Preliminary Security Analysis, Formalisation, and Verification of OpenTitan Secure Boot Code

Bjarke Hilmer Møller, Jacob Gosch Søndergaard, Kristoffer Skagbæk Jensen, Magnus Winkel Pedersen, Tobias Worm Bøgedal, Anton Christensen, Danny Bøgsted Poulsen, Kim Guldstrand Larsen, René Rydhof Hansen, Thomas Rosted Jensen, Heino Madsen and Henrik Uhrenfeldt

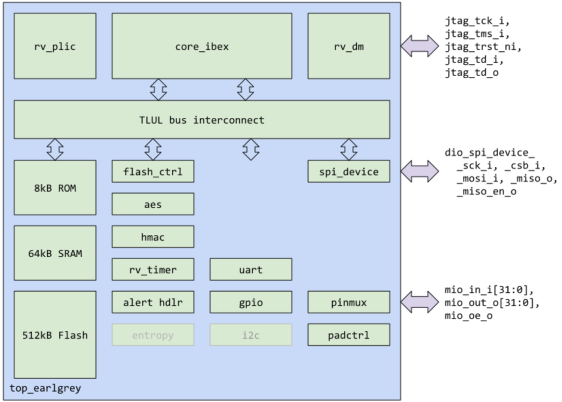
For Proceedings of the 26th Nordic Conference on Secure IT Systems (Nordsec 2021)

AALBORG UNIVERSITY
DENMARK

The case

The OpenTitan project

- RISC-V core
- Many security related built in peripheral modules accessible via MMIO
- Software implementation is lacking

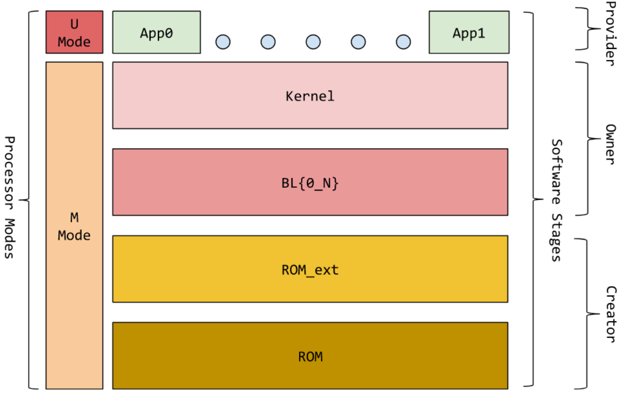


(cc) opentitan.org

The case

The OpenTitan project

- RISC-V core
- Many security related built in peripheral modules accessible via MMIO
- Software implementation is lacking



The goal

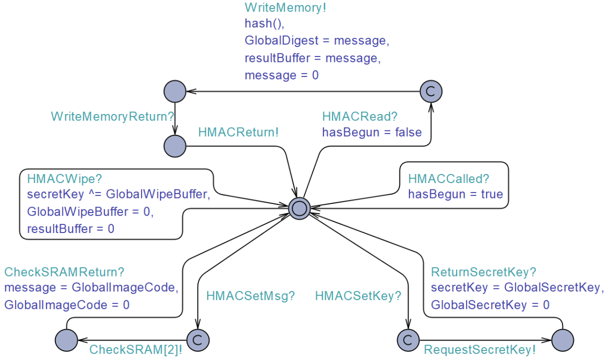
- Our device must always behave as expected.
- Only running the intended firmware.
- How can we ensure this?

