

How to Test the Security of an Arm Cortex-M IoT device using Open Source and Free Tools

Dr David Long

Principal Member Technical Staff, Doulos



Agenda

- What are the issues?
- Use of models
- Reverse engineering with Ghidra
- Unicorn testbench
- Extracting state from GDB with GEF
- Fuzz Testing
- Alternatives to Testing

What are the issues?

- Vulnerabilities not easy to find/investigate?
- Understanding what attacker might do
- Free Web and Linux security tools not appropriate for Cortex-M?
- Commercial security tools/testers too expensive?

Use of Models

- Hardware might not be available for vulnerability tests
- Virtual platform can model physical device
 - Enables invasive tests without damaging hardware
 - Provides similar visibility to hardware debugger
 - Tests easy to automate with scripts
- Open-source QEMU emulator widely used for SW development
 - Requires device models (not just CPU)
- Unicorn emulator based on cut-down QEMU core

Simple Example Vulnerability (1)

```
char user[40] = "";
```

Buffer for input string

```
for (int32_t nchars=0; nchars < 40; nchars++) {  
    int32_t c = uart0_getchar(UART0_BASE_PTR);  
    user[nchars] = c;  
}  
do_bad();
```

Get string from UART

Write string to buffer

Function to process input string

Simple Example Vulnerability (2)

```
void doBad() {  
    char outbuf[512];  
    char buffer[512];  
  
    sprintf(buffer, "ERR Wrong command: %.40s", user);  
  
    sprintf(outbuf, buffer);  
}
```

Local variables on stack

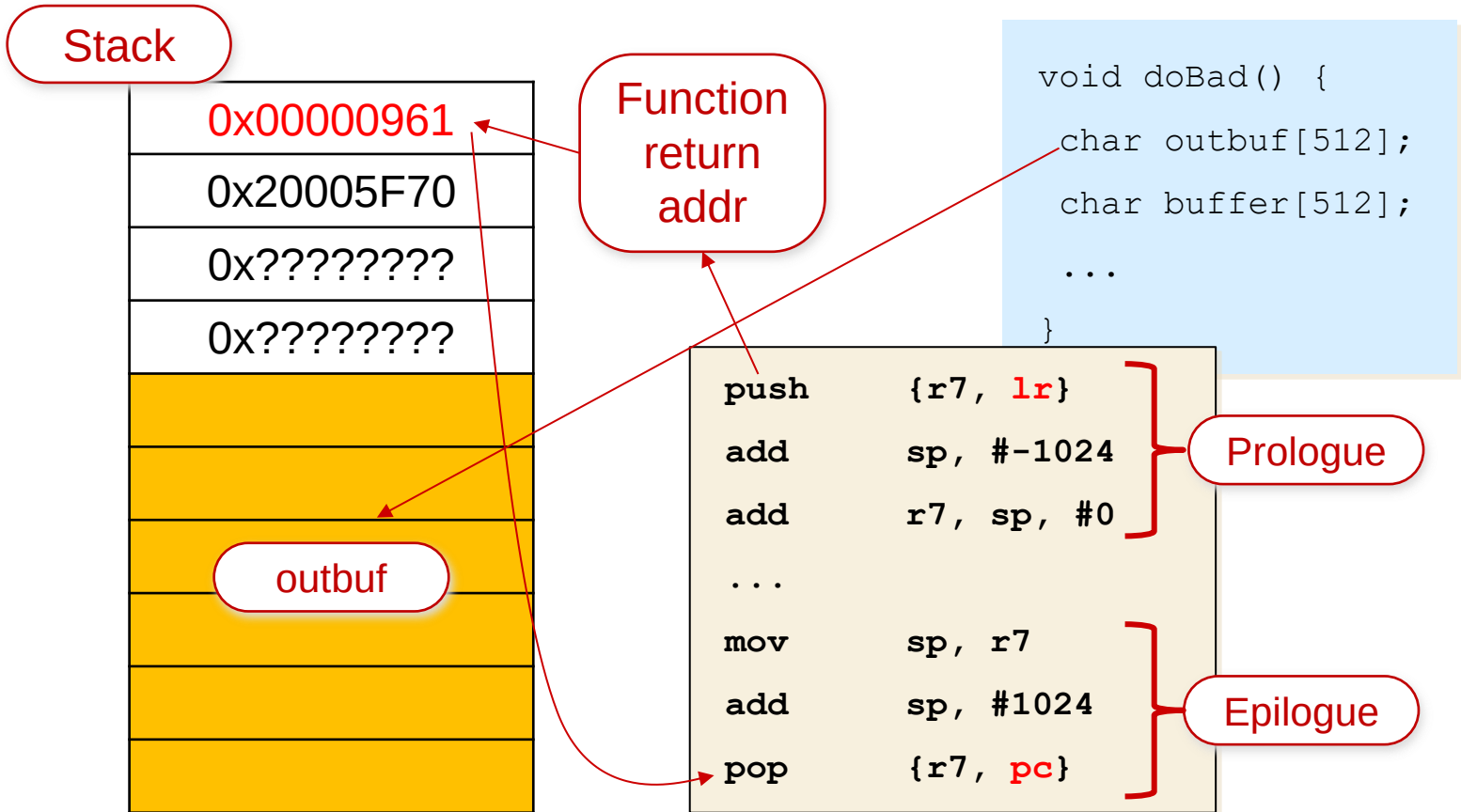
2nd arg is format string!

