Detecting Possible Timing Attack Vulnerabilities in OpenTitan Big Number Accelerator Programs

Master's Thesis

Rasmus Nørgaard Fjeldsø
Simon Svendsgaard Nielsen

Aalborg University

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerløfs Vej 300
Phone: +45 99 40 99 40
Fax: +45 99 40 97 98
http://www.cs.aau.dk

Title:

Detecting Possible Timing Attack Vulnerabilities in OpenTitan Big Number Accelerator Programs

Topic:

Security

Project period:

February 1st 2022 - June 10th 2022

Project group:

cs-22-ds-10-01

Authors:

Rasmus Nørgaard Fjeldsø
Simon Svendsgaard Nielsen

Supervisors:

René Rydhof Hansen
Danny Bøgsted Poulsen

Page count: 50
Total page count: 54

Abstract:

OpenTitan is a chip designed to secure a wide range of devices. We focus on the OpenTitan Big Number Accelerator, a co-processor of the OpenTitan chip, used for security-sensitive asymmetric cryptographic algorithms. In this work, we implement a tool to detect potential timing attack vulnerabilities in OTBN programs. The tool utilises different techniques, such as model checking, statistical model checking and symbolic execution. For model checking and statistical model checking, we use Uppaal and Uppaal SMC, respectively. We first construct a control flow graph (CFG) representing the OTBN program, which we use to construct various Uppaal models representing the program. We then apply model checking and statistical model checking to find possible time differences. Some models are over-approximating, detecting time-differing traces unreachable in the original OTBN program.
We extend the implemented interpreter to work with symbolic values, which we use for the TraceChecker capable of testing whether the traces are unreachable. Lastly, we use the implemented tool to show a timing difference in the *RSA 3072 verify*. We then remove the timing difference and use the implemented tool to verify that the program run in constant time.

Summary

OpenTitan is an open source silicon Root of Trust (RoT) project with many different use-cases ranging from data centre integrations to embedded security applications such as security keys and smart cards.

We focus on the Big Number Accelerator (OTBN), a co-processor designed to execute security-sensitive asymmetric algorithms like RSA or Elliptic Curve. We first examine and present the different parts of this co-processor, namely the host communication and operational states, register files, controller, memory, errors and the instruction set. The description of the instruction set focuses on the instructions that impact the control flow or, contrary to most of the instructions, uses more than a single cycle. After introducing the OTBN co-processor, we construct an interpreter for OTBN programs, which we use to test different programs, and as an initial point for making several different analyses. We use these analyses to detect possible timing attack vulnerabilities in OpenTitan Big Number Accelerator (OTBN) programs.

The different analyses all rely on a control flow graph. To create this control flow graph, we make some assumptions regarding sub-routines, which includes the assumption that the OpenTitan team checks with their tools.
We omit the `JALR` instruction, which uses a run-time computed address to determine where to jump, complicating the static CFG construction. Furthermore, the OpenTitan team never uses the instruction in any available programs. We also present and discuss the official guidelines for using the hardware-assisted loops, which we assume to be followed to ease the modelling of the hardware-assisted loops when constructing Uppaal models from the CFG.

The first model (Uppaal no-data) we create disregards data manipulation. We use this model to check whether it can be verified that the program does not have any time-differing traces when we disregard the data operations. We facilitate this test using the model checking capabilities of Uppaal, particularly the liveness property. However, this property only answers whether there is a timing difference but does not provide any information about the size of the possible difference. To answer this question, we make the Uppaal no-data-SMC model and use another tool from the Uppaal family, Uppaal SMC. We can then determine whether or not the timing difference is great enough to proceed with other types of analyses.

We make the Uppaal with-data model, which considers the data manipulation of the modelled program to remove the over-approximation from the Uppaal no-data model. Applying model checking to verify the Uppaal with-data model is unfeasible due to the input range for OTBN programs, which leads to state-space explosion. However, the Uppaal with-data model can still be used with model checking to verify small ranges of input.
As an alternative, we create the Uppaal with-data-SMC model, which we use for simulations instead of verification. With this approach, we simulate the model with randomly

selected input, trying to find the inputs, which results in a timing difference if such input exists.

We extend the implemented interpreter to do symbolic execution for OTBN programs to test whether both the two traces received as the counter-example from the Uppaal no-data model are reachable. This is done by directing the symbolic execution to follow the traces which we want to test. To be considered complete, this solution, however, still needs the additional feature of removing traces in order to make Uppaal propose other possible differences.

Lastly, we use the implemented analyses on a test case, the program *RSA 3072 verify*, which originates from the official repository of OpenTitan. We find a potential time difference ranging between 144942 and 148062 cycles. We find two branch instructions causing the difference, which we remove by inserting no-operation (`NOP`) instructions into the shorter branches of the program. We then use the implemented tool to verify that the program executes in constant time.

Contents

Chapter 1

Introduction

In the modern world, software systems are integrated into all sorts of products, ranging from complex systems in satellites to embedded software systems in coffee machines. According to Statista[1], the number of IoT devices will almost double from 16.4 billion in 2022 to 30.9 billion in 2025.

OpenTitan is a chip designed to secure a wide range of devices. According to the official web page, this ranges from data centre integrations to embedded security applications such as security keys and smart cards [1]. One of the core elements facilitating the security of the OpenTitan chip is the co-processor named OpenTitan Big Number Accelerator (OTBN) [2]. We will refer to our previous work [3] for an informal description of the OTBN co-processor and a formal definition of the execution of OTBN programs. However, in Section 2.1 we present the most relevant elements of the OTBN co-processor as preliminary for the rest of this work. For the description of the OTBN assembly instructions, we focus on the most interesting ones in terms of executed cycles and control flow. In Chapter 4 we present and discuss the guidelines to follow when using the hardware-assisted loops constructed with the `LOOP` and `LOOPI` instruction and the problems caused by not following them. For the models we construct later, we assume that the OTBN programs follow these guidelines.

The OTBN co-processor is designed to execute security-sensitive asymmetric cryptographic algorithms like RSA and Elliptic Curve. The programs for OTBN are written using the OTBN assembly language and will mostly be implementations of different cryptographic algorithms [2].
In [4], Kocher shows that an attacker may be able to break different cryptosystems by measuring the amount of time required to perform private key operations. The time differences are due to performance optimisations, branching and conditional statements, and instructions running in non-fixed time [4].

In this work, we focus on possible timing attack vulnerabilities in OTBN assembly programs. To look for time differing executions of OTBN programs, we implement a tool to generate various models representing the control flow of OTBN programs, which we use to apply various formal methods. We use model checking with Uppaal [5], statistical model checking with Uppaal SMC [6]. Some of the models we construct do not consider the program input, which is an over-approximation because some time-differing traces discovered using these models are possibly unreachable. To test traces for reachability, we implement a symbolic interpreter using CVC4 [7], which we describe in Chapter 6.

The OpenTitan project also has various tools to check for common problems in OTBN

[1] `https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/`

programs and compliance with assumptions made regarding OTBN programs [2]. One of the tools checks for correct use of the call stack. This tool checks that only the jump instructions use the call stack. Another tool checks for the correct use of the loop instructions. This includes checking if an inner loop is fully contained within an outer loop and that there are no branches into or out of loop bodies. This tool returns warnings rather than errors since this is allowed but cannot be checked statically.

In Chapter 7 we utilise the developed tool to test the *RSA 3072 verify* program. As expected from the description of the program, we find a timing difference. We then examine the program and add additional no-operation instructions to the shorter paths, making the program run in constant time, which we verify using model checking.

[2]The different tools currently developed: `https://github.com/lowRISC/opentitan/tree/master/hw/ip/otbn/util`

Chapter 2

OpenTitan Big Number Accelerator (OTBN)

OpenTitan is an open source silicon Root of Trust (RoT) project. Since the project is open source, it will make the silicon RoT design more transparent, trustworthy, and secure [8]. The use-cases for OpenTitan are many. Adopters can integrate the design into data centre servers, storage devices, and other types of hardware [1]. As stated on the official OpenTitan documentation page, the mission for OpenTitan is to raise the security bar industry-wide [1]. One of the critical elements of this mission is the OpenTitan Big Number Accelerator (OTBN), which we will describe in Section 2.1. In Section 2.2 we describe the interpreter of the OTBN instruction set developed as part of this work.

2.1 OTBN

In this section, we introduce the OTBN co-processor, which, as mentioned in the introduction, is designed to execute security-sensitive asymmetric algorithms like RSA or Elliptic Curve [2]. We will examine the host communication and operational states, controller, memory, register files and the instruction set for the OTBN co-processor. We first explore how the host processor communicates with the OTBN co-processor to start operations and thereby change the operational state of the OTBN co-processor.

2.1.1 Host Communication and Operational States

There are multiple registers that the host processor has access to, which include the Command Register, Status Register, and the Operation and Result Register [2, Register Table]. As the name suggests, the host processor uses the Command register to issue commands to the OTBN co-processor. There are three possibly commands, `SEC_WIPE_DMEM` securely wipes the data memory and is performed after a program has been executed. The `SEC_WIPE_IMEM` command is similar to `SEC_WIPE_DMEM` but wipes the instruction memory [2, Operations and Commands]. The most interesting command for this work is the `EXECUTE` command which makes the OTBN co-processor start executing the program instructions currently in the instruction memory.

When the host processor issues the `EXECUTE` command, the OTBN co-processor transitions from the **IDLE** state to the **BUSY** state [2, Operational States]. If OTBN is not in the **IDLE** state when the command is issued, it will be ignored. The last operational state is the **LOCKED**, which happens whenever a fatal error occurs. The operational states of OTBN is illustrated on Figure 2.1.

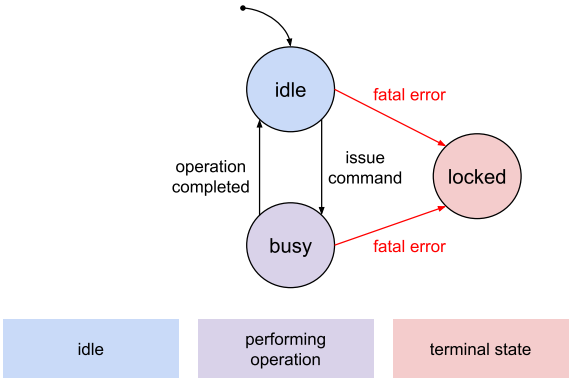


Figure 2.1: The operational states of OpenTitan [2, Operational States]

2.1.2 Register Files

In addition to the registers used to communicate with the host processor, the OTBN co-processor has two register files, both consisting of 32 registers. These are the General Purpose Registers (GPRs) and the Wide Data Registers (WDRs) [2, Processor State]. The registers in GPRs are 32-bit wide and named $x1$ to $x31$, while the wide registers in WDRs are 256-bit wide and named $w1$ to $w31$. There are two of the GPRs which have special behaviour, $x0$ ignores writes and always reads as zero, and $x1$ is the register used for reading or writing to the call stack. The registers in GPRs are accessible by the instructions in the base instruction subset, mainly used for control flow. In contrast, the WDRs can be accessed by the big number instruction subset, mainly used for data processing. Besides the GPRs and WDRs, OTBN also has 32-bit wide Control and Status registers (CSRs). These registers are used for special purposes, and can be seen on Table 2.2 and Table 2.1.

Register Number	Name
0x7C0	FG0
0x7C1	FG1
0x7C8	FLAGS
0x7D8	RND_PREFETCH
0xDC0	RND
0xFC1	URND

Table 2.1: This table shows one half of the CSRs

Register Number	Name
0x7D0	MOD0
0x7D1	MOD1
0x7D2	MOD2
0x7D3	MOD3
0x7D4	MOD4
0x7D5	MOD5
0x7D6	MOD6
0x7D7	MOD7

Table 2.2: This table shows the other half of the CSRs

The first three registers in Table 2.1 are for accessing the flag groups. In OTBN, there are two different flag groups, FG0 and FG1, which contain four different flags. The FLAGS register provides access to the collection of both FG0 and FG1 [2, Flags]. The four different kinds of flags are as follows.