The method

- Describe the intended properties of the system
- Model the system and ensure properties are present
- Compare and contrast with the system that's implemented

Tools

UPPAAL



CBMC

```
setKey(key);
setMsg(mes, size);
wipe(wipe_key);
char* hash = readResult();

__CPROVER_assert(
    __CPROVER_OBJECT_SIZE(hash) == 256 / 8,
    "PROPERTY 3: Hash is 256 bits");

__CPROVER_assert(
    __CPROVER_r_ok(hash, 256 / 8),
    "PROPERTY 3: hash is in readable address");

__REACHABILITY_CHECK

return hash;
```

Formal methods

Model checking

- Exhaustively checks if specified properties hold for a given model
- Clever pruning is done to reduce the work required
- The work lies in crafting the model and specifying the properties

Static analysis

- Iterates over a given programs to get as close to the exact result of a single property as possible
- May result in unhelpful but safe overestimates
- The work lies in specifying the property for each language construct

Theorem proving

- Validates the correctness of a given "pen and paper" proof
- The work lies in writing the proof, this is very time consuming
- Is well suited for almost any type of property

Secure boot- bootloader


```c
void mask_rom_boot(){
  policy_t boot_policy = read_boot_policy();
  rom_exts_manifests_t manifests = rom_ext_manifests(boot_policy);

  for (int i = 0; i < manifests.size; i++) {
    rom_ext_manifest_t current_rom_ext_manifest =
      manifests.rom_exts_mfs[i];
    pub_key_t rom_ext_pub_key = read_pub_key(current_rom_ext_manifest);
    if (!check_rom_ext_manifest(current_rom_ext_manifest) ||
      !check_pub_key_valid(rom_ext_pub_key) ||
      !verify_rom_ext_signature(rom_ext_pub_key,current_rom_ext_manifest))
        continue;
    pmp_unlock_rom_ext();
    if (!final_jump_to_rom_ext(current_rom_ext_manifest))
      boot_failed_rom_ext_terminated(boot_policy,current_rom_ext_manifest);
  }
  boot_failed(boot_policy);
}
```

Secure boot- bootloader - pseudo code

```
void mask_rom_boot(){
  signature = read_app_signature();
  app_entrypoint = get_app_entrypoint();
  if(check_signature(signature, app_entrypoint) != VALID) {
    handle_boot_failure();
  } else {
    pmp_unlock(app_entrypoint);
    app_entrypoint();
    handle_app_termination();
  }
}
```

Security analysis

The properties of the system being developed is broken down into three levels of increasingly specific detail

Policies The abstract properties

Goals May contain specific component level details

Properties Model specific description

Security analysis

P1: **The mask ROM must only execute code that securely transfers execution to a verified ROM EXT or terminates.**

P2: **Boot stages must only succeed in validating the following boot stage if the environment that the boot was initiated from is secure.**

P3: **Cryptographic material and other secrets must not be leaked.**

P4: **Access rights must be configured correctly.**

etc...

Security analysis

P1: The mask ROM must only execute code that securely transfers execution to a verified ROM EXT or terminates.

P2: Boot stages must only succeed in validating the following boot stage if the environment that the boot was initiated from is secure.

P3: Cryptographic material and other secrets must not be leaked.

P4: Access rights must be configured correctly.

- **G10:** Writing to a memory section requires writing privilege.
- **G11:** Reading from a memory section requires reading privilege.
- **G12:** Execution of a memory section requires execution privilege.

  etc...

Security analysis

P1: The mask ROM must only execute code that securely transfers execution to a verified ROM EXT or terminates.

