=thread's stack frame
  ...
  0519   0x00b02102   st r2,[r1,0002]             ;save r2
  051a   0x00b03103   st r3,[r1,0003]             ;save r3
  051b   0x00b04104   st r4,[r1,0004]             ;save r4
  051c   0x00b05105   st r5,[r1,0005]             ;save r5
  051d   0x00b06106   st r6,[r1,0006]             ;save r6
  051e   0x00b07107   st r7,[r1,0007]             ;save r7
  051f   0x00b08108   st r8,[r1,0008]             ;save r8
  0520   0x00b09109   st r9,[r1,0009]             ;save r9
  0521   0x00b0a10a   st r10,[r1,000a]            ;save r10
  0522   0x00b0b10b   st r11,[r1,000b]            ;save r11
  0523   0x00b0c10c   st r12,[r1,000c]            ;save r12
  0524   0x00b0d10d   st r13,[r1,000d]            ;save r13 (thread's ret addr)
  ...
  0526   0x00b0e10e   st r14,[r1,000e]            ;save r14 (thread's stack)
```

The dispatch of different threads is done by rotating the current thread's bitmask idx variable over a bit field of 4 bits (firmware data section at offset 0x4124):

```
l_0581   0x00a40049   ld r4,[r0,0049] // 0x4124      ;current thread bitmask idx
  ...
l_0585   0x00404300   tst r4,00
  0586   0x00881589   jz l_0589
  0587   0x00b04048   st r4,[r0,0048] // 0x4120     ;next thread bitmask idx
  ...
l_0589   0x00c04010   cmp r4,#10                     ;is bitmask idx == 0x10 ?
```

```
  058a  0x009c158d   jne,s l_058d                    ;-> no
  058b  0x00e40001   mov r4,#0001                    ;yes, start from bitmask 0x01
  058c  0x00d0581e   jmp l_058e

l_058d  0x00144001   shl r4,r4,#0001                 ;shift bitmask idx by 1 to
                                                     ;the left
  ...
  058f  0x00b04049   st r4,[r0,0049] // 0x4124       ;store new thread bitmask idx
```

The effect of the above becomes visible when thread's execution context gets restored by the main threads dispatching subroutine:

```
  05a2  0x00a20048   ld r2,[r0,0048] // 0x4120       ;next thread's bitmask idx
  05a3  0x00d22040   bitsrch topmost,r2,r2           ;next thread's dispatch idx
  05a4  0x00122004   shl r2,r2,#0004                 ;thread idx*16
  05a5  0x00ea0080   mov r10,#0080                   ;base addr of tmp stack frames
  05a6  0x00222a00   add r2,r2,r10,#0000             ;r2=thread's stack frame
  05a7  0x00d00090   sync
  05a8  0x00ae020e   ld r14,[r2,000e] // 0x0038      ;load r14 (thread's stack)
  05a9  0x00ad020d   ld r13,[r2,000d] // 0x0034      ;load r13 (thread's ret addr)
  05aa  0x00ac020c   ld r12,[r2,000c] // 0x0030      ;load r12
  05ab  0x00ab020b   ld r11,[r2,000b] // 0x002c      ;load r11
  05ac  0x00aa020a   ld r10,[r2,000a] // 0x0028      ;load r10
  05ad  0x00a90209   ld r9,[r2,0009] // 0x0024       ;load r9
  05ae  0x00a80208   ld r8,[r2,0008] // 0x0020       ;load r8
  05af  0x00a70207   ld r7,[r2,0007] // 0x001c       ;load r7
  05b0  0x00a60206   ld r6,[r2,0006] // 0x0018       ;load r6
  05b1  0x00a50205   ld r5,[r2,0005] // 0x0014       ;load r5
  05b2  0x00a40204   ld r4,[r2,0004] // 0x0010       ;load r4
  05b3  0x00a30203   ld r3,[r2,0003] // 0x000c       ;load r3
  05b4  0x00a10201   ld r1,[r2,0001] // 0x0004       ;load r1
  05b5  0x00a20202   ld r2,[r2,0002] // 0x0008       ;load r2
  05b6  0x00840d00   jmp r13                         ;continue execution in a new
                                                     ;thread context
```

STK commands' groups

The thread responsible for main STK command dispatch makes sure that certain commands are executed following an execution of some other commands. This state-machine is implemented by the means of state variables 0x4040-0x4054 (Table 3). This information makes it possible to associate certain TKD commands with each other (select their groups). The meaning of the commands can be also discovered upon the knowledge of the operation of a dependant commands (a prior command required to be executed). The results of such a grouping and a discovery of some unknown commands meaning is illustrated in Table 4.

COMMAND GROUP	STK COMMAND	TKD COMMAND	DESCRIPTION
CWPK1	0x01	01ff8101	Decrypt CWPK input with SCK key (key location 0x81[13]) and store it at key location 1
	idx<<8 | 0x04	10ff0101 + idx<<24	Decrypt key input with CWPK key at index 1 and store it at key location 10+idx (crypto DMA / AES keys)

[13] we verified that correct CWPK key at index 0 can be successfully set for the following TKD commands: 0x00ff8101, 0x00ff0101 and 0x00ff0001. Thus, we conclude that location 0x81 corresponds to SCK key.

CWPK0	0x02	00ff8101	Decrypt CWPK input with SCK key (key location 0x81) and store it at key location 0
	idx<<8 | 0x03	20ff0001 + idx<<24	Decrypt key input with CWPK key at index 0 and store it at key location 20+idx (descrambling keys)
	0x10	03ff0001	Decrypt key input with CWPK key at index 0 and store it at key location 3
	0x11	04000001	Decrypt CWPK key at index 0 with itself and store it at key location 4
TKD	0x05		Get public ID
	0x01	01ff8101	Decrypt CWPK input with SCK key (key location 0x81) and store it at key location 1
	0x02	00ff8101	Decrypt CWPK input with SCK key (key location 0x81) and store it at key location 0
	idx<<8 | 0x04	10ff0101 + idx<<24	Decrypt key input with CWPK key at index 1 and store it at key location 10+idx (crypto DMA / AES keys)
	idx<<8 | 0x80	10ff8001 + idx<<24	Decrypt key input with key at location 0x80 and store it at key location 10+idx (crypto DMA / AES keys)
UNKNOWN	0x10	03ff0001	Decrypt key input with CWPK key at index 0 and store it at key location 3
	idx<<8 | 0x03	20ff0001 + idx<<24	Decrypt key input with CWPK key at index 0 and store it at key location 20+idx (descrambling keys)
NONCE	0x11	04000001	Decrypt CWPK key at index 0 with itself and store it at key location 4 (NONCE)
	0x12	ffff0401	Decrypt key input with key at index 4 (NONCE)
SWREGS	0x20	02ff8101	Decrypt key input with SCK key (key location 0x81) and store it at key location 2
	0x21	80ff0203	??
	0x22	81ff0203	??
	0x23	82ff0203	??
	0x24	83ff0203	??

Table 4 STK commands groups and their description.

Core routines related to CWPK and CWs handling

Below, a more detailed description pertaining to keys handling related functionality implemented by the SlimCORE firmware is presented. This functionality is implemented by TKD commands handling thread.

Key initialization routine

Key initialization subroutine is called at the time of a firmware startup. At first, customer mode is checked for bit 0x40. If this bit is set, no keys are being initialized:

```
########################
SUB l_04e1
init keys
########################
```

```
l_04e1   0x000c0d3c   mov r12,r13
  04e2   0x00a7000c   ld r7,[r0,000c] //          ;customer mode
  04e3   0x00407040   tst r7,40
  04e4   0x009c14fc   jne,s l_04fc                ;-> jump to the end
```

In the next step, register r9 is set to the value 0 to indicate TDES cipher algorithm (a default cipher). If bit 0x02 of customer mode variable is set, the default cipher is changed to the value 1 (AES algorithm):

```
  04e5   0x0009003c   mov r9,r0                   ;r9 = 0 (TDES)
  04e6   0x00e10001   mov r1,#0001
  04e7   0x00407002   tst r7,02
  04e8   0x008814ea   jz l_04ea
  04e9   0x0009013c   mov r9,r1                   ;r9 = 1 (AES)
```

Following that, Control Words Pairing Key (CWPK) is initialized. This is accomplished by invoking a single crypto key initialization subroutine (location `04fd`) with register r8 indicating TKD Crypto core command to execute and r9 denoting the cipher. For CWPK key the TKD command is set to `0x00ff8101` value:

```
l_04ea   0x00a8001c   ld r8,[r0,001c] // 0x4070 = 0x00ff8101         ;setCWPK
  04eb   0x00ed04ed   mov r13,#04ed                                 ;sub ret addr
  04ec   0x008c04fd   j l_04fd                                      ;init single key
```

Next, customer mode is checked for bit value 0x20. If this bit is set, additional (pairing?) key initialization takes place with the use of a 0x03ff0001 TKD command:

```
  04ed   0x00407020   tst r7,20
  04ee   0x008814f2   jz l_04f2
  04ef   0x00a80022   ld r8,[r0,0022] // 0x4088 = 0x03ff0001         ;TKD CMD
  04f0   0x00ed04f2   mov r13,#04f2
  04f1   0x008c04fd   j l_04fd
```

Finally, all descrambling (Control Words) keys are initialized in a loop:

```
l_04f2   0x00e60032   mov r6,#0032                                ;number of CWs
  04f3   0x00e50020   mov r5,#0020                                ;base for TKD cmd
l_04f4   0x00a8001d   ld r8,[r0,001d] // 0x4074 = 0x20ff0001
  04f5   0x00785118   mov r8,r5&0xff<<24                          ;set highest byte
                                                                  ;in TKD cmd
  04f6   0x00ed04f8   mov r13,#04f8                               ;sub return addr
  04f7   0x008c04fd   j l_04fd                                    ;init single key
  04f8   0x00255001   add r5,r5,r0,#0001                          ;inc key idx
  ...
  04fa   0x00366001   sub r6,r6,r0,#0001                          ;dec loop counter
  04fb   0x008c14f4   jne l_04f4                                  ;-> loop jump if
                                                                  ;   counter not 0
```

The loop above initializes 0x32 descrambling keys (CWs).

Initialization of a single crypto key is implemented by the following subroutine:

```
########################
SUB l_04fd
initialization of a single crypto key
INPUT: r9 = 1 for AES, = 0 for TDES
       r8 = TKD command
########################
l_04fd   0x00409900   tst r9,00                                ;AES ?
  04fe   0x008c1506   jne l_0506                               ;-> jump for AES
```

```
  04ff   0x00fa4000   copTDES                                 ;handle TDES
  0500   0x000f083c   mov r15,r8                              ;TKD CMD -> OUT
  0501   0x008e1501   wait1
  0502   0x00d00004   rpt 4
  0503   0x000f003c   mov r15,r0                              ;rpt 4 r0 -> OUT
  0504   0x008e1504   wait1
  0505   0x008c050c   j l_050c

l_0506   0x00f54000   copAES                                  ;handle AES
  0507   0x000f083c   mov r15,r8                              ;TKD CMD -> OUT
  0508   0x008d8508   wait2
  0509   0x00d00004   rpt 4
  050a   0x00af0000   ld r15,[r0,0000] // 0x4000 = 0x00000000 ;rpt 4 [0x4000] -> OUT
  050b   0x008d850b   wait2

l_050c   0x00d00004   rpt 4                                   ;handle output result
  050d   0x00000f3c   mov r0,r15                              ;rpt 4 r0 < IN
  050e   0x00840d00   jmp r13
```

Initialization of a single crypto key is conducted in a similar way for both AES and TDES cipher. First, a coprocessor instruction corresponding to an argument in register r9 is executed indicating target crypto operation to perform. Then TKD command is sent to TKD core (OUT operation) through register r15. For TDES, it is followed by 4 consecutive out operations of 0 value. For AES, the 4 consecutive out operations are conducted with respect to the contents of firmware location 0x4000. The result of the key loading operation is always ignored (moved to r0).

`getPublicID` implementation

The `getPublicID` code has the following implementation:

```
l_01a1   0x00a5008b   ld r5,[r0,008b] // 0x5e2c                ;chip id
  01a2   0x00a9001f   ld r9,[r0,001f] // 0x407c = 0x00000000   ;TKD CMD = 0
  01a3   0x00b05008   st r5,[r0,0008] // 0x4020 = 0x00000000   ;DATA[0] = chip id
  01a4   0x00b00009   st r0,[r0,0009] // 0x4024 = 0x00000000   ;DATA[4] = 0
  01a5   0x00b0000a   st r0,[r0,000a] // 0x4028 = 0x00000000   ;DATA[8] = 0
  01a6   0x00b0000b   st r0,[r0,000b] // 0x402c = 0x00000000   ;DATA[c] = 0
```

The hardware value indicating chip ID is stored into the first word of STK command arguments buffer. It is followed by 3 consecutive store operations of 0 value.

`decryptKey` implementation

The `decryptKey` code sequence is responsible for loading encrypted crypto key values such as CWPK and CWs into TKD Crypto core. The code for this functionality is implemented as part of STK commands handling thread. Below, a more detailed description of TDES based implementation is given:

```
l_0206   0x00fa4000   copTDES                          ;TDES handling
  0207   0x000f093c   mov r15,r9                       ;TKD CMD -> OUT
  0208   0x008e1208   wait1
```

In the beginning, the TKD core is configured to operate in TDES mode. Then TKD command is sent to TKD core (OUT operation) through register r15. For descrambling key at index idx, the TKD command has the value of:


```
0x20ff0001+(idx<<24)
```

Finally, the encrypted key value contained in STK cmd buffer is sent to the TKD core.