- The Carry Flag (C) is 1 if the last arithmetic operation caused an overflow. Otherwise, it is 0.
- The MSb Flag (M) stores the most significant bit from the last arithmetic or shift

operation.

- The LSb Flag (L) stores the least significant bit from the last arithmetic or shift operation.
- The Zero Flag (Z) stores a 1 if the result of the last operation was zero. Otherwise, it stores a 0.

The remaining three CSRs in Table 2.1 are for random numbers. The random number which can be read from RND are fetched from an in-built Entropy Distribution Network (EDN). The random numbers coming from the EDN is primarily used for key generation [2, Wide Special Purpose Registers]. $RND\_PREFETCH$ is used to start a request to fill the RND cache, whenever the RND cache is empty [2, Wide Special Purpose Registers]. The random numbers read from $URND$ does not have guaranteed secrecy properties or specific statistical properties unlike RND [2, Wide Special Purpose Registers].

The $MOD0...MOD7$ in Table 2.2 are for reading a 32-bit subset of the MOD register, which can be found in WDRs. Since the register in WDRs is 256-bit, the MOD register only needs a single entry to retrieve the entire number. The RND and $URND$ registers in WDRs are similar to the RND and $URND$ registers in CSRs, but generate 256-bit random numbers. Last, we have the accumulator ACC register, which as the name states, is used to accumulate 256-bit information through the use of specific instructions.

Register Number	Name
0x0	MOD
0x1	RND
0x2	URND
0x3	ACC

Table 2.3: This table shows the WDRs, which are analogous to the CSRs but instead used by the big number instruction subset

2.1.3 Controller

The controller utilises a loop stack and a program counter (PC) to control the order in which the instructions are executed. For most instructions in the instruction set, the PC is incremented to hold the address of the consecutive instruction in the instruction memory. However, for some instructions, the PC is updated differently, for example, the branching (`BNE`) instruction, which sets the PC to a specified location if the values stored at two registers are distinct. If not, the PC is set to the consecutive instruction.

OTBN supports hardware-assisted loops, provided by the `LOOP` and `LOOPI` instructions. The controller maintains a loop stack, which it uses to update the program counter, such that the loop-body will be executed the correct number of times. Each element in the loop stack is a 3-tuple consisting of the number of iterations and the address of both the first and last instruction in the loop [2, Loop Stack]. `LOOP` and `LOOPI` each push a an element onto the loop stack, which allows for nested loops. However, the maximum depth of the stack is eight, meaning that a max of eight nested hardware-assisted loops is supported.

Algorithm 1 describes how the controller utilises the loop stack to update the program counter. The first condition in Algorithm 1 is a check to see if the loop stack is non-empty. If it is non-empty, the controller checks if the PC points to the end address of the loop at the top of the loop stack. If it does, it is checked whether the number of iterations is zero,

for which the loop-tuple at the top of the stack must be popped. If it is non-zero, the PC is updated to point at the start address of the loop. If the loop stack is empty, the PC will not be modified.

Algorithm 1 Pseudo code for the Controller

```
1: while STATUS = BUSY do
2:     if loopStack not empty then
3:         if CurrentPC = EndAddress then
4:             LoopCount − −
5:             if LoopCount = 0 then
6:                 LoopStack.pop
7:             else
8:                 ProgramCounter = StartAddress
9:     Run(InstructionAt(CurrentPC))
```

2.1.4 Memory

The layout for the memory follows the Harvard architecture [2, Design Details]. The two different memories in OTBN are both 4 kB. These are the instruction memory (IMEM) and the data memory (DMEM).

The instructions are stored in the instruction memory (IMEM). Each instruction in the OTBN instruction set, including both subsets, is 32-bit long, meaning that the instruction memory can contain a maximum of 1024 instructions. Since changes to the instruction in the IMEM are prohibited by the user, it cannot be read from or written to by user code.

The data memory contains 128 256-bit wide values and is accessible from both the base and the big number instruction subsets. Even though the DMEM, like the IMEM, is 4 kB, only the first 2 kB can be accessed by the host processor. This security feature makes OTBN applications able to store sensitive information in the second half of the DMEM, making it harder to leak confidential information [2, Design Details].

2.1.5 Errors

OTBN errors are classified into two categories, software errors and fatal errors [2]. If OTBN is not under attack, the software errors are due to a programmer's mistakes in the running OTBN program, and if OTBN is under attack, the software might not be the programmer's mistake, but this cannot be guaranteed [2, Errors]. Fatal errors are typically due to security properties being violated [2, Errors]. In this section, we only present the software errors, which can be seen on Table 2.4.

The BAD_INSN_ADDR occurs when an instruction memory access is out of bounds or unaligned. An instruction, which could cause this error, is `JAL` since the jump location could be unaligned. Similar to BAD_INSN_ADDR, there is also an error for illegal use of DMEM, BAD_DATA_ADDR, which occurs if data memory access is out of bounds or unaligned. The CALL_STACK error occurs if an instruction pops an element from the call stack when it is empty or if an instruction pushes an element onto the stack when it is full.

The LOOP error occurs if a branch, jump, or a loop instruction occurs as the last instruction in a loop, if the number of iterations in a loop is zero, or if the loop stack is full. The last software error is ILLEGAL_INSN, which occurs if the current instruction has illegal operands. This could be that the second operand of `CSRRS` is not a legal CSRs or that the value at *grd* is greater than 31 in `BN.LID`.

Software Errors
The BAD_INSN_ADDR error
The BAD_DATA_ADDR error
The CALL_STACK error
The LOOP error
The ILLEGAL_INSN error

Table 2.4: This table shows the possible software errors [2, Errors]

2.1.6 Instruction Set

The OTBN instruction set consists of the base instruction subset and the big number instruction subset. These contain many "standard" instructions such as `ADD` and `BN.ADD` for the addition of 32-bit and 256-bit words, respectively. However, in this section, we only present the instructions, which have a special impact on the techniques we use in later parts of this work.

The `LOOP` and `LOOPI` instructions are used to construct the hardware-assisted loops. These instructions take an immediate value k, representing the loop size. The instructions contained in the loop are then the k instructions following the loop instruction. How the number of iterations is defined is the only difference between the instructions. `LOOPI` uses an immediate value provided as an operand, and `LOOP` has to lookup the value of a provided register. The two different branching instructions are `BEQ` and `BNE`. They compare the two provided registers and jump to an address in the instruction memory provided as an immediate. The difference between the instructions is the condition on which they have to jump, `BNE` jumps when the values at the register are not equal, and `BEQ` jumps if they are equal. The `JAL` and `JALR` instructions jump to a specified address in the instruction memory. They writes address of the following instruction (return address) onto a register provided as an operand. If $x1$ is used, the return address is pushed onto the call stack. The significant difference is that `JALR` can add the values stored at a register to the jump address, resulting in a computed jump.

The `ECALL` instruction terminates the program execution by setting *INTR_STATE.done*, which signals to the host processor that the program is terminated. Additionally, when this instruction is executed, the error flags are cleared to ensure that the host processor sees the program termination as successful completion.
The `UNIMP` instruction aborts the execution of the program, and triggers the ILLEGAL_INSN error. The `NOP` instruction is a pseudo-operation, which have no effect, but it does increment the number of cycles.

Most of the instructions in the instruction set completes in a single cycle. However, some of them do not. The `LI` instruction uses two operands, namely a register and an immediate value, which it writes into the register. `LI` uses either one or two cycles depending on the value of said immediate. The reason for this is that `LI` is a pseudo operation, and it uses `LUI` for the 20 most significant bits and `ADDI` for the 12 least significant bits. The instruction `LA` takes the same operands as `LI`. The immediate, which is a named address, is stored in the register such that it is easier to refer to the value. `LA` is also a pseudo operation that, similar to `LI`, also varies between one and two cycles depending on the immediate value.

The instructions `LW` and `SW` both take two register and an immediate value. These instruction loads and stores to the data memory, respectively. `SW` takes a single cycles, while `LW` takes

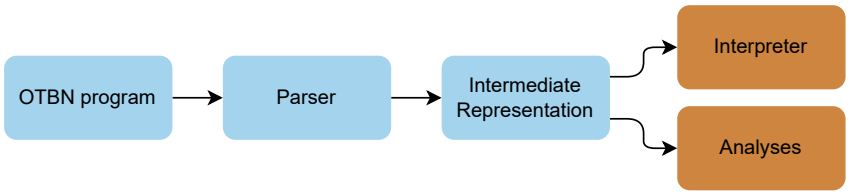


Figure 2.2: Representation of the different parts needed to make the interpreter and the further analysis.

two cycles. `LW` and `SW` are from the base instruction subset and therefore reads and writes 32-bit registers.

The `BN.LID` and `BN.SID` instructions are the big number instruction subset instructions for DMEM manipulations. The major difference from `LW` and `SW`, apart from using 256-bit values, is that `BN.LID` and `BN.SID` uses indirect addressing to specify which WDRs to use. This means that they both lookup values of the specified register – which must be from GPRs– to find which WDRs to write or read from. The instruction `BN.MOVR` also uses indirect addressing for selecting the content of a WDRs and copying it into another WDRs.

The `BN.SEL` instruction uses the value of a specified flag to select which of two provided registers to lookup. The result of the lookup is written to the specified destination register.

The `CSRRS` instruction is used to read and set bits in the special purpose register. The number of cycles changes if it uses the RND as the *csr* and the cache is empty. This is due to the generation of the random number. The time used is not specified in the [9]. However, observing the official simulator, we see that the amount of additional cycles is 18, which means that the total is 19 for one of these instructions when they use RND. This number of cycles is also the number we continue to use throughout this work. The instruction `CSRRW` reads/writes a special purpose register. This instruction is always completed in a single cycle. However, if it uses $RND\_PREFETCH$ as the *csr*, it starts the generation of a new random number, and the cache will therefore be non-empty. This means that `CSRRS` instruction does not take additional cycles, even when using RND. The instruction `BN.WSRR` can like `CSRRS` read from RND, which takes 18 cycles if the cache is empty. However, $RND\_PREFETCH$ is only accessible from the CSRs, which means that the instruction `BN.WSRW` does not take additional cycles.

To better reason about the behaviour of OTBN programs and measure the execution time, we implement an interpreter for all instruction in the instruction set.

2.2 Interpreter

To implement the interpreter, we first create a parser for OTBN programs, which stores the instructions as an intermediate representation, which will be the input for the interpreter. It will also be the basis for the analysis presented later. This is illustrated on Figure 2.2. To ease the parsing of the OTBN programs, we make minor changes to the OTBN label syntax. The changes add **lb:** as a prefix in label declarations, and when using a label, said label adds **lb** as a prefix.

The intermediate representation consists of all the information required to execute the instructions, namely the operands, i.e. the immediate and registers. Additionally, each instruction does also have a function describing how it modifies the state.
An example of such a function is shown on Listing 2.1. This function implements the se-

mantics for the SUB instruction. As seen in the function, it checks for the possible errors, in this case, the CALL_STACK error, and if the error occurs, the interpreter stops the entire execution and reports what instruction failed. Besides the error handling, it also implements the desired operation. In this instance, it is a subtraction between two registers and after that operation, it increments the program counter such that the interpreter knows which function should run next.

The interpreter itself runs the instructions in the OTBN program until it reaches an ECALL instruction or an error has occurred. During execution, the interpreter counts the number of cycles used. It is also the interpreter which implements the controller functionality described in Section 2.1.3.

```
1  void Sub::run(State &s) {
2      ErrorChecker::checkForCallStackError(s.errorStatus,
       ↪this->readsFrom(), this->writeTo(), s.callStack);
3      if (s.errorStatus.anyError()) {
4          return;
5      }
6
7      uint32_t grs1Value = s.lookUpGPR(grs1);
8      uint32_t grs2Value = s.lookUpGPR(grs2);
9      s.popCallStackIfNeeded(this->readsFrom());
10     s.writeGPR(grd, grs1Value - grs2Value);
11     s.programCounter++;
12 }
```

Listing 2.1: The function implementing the SUB instruction.

Chapter 3

Control Flow Graphs

For control-flow sensitive analysis, it is more convenient to view the program as a control flow graph. Frances E. Allen [10] expresses the control flow relationships in a directed graph, which was one of the first descriptions of control flow graphs. Control flow graphs are directed graphs with nodes representing basic blocks and edges representing control flow paths.