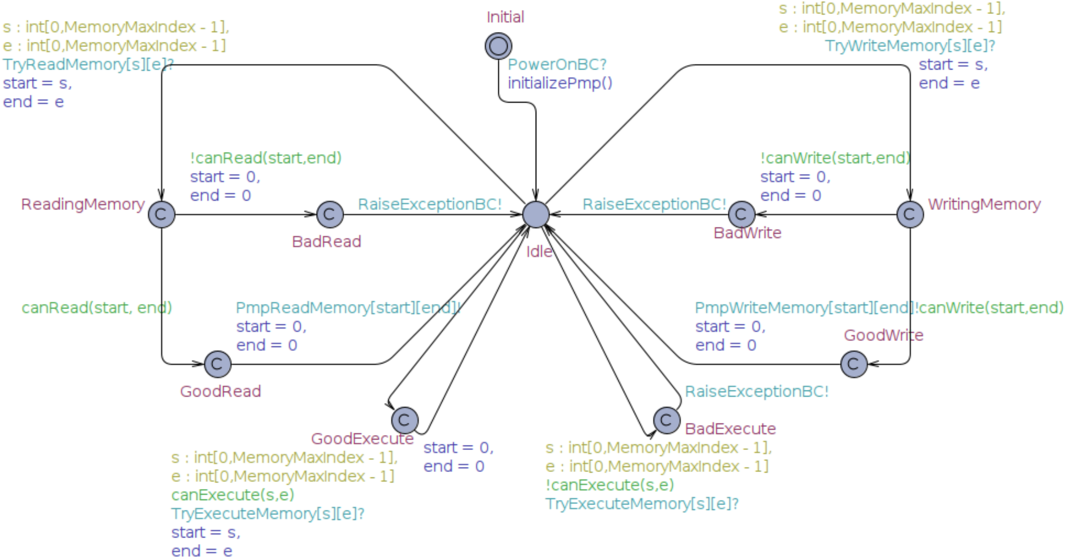
P2: Boot stages must only succeed in validating the following boot stage if the environment that the boot was initiated from is secure.

P3: Cryptographic material and other secrets must not be leaked.

P4: Access rights must be configured correctly.

- **G10:** Writing to a memory section requires writing privilege.
- **G11:** Reading from a memory section requires reading privilege.
- **G12:** Execution of a memory section requires execution privilege.
  - **Property 21:** `A[] !PmpModule.BadExecute`
  - **Property 30:** `E<> PmpModule.GoodExecute`
  - etc...

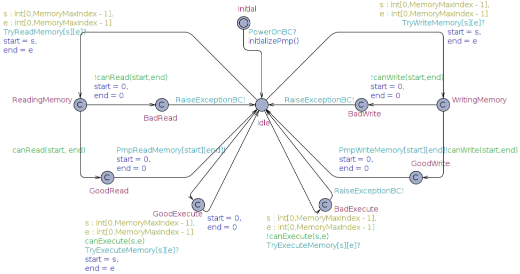
PMP module



Initial
PowerOnBC?
initializePmp()
s : int[0,MemoryMaxIndex - 1],
e : int[0,MemoryMaxIndex - 1]
TryReadMemory[s][e]?
start = s,
end = e
s : int[0,MemoryMaxIndex - 1],
e : int[0,MemoryMaxIndex - 1]
TryWriteMemory[s][e]?
start = s,
end = e
!canRead(start,end)
start = 0,
end = 0
!canWrite(start,end)
start = 0,
end = 0
ReadingMemory
BadRead
RaiseExceptionBC!
Idle
RaiseExceptionBC!
BadWrite
WritingMemory
canRead(start, end)
PmpReadMemory[start][end]!
start = 0,
end = 0
GoodRead
PmpWriteMemory[start][end]!
start = 0,
end = 0
canWrite(start,end)
GoodWrite
RaiseExceptionBC!
GoodExecute
start = 0,
end = 0
BadExecute
s : int[0,MemoryMaxIndex - 1],
e : int[0,MemoryMaxIndex - 1]
canExecute(s,e)
TryExecuteMemory[s][e]?
start = s,
end = e
s : int[0,MemoryMaxIndex - 1],
e : int[0,MemoryMaxIndex - 1]
!canExecute(s,e)
TryExecuteMemory[s][e]?