Malevolent Input string

%497d\x9d\x06\x00

Hex values

Simple Example Vulnerability (3)

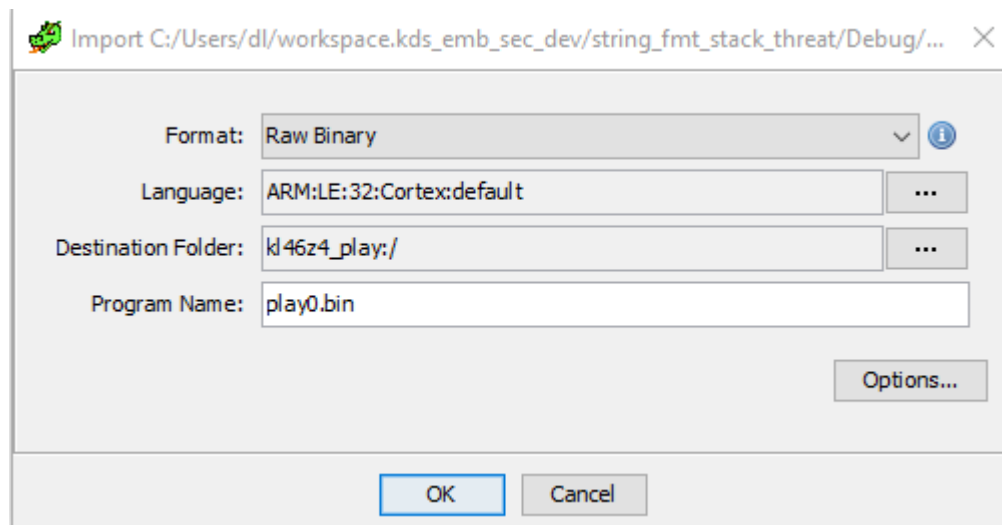


Ghidra

- Open Source Reverse Engineering tool developed by US National Security Agency (NSA)
- Runs on multiple platforms (requires Java JDK)
- Supports multiple architectures (including x86, Arm Cortex-M and Arm Cortex-A)
- Disassembles binary image (e.g. dumped from SPI FLASH)
- Generates readable C code from disassembled image
- Similar capabilities to IDA Pro (commercial tool)