A basic block is a sequence of program statements with a single entry point and a single exit point [10]. The programs written for OTBN are relatively small, and we, therefore, represent each instruction in a program as a node in the control flow graph for the program instead of constructing basic blocks. Figure 3.1 shows the control flow graph for the short OTBN program at Listing 3.1 consisting of four instructions.

In the following section, we examine how we can construct the control flow graphs.

```
1 li x2, 2
2 li x3, 3
3 add x4, x3, x2
4 ecall
```

Listing 3.1: A small OTBN program.

3.1 Control-Flow-Graph Construction

Due to the choice of having no basic blocks and instead a node in the graph for each instruction, the first step to construct a CFG is to create a node for each instruction in the program. To make sure that the control flow graphs have a single entry and exit point, the non-instruction nodes, **StartNode** and **EndNode** are added [11], as illustrated at Figure 3.1.

The next step will be to connect the nodes with edges representing the program's control flow. As illustrated by the control flow graph on Figure 3.1 this is done by connecting

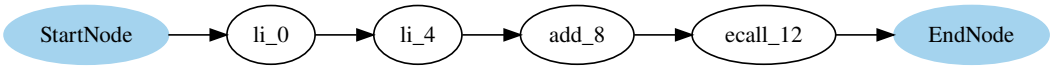


Figure 3.1: Control flow graph for the small OTBN program at Listing 3.1.

each instruction-node to the instruction-node for the instruction which will be executed next. For most instructions, this will be the instruction at the consecutive address in the instruction memory. However, this is not the case for the `JALR`, `JAL`, `BEQ`, `BNE`, `LOOP`, `LOOPI` and `RET` instructions. Furthermore, no instruction will be executed after an `UNIMP` or `ECALL` instruction, which we model as an edge from the `UNIMP` or `ECALL`-node to the **EndNode**. For the other instructions mentioned above, we start by omitting the `JALR` instruction, which performs a jump to a run-time computed instruction address. It would be necessary to over-approximate the possible addresses. However, we omit it since none of the currently available OTBN programs on the official GitHub page uses the `JALR` instruction[1].

To model the instructions in a control flow graph, we will make some assumptions about their use regarding sub-routines. However, the OTBN language does not have any function construct. To make separating the "main" program and the sub-routines easier, we assume that the programmer has annotated where each sub-routine starts and ends using `startf` and `endf`. Listing 3.2 shows a simple OTBN program with a single call (at line 2) to a sub-routine defined at lines 5-9. We can then trivially slice the program into sub-routines and the "main"-program, which are the instructions not encapsulated by `startf` and `endf`.

```
1 li x2, 20
2 jal x1, lbfun
3 ecall
4
5 startf
6 lb:fun
7     li x30, 123
8     ret
9 endf
```

Listing 3.2: OTBN program with an annotated sub-routine using `startf` and `endf`.

The assumptions we make are all created to prohibit jumping between sub-routines. We assume the following.

1. The call stack ($x1$) is only used for calls to sub-routines, such that the `RET` instruction can read the return address at the top of the call stack. The official tool mentioned in the Chapter 1 is used to check exactly this, making it a fair assumption. The instruction at line 3 in Listing 3.3 is therefore not allowed, while the instruction at line 2 is allowed.

2. The only approach used to call sub-routines is by using the `JAL` instruction with $x1$ as the first operand, such that the return value is stored on the call stack. The `JAL` instruction at line 1 in Listing 3.3 is therefore allowed, while the `JAL` instruction at line 2 is not.

3. `JAL` instructions with other registers than $x1$ as the first operand should only jump to instructions within the same sub-routine as the `JAL` instruction itself (a normal jump, which is not a call to a sub-routine). The `JAL` instruction at line 4 in Listing 3.3 is therefore not allowed, while the `JAL` instruction at line 5 is allowed.

[1] The programs in crypto `https://github.com/lowRISC/opentitan/tree/master/sw/otbn`

4. All sub-routines must end with a `RET` instruction. The instruction on line 17 in Listing 3.3 does not break this assumption allowed

5. `RET` instructions only appear within sub-routines. This instruction on line 10 in Listing 3.3 does break this assumption since the `RET` instruction is not within a sub-routines.

6. The `BEQ` and `BNE` instructions only jump to addresses within the same sub-routine as the `BEQ` or `BNE` instruction itself.

```
1  jal x1, lbfunc // good #1 and #2
2  jal x2, lbfunc // bad #2
3  li x1, 4 // bad #1
4  jal x2, lbInAnotherFunc // bad #3
5  jal x2, lbwithinSameFunc // good #3
6  nop
7  lb:withinSameFunc
8  nop
9
10 ret // bad #5
11
12 startf
13 lb:func
14     nop
15     lb:InAnotherFunc
16     nop
17     ret // good #4
18 endf
```

Listing 3.3: OTBN program exemplifying both allowed and disallowed instructions. The numbers in the comments (//) refer to the assumption number.

We can statically check if the assumptions are fulfilled. If they are, we can construct the control flow graph for a program by following a technique similar to the one Anders Møller et al. describe in [11].

The first step is to slice the program into sub-routines and the remaining instructions, which we refer to as the main-program. Similar to Anders Møller et al. [11] we use a **EnterFunc** and a **ExitFunc** node to represent calls to subroutines and returning from subroutines, respectively. By using the non-instruction nodes, the CFGs for the main program and the sub-routines can be constructed separately and glued together afterwards. To ease the CFG construction for sub-routines with multiple return instructions, we use additional non-instruction nodes to give each sub-routine a single entry and exit point (**StartFunc** and **EndFunc**).

We start by constructing the CFG for the main-program. If the original program contained calls to sub-routines, the graph would not be connected, but a **EnterFunc** and **ExitFunc** node-pair is inserted where a sub-routine CFG will be connected. For the program in Listing 3.2, the CFG for the main-program is the two left-most sub-graphs of Figure 3.2. For each pair of **EnterFunc** and **ExitFunc** nodes, we construct a CFG for the sub-routine and connect the **EnterFunc** to the **StartFunc**, and the **ExitFunc** to the **EndFunc** node.

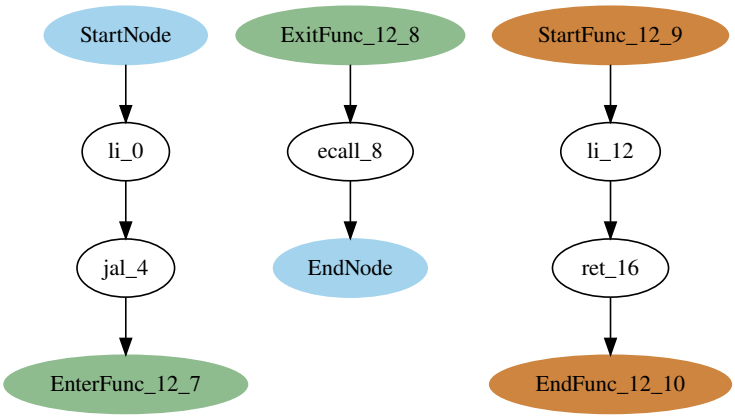


Figure 3.2: The two left-most sub-graphs is the CFG for the main-program in Listing 3.2. The right-most graph is the CFG for the sub-routine also from Listing 3.2.

This results in the control flow graphs for the sub-routines to be in-lined. After connecting the subroutine on Figure 3.2 to the main-program, we get the graph depicted on Figure 3.3.

We handle each type of OTBN instruction in the following manner to construct the sub-routine CFGs that we later connect to the main-program.

```
1  li x2, 20
2  jal x0, lbSkip
3  li x3, 30
4  lb:Skip
5
6  loopi 10, 3
7    beq x2, x3, lbjump
8      jal x1, lbfun
9      nop
10   lb:jump
11 ecall
12
13 startf
14 lb:fun
15     li x30, 123
16     ret
17 endf
```

Listing 3.4: OTBN program, which exemplifies all the rules for CFG construction.

- There are two different scenarios for the `JAL` instruction. The first scenario is when the register for `JAL` is $x1$ and is, therefore, a call to a sub-routine. For this scenario, we create a **EnterFunc** connected to the `JAL` instruction node and an **ExitFunc** node connected to the node for the instruction following the `JAL` instruction, namely the instruction to which the sub-routine will return. For the second scenario, where the `JAL` is a simple jump, we add an edge connecting the `JAL`-node to the node for the instruction at the jump address.

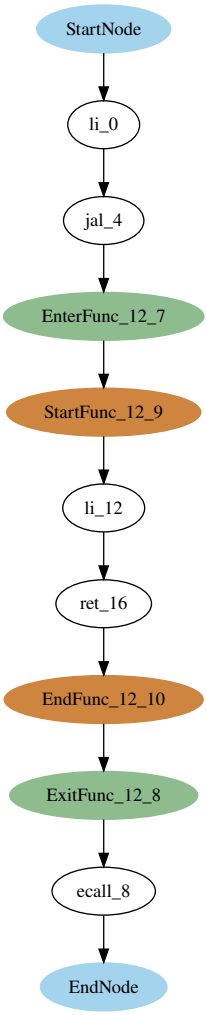


Figure 3.3: The final CFG for the program at Listing 3.2, after connecting the CFG for the sub-routine to the CFG of the main-program.

- To model `BEQ` and `BNE`, we create two edges from the branch instruction. One of the edges will connect to the node for the instruction to which the branch instruction can jump. We connect the second edge to the node for the instruction following the branch instruction.
- For the `LOOP` and `LOOPI` instructions, we add two edges—one from the node for the loop instruction to the node for the first instruction in the loop. The second edge is from the node for the last instruction in the loop to the node for the first instruction in the loop.
- For `RET` instructions, we create a **EndFunc** node and an edge from the `RET` instruction node to this node.
- `ECALL` and `UNIMP` are modelled by adding an edge from the node for the instruction to the **EndNode**.
- All other instructions are modelled by adding an edge from the node for the instruction to the node for the consecutive instruction in the instruction memory.

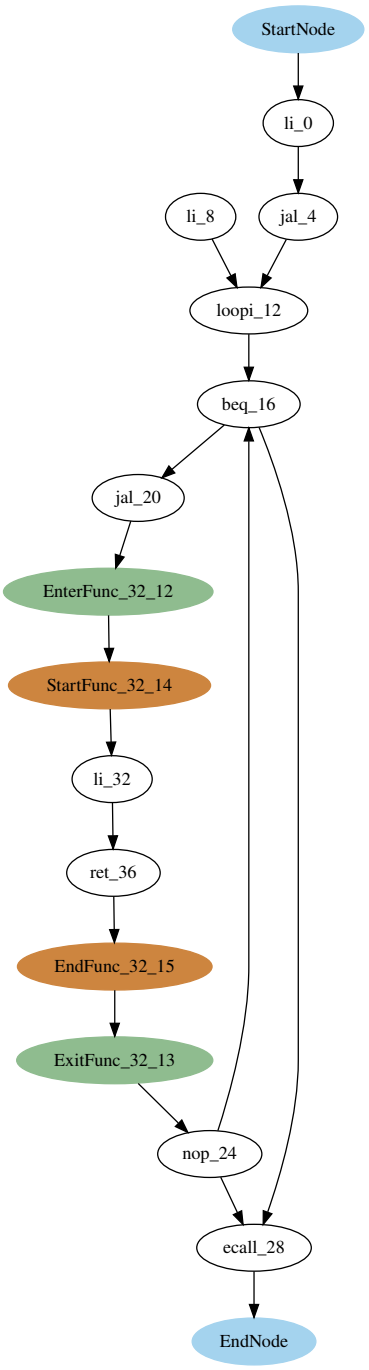


Figure 3.4: CFG using some of the special instructions, mentioned in Section 2.1.6.

We utilise all of the rules above to construct a control flow graph for the program at Listing 3.4, which includes a loop constructed using the `LOOPI` instruction. The loop contains a branching point that will either go to a function call or directly to the `ECALL` instruction, which is the end of the program.

Now that we have explored how to construct control flow graphs for OTBN programs, we can utilise them for multiple analyses.