PMP module code


```
bool canExecute(int startIndex, int endIndex){
    int i;
    for(i = 0; i < 16;i++)
        if (PmpRegions[i].startAddress <= startIndex &&
            PmpRegions[i].endAddress >= endIndex)
            if (PmpRegions[i].execute)
                return true;
    return false;
}
```



Conclusion

Many goals covered (to some extent)

- G1: The cryptographic signature of the ROM EXT image must be verified by mask ROM before it is executed, to ensure authenticity and integrity of the image
  - Covered by both methods
  - Covers mainly control flow
  - Can't feasibly check if hashing and encryption works
- G8: Only authorised applications have access to cryptographic keys.
  - Limited coverage
  - Difficult to prove that no information leaks

HMAC vulnerability

Verilog RTL code for function that supposedly clears the secret key from the message signing module.

```
always_ff @(posedge clk_i or negedge rst_ni) begin
    if (!rst_ni) begin
        secret_key <= '0;
    end else if (wipe_secret) begin
        secret_key <= secret_key ^ {8{wipe_v}};
    end else if
        ...
    end
end
```

HMAC typical usage pattern

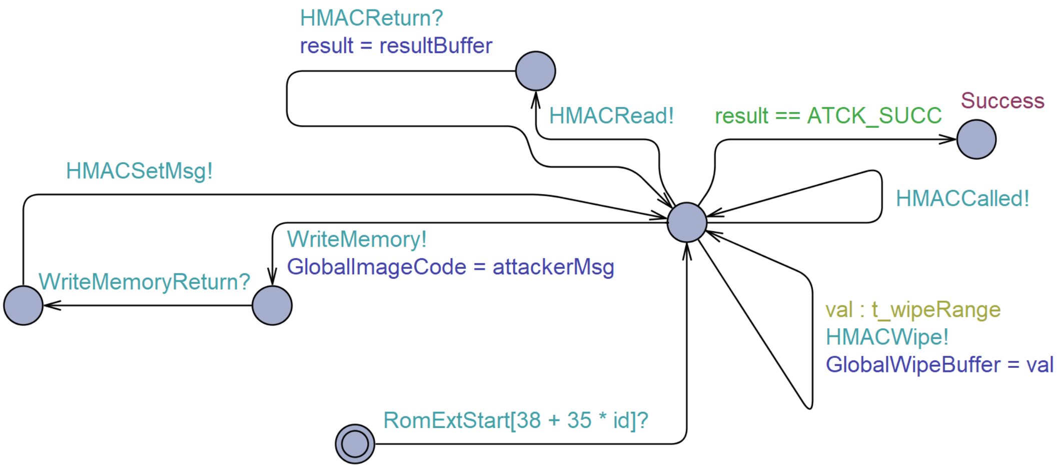
A typical use of the HMAC function

```
HMAC.setKey(key);
for(word in msg) {
    HMAC.setMsg(word);
}
HMAC.sign();
while(!HMAC.done()) {
    sleep();
}
result = HMAC.digest();
HMAC.wipe();
```

HMAC CBMC attacker model

```
unsigned int call_count = 3;
for(int i = 0; i < call_count; i++){
    BYTE ATTACKER_WIPE_KEY[WIPE_SIZE]; //32 bits arbitrary wipe key
    int n;
    __CPROVER_assume(n <= 2);
    switch(n){ //switch with nondeterministic n to model arbitrary call order
    case 0:
        setKey(ATTACKER_HMAC_KEY);
        break;
    case 1:
        char* mes = malloc(sizeof(char)*10);
        setMsg(mes, sizeof(char)*10);
        break;
    case 2:
        wipe(ATTACKER_WIPE_KEY);
        break;
    }
}
```

HMAC UPPAAL attacker model



Preliminary Security Analysis, Formalisation, and Verification of OpenTitan Secure Boot Code

Bjarke Hilmer Møller, Jacob Gosch Søndergaard, Kristoffer Skagbæk Jensen, Magnus Winkel Pedersen, Tobias Worm Bøgedal, Anton Christensen, Danny Bøgsted Poulsen, Kim Guldstrand Larsen, René Rydhof Hansen, Thomas Rosted Jensen, Heino Madsen and Henrik Uhrenfeldt

For Proceedings of the 26th Nordic Conference on Secure IT Systems (Nordsec 2021)

AALBORG UNIVERSITY
DENMARK