Ghidra Import Raw Binary

- Need to select format of raw binary
 - format of .elf is auto-detected



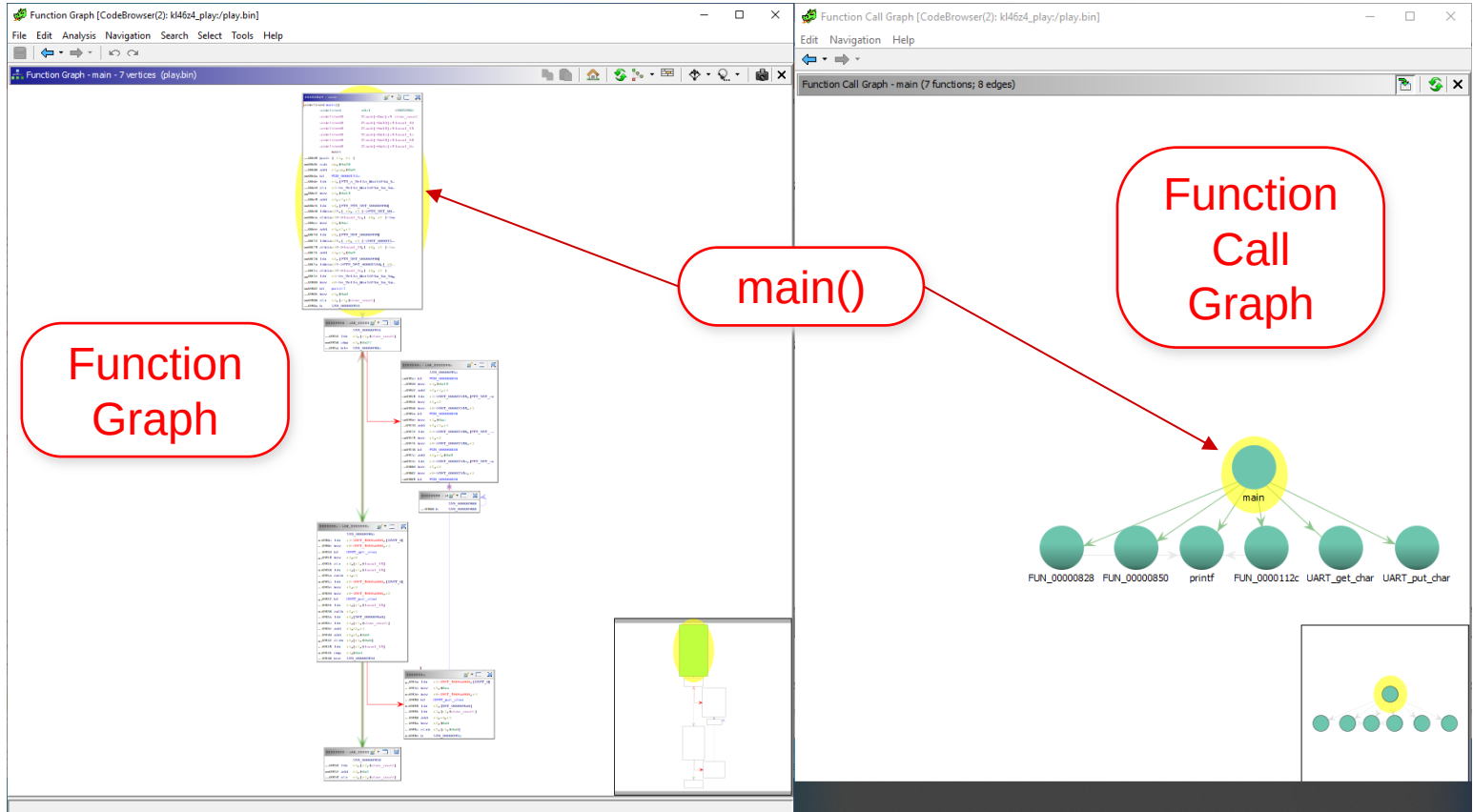
Ghidra Initial Results

The screenshot displays the Ghidra CodeBrowser interface for the file `ki46z4_play/play0.bin`. The interface is divided into several panes:

- Symbol Tree:** Located on the left, it lists various symbols including imports, exports, and functions. The function `FUN_000008d4` is highlighted.
- Disassembled:** The central pane shows the assembly code for the selected function. It includes instructions like `push {r7, lr}`, `sub sp, #0x28`, and `add r7, sp, #0x0`. A red circle labeled `main()` points to the function's entry point.
- Generated C:** The right pane shows the C code generated by Ghidra. It includes variable declarations like `undefined4 local_2c` and `uStack40`, and a loop structure with a `while( true )` block.
- Program symbols:** A red circle labeled `Program symbols` points to the `FUN_000008d4` entry in the Symbol Tree.

At the bottom, a console window shows the script `Console - Scripting`.

Ghidra Program Structure



Sources of Information

- If MCU known – look in Reference Manual!

39.2 Register definition

The UART includes registers to control baud rate, select UART options, report UART status, and for transmit/receive data. Accesses to address outside the valid memory map will generate a bus error.