```
  0209  0x00af0008  ld r15,[r0,0008] // 0x4020        ;DATA[0] -> OUT
  020a  0x00af0009  ld r15,[r0,0009] // 0x4024        ;DATA[4] -> OUT
  020b  0x00af000a  ld r15,[r0,000a] // 0x4028        ;DATA[8] -> OUT
  020c  0x00af000b  ld r15,[r0,000b] // 0x402c        ;DATA[c] -> OUT
  020d  0x008e120d  wait1
```

Next, register r10 indicating whether current TKD command has output is checked:

```
  020e  0x00500a00  tst r10,r10                   ;does this command have output ?
  020f  0x00881215  jz l_0215                     ;-> jump in case of no output
```

If a command has output, it is simply read via register r15 and stored into STK cmd buffer (IN operation):

```
  0210  0x00b0f008  st r15,[r0,0008] // 0x4020    ;DATA[0] <- IN
  0211  0x00b0f009  st r15,[r0,0009] // 0x4024    ;DATA[4] <- IN
  0212  0x00b0f00a  st r15,[r0,000a] // 0x4028    ;DATA[8] <- IN
  0213  0x00b0f00b  st r15,[r0,000b] // 0x402c    ;DATA[c] <- IN
  0214  0x00d02117  jmp l_0217
```

If register r10 indicates no output, the result of a key loading operation is always ignored (moved to r0):

```
l_0215  0x00d00004  rpt 4                       ;read output buffer, but ignore it
  0216  0x00000f3c  mov r0,r15                  ;r0 <- IN
```

For AES cipher, the sequence of instructions implementing `decryptKey` functionality is similar to the one of TDES cipher. There is however one difference. Following the OUT operation of a TKD command, there is a check for bit 0x08 of a firmware variable at 0x41c0 location:

```
  01ed  0x00a30070  ld r3,[r0,0070] // 0x41c0 = 0x00000070
  01ee  0x008d81ee  wait2
  01ef  0x00703c23  tst r3,#00000008
  01f0  0x008811f6  jz l_01f6
```

If this bit is set, instead of sending user provided (from STK cmd buffer location) arguments to the chip, data from some I/O locations is used for that purpose:

```
  01f1  0x00af0090  ld r15,[r0,0090] // 0x5e40      ;[0x5e40] -> OUT
  01f2  0x00af0091  ld r15,[r0,0091] // 0x5e44      ;[0x5e44] -> OUT
  01f3  0x00af0092  ld r15,[r0,0092] // 0x5e48      ;[0x5e48] -> OUT
  01f4  0x00af0093  ld r15,[r0,0093] // 0x5e4c      ;[0x5e4c] -> OUT
```

setCWPK implementation

The implementation of `setCWPK` makes use of the described above `decryptKey` functionality. For `setCWPK`, target TKD command is set to the value of `0x00ff8101`.

In Conax CAS environment, the value of CWPK key is passed to a set-top-box device by the means of a dedicated EMM message. We have observed that smartcard's response to it always starts with the same sequence of 6 bytes:

80 1b 40 19 01 17

The response to EMM message contains a TLV value and UPDATE_KEY tag in particular. The latter embeds pairing information in a form of a public chip ID and an encrypted CWPK key. This is illustrated on Fig. 16.

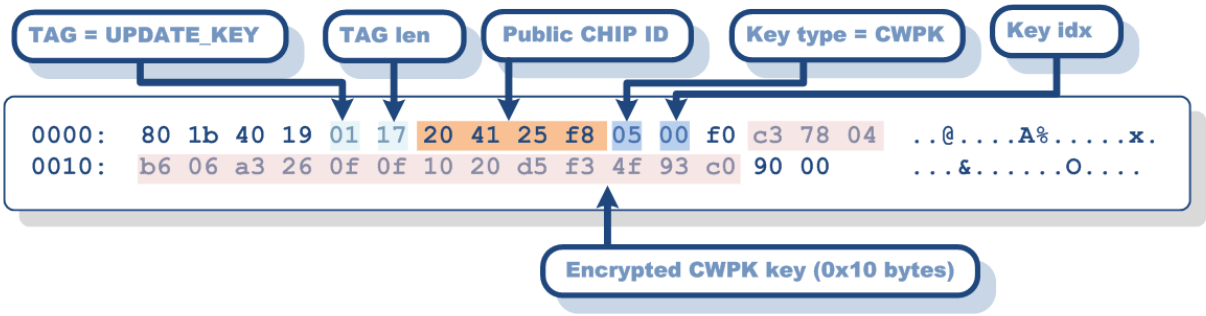


Fig. 16 A response to Conax CAS EMM message carrying chipset pairing information.

ADB set-top-boxes additionally encrypt the received encrypted pairing key and store it in a local file[14]. This is most likely for the purpose of a quick STB startup (no need to wait for a reception of a pairing key over the broadcast stream).

The additional encryption of CWPK should not be perceived in terms of a security mechanism though. This is primarily due to the following:

- the EMM message containing CWPK key seems to be continuously broadcasted and it can be easily detected upon smartcard's response pattern,
- in the environment of ADB set-top-boxes, the `cpm_SecGetDecryptedKeyPtr` function of `libstd_cai_client_conax7.so` library can be successfully used to obtain the original CWPK key.

Crypto DMA handling

While our reverse engineering efforts were primarily focused on SlimCORE firmware and its handling of CWPK and CW keys, some initial analysis of Crypto DMA functionality has been also conducted.

As a result of this analysis, the following code sequences were discovered as being likely primarily responsible for crypto DMA transfer implementation (standard DMA case):

1. Initialization of IV vector:

```
l_02c7  0x00a10001   ld r1,[r0,0001] // 0x4004   ;DMA CONFIG
  02c8  0x00711c2c   bitval r1,r1,#00001000      ;IV init ?
  02c9  0x008812cf   jz l_02cf                   ;-> jump if no need to init IV
  02ca  0x00b0deff   st r13,[r14,00ff]           ;temporary store r13
  02cb  0x003ee001   sub r14,r14,r0,#0001
  02cc  0x00ed02ce   mov r13,#02ce               ;subroutine return addr
  02cd  0x008c04bb   j l_04bb                    ;initialization of IV ?
  02ce  0x00ad1e01   ld r13,[r14,0001] // 0x0004 ;restore saved r13
```

[14] `/mnt/flash/secure/7/0` file.

The above sequence checks 0x1000 bit flag of a DMA CONFIG variable to see whether initialization vector[15] IV was provided at the time of a DMA setup operation. If it exists, a call to 04bb subroutine is made where IV gets initialized.

The called subroutine first checks whether target DMA channel in TKD DMA command is within the allowed range:

```
l_04bb  0x00a1002e   ld r1,[r0,002e] // 0x40b8   ;TKD CMD
  04bc  0x00d00090   sync
  04bd  0x00721d08   shr r2,r1,0x08             ;DMA channel id+0x10
  04be  0x00302010   sub r0,r2,r0,#0010         ;DMA channel id
  04bf  0x009844e0   jb l_04e0                  ;-> jump to the end if < 0x10
  04c0  0x00302018   sub r0,r2,r0,#0018
  04c1  0x009c44e0   jae l_04e0                 ;-> jump to the end if >= 0x18
```

For IV init, only channels 0 and 7 seem to be used:

```
  04c2  0x00222001   add r2,r2,r0,#0001         ;r2 = in the range of 0x11 do 0x18
  04c3  0x00302017   sub r0,r2,r0,#0017
  04c4  0x008874c7   jz l_04c7                  ;-> jump if r2 == 0x17
  04c5  0x00e20010   mov r2,#0010
  04c6  0x00d00090   sync
```

One of these channels is set in a target TKD DMA command. Additionally, its IV bit is cleared to indicate that IV has been configured and algorithm mode is set to ECB:

```
l_04c7  0x00712108   mov r1,r2&0xff<<8
  04c8  0x00710026   bitset r1,r0&0x01<<6       ;clear bit 0x40 (IV seed?)
  04c9  0x00710041   mov r1,r0&0x03<<1          ;clear bits xxxxx00x TKD CMD
```

Later on a check is made to see whether the IV is for AES or TDES algorithm and the actual initialization takes place:

```
  04ca  0x00500300   tst r3,r3                  ;AES ?
  04cb  0x009814d6   je l_04d6                  ;-> jump for TDES
  04cc  0x009d84cc   wait4                      ;AES handling
  04cd  0x00f54000   copAES
  04ce  0x00d00090   sync
  04cf  0x000f013c   mov r15,r1                 ;TKD CMD -> OUT
  04d0  0x00d00004   rpt 4
  04d1  0x000f003c   mov r15,r0                 ;rpt 4 r0 -> OUT
  04d2  0x008d84d2   wait2
  04d3  0x00d00004   rpt 4
  04d4  0x00000f3c   mov r0,r15                 ;rpt 4 r0 <- IN
  04d5  0x008c04e0   j l_04e0                   ;-> jump to the end
```

The IV initialization implementation is a little bit confusing. It seems to be initializing the IV with the use of a target cipher (AES or TDES), but the actual value used for the IV is always 0. It could be that the IV seed in DMA CONFIG indicates that a default zero vector for the IV should be used.

2. Configuration of source and target addresses for crypto DMA transfer:

```
l_02e4  0x00f10000   UNK                                ;unknown coprocessor
                                                         instruction
```

[15] required for the CBC cipher mode of AES algorithm operation.

```
02e5   0x00af0032   ld r15,[r0,0032] // 0x40c8       ;0x3051 or 0x1051 DMA src
                                                       config cmd -> OUT
02e6   0x002fb000   add r15,r11,r0,#0000              ;DMA src
02e7   0x002bb020   add r11,r11,r0,#0020              ;DMA src+=0x20
02e8   0x00366008   sub r6,r6,r0,#0008
02e9   0x00af0031   ld r15,[r0,0031] // 0x40c4       ;0x4052 or 0x6052 DMA dst
                                                       config cmd -> OUT
02ea   0x000f0a3c   mov r15,r10                       ;DMA dst
02eb   0x009d82eb   wait4
```

The above sequence initializes source and destination addresses for DMA transfer. The transfer is conducted by the means of 0x20 bytes at a time (eight 32-bit words).

There are different TKD DMA configuration commands depending on whether they pertain to the source and destination address as well as the actual cipher operation (encryption vs. decryption). This is illustrated in Table 5.

TKD DMA COMMAND	MEMORY ADDRESS	OPERATION
0x3051	DMA source	Encryption
0x1051	DMA source	Decryption
0x4052	DMA destination	Encryption
0x6052	DMA destination	Decryption

Table 5 TKD DMA configuration commands.

3. Actual DMA transfer:

```
02f1   0x00f44000   copAES_dma
02f2   0x00af002e   ld r15,[r0,002e] // 0x40b8     ;TKD CMD -> OUT
02f3   0x00d00004   rpt 4
02f4   0x000f0f3c   mov r15,r15                     ;do the DMA transfer
02f5   0x002aa020   add r10,r10,r0,#0020            ;DMA dst+=0x20
02f6   0x00399008   sub r9,r9,r0,#0008              ;decrease number of dwords by 8
02f7   0x009d82f7   wait4
```

The above sequence seems to be configuring the target crypto algorithm for the DMA transfer (`copAES_dma` instuction). Then, it issues TKD DMA command (Fig. 13 and Fig. 14) to the TKD core. Finally, the transfer is performed by the means of a `mov r15,r15` instruction within the repeat loop.

It should be noted, that for TDES crypto algorithm, the configuration takes place with the use if the following instruction:

```
0x00f84000 copTDES_dma
```

There are many other peculiarities pertaining to the crypto DMA implementation such as the use of 0x00f00000 and 0x00f20000 coprocessor instruction, swapping bytes and `decryptContainer` implementation in particular. As this functionality didn't seem to be relevant from a point of view of CWPK and CW handling, it hasn't been analyzed / reversed engineered further (only basic understanding of TKD DMA implementation was acquired).

Original reverse engineering annotations

Upon successful reverse engineering of a SlimCORE processor instruction format and a disassembly dump of TKD firmware code, we conducted an analysis of its operation. This analysis was performed in the context of the information acquired by the means of both static[16] and dynamic[17] analysis of the firmware's code. Along the analysis process, firmware code corresponding to STTKDMA-REL_3.1.6 was being annotated with comments and description of the instructions' operation.

These original annotations are available as part of SRP-2018-01 project. The annotation file has the following format:

```
!/*## (c) SECURITY EXPLORATIONS    2011 poland                                #*/
!/*##     http://www.security-explorations.com                                #*/
!
!/* RESEARCH MATERIAL:     SRP-2018-01
*/
!/* Reverse engineering annotations                                            */
!/* SlimCORE firmware ver      : STTKDMA-REL_3.1.6                             */
!/*                    code size: 5852 (0x16dc)                                */
!/*                    sha-1    : afe518789d1b0b1d3c0f8efd2704ac84a69140ed     */
!
!/* THIS SOFTWARE IS PROTECTED BY DOMESTIC AND INTERNATIONAL COPYRIGHT LAWS    */
!/* UNAUTHORISED COPYING OF THIS SOFTWARE IN EITHER SOURCE OR BINARY FORM IS   */
!/* EXPRESSLY FORBIDDEN. ANY USE, INCLUDING THE REPRODUCTION, MODIFICATION,    */
!/* DISTRIBUTION, TRANSMISSION, RE-PUBLICATION, STORAGE OR DISPLAY OF ANY      */
!/* PART OF THE SOFTWARE, FOR COMMERCIAL OR ANY OTHER PURPOSES REQUIRES A      */
!/* VALID LICENSE FROM THE COPYRIGHT HOLDER.                                   */
!
!/* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS    */
!/* OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,*/
!/* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL    */
!/* SECURITY EXPLORATIONS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, */
!/* WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF  */
!/* OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE     */
!/* SOFTWARE.                                                                  */
0 ########################
0 DISPATCH idx 0x04 -> 0x2000000 (init code)
0 ########################

b ;counter = 0
c ;memory idx of 0x4040 addr
f ;store 0 to [0x4040-0x4060]
10 ;chip customer mode
12 ;low nibble of chip customer mode
14 ;-> chip customer mode == 0x05
16 ;-> chip customer mode == 0x02
18 ;-> chip customer mode == 0x06
1a ;-> chip customer mode == 0x0b
1c ;-> chip customer mode == 0x0f
1e ;-> chip customer mode == 0x03
...
```

Each line starts with a hexadecimal number indicating the instruction at a given code location. It is followed by a space separator and one of the following:

[16] static analysis of firmware code, STTKDMA device driver files and user level libraries.
[17] dynamic analysis conducted with the help of a SlimCORE tracer tool.

- a "!" character indicates a comment in the annotations file itself and it is ignored,
- a ";" character indicates a comment following a given instruction,
- any other character indicates a comment proceeding a given instruction.

The annotation can be applied to a target disassembly dump[18] with the use of our SCDisasm tool. The results of doing this is presented below:

```
0009  0x00b04084   st r4,[r0,0084] // 0x5e10
000a  0x00b03085   st r3,[r0,0085] // 0x5e14
000b  0x00b0002c   st r0,[r0,002c] // 0x40b0   ;counter = 0
000c  0x00e60010   mov r6,#0010                ;memory idx of 0x4040 addr
000d  0x00d00090   sync
000e  0x00d00009   rpt 9
000f  0x00b10601   st r0,[r6],r6+=#0001        ;store 0 to [0x4040-0x4060]
0010  0x00a5008a   ld r5,[r0,008a] // 0x5e28   ;chip customer mode
0011  0x00e40040   mov r4,#0040
0012  0x00735c80   and r3,r5,0x0f              ;low nibble of chip customer mode
0013  0x00c03005   cmp r3,#05
0014  0x00981026   je l_0026                   ;-> chip customer mode == 0x05
0015  0x00c03002   cmp r3,#02
0016  0x00981028   je l_0028                   ;-> chip customer mode == 0x02
0017  0x00c03006   cmp r3,#06
0018  0x0098102a   je l_002a                   ;-> chip customer mode == 0x06
0019  0x00c0300b   cmp r3,#0b
001a  0x0098102c   je l_002c                   ;-> chip customer mode == 0x0b
001b  0x00c0300f   cmp r3,#0f
001c  0x0098102e   je l_002e                   ;-> chip customer mode == 0x0f
001d  0x00c03003   cmp r3,#03
001e  0x00981030   je l_0030                   ;-> chip customer mode == 0x03
```

Recent firmware changes

Over the years, the SlimCORE firmware for STi7111 SoC has not changed much. There are not many differences between firmware version 3.1.6 and 3.5.0. The functionality and implementation of both firmwares is almost identical (the offsets for all main subroutines differ only by a few bytes).

The biggest changed was observed in the most recent firmware version available in ITI-2849ST and ITI-2850ST set-top-boxes (Table 6).

FIRMWARE VERSION	DATE	CODE SIZE	INSTRUCTION COUNT	DIFFERENCE VS. 3.1.6
STTKDMA-REL_3.1.6	2010	5852	0x05b7	Same
STTKDMA-REL_3.5.0	2011	5944	0x05ce	+23 instructions
STTKDMA-REL_3.9.2	2015	6324	0x062c	+117 instructions

Table 6 SlimCORE firmware versions and their differences.

More specifically, the code of the latest firmware is bigger by 117 instructions than the previous one. This is primarily due to the addition of the following code:

- TKD commands obfuscation subroutine (15 instructions),
- multiple invocations of TKD commands obfuscation subroutine (13 locations with 3 instructions each),

[18] corresponding to the firmware version for which it was suited for.

- modified implementation of the main TKD commands execution subroutine such as `decryptKey` (prolog and epilog subroutines, with 15 and 17 instructions respectively),
- implementation of 3 new commands (40+ instructions[19]).

These are the only differences observed - the core functionality related to key management and crypto DMA is implemented in a similar way as for old firmwares. Below, a more detailed description pertaining to the new code is given.

TKD commands obfuscation

New firmware does not store TKD commands in data memory in plaintext form. They are obfuscated instead and need to be processed before being sent to TKD crypto core.

Below, an code sequence handling `setCWPK` key command (STK cmd 0x02) is shown:

```
  01cc  0x00c05002   cmp r5,#02
  01cd  0x00981209   je l_0209                         ;-> jump for STK CMD == 0x02
  ...
l_0209  0x00a10018   ld r1,[r0,0018] // 0x4060 = 0xa3cedbeb
  020a  0x00ed020c   mov r13,#020c                     ;subroutine return addr
  020b  0x008c0059   j l_0059                          ;call deobfuscation subroutine
  020c  0x0009013c   mov r9,r1                         ;move real (deobfuscated) TKD
                                                       ;command value to r9
```

The code above loads an obfuscated TKD command value (`0xa3cedbeb`) to register r1 and invokes a deobfuscation subroutine at `0x0059` location. The result value (real TKD command value `0x00ff8101`) is returned in register r1.

The implementation of TKD commands' deobfuscation subroutine is as follows:

```
l_0059  0x00b02eff   st r2,[r14,00ff]          ;save r2 on stack
  005a  0x00b03efe   st r3,[r14,00fe]          ;save r3 on stack
  005b  0x003ee002   sub r14,r14,r0,#0002      ;adjust stack pointer for tmp space
  005c  0x00721e03   mov r2,(r1>>3)&0xffff     ;bits 3-18 of input cmd
  005d  0x00731db3   mov r3,(r1>>19)&0x1fff    ;bits 19-31 of input cmd
  005e  0x00712210   movhi r1,r2<<16           ;bits 3-18 become bits 16-31
  005f  0x007131a3   mov r1,r3&0x1fff<<3       ;bits 19-31 become bits 3-15
  0060  0x00e2db82   mov r2,#db82
  0061  0x00e322ca   mov r3,#22ca
  0062  0x00d00090   sync
  0063  0x00732210   movhi r3,r2<<16           ;=0xdb8222ca (fixed constant)
  0064  0x00611300   xor r1,r1,r3              ;perform deobfuscation through xor
  0065  0x00a31e01   ld r3,[r14,0001]          ;restore r3
  0066  0x00a21e01   ld r2,[r14,0001]          ;restore r2
  0067  0x00840d00   jmp r13                   ;return from subroutine
```

The deobfuscation process is very simple - it involves arbitrary bits shifting and an exclusive or (xor) operation with a constant value (Fig. 17).

[19] the total of new instructions exceeds 117, but this is compensated by the compression of some other code parts such as the one related to chip customer mode handling and STK commands' state maintenance in particular.

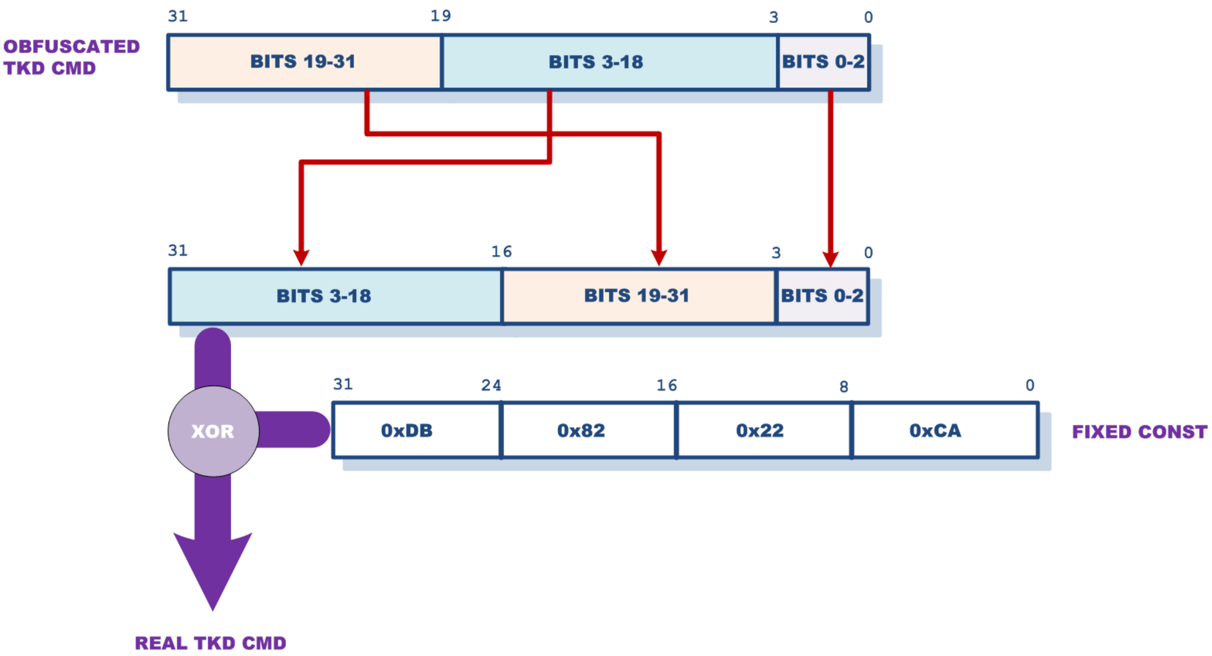


Fig. 17 Deobfuscation of TKD commands.

The following Java code can be used to successfully deobfuscate arbitrary TKD command value from new STi7111 firmware:

```
public static int deobfuscate(int v) {
  int v1=(v>>3)&0xffff;
  int v2=(((v>>19)&0x1fff)<<3)|((v&0x07)&0xffff);
  int vv=(v1<<16)|v2;
  int res=vv^0xdb8222ca;
  return res;
}
```

Prolog and epilog routines

For certain TKD commands, additional prolog and epilog functions are executed by the new firmware. This in particular includes, but is not limited to core routines related to CWPK and CWs handling.

The following prolog code is used prior to the execution of TKD commands:

```
l_036a  0x00b04eff   st r4,[r14,00ff]              ;save r4
  036b  0x00b05efe   st r5,[r14,00fe]              ;save r5
  036c  0x003ee002   sub r14,r14,r0,#0002          ;adjust stack for tmp space
  036d  0x00e44042   mov r4,#4042
  036e  0x00e50030   mov r5,#0030
  036f  0x00745210   movhi r4,r5<<16               ;=0x00304042 (TKD CMD)
  0370  0x00a50043   ld r5,[r0,0043] // 0x410c = 0xfe248000
  0371  0x00f00000   UNK                          ;unknown coprocessor
                                                   instruction
  0372  0x000f043c   mov r15,r4                    ;TKD CMD -> OUT
  0373  0x002f5010   add r15,r5,r0,#0010           ;0xfe248010 addr   -> OUT
  0374  0x00d00004   rpt 4
  0375  0x000f003c   mov r15,r0                    ;rpt 4 r0 -> OUT
  0376  0x00a51e01   ld r5,[r14,0001] // 0x0004    ;restore r5
  0377  0x00a41e01   ld r4,[r14,0001] // 0x0004    ;restore r4
  0378  0x00840d00   jmp r13                       ;return from subroutine
```

The code above seems to initialize several internal registers[20] of a SlimCORE processor with zero values. This is accomplished by the means of a TKD command configuring destination address of a TKD operation in a similar way to DMA transfer. In this particular case, the 0x00304042 is however used instead of the usual 0x4052 DMA destination addr configuration command (Table 5).

The epilog code invoked after the execution of arbitrary TKD commands is very similar to the prolog one:

```
l_0379  0x00f00000  UNK                              ;unknown coprocessor
                                                      instruction
  037a  0x00b01eff  st r1,[r14,00ff]                 ;save re
  037b  0x003ee001  sub r14,r14,r0,#0001             ;adjust stack for tmp space
  037c  0x00a10043  ld r1,[r0,0043] // 0x410c = 0xfe248000
  037d  0x00ef4042  mov r15,#4042                    ;TKD CMD -> OUT
  037e  0x002f1010  add r15,r1,r0,#0010              ;0xfe248010 addr   -> OUT
  037f  0x00d00004  rpt 4
  0380  0x000f003c  mov r15,r0                       ;rpt 4 r0 -> OUT
  0381  0x00a11e01  ld r1,[r14,0001] // 0x0004       ;restore r1
  0382  0x00840d00  jmp r13                          ;return from subroutine
```

There is however a difference in TKD CMD used (0x4042) to configure the destination address.

New commands

New firmware implements 3 new STK commands. These are briefly described below.

STK command 0x43

This command seems to directly initialize a key slot from a descrambling keys' memory location (offset 0x3100) with given input values.

First, target memory address corresponding to descrambling key index indicated by register r4 is computed and stored in same register:

```
l_026d  0x00a30043  ld r3,[r0,0043] // 0x410c        ;= 0xfe248000 (base addr)
  026e  0x00e53100  mov r5,#3100                     ;descrambling keys offset
  026f  0x00d00090  sync
  0270  0x00233500  add r3,r3,r5,#0000               ;descrambling keys addr
  0271  0x000c003c  mov r12,r0                       ;=0

  0272  0x008c0276  j l_0276
  ...
l_0276  0x00144004  shl r4,r4,#0004                  ;key idx<<4
  0277  0x00244300  add r4,r4,r3,#0000               ;addr for a descrambler key
```

After that, source memory address from where key data is to be obtained is also computed:

```
l_0278  0x00a50008  ld r5,[r0,0008] // 0x4020        ; DATA[0] - src idx
  0279  0x00e30120  mov r3,#0120                     ; memory idx of 0x4480 addr
  027a  0x00155002  shl r5,r5,#0002                  ;src idx<<2
  027b  0x00233500  add r3,r3,r5,#0000               ;src addr
  027c  0x0040c001  tst r12,01
  027d  0x00881284  jz l_0284                        ;-> jump for STK cmd == 0x43
```

[20] SlimCORE processor space is mapped at base address `0xfe24800` of the host operating system. According to [7], internal processor registers occupy the beginning of this address space.

Finally, key data from source memory location is moved into the target descrambling memory slot:

```
l_0284   0x00f00000   UNK                                 ;unknown coprocessor
                                                           instruction
  0285   0x00d00090   sync
  0286   0x00d00090   sync
  0287   0x00af0044   ld r15,[r0,0044] // 0x4110          ;= 0x23104022 (TKD CMD) -> OUT
  0288   0x000f043c   mov r15,r4                          ;addr for a descrambler key
  0289   0x00af0300   ld r15,[r3,0000] // 0x0000          ;src data[0] -> OUT
  028a   0x00af0301   ld r15,[r3,0001] // 0x0004          ;src data[4] -> OUT
  028b   0x00af0302   ld r15,[r3,0002] // 0x0008          ;src data[8] -> OUT
  028c   0x00af0303   ld r15,[r3,0003] // 0x000c          ;src data[c] -> OUT
```

Following that, a dummy delay loop is executed:

```
  028d   0x00ec0064   mov r12,#0064                       ;loop counter = 100
l_028e   0x003cc001   sub r12,r12,r0,#0001                ;decrease counter
  028f   0x008c128e   jne l_028e                          ;-> loop jump if counter != 0
```

From the above implementation, we conclude that STK command 0x43 makes it possible to set a given descrambling key directly in descramblers' key memory.

STK command 0x44

STK command 0x44 starts with an initialization of register r4 with a key index provided as part of STK command itself (byte 1 denoting 0x44+idx<<8 value):

```
l_0273   0x0004033c   mov r4,r3                           ;key idx
  0274   0x00ec0001   mov r12,#0001                       ;indicate STK command 0x44
  0275   0x008c0278   j l_0278
```

Following that, similarly to STK command 0x43, the source memory address is computed from where key data for a given source key index is to be obtained:

```
l_0278   0x00a50008   ld r5,[r0,0008] // 0x4020           ; DATA[0] - src idx
  0279   0x00e30120   mov r3,#0120                        ; memory idx of 0x4480 addr
  027a   0x00155002   shl r5,r5,#0002                     ;src idx<<2
  027b   0x00233500   add r3,r3,r5,#0000                  ;src addr
  027c   0x0040c001   tst r12,01
  027d   0x00881284   jz l_0284                           ;-> jump for STK cmd == 0x43
```

The difference is that the jump at location `0x027d` is not taken (r12 is set to 1) and consecutive instructions get executed. These instruction modify the key index value to be in the range 0-7 (modulo 8) and transfer key data from a computed source location to registers:

```
  027e   0x00444007   and r4,r4,#0007                     ;key idx modulo 8
  027f   0x00a50301   ld r5,[r3,0001] // 0x0004           ;src data[4]
  0280   0x00a90302   ld r9,[r3,0002] // 0x0008           ;src data[8]
  0281   0x00ac0303   ld r12,[r3,0003] // 0x000c          ;src data[c]
  0282   0x00a30300   ld r3,[r3,0000] // 0x0000           ;src data[0]
  0283   0x008c0296   j l_0296
```

Following that, a jump to prolog routine is made:

```
l_0296   0x00ed0298   mov r13,#0298                       ;subroutine return addr
  0297   0x008c036a   j l_036a                            ;invoke prolog subroutine
```

Later on, two similar sequences corresponding to two TKD core operations are executed one after another.

- OPERATION 1

The TKD core is put into AES mode and TKD command `0x324923eb` gets deobfuscated. As a result, cleartext TKD command `ffff1081` is obtained in register r1:

```
0298  0x00f54000   copAES
0299  0x00a10028   ld r1,[r0,0028] // 0x40a0       ;= 0x324923eb (obfuscated TKD
                                                      command)
029a  0x00b0deff   st r13,[r14,00ff]               ;save r13
029b  0x003ee001   sub r14,r14,r0,#0001            ;adjust stack for tmp space
029c  0x00ed029e   mov r13,#029e                   ;subroutine return addr
029d  0x008c0059   j l_0059                        ;invoke deobfuscation sub
029e  0x00ad1e01   ld r13,[r14,0001] // 0x0004     ;restore r13
```

Selected bits of TKD command `0xffff1081` are further modified. As a result, TKD command `0xffff8000` is produced:

```
029f  0x00e80080   mov r8,#0080                    ;=0x80
02a0  0x00710020   bitset r1,r0&0x01<<0            ;=0xffff1080 (bit 0 cleared)
02a1  0x00718108   mov r1,r8&0xff<<8               ;=0xffff8080 (bit 15 set)
02a2  0x00710027   bitset r1,r0&0x01<<7            ;=0xffff8000 (bit 7 cleared)
```

This command along arguments data contained in registers are sent to the TKD Crypto core (OUT operation):

```
02a3  0x000f013c   mov r15,r1                      ;0xffff8000 (TKD CMD) -> OUT
02a4  0x008d82a4   wait2
02a5  0x000f033c   mov r15,r3                      ;r3 -> OUT
02a6  0x000f053c   mov r15,r5                      ;r5 -> OUT
02a7  0x000f093c   mov r15,r9                      ;r9 -> OUT
02a8  0x000f0c3c   mov r15,r12                     ;r12 -> OUT
02a9  0x008d82a9   wait2
```

The result of the operation is read from TKD Crypto core (IN operation) and stored back to registers:

```
02aa  0x00030f3c   mov r3,r15                      ;r3 <- IN
02ab  0x00050f3c   mov r5,r15                      ;r5 <- IN
02ac  0x00090f3c   mov r9,r15                      ;r9 <- IN
02ad  0x000c0f3c   mov r12,r15                     ;r12 <- IN
```

- OPERATION 2

The TKD core is again put into AES mode and TKD command `0x23ce5beb` gets deobfuscated. As a result, cleartext TKD command `10ff0101` is obtained in register r1:

```
02ae  0x00f54000   copAES
02af  0x00a1001a   ld r1,[r0,001a] // 0x4068       ;= 0x23ce5beb (obfuscated TKD
                                                      command)
02b0  0x00b0deff   st r13,[r14,00ff]               ;save r13
02b1  0x003ee001   sub r14,r14,r0,#0001            ;adjust stack for tmp space
02b2  0x00ed02b4   mov r13,#02b4                   ;subroutine return addr
02b3  0x008c0059   j l_0059                        ;invoke deobfuscarion sub
02b4  0x00ad1e01   ld r13,[r14,0001] // 0x0004     ;restore r13
```


Selected bits of TKD command `10ff0101` are further modified. As a result, TKD command `0x10ff8101|(idx<<24)` is produced in register r1:

```
02b5   0x00e80080   mov r8,#0080                    ;=0x80
02b6   0x00714098   mov r1,r4&0x0f<<24              ;set key idx in highest byte
02b7   0x00718108   mov r1,r8&0xff<<8               ;=0x10ff8101 (bit 15 set)
```

This command along arguments data contained in registers are sent to the TKD Crypto core (OUT operation):

```
02b8   0x00d00090   sync
02b9   0x000f013c   mov r15,r1                      ;TKD CMD -> OUT
02ba   0x008d82ba   wait2
02bb   0x000f033c   mov r15,r3                      ;r3 -> OUT
02bc   0x000f053c   mov r15,r5                      ;r5 -> OUT
02bd   0x000f093c   mov r15,r9                      ;r9 -> OUT
02be   0x000f0c3c   mov r15,r12                     ;r12 -> OUT
02bf   0x008d82bf   wait2
```

The result of the operation is read from TKD Crypto core (IN operation), but it is ignored:

```
02c0   0x00d00004   rpt 4
02c1   0x00000f3c   mov r0,r15                      ;rpt 4 r0 <- IN
```

A summary of both operations implemented by STK command 0x44 is presented in Table 7.

OPERATION	TKD COMMAND	DESCRIPTION
OP1 (Calc pairing key)	0xffff8000	Encrypt input with SCK key (key location 0x80[21]) and make it available as the output
OP2 (Calc crypto DMA key with the use of a pairing key)	0x10ff8101|(idx<<24)	Decrypt input with SCK key (key location 0x81) and store it at key location 10+idx (crypto DMA / AES keys)

Table 7 Summary of operations implemented by STK command 0x44.

At the end of STK command 0x44 implementation, an epilog subroutine is invoked:

```
02c2   0x00ed02c4   mov r13,#02c4                   ;subroutine return addr
02c3   0x008c0379   j l_0379                        ;invoke epilog subroutine
```

From the above implementation, we conclude that STK command 0x44 serves as either:

- a debug command making it possible to test encryption and decryption of operations of a arbitrary pairing key (if keys at locations 0x80 and 0x81 are the same),
- an implementation of a pairing functionality making use of two SCK keys (if keys at locations 0x80 and 0x81 are different).

STK command 0x48

[21] during our tests, commands `ffff8001` and `ffff8101` produced same results, thus we associate key locations ox80 and 0x81 with same SCK key.

Implementation of STK command 0x48 is similar to command 0x44. The only difference is in the source for the input key data. For STK command 0x48, the input key comes from STK command buffer, not the 0x4480 based memory area:

```
l_0291  0x0004033c   mov r4,r3                          ;key idx
  0292  0x00a30008   ld r3,[r0,0008] // 0x4020          ;DATA[0]
  0293  0x00a50009   ld r5,[r0,0009] // 0x4024          ;DATA[4]
  0294  0x00a9000a   ld r9,[r0,000a] // 0x4028          ;DATA[4]
  0295  0x00ac000b   ld r12,[r0,000b] // 0x402c         ;DATA[4]
```

The processing of the input data is further handled from code location 0x0296 (shared code path for both 0x44 and 0x48 STK commands).

Potential vulnerabilities and further research

While analysis of STi7111 SlimCORE firmware and TKD operation has lead to the discovery of 2 security vulnerabilities in the SoC implementation, some other vulnerabilities could be still present in the chip. Below, a brief description of several interesting candidates is given that in our opinion deserve a deeper attention and verification as they could be the source of additional security vulnerabilities of STi7111 SoC.

Privileged customer mode

STTKDMA-REL_3.1.6 firmware contains multiple checks of a customer mode variable. While hardware customer mode does not seem to matter much (it is mapped to a corresponding SW variable, which can be easily bypassed), the checks conducted indicate that some STK / TKD commands could be more sensitive than others. More specifically, it is reasonable to assume that a privileged / unique customer mode exists (such as the chipset vendor related one) that allows for some security sensitive commands to be executed.

Table 8 illustrates customer mode values and corresponding STK commands (explicitly invalid or valid).

CUSTOMER MODE		STK COMMANDS	
HW	SW	INVALID	VALID
00, 01, 04, 09, 0a, 0d, 0e	40	0x00, 0x05, 0x02, 0x03, 0x40 0x01, 0x04	
02	04		0x20, 0x21, 0x22, 0x23
03	10		0x80 0x20, 0x21, 0x22, 0x23
05	02		0x06 0x20, 0x21, 0x22, 0x23
06	05		0x20, 0x21, 0x22, 0x23
07	11		0x80 0x20, 0x21, 0x22, 0x23
08	20	0x01, 0x04	0x10, 0x11, 0x12 0x20, 0x21, 0x22, 0x23
0b	08	0x01, 0x04	0x80 0x20, 0x21, 0x22, 0x23
0c	21	0x01, 0x04	0x10, 0x11, 0x12 0x20, 0x21, 0x22, 0x23

0f	09	0x01, 0x04	0x80 0x20, 0x21, 0x22, 0x23

Table 8 Customer mode values and corresponding STK commands.

One can notice that for SW customer mode 0x02, STK command 0x06 is explicitly allowed. This commands corresponds to the unusual bit combination for the least significant byte of an associated TKD command (20ff0010 + idx<<24). It also targets descrambling keys memory (TKD cmd target is 0x20 based), which makes this command a natural candidate for a more thorough investigation.

Similarly, Table 9 shows some of the special modifications applied to TKD commands with respect to the customer mode value. These modifications concern CWPK and CW keys handling commands in particular, which again make them primary candidates for an in-depth investigation.

SW CUSTOMER MODE	STK COMMAND	OPTIONAL SPECIAL HANDLING
!=0x02	0x01	set xxxx82xx in TKD CMD
0x02	0x02	set bit 0x08 in TKD CMD
0x10, 0x08, 0x04	0x02	set xxxx82xx in TKD CMD
0x02	0x03	set bit 0x08 in TKD CMD
0x21	0x03	set xxxx03xx in TKD CMD set bit 0x80 in TKD CMD
0x10, 0x11	0x03	set xxxx82xx in TKD CMD
0x08	0x03	set xxxx81xx in TKD CMD
0x10, 0x11, 0x04	0x04	set xxxx80xx in TKD CMD
!=0x10,!=0x11,!=0x04	0x04	set bit 0x80 in TKD CMD

Table 9 Customer mode value and special handling of STK commands.

Additionally, the changes introduced in SlimCORE firmware 3.9.2 still take customer mode into account. For instance, the firmware makes sure that bit values 0x01 and 0x08 of HW customer mode are always 0:

```
  000c   0x00a1008a   ld r1,[r0,008a] // 0x5e28      ;HW customer mode
  000d   0x0045100b   and r5,r1,#000b                ;r5=bits 0, 1 and 3 of HW
                                                         customer mode
  000e   0x00c05002   cmp r5,#02                     ;is only bit 1 set ?
l_000f   0x009c100f   jne,s l_000f                   ;endless loop if not
```

Privileged chip configuration state

TKD Crypto Core configuration state is primarily maintained in memory by the means of TK and DMA CONFIG variables.

In this context, TK CONFIG seems to be in particular interesting as it could decide about whether the chip is put into insecure / privileged state or not. For example, bit 1 of TK CONFIG variable implicates setting of bit 0 at 0x5e30 I/O register location.

Additionally, bits 0, 5 and 7 of TK CONFIG variable directly influence the operation of a descrambler.

Crypto DMA for read / write kernel access

The environment of ITI-2840ST and ITI-2850ST set-top-boxes contain user level libraries that provide support for STTKDMA device driver functionality related to DMA transfers. As Crypto DMA hasn't been the focus of our research, it is still worth to verify whether kernel addresses can be used as either source or destination of arbitrary DMA transfers. If so, such an implementation weakness could be exploited to either modify kernel of the underlying OS[22] or SlimCORE firmware.

Kernel modification is in particular interesting here as this would make it possible to conduct a successful privilege elevation attack[23] in a target OS.

Crypto DMA for chip registers / memory access

SlimCORE firmware 3.9.2 implicitly access memory area mapped by TKD Crypto core with the use of Crypto DMA related TKD commands (0x10 from the SoC base).

In that context, crypto DMA could be potentially used to bypass SoC protections aimed at guarding access to chipset's keys (descrambling keys at 0x3100 offset or internal locations corresponding to CWPK key) or internal registers. The latter seems to be an interesting option to consider taking into account the prolog and epilog functions introduced to firmware 3.9.2. These functions do only one thing - overwrite chip locations likely mapped to internal SlimCORE processor registers as indicated by the `slim_core_map` structure [7] (Fig. 18).

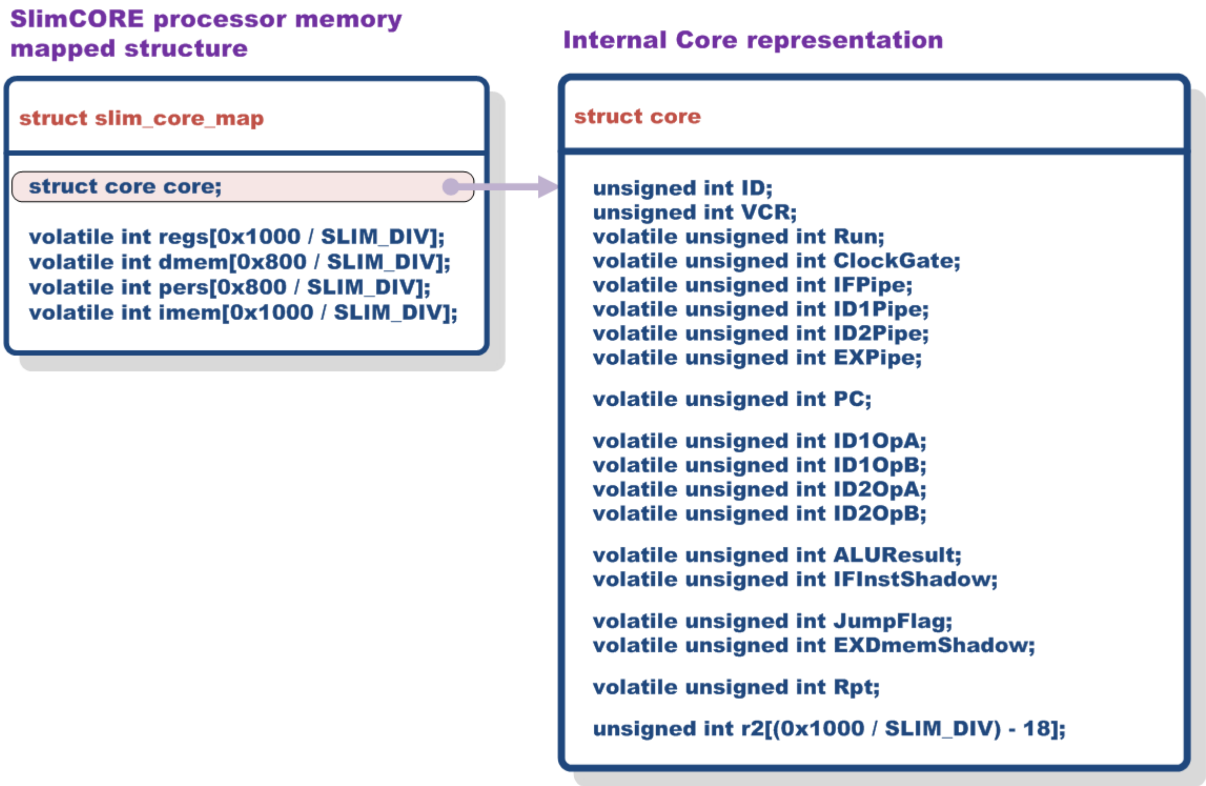


Fig. 18 Internal SlimCORE processor structure.

If this is the case, it could mean that these registers leak key data as part of computations performed.

---

[22] some device drivers such as `/dev/memdev` of ITI-2849ST and ITI2850ST set-top-boxes opened access to limited I/O space of a STi7111 chipset such as SlimCORE processor memory, a vulnerability in Crypto DMA could be abused to gain access to whole kernel memory of the underlying OS.

[23] as a result of fixing the vulnerabilities discovered by Security Explorations the main MHP application is currently executed as unprivileged user and with no capabilities on ADB set-top-boxes.

Arbitrary transfer from / to key memories would need to be accomplished by the means of a custom SlimCORE processor code sequence executed from within the firmware code.

TKD commands for registers access

STK command 0x24 seems to be accessing some software register. This is indicated by the following:

- `sttkdmaHal_GetSWReg` name associated with a code function implementing the command,
- reading of the function execution result from some strange memory locations corresponding to chipset's memory space (0x3024 and 0x3028 offsets from chipset base),
- TKD 0x83ff0203 command format and a target of the operation likely indicating the register (value 0x83).

Beside STK command 0x24, there are other similar STK commands (0x21-0x23) that make use of TKD command targets likely indicating a SoC register (values 0x80-0x82).

This goes along the `setCWPK` command[24] that makes use of SCK key (key implicitly associated with 0x81 location).

Thus, it is worth to investigate these commands in a little bit more detail in order to find out whether SCK key could be accessed / leaked.

With the ability to extract arbitrary pairing key (such as the one from 0x02 key location), TKD command 0x02ff8101 should be treated as under attacker's control. This should make it easier to proceed with the investigation of STK commands 0x21-0x24 from SWREGS group (Table 4) in order to verify whether access to some sensitive SW registers and SCK key in particular could be actually gained.

It is also worth checking whether the plaintext value of a CWPK key set as a result of the usual pairing key configuring commands (STK 0x01 and 0x02) could be accessed through target TKD command locations 0x80-0x83 (through memory offsets around / at 0x3024 and 0x3028 from chipset base).

Coprocessor related commands

There are many coprocessor related commands (opcode 0x0f and `wait` commands) of which meaning and format has not been fully discovered.

These commands seem to be configuring single TKD core components (such as AES / TDES engine) or actual pathways / routing between given TKD core parts (key locations, memory addresses and I/O ports). The latter is concluded from the implementation of crypto DMA and its use of `mov r15, r15` instruction in particular (it can move data between implicitly configured source and destination location). The nature of TKD commands seem to confirm this as well (commands indicate a source and destination for a given operation).

It is worth to explore coprocessor related commands as there might exist a way to configure a pathway from a secret key location to a memory or I/O port. It could be that these commands

[24] or all commands that configure a pairing key such as 0x01, 0x02, 0x10 and 0x11 STK commands.

influence whether the output of a command execution is provided or not (this is in particularly important for pairing key configuration commands - some of them provide output, some do not).

PTI

PTI (Programmable Transport Interface) core is responsible for handling MPEG transport streams, their filtering, descrambling and dispatch. PTI runs firmware code (embedded in and initialized by `ptiinit.ko` device driver), which implements an unknown CPU instruction set.

Some initial analysis of this core along the approach taken has been presented in our paper from 2017 [9]. That analysis has lead us to the conclusion that key contents held in PTI's memory location pointed by `DescramblerKeysStart` address were offsets to some other memory location (such as a descrambler memory), which might have been used by the PTI DMA engine or a descrambler itself.

The analysis of TKD core operation and associated user level libraries[25] seem to confirm that (PTI seems to interact with TKD crypto core by the means of offsets to descrambling key locations).

Taking into account the functionality of PTI component, its complexity (device driver binary is 250KB+ in size), SoC location, interaction with a descrambler and use across various ST chipset generations, PTI seems to be a primary target for any further security investigation of DVB chipsets from STMicroelectronics for all concerned parties (PTI is a common core for many ST DVB chipsets generations).

FDMA and STBUS

SlimCORE processor executing firmware for TKD core control is not the only SlimCORE CPU available as part of STi7111 SoC. There is also one more SlimCORE processor that runs firmware implementing FDMA (Flexible Direct Memory Access) transfers.

In the environment of ITI-2849ST and ITI2850ST set-top-boxes, this firmware can be successfully extracted and disassembled from `fdma.ko` device driver file[26]. Its analysis might provide additional hints regarding SlimCORE instruction set and coprocessor instructions in particular (FDMA firmware makes heavy use of these instructions).

Finally, as indicated on Fig. 1, all components of STi7111 SoC are interconnected with the use of an STBus [10] system interconnect. It could be that SlimCORE coprocessor instructions are in some way related to STBus (that they influence an interact with this system interconnect). Therefore, it is also an interesting area to check in order to verify whether some protected SoC parts can be accessed by the means of STBus.

OTP security fuses

STi7111 contains a dedicated OTP (one time programming) memory area containing various configuration settings of the SoC. This area is mapped at 0xFE00D000 address and it contains such settings as chipset security state and chip id. There are however many other interesting settings as illustrated below:

[25] `CopyTKDMAOffsetToTCsdKey` and `CopyTCsdKeytToTKDMAKey` functions of libstd_drv_scds.so library.

[26] FDMA SlimCORE firmware initialization takes place in `stfdma_FDMA2Conf` subroutine. References to firmware code and data sections are immediately followed by pointers to magic strings ("DATA" and "PROG" respectively).