Chapter 4

Rules for Hardware-Assisted Loops (`LOOP` and `LOOPI`)

In an earlier work [3], we explored the guidelines for the programmer to follow in order to avoid polluting the loop stack. In the this chapter, we revisit the rules, the consequences of not following them, and describe how to statically check if a program adheres to some of the rules.

A loop-tuple is pushed to the stack using the `LOOP` or `LOOPI` instructions. The controller afterwards uses the information in the tuple to execute the loops correctly. OTBN permits loop nesting and both branching and jumps inside a loop. However, the only way a loop-tuple can be popped from the loop stack is by executing the last instruction of the loop. Therefore, there is no support for early termination of loops, and only one loop-tuple can be popped per instruction. These limitations give rise to unexpected behaviour if a programmer tries to short circuit a hardware-assisted loop. Because of this, the documentation states the following three guidelines a program must adhere to in order to avoid polluting the loop stack and experiencing surprising behaviour[2, Using hardware loops].

- Even if there are branches and jumps within a loop body, the final instruction of the loop body gets executed exactly once per iteration [2, Using ardware loops].
- Nested loops have distinct end addresses [2, Using ardware loops].
- The end instruction of an outer loop is not executed before an inner loop finishes [2, Using ardware loops].

4.1 Discussion

The first and third guidelines are a bit unclear, and we will therefore discuss them in the following. We know that the only possible way the iteration count in the loop stack can be decremented is by running the last instruction in the loop. Therefore, it is impossible to run the last loop instruction more than once per iteration due to how the controller works. Therefore, we interpret the first guideline as saying that if the execution flow leaves the loop-body, it must return to run the last instruction. The consequence of not following the first guideline is that the loop-tuple representing the loop will never be popped from the loop stack, which does not necessarily lead to surprising behaviour. However, if the guideline is violated multiple times, the loop stack will eventually run out of space, which leads to a software error and a stopped program.

```
1 li x2, 3000
2 loop x2, 2
3   loopi 4, 1
4     addi x3, x3, 2
5
6 li x7, 12
7
8 Output:
9 x3 = 8
```

Listing 4.2: Inner and outer loop with the same end-address.

```
1 li x2, 3000
2 loop x2, 3
3   loopi 4, 1
4     addi x3, x3, 2
5   nop
6 li x7, 12
7
8 Output:
9 x3 = 24000
```

Listing 4.3: Inner and outer loop with distinct end-addresses.

The program in Listing 4.1 violates the first guideline by skipping the last instruction in the loop (the LI instruction) and never returning to the loop-body again. If this is done for eight different loops, the program will terminate due to a call stack error.

```
1 loopi 3, 3
2     lui x4, 6
3     jal x2, lbout_of_loop
4     li x4, 6
5 lb:out_of_loop
6 nop
7 ecall
```

Listing 4.1: OTBN program not following the third guideline.

The second guideline exists because the controller cannot pop more than a single loop-tuple from the loop stack at a time. This means that when the last instruction of the inner loop is executed, the controller will not consider the new top of the stack and restart the loop if there are more iterations left of the outer loop but instead move on with the next instruction in the instruction memory. As a side note, there is a GitHub Issue[1] for OpenTitan where some contributors discuss the need for a style/secure guideline for OTBN programs. One of the mentioned guidelines is that the versions of LOOP and LOOPI supporting labels should be used when available, which indicates that the team know the syntax where the programmer needs to count instructions to see where a loop ends is problematic.

The consequence of not following the second rule is that the outer loop will only be executed once. An example is provided in the following. Listing 4.2 and Listing 4.3 contains almost identical programs with the only exception that Listing 4.3 have an additional NOP instruction to make sure that the two loops have distinct end addresses. For the program in Listing 4.2, where the rule is violated, the controller will not make the control flow return to the beginning of the outer loop, which results in the behaviour that the outer loop is only executed once. The erroneous program's output is therefore 8, while the correct program's output is 24000.

The third guideline is somewhat similar to the first. To execute the outer loop's last instruction before the inner loop is finished, some jump instruction must have been used. Again, the loop-tuple is left on the loop stack and never popped unless a jump into the

[1] https://github.com/lowRISC/opentitan/issues/2967

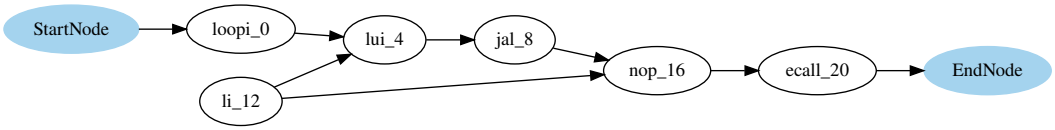


Figure 4.1: CFG representing the program Listing 4.1 illustrating the unreachable end of loop (li_12).

loop is performed afterwards. The consequence of not following the guideline is similar to the consequences of violating the second guideline. When the last instruction of the outer loop is reached after the jump out of the inner loop, the controller does not restart the loop because it only compares the current program counter to the top of the loop stack, which is the inner loop that was skipped.

4.2 Checking Compliance with Guidelines

Instead of polluting the loop stack and possibly getting unexpected behaviour, it would be helpful for OTBN programmers if they would get some error if they do not follow the guidelines. The team working on OpenTitan is already working on some tools that help the programmer follow the guidelines.

For the second guideline, the one regarding nested loops, we have implemented a check similar to the one in the tool built by the OpenTitan team. Whether loops are properly nested can be checked relatively easily. The start and end address of a loop can be determined statically. Using these values, we need to check if the outer loop surrounds the inner loop as required. The implementation of this can be seen on Algorithm 2. We loop through all the loops in the program and check if a loop's start address is contained inside another loop, and if that is the case, the inner loop's end address also has to be contained inside the same loop.

Algorithm 2 Pseudo code for checking the second rule.

```
1: for all outerLoop do
2:     for all innerLoop do
3:         if innerLoop.start > outerLoop.start and innerLoop.start < outerLoop.end then
4:             assert innerLoop.end < outerLoop.end
```

The first guideline is satisfied if a jump out of the loop is followed by a jump back into the loop. Due to our assumptions listed concerning the control flow graph construction in Chapter 3, calls to sub-routines within a loop-body will always return to the loop-body again. The remaining possible jumps are `BEQ`, `BNE` and `JAL` that do not uses the call stack ($x1$). The tool built by the OpenTitan team [2] looks for these instructions and checks if the jump is to an address outside of the loop-body where the instruction is found. Even if the jump address is outside the loop, it is not necessarily a problem since a jump back to the loop-body may be made. For this reason, the tool only issues a warning about the potential problem. However, in some situations, it is relatively simple to check that it is impossible to reach the end instruction of the loop-body. One example is the program at Listing 4.1, where the last instruction in the loop-body (`LI`) is made unreachable from the **StartNode** because of the `JAL` instruction, illustrated on Figure 4.1.

[2]The tool for checking loop rules can be found at `https://github.com/lowRISC/opentitan/blob/master/hw/ip/otbn/util/check_loop.py`

The problem remains with the BEQ and BNE instructions, for which we do not know if the jump branch will ever be executed. To test this, other techniques such as symbolic execution can be utilised.

Chapter 5

Timing Attacks

For the remaining part of this work, we focus on the problem regarding timing channels and different techniques we can use to look for possible timing attack vulnerabilities. Before we begin exploring how we can utilise different formal methods to unveil the existence of timing attack vulnerabilities in OTBN assembly programs, we first examine what a timing attack vulnerability is.

As described by François-Xavier Standaer [12], a cryptographic primitive can be viewed as an abstract mathematical object or black box, which transforms some input, possibly parameterised with a key, into some output. However, to provide practical value, the abstract mathematical object has to be implemented in a program that a processor will execute in a physical environment.
The physical environment enables a lot of different physical attacks. According to François [12] the numerous physical attacks are usually sorted among two orthogonal axes, namely invasive vs. non-invasive attacks and active vs. passive attacks.

- **Invasive vs. non-invasive**. Invasive attacks involve opening the chip to get physical access to the inside components. Non-invasive attacks exploit externally available information, which is often unintentionally available.
- **Active vs. passive attacks**. Active attacks try to tamper with the device's normal functioning. Passive attacks simple observes the device's behaviour during its processing.

The OTBN chip has security features to cope with invasive physical attacks[2, Security Features]. However, these features do not cope with side-channel attacks since, as mentioned in the Chapter 1, a naive cryptographic implementation can leak information through various side channels.
The information used in the non-invasive attacks can be execution time [13], power consumption, electromagnetic radiation [14] and more. However, we only focus on non-invasive attacks using execution time as a side-channel, called timing attacks. These attacks can leak information by an unintentional correlation between the secret information in the program and the program's execution time.

As a naive example, one can imagine an algorithm to check if a given input is a specific secret password. The algorithm checks if one input character at a time corresponds to the known password's character at the same index. A tempting performance optimisation for such an algorithm is to stop checking characters when the program discovers the first non-correct

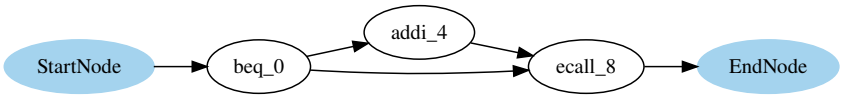


Figure 5.1: CFG for program vulnerable to timing attacks.

character. However, this results in a correlation between the execution time of the program and the correctness of the input. This example is trivial, but as the complexity rises, it can be tough to keep track of whether the execution time of a program depends on secret information.

5.1 (Time channel attacks) Discovering time-channels

For a program to be vulnerable to timing attacks, the program's control flow must depend on some secret value, and these variations in control flow should lead to different execution times. Like other safety properties, this property cannot be expressed as properties for individual execution traces of a system because it relates to events of two or more executions. Another example is *noninterference*, which is a confidentiality policy prescribing that actions performed by one group of users using a certain ability do not affect what some other group of users sees [15]. These types of properties are known as hyper-properties [16].

The control flow graph on Figure 5.1 is constructed for a program with a branch instruction (`BEQ`) that, if the condition is true, skips one of the program instructions, thereby making one of the possible traces through the program one instruction shorter than the other possible trace. If the value stored in at least one of the compared registers are secret values, the program is vulnerable to timing attacks. On a practical level, a difference on a single cycle will, in most cases, not lead to a vulnerability, which we will explore further in Section 5.4.

In the remaining part of this work, we use different techniques to look for possible timing attack vulnerabilities in OTBN programs. To begin with, we will construct a Uppaal [5] model similar to the control flow graph of the program without considering the data operations of the program. Because we disregard the data operations, we do not consider the comparison of registers usually used in `BEQ`, `BNE`, and both branches from these instructions will be feasible. Likewise, the computed number of iterations for `LOOP` constructed loops cannot be computed, and both the edge restarting and exiting the loop is always feasible. We use Uppaal [5] to test whether all possible paths in the model will lead to the same amount of executed cycles, even if we disregard the data operations performed by the program. This means that `BEQ` and `BNE` instructions jumping backwards and `LOOP` constructed loops effectively will be unbounded loops, making the set of possible execution paths seem infinite. However, loops constructed with the `LOOPI` instruction have statically known loop constraints, which we will consider in the analyses.

For Uppaal to verify the property, none of these unbounded loops can be present in the program, making the number of traces finite. Furthermore, all the traces must have the same execution time. If Uppaal can verify the property, the program cannot vary in execution time, and we do not have to analyse the program further.
Before examining how we construct the Uppaal models without data, we explore model checking and the Uppaal model checker tool.

5.2 Model Checking

In the following section, we briefly introduce model checking and Uppaal [5, 6]. We base the description of model checking on [17, p. 1-7]. There are overall three elements in model

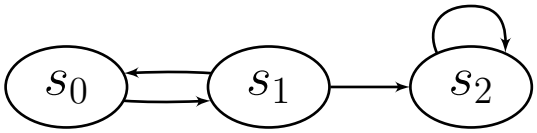


Figure 5.2: A simple Kripke structure.

checking. These are a model, a specification and a decision procedure for determining whether the model models the specification. The model of the system, which we want to verify, can be expressed in different ways. One of them is the Kripke structure, which represents a system as some model P in the form of a state-transition graph. The Kripke structure is a directed labelled graph, where vertices are states and edges is transitions. In [17] the structure are defined as $K = \langle S, R, L \rangle$ where S is a finite set of states, often referred to as state space. $R \subseteq S \times S$ is the transition relation, and $L : S \to 2^A$ is a function that assigns a set of atomic propositions to states. An example of a Kripke structure can be seen on Figure 5.2, where the path through this structure describes the possible behaviour of the model. The specification can be expressed as a temporal-logic formula denoted as ϕ, and the algorithm, the model checker itself, can determine if the model models the specification written as $P \vDash \phi$.