UART memory map

Absolute address (hex)	Register name	Width (in bits)	Access	Reset value	Section/ page
4006_A000	UART Baud Rate Register High (UART0_BDH)	8	R/W	00h	39.2.1/749
4006_A001	UART Baud Rate Register Low (UART0_BDL)	8	R/W	04h	39.2.2/750
4006_A002	UART Control Register 1 (UART0_C1)	8	R/W	00h	39.2.3/750
4006_A003	UART Control Register 2 (UART0_C2)	8	R/W	00h	39.2.4/752
4006_A004	UART Status Register 1 (UART0_S1)	8	R/W	C0h	39.2.5/753
4006_A005	UART Status Register 2 (UART0_S2)	8	R/W	00h	39.2.6/755
4006_A006	UART Control Register 3 (UART0_C3)	8	R/W	00h	39.2.7/757
4006_A007	UART Data Register (UART0_D)	8	R/W	00h	39.2.8/758
4006_A008	UART Match Address Registers 1 (UART0_MA1)	8	R/W	00h	39.2.9/759

Table continues on the next page...

Ghidra Annotated Results

The screenshot displays the Ghidra IDE interface with the following components and annotations:

- Symbol Tree (Left):** Lists symbols including `FUN_000010`, `main`, `UART_get_char`, and `UART_put_char`. The `UART_0` label is highlighted.
- Listing: playbin (Center):** Shows assembly code for the `UART_0` device. Annotations include:
 - UART_0_S1:** Points to the `UART_0` label in the assembly listing.
 - UART_0_D:** Points to the `UART_0` label in the assembly listing.
 - UART_0:** Points to the `UART_0` label in the assembly listing.
- Decompile: UART_get_char (Top Right):** Shows the decompiled C code for the `UART_get_char` function. An annotation points to the `UART_0` label in the assembly listing.
- Decompile: main (Bottom Right):** Shows the decompiled C code for the `main` function. An annotation points to the `main()` label in the assembly listing.

Unicorn Emulator

- Uses QEMU JIT CPU emulation – fast!
- Written in C but also has Python API
- Supports multiple architectures, including Cortex-M
- Various levels of instrumentation provided:
 - Single-step or break on particular instruction
 - Memory accesses
 - Dynamic read/write of registers and memory
 - Exceptions/events with user-defined callbacks

Unicorn Basic Python Testbench

```
finput = open('idata.txt', 'rb')  
  
mu = Uc(UC_ARCH_ARM, UC_MODE_ARM)  
  
mu.mem_map(0x0, STACK_ADDR)  
mu.mem_map(0x40000000, 0x100000)  
  
#Open code from file  
file = open("play.bin", "rb")  
CODE = file.read(0x168c)  
  
#Write machine code to previously mapped memory  
mu.mem_write(0x0, CODE)  
  
mu.reg_write(UC_ARM_REG_SP, STACK_ADDR)  
  
mu.emu_start(ADDRESS+1, ADDRESS + 80)
```

ADDRESS = 0x00008d4

STACK_ADDR = 0x20006000

Unicorn – Adding a Callback

```
# callback for read of UART  
def hook_uart_read(mu, access,  
    address, size, value, user_data):  
    print("UART read")  
    mu.mem_write(address, finput.read(1))
```

```
UART0_D = 0x4006A007
```

Address of UART0_D register

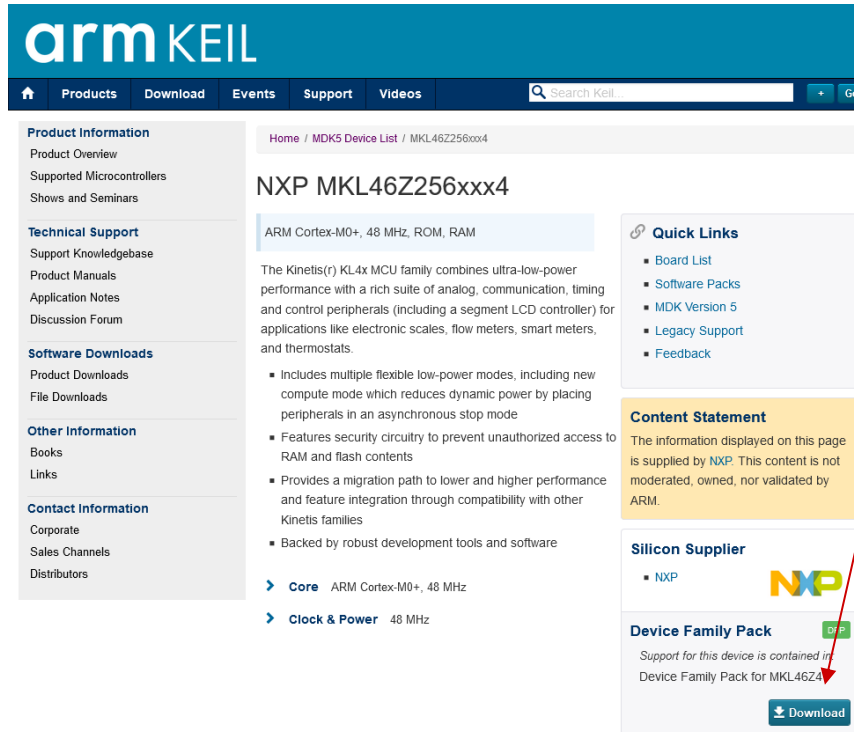
```
mu.hook_add(UC_HOOK_MEM_READ, hook_uart_read,  
    begin=UART0_D, end=UART0_D)
```