```
STSECTOOLFUSE_ReadItem 00000001 00000005 netjtag_portstate (lock bit) @jtag_protect
(addr FE00D000,mask 0x0f, shift 0x0c)
STSECTOOLFUSE_ReadItem 00000002 00000001 @engineering_test_000 (FE00D028,0x01,0x06)
STSECTOOLFUSE_ReadItem 00000003 00000001 secure chipset (lock bit) @trans_cw_secure
(FE00D03c,0x01,0x01)
STSECTOOLFUSE_ReadItem 00000004 00000001 @trans_cw_enable (FE00D02c,0x01,0x05)
STSECTOOLFUSE_ReadItem 00000005 00000000 @crypt_cpu0_ifetch_src_rst
STSECTOOLFUSE_ReadItem 00000006 00000000 @crypt_cpu1_ifetch_src_rst
STSECTOOLFUSE_ReadItem 00000007 00000000 @crypt_cpu2_ifetch_src_rst
STSECTOOLFUSE_ReadItem 00000008 00000001 @crypt_sigdma_src_rst
STSECTOOLFUSE_ReadItem 00000009 00000001 @crypt_sigchk_src_rst
STSECTOOLFUSE_ReadItem 0000000a 00000001 @crypt_watchdog_src_rst
STSECTOOLFUSE_ReadItem 0000000b 00000000 @crypt_hash_include_addr
STSECTOOLFUSE_ReadItem 0000000c 00000001 enable_scs (lock bit) @crypt_sigchk_enable
STSECTOOLFUSE_ReadItem 0000000d 00000001 @mes0_enable
STSECTOOLFUSE_ReadItem 0000000e 00000000 @mes0_src_id_mon_enable
STSECTOOLFUSE_ReadItem 0000000f 00000001 @mes0_encrypt_all_enable
STSECTOOLFUSE_ReadItem 00000010 00000000 (lock bit) @t1_filter_enable
STSECTOOLFUSE_ReadItem 00000011 00000001 (lock bit) @dirt_disable
STSECTOOLFUSE_ReadItem 00000012 00004872 @engineering_0
STSECTOOLFUSE_ReadItem 00000013 0000251b @engineering_1
STSECTOOLFUSE_ReadItem 00000014 0000a642 @engineering_2
STSECTOOLFUSE_ReadItem 00000015 0000ba4b @engineering_3
STSECTOOLFUSE_ReadItem 00000016 00000000 @metal_fix_nb
STSECTOOLFUSE_ReadItem 00000017 00000001 @proc_type
STSECTOOLFUSE_ReadItem 00000018 00000002 @fab_loc
STSECTOOLFUSE_ReadItem 00000019 00000000 @customer_otp0
STSECTOOLFUSE_ReadItem 0000001a 00000000 @customer_otp1
STSECTOOLFUSE_ReadItem 0000001b 00000000 @customer_otp2
STSECTOOLFUSE_ReadItem 0000001c 00000000 @customer_otp3
```

We verified that arbitrary OTP programming of this area is possible, which makes it an interesting, but also dangerous target for exploration.

It could be that overall chip security could be weakened (or even disabled) by the means of some of the OTP settings.

The following OTP settings could be in particular interesting from a security point of view:

- all *lock bit* fuses that are not enabled (set to the value of 0) as they likely influence SoC security (i.e. secure chipset setting),
- `crypt_cpuX_ifetch_src_rst` settings as these could influence whether the source (such as a key) of an instruction fetch operation is leaked.

T1 bus configuration

T1 seems to be the internal bus associated with CCORE. The existence of this bus is mentioned in several locations (PhD thesis [8], STM Linux distribution[27], and `t1_filter_enable` OTP security fuse among others).

In some previous distributions of ADB software for ITI-2849ST and ITI-2850ST set-top-boxes, the `libstd_drv_ccore.so` library contained a `ccore_T1Configure` symbol associated with a subroutine doing memory writes to 0xFE216400 based chipset memory area.

[27] linux-2.6.32.16_stm24_sh4_0205.patch

While the written values are not in particular interesting (mostly zero), the unused data immediately following it formed what looked like blocks and their values seemed to follow a pattern:

```
.rodata:00005414                  .data.l h'FC40
.rodata:00005418                  .data.l h'FC04
.rodata:0000541C                  .data.l h'FC08
.rodata:00005420                  .data.l h'FC00

[block 1]

.rodata:00005424                  .data.l 8
.rodata:00005428                  .data.l h'30100
.rodata:0000542C                  .data.l h'B
.rodata:00005430                  .data.l h'60200
.rodata:00005434                  .data.l h'1E
.rodata:00005438                  .data.l h'10300
.rodata:0000543C                  .data.l h'32
.rodata:00005440                  .data.l h'10400
.rodata:00005444                  .data.l h'34
.rodata:00005448                  .data.l h'40600
.rodata:0000544C                  .data.l h'35
.rodata:00005450                  .data.l h'60100
.rodata:00005454                  .data.l h'36
.rodata:00005458                  .data.l h'30400
.rodata:0000545C                  .data.l h'37
.rodata:00005460                  .data.l h'30400
.rodata:00005464                  .data.l h'41
.rodata:00005468                  .data.l h'30800
.rodata:0000546C                  .data.l h'44
.rodata:00005470                  .data.l h'10600
.rodata:00005474                  .data.l h'45
.rodata:00005478                  .data.l h'10400
.rodata:0000547C                  .data.l h'51
.rodata:00005480                  .data.l h'20000

.rodata:00005484                  .data.l h'FFFF
.rodata:00005488                  .data.l h'FFFF

[block 2]

.rodata:0000548C                  .data.l 8
.rodata:00005490                  .data.l h'50202
.rodata:00005494                  .data.l h'B
.rodata:00005498                  .data.l h'30101
.rodata:0000549C                  .data.l h'1E
.rodata:000054A0                  .data.l h'10400
.rodata:000054A4                  .data.l h'32
.rodata:000054A8                  .data.l h'20100
.rodata:000054AC                  .data.l h'34
.rodata:000054B0                  .data.l h'40700
.rodata:000054B4                  .data.l h'35
.rodata:000054B8                  .data.l h'60500
.rodata:000054BC                  .data.l h'36
.rodata:000054C0                  .data.l h'40100
.rodata:000054C4                  .data.l h'37
.rodata:000054C8                  .data.l h'40300
.rodata:000054CC                  .data.l h'41
.rodata:000054D0                  .data.l h'40300
.rodata:000054D4                  .data.l h'44
```

```
.rodata:000054D8                .data.l h'10700
.rodata:000054DC                .data.l h'45
.rodata:000054E0                .data.l h'10402
.rodata:000054E4                .data.l h'51
.rodata:000054E8                .data.l h'10500

.rodata:000054EC                .data.l h'FFFF
.rodata:000054F0                .data.l h'FFFF
...
```

It could be that these memory writes configure the possible interconnections (filter as in OTP fuse name) between TKD Crypto core components (whether given key locations could be accessed, whether the results of TKD commands produce results, etc.).

Key initialization quirks

Starting from firmware 3.5.0, some strange detail pertaining to the implementation of a key initialization subroutine could be noticed:

```
l_0511  0x00409900   tst r9,00                        ;AES ?
  0512  0x008c151c   jne l_051c                       ;-> jump for AES
  0513  0x00fa4000   copTDES                          ;handle TDES
  0514  0x000f083c   mov r15,r8                       ;TKD CMD -> OUT
  0515  0x008e1515   wait1
  0516  0x00d00002   rpt 2
  0517  0x000f003c   mov r15,r0                       ;rpt 2 r0 -> OUT
  0518  0x00d00002   rpt 2
  0519  0x000f0c3c   mov r15,r12                      ;rpt 2 r12 -> OUT
  051a  0x008e151a   wait1
  051b  0x008c0522   j l_0522
```

What's interesting in the code above is that as part of a single key initialization routine, r12 register is used instead of the usual r0 (zero value). This register holds subroutine return addr for the invocation of a key initialization code:

```
  0038  0x00ed003a   mov r13,#003a                    ;subroutine return addr
  0039  0x008c04f5   j l_04f5                         ;init all of the keys (CWPK,
                                                       CWs)

  ...

l_04f5  0x000c0d3c   mov r12,r13                      ;r12 = subroutine return addr
  04f6  0x00a7000c   ld r7,[r0,000c] // 0x4030        ;customer mode
  04f7  0x00407040   tst r7,40
  04f8  0x009c1510   jne,s l_0510                     ;-> jump to the end
  04f9  0x0009003c   mov r9,r0                        ;r9 = 0 (TDES)
  04fa  0x00e10001   mov r1,#0001
  04fb  0x00407002   tst r7,02
  04fc  0x008814fe   jz l_04fe
  04fd  0x0009013c   mov r9,r1                        ;r9 = 1 (AES)

l_04fe  0x00a8001c   ld r8,[r0,001c] // 0x4070        ;= 0x00ff8101 (setCWPK)
  04ff  0x00ed0501   mov r13,#0501                    ;subroutine return addr
  0500  0x008c0511   j l_0511                         ;init single crypto key
```

It's rather unusual to tie an initialization of a cryptographic key with a firmware code return addr. This alone requires further investigation in our opinion (whether such a key initialization is required for proper CWPK and CW decryption, etc.).

TOOLS

SlimCORE disassembler

SlimCORE disassembler (SCDisasm) is a tool to disassemble SlimCORE processor instruction streams from various firmwares used by STi7111 DVB chipsets. It implements the following features:

- SlimCORE instruction stream disassembly from a device driver file or input files corresponding to firmware code / data sections,
- extraction of SlimCORE firmware data / code sections from a device driver file to output files,
- statistics information regarding the usage of SlimCORE instructions (i.e. unknown, recognized instructions).

Description

Table 10 describes command line arguments available in SCDisasm tool.

ARGUMENT	DESCRIPTION
`-dis`	The argument specifies a disassemble command.
`-m drv|file`	The argument indicates whether a driver file or code dumps should be used as a source for the tool operation.
`-f drv_name`	The argument denotes the name of a device driver file to use.
`-a ann_name`	The argument denotes the name of an annotation file to use.
`-c code_file`	The argument denotes the name of a SlimCORE code dump file to use (either input for a disassemble command or an output for the extraction command)
`-d data_file`	The argument denotes the name of a SlimCORE data dump file to use (either input for a disassemble command or an output for the extraction command)
`-stat unk|all`	The argument indicates a statistics command and whether statistics for unknown or all instructions should be given.
`-ext code|data`	The argument indicates extraction command and whether SlimCORE code or data section dumps should be extracted from a device driver file.

Table 10 Command line arguments of SCDisasm tool.

Sample uses

1. Disassemble SlimCORE firmware from a default device driver file and with the use of a given annotations file:

```
run -dis -m drv -a rea\3.1.6.txt

/*## (c) SECURITY EXPLORATIONS    2011 poland                               #*/
/*##     http://www.security-explorations.com                              #*/

SlimCore disassembler
- loading sttkdma_core_user.ko
   ver: STTKDMA-REL_3.1.6
- locating SlimCore firmware
   code at 0x00003820 size 5852 (0x16dc)
    - sha1 afe518789d1b0b1d3c0f8efd2704ac84a69140ed
   data at 0x00004efc size 1156 (0x0484)
    - sha1 d00044a77407b5a530f94c53bacbbf5b3ee3a0b4
- loading annotations rea\3.1.6.txt
- disassembling
[CODE]
```



```
#######################
DISPATCH idx 0x04 -> 0x2000000 (init code)
#######################
l_0000  0x00200000   add r0,r0,r0,#0000
  0001  0x00200000   add r0,r0,r0,#0000
  0002  0x00d00080   sync
  0003  0x00e30374   mov r3,#0374
  0004  0x00743210   movhi r4,r3<<16
  0005  0x00e4ffff   mov r4,#ffff
  0006  0x00e3ffff   mov r3,#ffff
  0007  0x00743210   movhi r4,r3<<16
  0008  0x00e30001   mov r3,#0001
  0009  0x00b04084   st r4,[r0,0084] // 0x5e10
  000a  0x00b03085   st r3,[r0,0085] // 0x5e14
  000b  0x00b0002c   st r0,[r0,002c] // 0x40b0 = 0x00000000          ;counter = 0
  000c  0x00e60010   mov r6,#0010                                    ;memory idx of 0x4040 addr
  000d  0x00d00090   sync
  000e  0x00d00009   rpt 9
  000f  0x00b10601   st r0,[r6],r6+=#0001                            ;store 0 to [0x4040-0x4060]
  0010  0x00a5008a   ld r5,[r0,008a] // 0x5e28                       ;chip customer mode
  0011  0x00e40040   mov r4,#0040
  0012  0x00735c80   and r3,r5,0x0f                                  ;low nibble of chip customer mode
  0013  0x00c03005   cmp r3,#05
  0014  0x00981026   je l_0026                                       ;-> chip customer mode == 0x05
  ...
```

2. Extract code section of SlimCORE firmware from a default device driver file and save it into given output file:

```
run -ext code -m drv -c code.dat

/*## (c) SECURITY EXPLORATIONS    2011 poland                              #*/
/*##     http://www.security-explorations.com                              #*/

SlimCore disassembler
- loading sttkdma_core_user.ko
   ver: STTKDMA-REL_3.1.6
- locating SlimCore firmware
   code at 0x00003820 size 5852 (0x16dc)
    - sha1 afe518789d1b0b1d3c0f8efd2704ac84a69140ed
   data at 0x00004efc size 1156 (0x0484)
    - sha1 d00044a77407b5a530f94c53bacbbf5b3ee3a0b4
- saving code.dat
```

3. Extract data section of SlimCORE firmware from a given device driver file and save it into given output file:

```
run -ext data -m drv -f sttkdma_core_user.ko -d data.dat

/*## (c) SECURITY EXPLORATIONS    2011 poland                              #*/
/*##     http://www.security-explorations.com                              #*/

SlimCore disassembler
- loading sttkdma_core_user.ko
   ver: STTKDMA-REL_3.1.6
- locating SlimCore firmware
   code at 0x00003820 size 5852 (0x16dc)
    - sha1 afe518789d1b0b1d3c0f8efd2704ac84a69140ed
   data at 0x00004efc size 1156 (0x0484)
    - sha1 d00044a77407b5a530f94c53bacbbf5b3ee3a0b4
- saving data.dat
```

4. Show statistic regarding unknown instructions embedded in SlimCORE firmware loaded from a given set of files corresponding to firmware code and data sections:

```
run -stat unk -m files -c code.dat -d data.dat

/*## (c) SECURITY EXPLORATIONS    2011 poland                              #*/
/*##     http://www.security-explorations.com                              #*/
```

```
SlimCore disassembler
- loading code.dat
- loading data.dat

[UNKNOWN INSTRUCTIONS STATS]
opcode 008cxxxx cnt 1
opcode 008exxxx cnt 2
opcode 00b2xxxx cnt 2
opcode 00f0xxxx cnt 6
opcode 00f1xxxx cnt 4
opcode 00f2xxxx cnt 4
opcode 00f4xxxx cnt 4
opcode 00f8xxxx cnt 4
opcode 00ffxxxx cnt 1
total 9 opcodes
```

SlimCORE tracer

SlimCORE tracer is a tool that makes it possible to trace execution flow of SlimCORE processor instructions. It implements the following features:

- tracing the execution of SlimCORE processor instructions (single stepping, dump of register contents with proper indication of register changes),
- logging of a trace of executed instructions.

The tool was developed as part of SE-2011-01 project and its operation was suited to the environment of fully compromised (OS root, JVM root and kernel level access privileges) ITI-2849ST / ITI-2850ST set-top-boxes and SE-2011-01 Proof of Concept code in particular. A successful operation and use of SlimCORE tracer may require customization and/or porting to the target STi7111 environment (target STB).

Tracer API

Proper operation of the tracer requires that arbitrary access to STi7111 chipset's memory is possible. This in particular includes access to SlimCORE firmware's code and data sections at the time of its execution.

Access to firmware code is necessary due to the fact that traced instructions are modified on the fly. Access to firmware data stems from the fact that it is used by the tracer to keep state of its execution.

Tracer's `API` class contains routines that need to be adopted to the requirements of a target STB environment in order to provide the tracer with read and write access to STTKDMA memory. These are illustrated in Table 11.

TRACER API	SE-2011-01 POC ROUTINES	DESCRIPTION
`STTKDMA_READ(int addr)`	`STTKDMA.tkdma_read`	The base routine making it possible to read kernel memory address.
`STTKDMA_WRITE(int addr,int val)`	`STTKDMA.tkdma_write`	The base routine making it possible to write kernel memory address with a given value.
`LOG(String s)`	`ApiMonitor.log`	The base routine to log tracer's output.

Table 11 Tracer's API subroutines.

Additionally, tracer's `Config` class contains several variables describing target location for a tracer core routine (firmware hijacking location and location where tracer code could be appended). They are described in Table 12.

TRACER VARIABLE	SE-2011-01 POC VALUE	DESCRIPTION
`STTKDMA_BASE`	`0xFE248000`	Chip base address
`STTKDMA_DATA`	`0x4000`	Offset of SlimCORE firmware data section start (relative to chip base)
`STTKDMA_CODE`	`0x6000`	Offset of SlimCORE firmware code section start (relative to chip base)
`TRACER_DATA`	`0x0140`	Offset of tracer's state variables (relative to STTKDMA_DATA)
`TRACER_CODE`	`0x05b7`	Offset of tracer's core routine (relative to STTKDMA_CODE) / starting location past the firmware code section

Table 12 Tracer's API variables.

Description

Instead of making use of the hardware features of a SlimCORE processor[28], tracer's implementation is based on an idea of a binary instrumentation. Traced instructions are translated into other instructions or their sequences. These instructions are executed by the tracer in such a way so that it is possible to maintain information about the contents of registers and jump targets in particular (whether conditional jumps were taken or not).

The tracer is composed of the following two parts:

- SlimCORE instruction disassembler and rewriter,
- core tracer routine.

The core tracer routine is copied at the end of an original firmware's code section[29]. It executes binary translated instruction sequences produced by the disassembler and rewriter as illustrated on Fig. 19.

[28] the `Run` I/O register from `slim_core_map`'s embedded `core` structure and SLIM_RUN_STOP SLIM_RUN_ENABLE and SLIM_RUN_STOPPED flags [7]

[29] code location 0x05b7 as original SlimCORE tracer's code has been implemented for firmware 3.1.6.

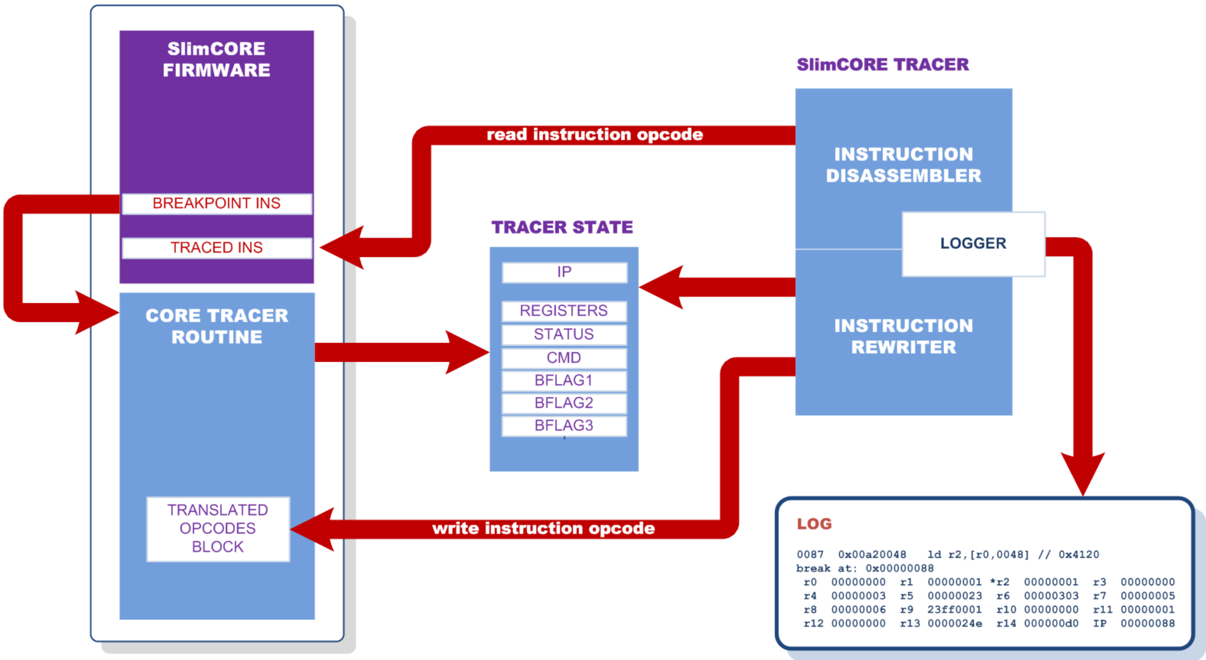


Fig. 19 SlimCORE tracer architecture.

The core tracer routine is entered when a breakpoint[30] is hit and it never exits. Its code executes in a loop as a response to notifications received from the tracer's disassembler and rewriter. The disassembler and rewriter parses SlimCORE instruction to execute from a given firmware location (denoted by tracer's IP variable), translates its opcode into a form suitable for the tracer and writes it back into a dedicated execution block of the core routine.

The tracer maintains state information in firmware data section location starting at offset 0x4140. The meaning of tracer state variables is illustrated in Table 13.

TRACER VARIABLE	OFFSET IDX	DESCRIPTION
R1-R14	0x00-0x0d	Variables holding saved SlimCORE registers (saved execution context)
DUMMY	0x0e	A dummy variable used by the tracer NOP instruction
STATUS	0x0f	A variable indicating that a core tracer routine has been reached (a breakpoint has been hit)
CMD	0x10	A variable indicating whether the tracer should proceed with execution of any translated instructions
BFLAG1, BFLAG2, BFLAG3	0x11-0x13	Variable indicating, which branch (1, 2 or 3) has been taken as a result of a given translated instructions' sequence execution

Table 13 Tracer's state variables.

Tracer gets executed as a result of hitting a breakpoint instruction. This instruction is a simple jump to the beginning of a tracer core routine:

```
public static final int BREAK       = 0x00d05b17; //JMP 0x5b7
```

[30] the interception breakpoint, there can be only one of it set.

CORE ROUTINE

The structure of a core tracer's routine is illustrated on Fig. 20.

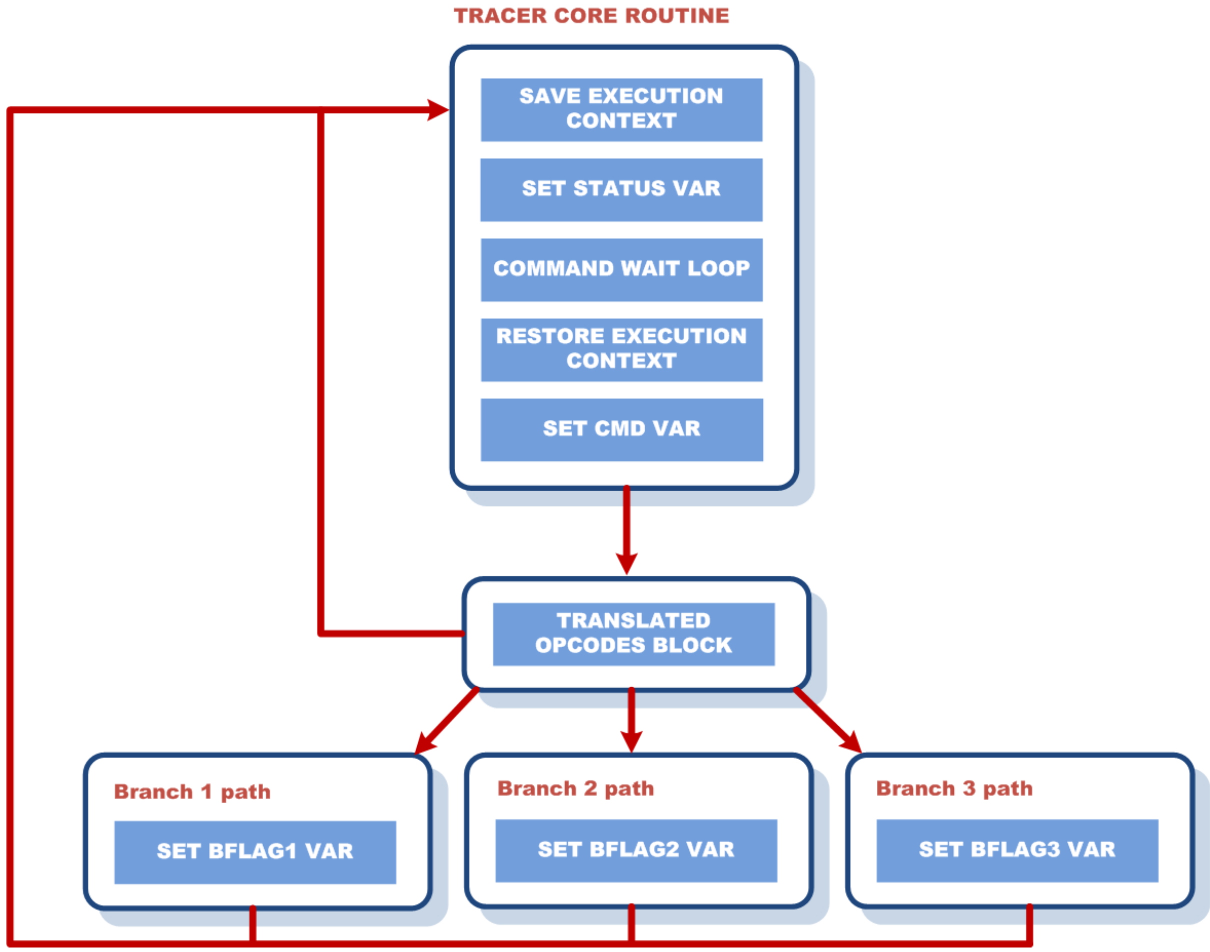


Fig. 20 Tracer's core routine implementation.

The core routine starts with an instruction sequence responsible for the saving of an original execution context. As a result, the contents of SlimCORE registers are stored into memory (variables R1-R14):

```
0x00b01050,//   st r1,[r0,0050] offset 0x05b7
0x00b02051,//   st r2,[r0,0051] offset 0x05b8
0x00b03052,//   st r3,[r0,0052] offset 0x05b9
0x00b04053,//   st r4,[r0,0053] offset 0x05ba
0x00b05054,//   st r5,[r0,0054] offset 0x05bb
0x00b06055,//   st r6,[r0,0055] offset 0x05bc
0x00b07056,//   st r7,[r0,0056] offset 0x05bd
0x00b08057,//   st r8,[r0,0057] offset 0x05be
0x00b09058,//   st r9,[r0,0058] offset 0x05bf
0x00b0a059,//   st r10,[r0,0059] offset 0x05c0
0x00b0b05a,//   st r11,[r0,005a] offset 0x05c1
0x00b0c05b,//   st r12,[r0,005b] offset 0x05c2
0x00b0d05c,//   st r13,[r0,005c] offset 0x05c3
0x00b0e05d,//   st r14,[r0,005d] offset 0x05c4
```


Next, the value of a STATUS variable is set to 0 to indicate that a breakpoint has been hit (that tracer's code has been reached):

```
0x00b0005f,//   st r0,[r0,005f] offset 0x05c5
```

Following that, the tracer waits in a loop for the CMD variable to change to the non-zero value. This happens when a tracer is notified by the instruction rewriter to execute next instruction (to single step over an instruction):