One challenge in relation to model checking is the *state explosion problem*, which happens because each state represents the system status at a given point in time, and each state is a memory snapshot of the modelled system. The state space size is therefore exponential in the size of the memory [17, p. 3].

An approach to mitigate the state explosion problem is by simulation instead of model-checking. Techniques using this approach are known as *Statistical Model Checking* (SMC). In this approach, a finite number of simulations is made, and well-known techniques from statistics are applied to infer, through statistical evidence, whether the specification is violated or satisfied. This also gives the advantage of easy parallelization of the algorithms, which can help scale to large systems [18].

The techniques in SMC can be used to answer *Qualitative* and *Quantitative* questions. The quantitative question asks what the probability that a stochastic system satisfies a property ϕ is [18]. The qualitative question asks whether a stochastic system's probability of satisfying a property ϕ is greater or equal to a certain threshold.

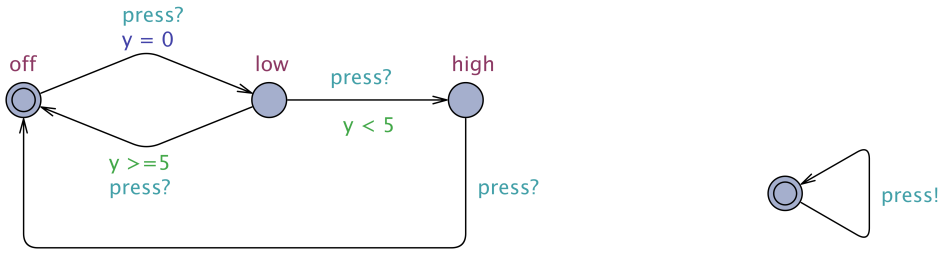
5.2.1 Uppaal

The model checking tool Uppaal is used for verification of real-time systems [5]. The models developed and used in Uppaal are either a timed automata or a network of timed automata, and consists of locations and edges. The timed automata are extended with bounded integers, urgency, broadcast channels, committed locations, user-defined types, functions, and more. The system state are defined as the locations of all automata, the clock values, and the values of the variables [5].

The edges can be labelled with select statements, guards, synchronisations, and updates. Guards are side-effect free expressions evaluating to a boolean variable. If the guard statement evaluates to `false`, the transition is disabled and cannot be taken. Otherwise, it can. There are two different kinds of synchronisation, binary synchronisation, which is declared as `chan Expression` and broadcast synchronisation, which is declared as `broadcast chan Expression`. Binary synchronisation on a edge can labelled with `Expression!` synchronises with another edge labelled `Expression?`. If no receiver is available, the edge labelled `Expression!` are not enabled. The broadcast synchronisation `Expression!` broad-

(a) **select** statement. (b) Unrolled **select** statement.

Figure 5.3: Illustrating the effect of the **select** statement.



(a) Lamp (b) User

Figure 5.4: This shows a simple Uppaal model of the interaction between a lamp and a user [5].

cast on the specified channel, and any receivers with the corresponding `Expression?`, which must synchronise. Unlike binary synchronisation, ff there are no receivers, then the sender can still execute the action. An Update is an expression with a side-effect, which can change the values of clocks, integers, and constants. An update expression can also be a function which updates some variables. A Select statement is used to non-deteministically select between a range of values. Figure 5.3 shows how the **select** statement works. On Figure 5.3b the integer i is non-deterministically chosen from the interval $[0:2]$, which results int the same behaviour as the automaton on Figure 5.3a.

A location can have an invariant, which is a side-effect-free expression that evaluates to a boolean variable. For a system state to be valid, the invariants of all active locations must evaluate to be true.

The example on Figure 5.4 shows some of the different labels. The example is a model of a lamp and a user. The lamp can be either be `off`, `low` or `high`, represented by the three locations in Figure 5.4a. The edge on Figure 5.4a, from the initial state `off` to `low`, can be taken when the binary synchronisation `press!` from the User, Figure 5.4b, is sent to the lamp. This binary synchronisation is the same for every edge in Figure 5.4a. The transition `off` to `low`, then perform the update statement `y = 0`, resetting the clock. The opposite edge from `low` to `off`, is only available if `y >= 5`. The edge `low` to `high` is transversely only available if `y < 5`, which represents a fast double-click by the user.

The queries for verifying properties in Uppaal is a simplified version of Timed Computation Tree Logic (TCTL) [5, p.7]. The TCTL queries that are used to describe a single state are called state formulae, which are expressions describing which location a process is in [5]. A path or trace through a system is a path formula categorised into reachability, safety, or liveness properties. There are five different types of Uppaal queries [5].

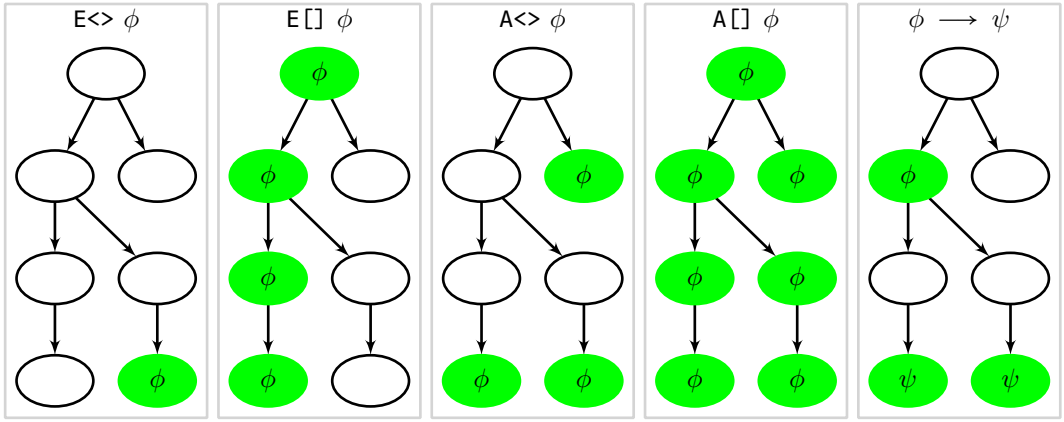


Figure 5.5: The different path formulae possible in UPPAAL. ϕ or ψ are satisfied in the green nodes [5].

- `E<>` ϕ is used to express reachability properties. It is satisfied if there exists a state where ϕ is satisfied, as illustrated on the leftmost sub-figure of Figure 5.5.
- `E[]` ϕ is used for checking safety proprieties. It is satisfied if there exists a path where all states satisfies ϕ. This is illustrated on the second sub-figure of Figure 5.5.
- `A<>` ϕ is used for checking liveness properties. It is satisfied if ϕ eventually will be satisfied for each path. This is illustrated as the middle sub-figure of Figure 5.5.
- `A[]` ϕ is used for checking safety properties. It is satisfied if all states satisfies ϕ. An illustration of this is the fourth sub-figure of Figure 5.5.
- $\phi \longrightarrow \psi$ is also used for checking liveness. It is satisfied if ϕ is satisfied and ψ eventually becomes satisfied in all following traces. This is illustrated as the right-most sub-figure of Figure 5.5.

In addition to the standard model checking queries above, UPPAAL SMC, another model-checking tool in the UPPAAL family, provides additional queries. These queries from UPPAAL SMC relate to the stochastic interpretation of timed automata [6]. UPPAAL SMC gives five new possibilities.

- `simulate N [<= bound]{expr1,...,exprk}`, where N indicates the number of simulations to be performed, `bound` is the time bound for the simulation, and `expr1`, ...,`exprk` are the `k` expressions to be monitored or visualised.
- `Pr[bound] (ap)`, where `ap` is a conjunction of predicates over the state of a Networks of Stochastic Timed Automata. This query is used for probability estimation, which computes the number of runs needed to produce an approximated interval with confidence $1 - \alpha$.
- `Pr[bound](`ψ`)` $\geq P_0$ where, `bound` is one of three different ways to bound a run, implicit by time $\leq$ `M`, explicit by cost `x` $\leq$ `M` (x is a specific clock) or a number of discrete steps `#` $\leq$ `M`. The formula ψ is either `<>q` or `[]q` where `q` is a state predicate. This query is used for hypothesis testing, and the probability to test for is specified as P_0.
- `Pr[bound1](`ψ_2`)` $\geq$ `Pr[bound2](`ψ_2`)` where `bound1` and `bound2` is the same possibility as the item above, and ψ_2 and ψ_2 is likewise the same as ψ from the item above. This query is used for probability comparison.

- `E[bound; N] (min: expr)` or `E[bound; N] (max: expr)` where `bound` is the same as the previous queries, `N` is the number of runs, and `expr` is the expression to evaluate. This query is used for evaluation of expected values for min or max respectively.

5.3 Uppaal no-data Model Construction

To use Uppaal to test if all possible paths through the control flow graph results in the same number of executed cycles, even when disregarding the data, we construct a Uppaal template representing the CFG of a given OTBN program. As mentioned in Section 5.1 hyper-properties relates the events of multiple executions. To handle this in Uppaal we use the constructed template to instantiate two processes denoted as `Process0` and `Process1`. We will refer to processes instantiated using the above described template as interpreter processes. We make the end-location, representing the **EndNode** of the control flow graph, of the constructed Uppaal model urgent such that only one of the models can reach the end of the program unless they reach the end at the same time and thereby have used the same amount of cycles. To keep the processes synchronised, we will construct a third process, representing a CPU, which broadcasts a message for each time unit, where each of the processes must respond with `done?` if they can. The models will be constructed such that this is always the case.
We refer to the model, constructed in this section as Uppaal no-data, which consists of `Process0`, `Process1` and a `CPU` process.

We make Uppaal try to verify that all possible traces reaches the end node of both processes. If Uppaal can verify the query, the program is proved not to be vulnerable to timing attacks. However, if Uppaal cannot verify the query, it can show two traces through the program with a different number of executed instructions.

The first step in constructing a Uppaal template corresponding to the given control flow graph is to transfer the vertices and edges from the CFG into Uppaal locations and edges, respectively. The locations representing the non-instruction nodes, such as the **StartNode** and **EndNode** on Figure 5.1, added to the CFG for easier assemble, are made urgent such that no time is spent in these locations. The urgent nodes can be seen on Figure 5.7.

The second step is to handle the instructions, which take more than a single cycle and the instructions that differ in the number of cycles taken depending on their operands. Examples of these instructions are `LI`, `LW`, and when `CSRRS` and `BN.WSRR` are used to read a random number from the RND special purpose register. We will examine how we model `LI`, which is a pseudo instruction which takes either one or two cycles, as mentioned in the Section 2.1.6. `LI` takes a single instruction when the most significant 20 bits are zeros, or the least significant 12 bits are zeros, handled by a single `LUI` or `ADDI` instruction, respectively. Otherwise, it uses both `LUI` and `ADDI` and thereby uses two cycles. This is shown at Figure 5.7 at `li_0` and `li_4`. However, if the most significant 20 bits and the 12 least significant bits are non-zero, the instruction requires both a `LUI` and an `ADDI` instruction, and it is represented as `li_8` where the guards ensure that the simulation visits the location twice since, making it take two time cycles.