Unicorn – Display Registers

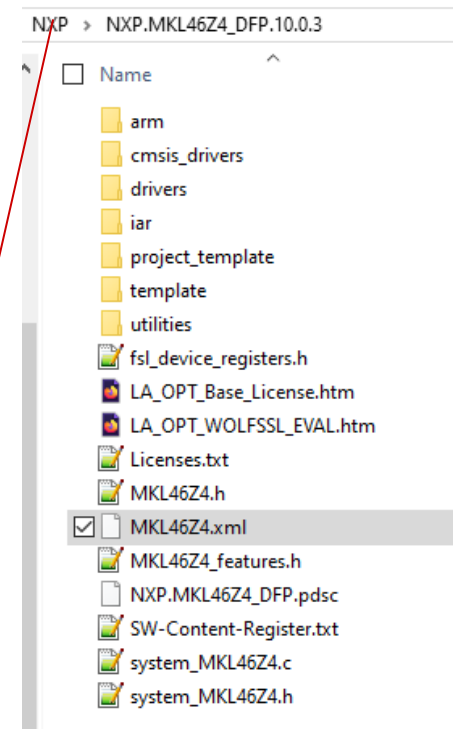
```
def dump_regs(mu):  
    r0 = mu.reg_read(UC_ARM_REG_R0)  
    r1 = mu.reg_read(UC_ARM_REG_R1)  
    r2 = mu.reg_read(UC_ARM_REG_R2)  
    ...  
    r6 = mu.reg_read(UC_ARM_REG_R6)  
    r7 = mu.reg_read(UC_ARM_REG_R7)  
    sp = mu.reg_read(UC_ARM_REG_SP)  
    print("Regs = {0x%0x,0x%0x,0x%0x,  
              0x%0x,0x%0x,0x%0x,0x%0x,0x%0x,0x%0x} "  
          %(r0,r1,r2,r3,r4,r5,r6,r7,sp))
```

Unicorn – Peripheral Reg Names

- Peripheral register details listed in CMSIS pack



The screenshot shows the ARM KEIL website. The header includes the 'armKEIL' logo and navigation links: Home, Products, Download, Events, Support, Videos, and a search bar. The left sidebar contains sections for Product Information, Technical Support, Software Downloads, and Other Information. The main content area displays the product 'NXP MKL46Z256xxx4' with its specifications: ARM Cortex-M0+, 48 MHz, ROM, RAM. It includes a description of the Kinetis(r) KL4x MCU family, a list of features (flexible low-power modes, security circuitry, migration path, and robust development tools), and quick links to Board List, Software Packs, MDK Version 5, Legacy Support, and Feedback. A content statement and silicon supplier (NXP) information are also present. At the bottom, there is a 'Device Family Pack' section with a 'Download' button.



The screenshot shows a file explorer view of the NXP MKL46Z4_DFP.10.0.3 CMSIS pack. The file list includes: arm, cmsis_drivers, drivers, iar, project_template, template, utilities, fsL_device_registers.h, LA_OPT_Base_License.htm, LA_OPT_WOLFSSL_EVAL.htm, Licenses.txt, MKL46Z4.h, MKL46Z4.xml (selected), MKL46Z4_features.h, NXP.MKL46Z4_DFP.pdsc, SW-Content-Register.txt, system_MKL46Z4.c, and system_MKL46Z4.h. A red arrow points from the 'Download' button in the KEIL website screenshot to the 'MKL46Z4.xml' file in this pack.