```
0x00a50060,//   ld r5,[r0,0060] offset 0x05c6
0x00c05000,//   cmp r5,#00 offset 0x05c7
0x009815c6,//   je  0x05c6  offset 0x05c8
```

Following that, the CMD variable state is restored to indicate a default state (a stop after an instruction execution):

```
0x00b0005e,//   st r0,[r0,005e] offset 0x05c9
0x00b00060,//   st r0,[r0,0060] offset 0x05ca
```

Next, saved SlimCORE registers context is restored to the original values:

```
0x00a10050,//   ld r1,[r0,0050] offset 0x05cb
0x00a20051,//   ld r2,[r0,0051] offset 0x05cc
0x00a30052,//   ld r3,[r0,0052] offset 0x05cd
0x00a40053,//   ld r4,[r0,0053] offset 0x05ce
0x00a50054,//   ld r5,[r0,0054] offset 0x05cf
0x00a60055,//   ld r6,[r0,0055] offset 0x05d0
0x00a70056,//   ld r7,[r0,0056] offset 0x05d1
0x00a80057,//   ld r8,[r0,0057] offset 0x05d2
0x00a90058,//   ld r9,[r0,0058] offset 0x05d3
0x00aa0059,//   ld r10,[r0,0059] offset 0x05d4
0x00ab005a,//   ld r11,[r0,005a] offset 0x05d5
0x00ac005b,//   ld r12,[r0,005b] offset 0x05d6
0x00ad005c,//   ld r13,[r0,005c] offset 0x05d7
0x00ae005d,//   ld r14,[r0,005d] offset 0x05d8
```

The block containing the traced (binary translated by the rewriter) instruction sequence gets executed:

```
0x00d00090,//   ins1 offset 0x05d9
0x00d00090,//   ins2 offset 0x05da
0x00d00090,//   ins3 offset 0x05db
0x00d00090,//   ins4 offset 0x05dc
0x00d00090,//   ins5 offset 0x05dd
```

As a result of the above, one of the code paths corresponding to branches of conditional instructions could be taken. If this is the case, proper BFLAG variable is set accordingly:

```
//branch 1
0x00b00061,//   st r0,[r0,0061] offset 0x05df
0x00d05b17,//   jmp 0x5b7 offset 0x05e0

//branch 2
0x00b00062,//   st r0,[r0,0062] offset 0x05e1
0x00d05b17,//   jmp 0x5b7 offset 0x05e2
```

```
//branch 3
0x00b00063,//   st r0,[r0,0063] offset 0x05e3
0x00d05b17,//   jmp 0x5b7 offset 0x05e4
```

After that, the core tracer's routine starts execution from the beginning (it waits in a loop for CMD flag to be set by the disasssembler and rewriter part indicating next instruction to execute).

INSTRUCTION DISASSEMBLER AND REWRITER

The SlimCORE instruction opcodes disassembler and rewriter processes SlimCORE firmware instructions and translates them into corresponding sequences for execution by the core tracer's routine.

Upon processing of a given instruction, the translated instruction or their sequence is written into the translated opcode block of a core tracer routine (indicated by the INS_OFF variable). The core tracer routine is notified via CMD variable that a next instruction is ready to be traced (that is should be executed in a single step manner).

Specific translation rules used by the instruction rewriter are briefly described in Table 14.

SOURCE INSTRUCTION (OPCODE)	TRANSLATED INSTRUCTION	DESCRIPTION
`WAITx`	`(opcode&0xfff000)|(INS_OFF&0xfff)`	Wait instructions are translated directly to the target PC location (INS_OFF)
`JMP reg`	`0xd00010|((BRANCH1_OFF&0xff0)<<4)|(BRANCH1_OFF&0x0f)`	Jumps through registers are translated to go through branch1 code path
`JMP imm`	`0xd00010|((BRANCH1_OFF&0xff0)<<4)|(BRANCH1_OFF&0x0f)`	Absolute jumps are translated to go through branch1 code path
`J imm`	`opcode&0xfffff000|(BRANCH1_OFF&0xfff)`	Absolute jumps are translated to go through branch1 code path
`RPT`	`opcode` `opcode2`	Repeat opcodes are translated directly along the instruction that follows it

`Jxx off1`
`Jxx off2`
`...`
	`opcode&0xfffff000|(BRANCH1_OFF&0xfff)`
	`opcode2&0xfffff000|(BRANCH2_OFF&0xfff)`
	`...`
		A sequence of conditional jumps following a given

		instruction is translated into corresponding conditional jumps going through branch code paths (1, 2 or 3)

Table 14 Translation rules used by the instruction rewriter.

Sample uses

The following code sequence starts tracing the execution of SlimCORE instructions from 0x86 firmware location:

```
STTKDMADebug.trace(0x86);
```

The above invocation produces the following output by the tracer logging routine:

```
break at: 0x00000086
 r0  00000000 *r1  00000001 *r2  00000100  r3  00000000
*r4  00000003 *r5  00000023 *r6  00000303 *r7  00000005
*r8  00000006 *r9  23ff0001  r10 00000000 *r11 00000001
 r12 00000000 *r13 0000024e *r14 000000d0  IP  00000086
0086  0x00e10001   mov r1,#0001

0086  0x00e10001   mov r1,#0001
break at: 0x00000087
 r0  00000000  r1  00000001  r2  00000100  r3  00000000
 r4  00000003  r5  00000023  r6  00000303  r7  00000005
 r8  00000006  r9  23ff0001  r10 00000000  r11 00000001
 r12 00000000  r13 0000024e  r14 000000d0  IP  00000087

0087  0x00a20048   ld r2,[r0,0048] // 0x4120
break at: 0x00000088
 r0  00000000  r1  00000001 *r2  00000001  r3  00000000
 r4  00000003  r5  00000023  r6  00000303  r7  00000005
 r8  00000006  r9  23ff0001  r10 00000000  r11 00000001
 r12 00000000  r13 0000024e  r14 000000d0  IP  00000088

0088  0x00722c21   bitval r2,r2,#0002
0089  0x00881091   jz l_0091
break at: 0x00000091
 r0  00000000  r1  00000001 *r2  00000000  r3  00000000
 r4  00000003  r5  00000023  r6  00000303  r7  00000005
 r8  00000006  r9  23ff0001  r10 00000000  r11 00000001
 r12 00000000  r13 0000024e  r14 000000d0  IP  00000091

0091  0x00a20070   ld r2,[r0,0070] // 0x41c0
break at: 0x00000092
 r0  00000000  r1  00000001  r2  00000000  r3  00000000
 r4  00000003  r5  00000023  r6  00000303  r7  00000005
 r8  00000006  r9  23ff0001  r10 00000000  r11 00000001
 r12 00000000  r13 0000024e  r14 000000d0  IP  00000092

0092  0x00721020   bitset r2,r1&0x01<<0
break at: 0x00000093
 r0  00000000  r1  00000001 *r2  00000001  r3  00000000
```



```
 r4  00000003  r5  00000023  r6  00000303  r7  00000005
 r8  00000006  r9  23ff0001  r10 00000000  r11 00000001
 r12 00000000  r13 0000024e  r14 000000d0  IP  00000093

0093  0x00d00090   sync
break at: 0x00000094
 r0  00000000  r1  00000001  r2  00000001  r3  00000000
 r4  00000003  r5  00000023  r6  00000303  r7  00000005
 r8  00000006  r9  23ff0001  r10 00000000  r11 00000001
 r12 00000000  r13 0000024e  r14 000000d0  IP  00000094

0094  0x00b02070   st r2,[r0,0070] // 0x41c0
break at: 0x00000095
 r0  00000000  r1  00000001  r2  00000001  r3  00000000
 r4  00000003  r5  00000023  r6  00000303  r7  00000005
 r8  00000006  r9  23ff0001  r10 00000000  r11 00000001
 r12 00000000  r13 0000024e  r14 000000d0  IP  00000095
...
```

REFERENCES

[1] STMicroelectronics
http://www.st.com

[2] STi7111 Low-cost HDTV satellite set-top box decoder for Microsoft VC-1, H.264 and MPEG-2
http://www.st.com/content/st_com/en/products/digital-set-top-box-ics/legacy-products/legacy-processors/sti7111.html

[3] SE-2011-01 Security weaknesses in a digital satellite TV platform
http://www.security-explorations.com/en/SE-2011-01.html

[4] SE-2011-01 Issues #17-19
http://www.security-explorations.com/materials/se-2011-01-st.pdf

[5] Security vulnerabilities of Digital Video Broadcast chipsets
http://www.security-explorations.com/materials/se-2011-01-hitb2.pdf

[6] IST FP6 PROSYD EU project
http://www.prosyd.org

[7] OpenDuckbox project
https://gitorious.org/open-duckbox-project-sh4/tdt

[8] Separation of Functional and Non-Functional Aspects in Transactional Level Models of Systems-on-Chip
https://pdfs.semanticscholar.org/82dd/50f218bb85c9ff221c2c766e3597fab68bdc.pdf

[9] Ideas regarding vulnerabilities in ST DVB chipsets
http://www.security-explorations.com/materials/se-2011-01_ideas.pdf

[10] STBus communication system concepts and definitions
http://www.st.com/content/ccc/resource/technical/document/user_manual/39/81/fa/c8/2e/4d/41/f5/CD00176920.pdf/files/CD00176920.pdf/jcr:content/translations/en.CD00176920.pdf

[11] Security threats in the world of digital satellite television
http://www.security-explorations.com/materials/se-2011-01-hitb1.pdf

APPENDIX A

By issuing different TKD commands we found out the following:

- bit 0 (encrypt / decrypt) of a TKD command did not influence the result of the command if destination was a key slot (commands such as 01xxxxxx, 04xxxxxx or 15xxxxxx). In such cases, a conducted operation was always the same. Upon the test done with respect to the 04ff0000 TKD command we conclude that this was always the decryption operation,
- bit 0 (encrypt / decrypt) influenced the result of the TKD command if destination was set to 0xff (ffxxxxxx commands).

The test below verifies the nature of the 0x04ff0000 TKD command. The test was conducted with the following values of the plaintext / encrypted Control Words:

```
CW1 [ 54 29 09 86 26 55 85 00 ] CW2 [ f2 cd 09 c8 d3 bf 30 c2 ] plaintext
CW1 [ 4e cd c9 e0 a0 52 bd 2f ] CW2 [ 35 39 76 bb a2 f3 9f 80 ] encrypted
```

1) First, the input data is set to the value of the encrypted Control Word:

```
test> input "e0 c9 cd 4e 2f bd 52 a0 e0 c9 cd 4e 2f bd 52 a0"
INPUT: e0 c9 cd 4e 2f bd 52 a0 e0 c9 cd 4e 2f bd 52 a0
```

2) Next, 0x04ff0000 TKD command is issued. Bit 0 (encryption / decryption) of the command is not set and this should indicate that the command does the encryption operation:

```
test> ed 0x04ff0000 0x00fa4000 0x008e1abc
tkcmd 04ff0000
[running SLIM code]
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

3) In the next step, input data is set to the block of zero values:

```
test> input "00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00"
INPUT: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

4) Then, 0xffff0401 TKD command is issued, which makes use of the key at slot 04 and does the decryption operation (due to the value of bit 0 set to 1):

```
test> ed 0xffff0401 0x00fa4000 0x008e1abc
tkcmd ffff0401
[running SLIM code]
b9 6a 0c e8 6c d6 44 2e b9 6a 0c e8 6c d6 44 2e
```

The result of the decryption operation is the following vector of data:


```
b9 6a 0c e8 6c d6 44 2e b9 6a 0c e8 6c d6 44 2e
```

5) Finally, a test is conducted that decrypts the input block of zero values with the use of the plaintext Control Word used as a decryption key. Pure Java API is used for that purpose:

```
test> tdes d "00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00" "26 55 85 00 54 29
09 86 26 55 85 00 54 29 09 86"

e8 0c 6a b9 2e 44 d6 6c e8 0c 6a b9 2e 44 d6 6c
```

In a result, the same data "b9 6a 0c e8 6c d6 44 2e b9 6a 0c e8 6c d6 44 2e" is obtained.

This confirms that the operation at step 2 did the DECRYPTION operation in a result of which, key slot at index 4 was loaded with plaintext Control Word value (encrypted Control Word was decrypted).

Finally, a quick test is conducted in order to verify whether bit 0 has any influence on the 0x04ff0000 command:

```
test> input "e0 c9 cd 4e 2f bd 52 a0 e0 c9 cd 4e 2f bd 52 a0"
test> ed 0x04ff0001 0x00fa4000 0x008e1abc
tkcmd 04ff0001
[running SLIM code]
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

test> input "00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00"
test> ed 0xffff0401 0x00fa4000 0x008e1abc
tkcmd ffff0401
[running SLIM code]
b9 6a 0c e8 6c d6 44 2e b9 6a 0c e8 6c d6 44 2e
```

The above proves that both 0x04ff0000 and 0x04ff0001 TKD commands give same results, thus bit 0 does not matter.

The test above also proves that the value of bit 0 (encrypt / decrypt) of TKD commands is not consistent across the whole TKD command space. It may either indicate encrypt / decrypt functionality or be fixed to the given operation (such as decryption).