The Uppaal model must also imitate the behaviour of the OTBN controller, which changes the program counter when needed, to execute the hardware-assisted loops with the correct number of iterations. To do this, the Uppaal model needs the same information as the controller, namely the loop stack. However, since we assume that the program complies with the loop guidelines, we will model the controller without the loop stack and instead store a number for each loop, holding the number of iterations.
In the following, we describe how we model the controller and refer to the Uppaal model at

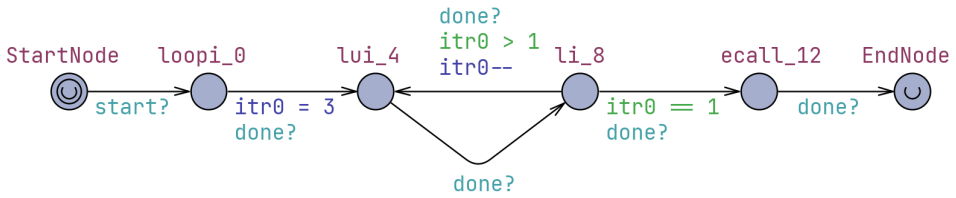


Figure 5.6: UPPAAL model for describing the controllers behaviour.

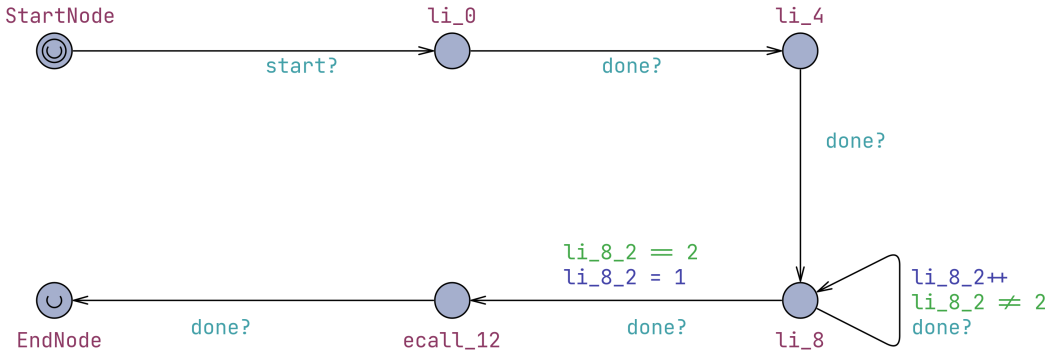


Figure 5.7: Interpreter template.

Figure 5.6 as an example. The edge going into the location representing the first instruction of the loop (`lui_4`) have an update statement `itr0 = 3` specifying the number of iterations. The edge leaving the last location in the loop, `li_8`, has the guard `itr0 == 1`, disabling the edge until the loop has been executed the correct number of times. The edge going from the last location, `li_8`, in the loop to the first location, `lui_4`, in the loop also has a guard `itr0 > 1`, that enables said edge until the number of iterations is equal to one. This edge also has an update statement `itr0--`, which decrements the remaining number of iterations.

Without the assumptions from Section 3.1 and Section 4.1 this modelling will result in a different behaviour than the one described in Section 2.1.3. The reason is that the controller can read the number of iterations for all loops and not just the one at the top of the loop stack. This means that when a jump from an inner nested loop to the outer loop is performed, the controller will be able to restart the outer loop when the last instruction is reached, which is normally not possible, as exemplified in Section 4.1.

Since `LOOP` reads the number of iterations from a register, the value is only known at run time. Therefore, we cannot set the number of iterations statically in the model. Since the model abstracts away all the data manipulation, it will not compute the value at run time. Therefore, the edges in a `LOOP` constructed loop will not have any guards, which leads to unbounded loops, as mentioned in Section 5.1.

The last step is to construct the CPU template and annotate the interpreter template with synchronisations `start?` and `done?`, which allows for the synchronisation with the CPU process.

The construction of the CPU template is the same for all OTBN programs. The model is shown on Figure 5.8 and consists of the start location, which synchronises on the broadcasting channel `start!`, making the interpreter process transition from `StartNode` to the following location. The `Error` location is only relevant for models considering the data operations, which we explore in Section 5.5.
The loop from `Idle` to `Run` and back again is the *main* loop, where the CPU broadcasts

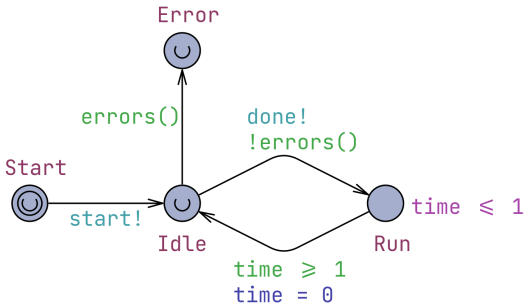


Figure 5.8: CPU template.

on the synchronisation channel `done!` to ensure that progression through the interpreter processes happens in a synchronised manner.

We construct a UPPAAL no-data model using the template at Figure 5.7, which UPPAAL tries to verify using the query `A<> Process0.EndNode && Process1.EndNode`. The result of this query is `Property is satisfied`, meaning that there is not any possibility for a timing difference. However, when trying to verify the UPPAAL template corresponding to Figure 5.1 with the same query, the result is `Property is not Satisfied` meaning there is a timing difference. This is because there are two possible traces through the model.

With this implementation, we can determine whether there is a timing difference in the UPPAAL no-data model. However, it does not show how much the different traces differentiate in time, which is important since a few cycles of difference are often acceptable.

5.4 Statistical Model Checking on UPPAAL no-data

Due to the `EndNode` being urgent in the UPPAAL no-data model, we only know that one of the processes has reached the `EndNode` and the other has not. We cannot determine the number of cycles required for the second process to reach the `EndNode`. Additionally, the standard UPPAAL queries only answer whether or not a property is satisfied, and therefore, we use the queries from UPPAAL SMC to get the timing difference. We also have to insert a variable indicating the number of cycles, `cycles`, for each of the processes, such that we can compare their amount of cycles at the end of the simulation. In this section, we, therefore, modify the UPPAAL no-data model such that both processes can reach their `EndNode`. We will refer to the modified model as UPPAAL no-data-SMC.

The models in UPPAAL SMC do not support input-determinism, which requires that the state following a synchronisation can be uniquely determined. This is not the case when the processes have multiple options following the `done?` synchronisation. We remove the input-nondeterminism by adding an additional committed location immediately after every location with multiple outgoing edges to the UPPAAL no-data-SMC models.
On Figure 5.9, we have added the update statement `cycles++` to the edges which have the synchronisation `done?`, and the committed location `additional`. This location is placed after the branch instruction `BNE`, and the outgoing edges from `BNE` is then added to the committed location.

As an example, we simulate the UPPAAL no-data-SMC model constructed with the template on Figure 5.9, with the query `simulate 1 [<=6]{Process1.EndNode, Process0.EndNode, ↪(Process0.cycles), Process1.cycles}`. The result is the simulation presented on Figure 5.10, which shows that `Process0` reached its `EndNode` after 5 cycles and `Process1` reached its `EndNode` after only 2 cycles.

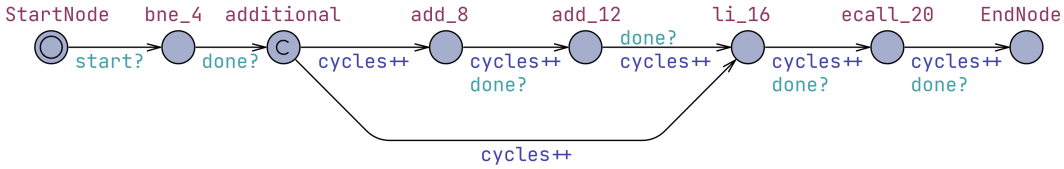


Figure 5.9: This example shows an additional committed location, which removes the input non-determinism, and update statement `cycles++`.

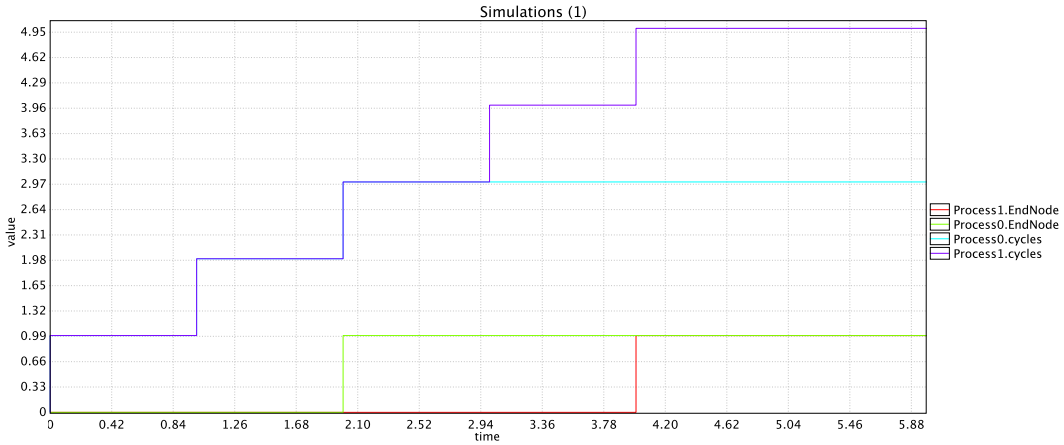


Figure 5.10: This graph shows that the UPPAAL no-data-SMC model made from the template on Figure 5.9 have a timing difference, since `Process0.EndNode` is entered after 3 cycles, and `Process1.EndNode` is entered after 5 cycles.

Both the UPPAAL no-data and UPPAAL no-data-SMC models are over-approximations because they do not include the execution of the instructions in the original program. This means that the time-differing traces might be unreachable in the original program. To accommodate for this we make UPPAAL no-data and UPPAAL no-data-SMC perform computations.

5.5 UPPAAL with-data

To remove the over-approximation, one theoretically possible solution is to implement the interpreter form Section 2.2 into UPPAAL and make UPPAAL select the input values through a select-statement. This solution is not practically possible due to the range the input can be within, leading to state-space explosion, which makes model checking infeasible. However, we implement the solution anyway and try to verify the property with 8-bit input. If UPPAAL can find a counterexample, we have proven the existence of a timing difference. If UPPAAL cannot find a counterexample, timing differences might still exist for non-explored input.

As mentioned in Section 2.2 the operations of each instruction in the instruction set is implemented as a function, modifying the state. To extend the UPPAAL no-data and UPPAAL no-data-SMC models we create an external library including all the features of the interpreter from Section 2.2. The function implementing the behaviour of `LI` in the external library is shown on Listing 5.1.

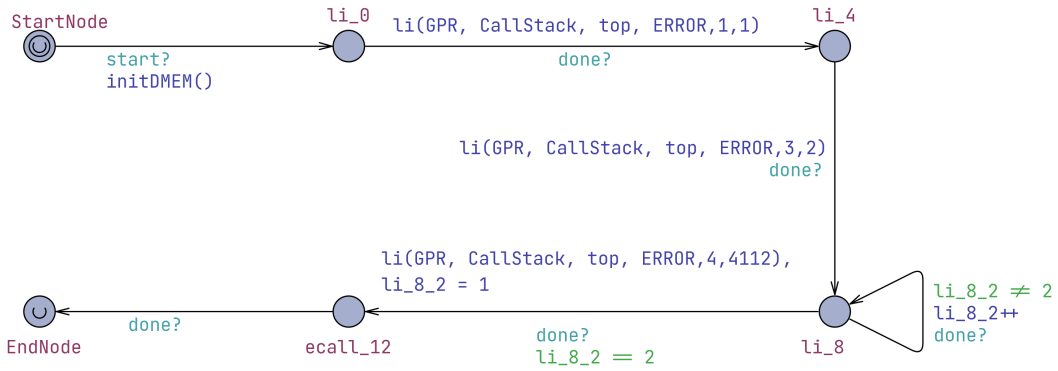


Figure 5.11: The same model as Figure 5.7, but annotated with extra information for data manipulation.

```
1  extern "C" void li(uint32_t *gprs, uint32_t *
       ↪stack, uint32_t *top, bool *err, uint32_t
       ↪grd, uint32_t imm) {
2      State state = createState(gprs, stack, top,
       ↪ err);
3
4      ErrorChecker::checkForCallStackError(state.
       ↪errorStatus, {}, {grd}, state.callStack);
5      if (state.errorStatus.anyError()) {
6          return;
7      }
8
9      state.writeGpr(grd, imm);
10 }
```

Listing 5.1: The implementation of `LI` instruction from the external library.