Unicorn – Peripheral Reg Names

Python cmsis_svd module can convert to pickle file

```
import pickle

from cmsis_svd.parser import SVDParser

parser = SVDParser.for_xml_file('MKL46Z4.xml')

svns = {}

for peripheral in parser.get_device().peripherals:
    for register in peripheral.registers:
        regname = "%s_%s"%(peripheral.name,register.name)
        regaddr = peripheral.base_address + register.address_offset
        svns.update ({regname:regaddr})

f = open('MKL46Z4.pickle', 'wb')
pickle.dump(svns, f)
f.close
```

Unicorn – Peripheral Reg Names

- Python testbench can load pickle file

```
import pickle  
  
fregs = open('MKL46Z4.pickle', 'rb')  
  
PREGS = pickle.load(fregs)
```

- Testbench code can now use register names

```
mu.mem_write(PREGS['UART0_S1'], struct.pack('I', 0xffffffff) )  
print("UART0_S1  0x%0x" %(PREGS['UART0_S1']))  
  
mu.hook_add(UC_HOOK_MEM_READ, hook_uart_read,  
            begin=PREGS['UART0_D'], end=PREGS['UART0_D'])
```

Unicorn - Disassembly

- Python capstone module disassembles code

```
import capstone

md = Cs(CS_ARCH_ARM, CS_MODE_THUMB + CS_MODE_LITTLE_ENDIAN)
```

- Callback can trace instructions

```
def hook_code(uc, address, size, user_data):
    tmp = CODE[address:address+size]
    for i in md.disasm(tmp, address):
        if (sz == 2):
            print("#0x%x:\t%s\t%s" % (u16(tmp), i.mnemonic, i.op_str))
    dump_regss(uc)
```

Unicorn Testbench Results

[Init]

UART0_S1 0x4006a004

[Emulation]

>>> Tracing instruction at 0x8d4,
instruction size = 0x2

>>> Instruction code at [0x8d4] =#0xb580: push {r7, lr}

Regs = {0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x20006000}

...

>>> Instruction code at [0x13e6] =#0x79db: ldrb r3, [r3, #7]

Regs =

{0x4006a000,0xa,0xffffffffe0,0x4006a000,0x0,0x0,0x0,0x20005fc0,0x20005fc0}

UART read ...

Improving Emulator Accuracy

- Can pre-load register and memory contents
- Memory and registers can be captured from hardware debugger
- GDB includes Python API
- GEF (gef.py) includes features to dump reg/memory

```
def dump_regs():  
    for reg in current_arch.all_registers:  
        reg_val = get_register(reg)  
        reg_state[reg.strip().strip('$')] = reg_val  
    return reg_state
```

Automating Vulnerability Tests

- Fuzzing is a technique to generate and apply a large set of input data to see if anything breaks!
- afl-unicorn combines popular "American Fuzzy Lop" fuzzer with Unicorn emulator
 - Support includes Cortex-A
 - Cortex-M requires some additional work

Alternatives to Vulnerability Tests

- Check for violations of CERT C coding rules
 - Requires static analysis tool
- Perform thorough Threat Modelling at design stage
 - Microsoft Threat Modeling Tool is free!
- Use a secure design framework
 - Arm PSA – see next slide

Arm PSA

Platform Security Architecture (PSA)

The open device security framework, with independent testing



PSA: enabling right-sized device security

1 © 2019 Arm Limited

arm

Arm PSA

Key Takeaways

PSA and PSA Certified Builds Trust in Devices and Data

PSA is a framework for security, enabling the right amount of security to be designed into devices through the PSA Root of Trust, **unlocking innovation opportunities and digital transformation**.

arm.com/psa

PSA Certified assesses the framework through



Testing solutions have **consistent developer APIs** for fundamental security functions



Providing **multi-level assurance and robustness testing** of the PSA-RoT

psacertified.org

For any questions contact Anurag.gupta@arm.com

3

© 2019 Arm Limited

arm

SoC Design & Verification

» SystemVerilog » UVM
» SystemC » TLM-2.0 » Arm

FPGA & Hardware Design

» VHDL » Verilog » Perl » Tcl
» Xilinx » Intel (Altera)

Embedded Software

» Emb C/C++ » Emb Linux
» Yocto » RTOS » **Security** » Arm

Python & Deep Learning