The library functions are added as update labels to the UPPAAL models, which is shown on Figure 5.11. The edge leaving `li_8`, have the update function `li(GPR, CallStack, top, ERROR, 3, 4112)`. The `LI` functions parameter is the information necessary for implementing the behaviour of `LI`. The first four parameters are the same across every instruction in base instruction subset, and the last parameters – in this case, two – are the register and immediate values. This means that in the update function leaving `li_8`, we write the immediate value 4112 into the fourth register, $x3$.

Another important addition to this model is the *select* statement, `i : int[0,2048]`, on the transition from `StartNode` to `li_0`. This addition allows us to specify the range of inputs which UPPAAL must non-deterministically choose from. This can be a register or a DMEM address where the initial secret should be saved. This is done through an update, `GPR[2] = i`, which assigns `i` to register $x2$. This feature allows for verifying whether any integer in the specified range impacts the control flow or not.

Instead of testing a range of input but random values instead, we can use a similar approach to the one in Section 5.4, except that the added data in this model prevent input non-determinism, so the extra committed location is not needed. Another modification needs to run the with-data model is to remove the *select* statement since we only can utilise this

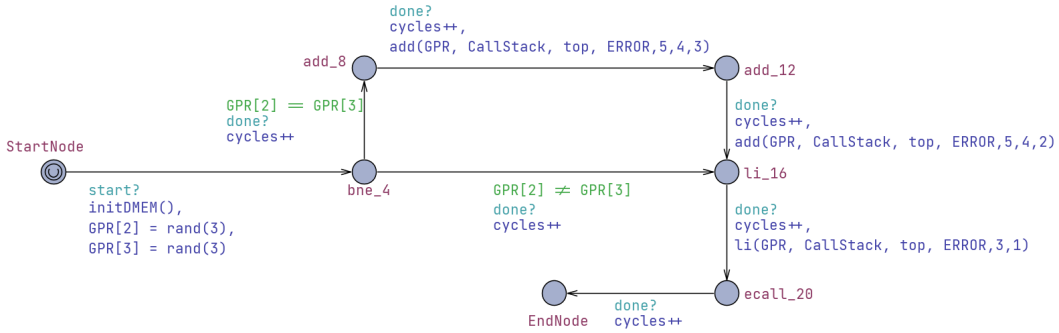


Figure 5.12: This shows the UPPAAL template from Figure 5.9 with data, and the addition of saving random input into $x2$ and $x3$.

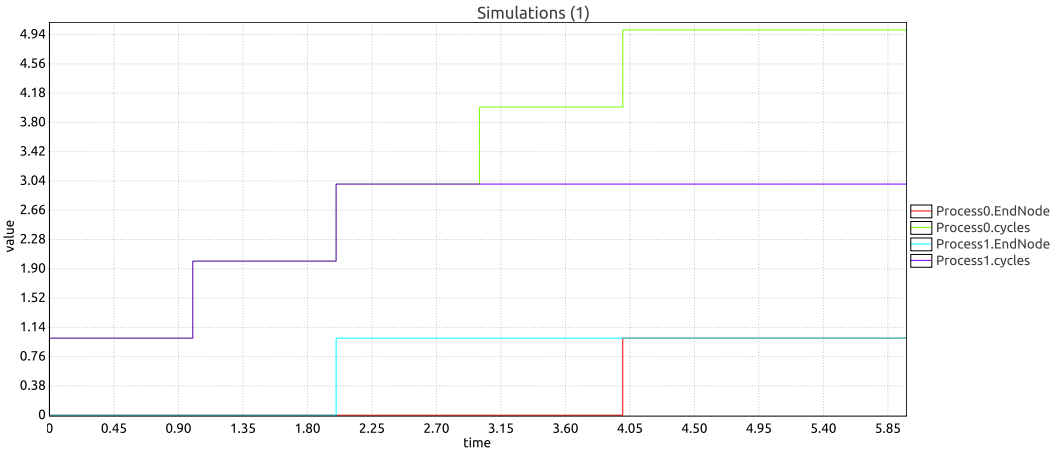


Figure 5.13: The result of simulating the UPPAAL with-data-SMC model using the template on Figure 5.12 a single time. The result shows that `Process0.EndNode` is entered at time 3, and `Process1.EndNode` entered at time 5.

in model-checking. Instead, we need a function which can create random numbers for each run and check if this number changes the execution, meaning that a specific number could impact the timing.

Figure 5.12 shows an example of the implementation described above, where the update functions `GPR[2] = rand(3)` and `GPR[3] = rand(3)` saves the input into different registers $x2$ and $x3$, respectively. With these update functions, we then perform a simulation to see if either of the two inputs affects the execution. In this case, it is trivial since the execution relies on a comparison between these inputs. The result of the simulation query, `simulate 1 [6]{Process0.EndNode, Process0.cycles, Process1.EndNode, Process1.cycles}` is shown on Figure 5.13. It should be noted that we only ran this simulation once since, in this case, the input values range is small. We see that the simulation gives an identical result to Figure 5.10, and can therefore see that the timing problem found is present even when the data manipulation is considered.

With this solution, we try to guess the inputs, showing a timing difference. Another approach is to test whether the time-differing traces found using the UPPAAL no-data and UPPAAL no-data-SMC are executable. For this, we will examine symbolic execution.

Chapter 6

Symbolic execution

In this chapter, we explore the overall theoretical idea behind symbolic execution and examine how we can extend the interpreter described in Section 2.2 to do symbolic execution.

6.1 Introducing Symbolic Execution

In this section, we briefly review the theory behind symbolic execution. We start by describing the general idea, followed by an exploration of some of the key challenges, such as symbolic memory addresses and symbolic loop conditions.

Symbolic execution was first presented by James C. King in [19]. The fundamental idea is to execute a program with symbolic input values rather than concrete values, which allows to reason about the behaviour of a program on many different inputs [20].
When a program is symbolically executed, input values are represented as symbols, and every operation must therefore be able to operate on both concrete and symbolic values. According to [21] a symbolic execution engine maintains the three following elements for each operation.

- *stmt*: The statement to be executed next.
- *store*: A symbolic store, which associates program variables with either concrete or symbolic values.
- *path condition* (π): A formula that describes all the constraints for the symbols, such that *stmt* can be reached. The *path condition* starts being *true*.

How the symbolic engine updates *store* and π depends on *stmt*. If *stmt* is an assignment expression $x = e$, the *store* must be updated such that x is associated with e ($x \mapsto e$), which is evaluated in the context of the current *store* and can range over both symbols and concrete values [21].

If *stmt* is a conditional branch (`if` *expr* `then` e_{true} `else` e_{false}), the *path condition* must be updated. The symbolic execution engine then forks the execution by creating two execution states, one for the true branch and one for the false branch, having the path conditions $\pi_{trueBranch}$ = (`true` $\wedge$ *expr*) and $\pi_{falseBranch}$ =(`true` $\wedge \neg$*expr*), respectively [21].

If *stmt* is a `jump` *label* statement, the *stmt* is updated to be the instruction at *label* [21]. The *stmt* can also be a loop, which for OTBN programs are pushed to the loop stack and

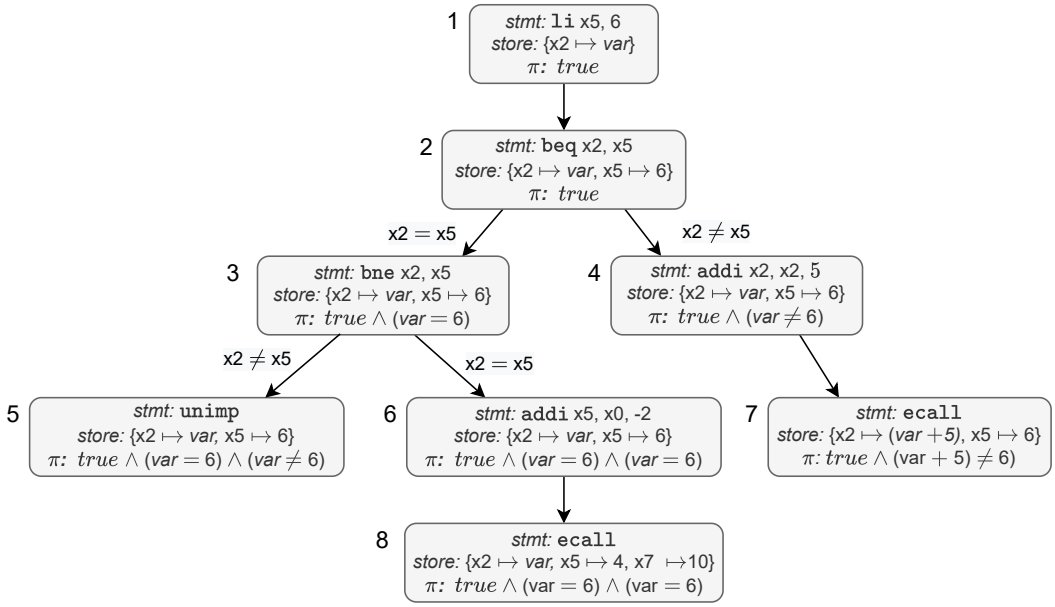


Figure 6.1: The program from Listing 6.1 as a symbolic execution graph.

then handled by the controller. For now, we will not go further into loops, but we describe the challenge of unbounded loops later.

```
1  li x5, 6
2  beq x2, x5, lbj1
3      addi x2, zero, 5
4      ecall
5
6      lb:j1
7      bne x2, x5, lbj2
8          addi x5, x2, -2
9          ecall
10
11         lb:j2
12         unimp
```

Listing 6.1: OTBN program used to exemplify Symbolic Execution. $x2$ holds a symbolic variable.

In the following we exemplify how the *store* and *path condition* (π) are updated during symbolic execution, based on the OTBN program at Listing 6.1. For this example, the program are represented as a graph at Figure 6.1, which illustrates the control flow together with the current state of the *store* and *path condition* for each instruction (*stmt*). The numbers at the top left corner of each node in the graph are used to refer to specific nodes.

Through the example, register $x2$ is considered the register holding the input value. Therefore, the *store* includes the binding $x2 \mapsto var$ at node number 1, where *var* is a symbolic value. All other registers and data-memory addresses store the value 0, to begin with, but

we omit zero values in the figure. When a program starts, the *path condition* will be *true*. The *stmt* at node 1 is a `LI` instruction writing the value 6 to register $x5$.

For node number 2, the *store* is updated according to the `LI` instruction, and the new *stmt* is a `BEQ` instruction, which is a conditional branch and the execution is forked. The two following nodes (labelled 3 and 4) then gets the path conditions $true \wedge (var = 6)$ and $true \wedge (var \neq 6)$ for the true and false branch, respectively. When branching, an SMT solver can be utilised to test whether both of the path conditions are satisfiable and, therefore, should be further explored.

There are two possible states that the symbolic execution engine can explore. The example continues at node 3 with *stmt* being a `BNE` instruction. As before, this is a conditional branch, and the execution is split into the nodes labelled 5 and 6. For node 5, where the *stmt* is an `UNIMP` instruction, the *path condition* will then be $true \wedge (var = 6) \wedge (var \neq 6)$. The path should not be explored any further since the *path condition* cannot be satisfied, which is lucky since the `UNIMP` instruction triggers an ILLEGAL_INSN error.
The other path starting from node 6, where *stmt* is an `ADDI` instruction, updates the value stored at register $x5$, which can be seen at node 8, the end of the path.

Now, we continue where we left of with the other branch from node 2. The *stmt* is an `ADDI` instruction, which adds 5 to the symbolic *var*, updating $x2$, such that $x2 \mapsto (var + 5)$. Node 7 is an `ECALL`, which terminates the execution.

6.1.1 Satisfiability Modulo Theory (SMT) Solvers

Satisfiability modulo theory (SMT) solvers extends Boolean satisfiability solvers (SAT solvers) and can check the satisfiability of first-order formulas using operations from multiple theories such as bit-vectors, arithmetic, arrays and more. One SMT solver is the Cooperating Validity Checker (CVC4) [7], which we will utilise for the implementation of symbolic execution for OTBN programs.

6.1.2 Symbolic Execution Challenges

Multiple challenges exist for symbolic execution, one being the *symbolic memory address problem* addressed in [20]. In the concrete setting, any expression representing a memory address can be evaluated to a value representing a particular memory address. However, when doing symbolic execution, the expression can include symbolic variables, and we can, in most cases, not determine a particular memory address. In the OTBN setting, the problem arises with the `LW`, `SW`, `BN.LID`, and `BN.SID` instructions. For a load from an address specified by a symbolic expression, a sound strategy considers it a load from any possible concrete address created with a satisfying assignment of the variables in the symbolic expression [20]. The same is the case for writes to symbolic addresses, which can be considered a overwrite/assignment to any satisfying assignment to the address-expression [20].

Fully symbolic memory can be modelled using multiple different strategies like state forking or `if then else` formulas. However, some SMT solvers have the expressive power to model fully symbolic addresses. By using *a theory of arrays*, both load and store to arrays can be expressed as first-class entities in constraint formulas. This is utilised by tools such as EXE [22] and KLEE [23].

Another challenge when implementing symbolic execution is symbolic loop conditions. For OTBN programs, the problem arises with the use of the `LOOP` instruction, which reads the number of iterations from a register, potentially being a symbolic value. If we, for each iteration, apply the SMT solver to check whether the path condition still can be verified

while the execution is potentially forked for each iteration, we can end up with a path explosion problem. To avoid this, it is common to bound the exploration of loops up to a limited number of iterations. However, potentially interesting paths are left unexplored [21].

6.2 Implementation

To test whether any concrete input will lead to executing the traces proposed by UPPAAL, we have extended the interpreter described in Section 2.2 to work with both concrete values and symbolic expressions. The data type representing the values stored at different locations in the state is extended to be a wrapper around a symbolic expression and a concrete value, 32-bit integers for general purpose registers and 256-bit integers for wide data registers and data memory. The wrapper also holds an additional boolean to specify whether it is a symbolic value or not, which will be updated according to the operations performed on it. For example, if at least one of the operands of addition is symbolic, the result will also be a symbolic value. We refer to the wrapper datatype as `Value`.

As mentioned in Section 2.2, we have implemented a function for all instructions, which modifies the input state. For the symbolic execution version of the interpreter, all the instruction functions are updated to either do the same operation as the concrete interpreter presented in Section 2.2, if the values used are concrete, or construct a symbolic expression representing the operation performed by the instruction.
For example, the `ADD` instruction computes the usual addition and writes the result to the destination register if both the operands are concrete values. If not, the instruction creates a symbolic expression and writes it to the destination register. If $x2$ and $x3$ stores the symbolic expressions $e1$ and $e2$, respectively, the destination register will hold the expression $e1 + e2$.

A subset of the instructions in the big number instruction subset updates the flags described in Section 2.1.2. One example is the `BN.ADD` instruction, which updates all four flags in the specified flag group, according to the result of the performed addition. Since the result will be a symbolic value if one of the operands is symbolic, the flags must also store a `Value` variable. If the result of the operation are $e1 + e2$, where $e1$ and $e2$ are some kind of expression, which can be evaluated to a bit-vector the flags are updated as follows.

- For the L (least) flag we use the bit-vector extract functionality of CVC4, such that the flag after `BN.ADD` holds the expression: $extractLSB(e1 + e2)$.
- The M (most) flag are handled almost similar, but with $extractMSB(e1 + e2)$.
- The Z (zero) flag value cannot be extracted from the result expression. The flag should be 0 if the result is non-zero and 1 otherwise. One possible strategy to test both cases is by forking the state and making the symbolic execution engine explore both possibilities. However, when using this solution to test if a given trace proposed by UPPAAL is reachable, we will have to test if at least one of these forks is reachable. Instead, we will utilise the if-then-else expression supported by CVC4, such that the Z flag will hold the expression $if\ (var1 + var2 == 0)\ then\ 1\ else\ 0$. If the result-expression evaluates to 0, the flag is 1 and vice versa.
- The C (carry) flag is handled likewise, but instead of testing for evaluation to 0, we test for overflow.

As mentioned in Section 2.1.6, `BN.SEL` writes the value stored at one of two registers to a destination register, depending on a flag value. To handle this symbolically, we utilise the if-then-else statements provided by CVC4.

6.2.1 Symbolic Data Memory

CVC4 supports the *theory of arrays*, which we use to model a fully symbolic data memory. To ease the load and store for 32-bit integers with `LW` and `SW`, respectively, we model the memory as a 32-bit data memory instead of 256-bit. Because of this, the 256-bit load instruction (`BN.LID`) reads eight values from the date memory starting from the provided address and concatenates the bit-vectors into one, using the SMT library's concatenation functionality.

6.2.2 Indirect Addressing

As mentioned in Section 2.1.6, the `BN.SID` and `BN.LID` instructions use a wide register through indirect addressing. The value stored at a general-purpose register determines which wide register to use. Because the value at the register can be a symbolic expression, this leaves us with the same problem as with the symbolic memory addresses. However, none of the available OTBN programs on the official GitHub project utilises this feature. Every time either `BN.SID` or `BN.LID` is used, the value at the general-purpose register is explicitly assigned to an immediate using `LI`. Therefore, we assume that the register stores a concrete value when `BN.SID` and `BN.LID` are executed, and we will not provide support for symbolic indirection.

6.3 TraceChecker: Testing Uppaal Traces for Reachability

We utilise the symbolic extension of the interpreter to execute the traces proposed by Uppaal symbolically and then apply the SMT solver to check whether the *path condition* is satisfiable or not. If we find that both traces proposed by Uppaal are reachable, we have found a potential timing attack vulnerability. Because the proposed Uppaal trace leads the execution, the TraceChecker implementation avoids the problems related to the symbolic loop conditions. Generally, we do not have to explore more than the paths Uppaal propose.

To follow a trace proposed by Uppaal, we will use the control flow graph originally used to construct the Uppaal model, the trace to be tested, and information about what parts of the *store*/state should store symbolic variables to begin with.

In the following, we present how the TraceChecker is implemented. For the presentation we refer to Algorithm 3. The TraceChecker works by traversing the same path in the CFG as Uppaal did while symbolically executing the instructions at the path.

The trace is followed in the while loop from line 6 to 35 in Algorithm 3. Each iteration in the while-loop corresponds to one step through the trace. Overall, two different cases are considered in the while-loop. The first case (from lines 8 to 11 in Algorithm 3) is where the current node only has a single adjacent node. For this case, we update the current node to be the adjacent node, and if the previous, current node was an instruction node, we execute the instruction such that the *store* is updated. At line 38 in Algorithm 3 the current position in the given trace is incremented for the next iteration of the while loop.

The second case in the while-loop is when the current node has more than one adjacent (from line 13 to 35 in Algorithm 3). This reflects the situation where Uppaal took a non-deterministic choice.
First, we find the edge from the current node to the node corresponding to the choice made by Uppaal at lines 16-17. The edge is used to provide information about what kind of choice Uppaal took. There exist two different scenarios where Uppaal can take a choice leading to four different outcomes. Uppaal can non-deterministic choose to either restart

Algorithm 3 Pseudo code for TraceChecker.

```
1: maxPosition = trace.size()
2: currentPositionInTrace = 0
3: currentNode = cfg.startNode
4: pathCondition = true
5:
6: while currentPositionInTrace ≠ maxPosition do
7:
8:     if currentNode.numberOfAdjacent == 1 then
9:         if currentNode.isInstructionNode then
10:            currentNode.instruction.run
11:        currentNode = currentNode.adjacent[1]
12:
13:    uppaalChoice = trace.at(currentPositionInTrace)
14:    if currentNode.numberOfAdjacent > 1 then
15:        adjacentList = currentNode.adjacent
16:        uppaalChoiceNode = adjacent.find(uppaalChoice)
17:        edge = cfg.findEdge(currentNode, uppaalChoiceNode)
18:
19:        if edge is loopRestart then
20:            loopStack.top.loopCount − −
21:
22:        else if edge is loopExit then
23:            pathCondition = pathCondition ∧ (loopstack.top.loopCount == 0)
24:
25:        else if edge is trueBranch then
26:            if edge.from is BEQ then
27:                pathCondition = pathCondition ∧ (reg1.val == reg2.val)
28:            if edge.from is BNE then
29:                pathCondition = pathCondition ∧ (reg1.val ≠ reg2.val)
30:
31:        else if edge is falseBranch then
32:            if edge.from is BEQ then
33:                pathCondition = pathCondition ∧ (reg1.val ≠ reg2.val)
34:            if edge.from is BNE then
35:                pathCondition = pathCondition ∧ (reg1.val = reg2.val)
36:
37:        currentNode = uppaalChoiceNode
38:    currentPostionInTrace + +
39: Test path condition using SMT solver and return result
```

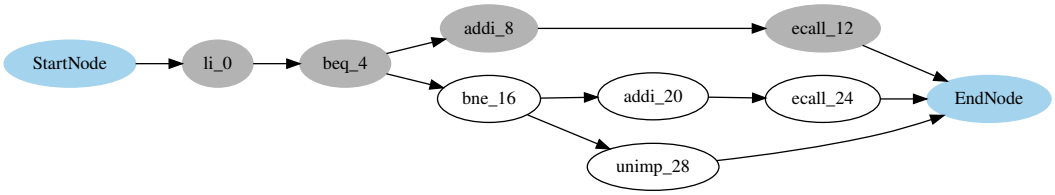


Figure 6.2: CFG for program at Listing 6.1, with a marked trace.

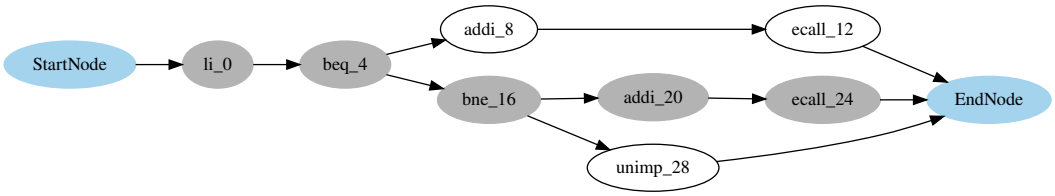


Figure 6.3: CFG for program at Listing 6.1, with a marked trace.

or exit a loop when the last instruction in a loop body is reached. Furthermore, for `BEQ` and `BNE` instructions, UPPAAL can choose to either jump to the specified instruction or continue with the consecutive instruction in the instruction memory.

On line 19, we test if the edge is a loop-restart edge. If it is the case, the top of the loop stack is updated such that the number of iterations are decremented, either concrete or symbolic, depending on value originally used to construct the loop.
On line 22, we test if the edge is a loop-exit edge. If it is, the *path condition* will be updated to reflect that the expression or value at the top of the loop stack must be equal to zero, $\pi = (\pi \wedge loopStack.top.loopCount == 0)$.
On lines 25 and 30, we check if the edge is a true or false branch, respectively. Taking into account if the instruction is a `BEQ` or `BNE`, the *path condition* are updated accordingly.

When the while-loop terminates, the SMT solver is applied to test whether the *path condition* is satisfiable. Figure 6.2 show the CFG for the program at Listing 6.1 with a trace highlighted with grey. If the program at Listing 6.1, together with the CFG and the grey trace, are given as input to the TraceChecker, it will start following the trace while executing the instructions symbolically. When the `BEQ` instruction is reached, it will look in the trace to check which path to follow. At the end, when the last node in the trace is reached (in this case `ECALL`), the SMT solver is applied, which in this case will return SAT.

6.4 TraceChecker Example

Using the UPPAAL no-data model, UPPAAL can either verify that the program is not vulnerable to timing attacks or show two traces through the program with different execution times. However, there is no guarantee that program input exists, such that the traces proposed by UPPAAL can be executed. To test whether the traces are executable, we use symbolic execution with the TraceChecker. If both of the traces are reachable, we have found a potential timing attack vulnerability. As an example, we look at the program from Listing 6.1. Using this program as input, UPPAAL could find the two traces shown on Figure 6.2 and Figure 6.3, which differ in the number of executed cycles. Using the TraceChecker, we see that both traces are reachable, and we have thereby shown the existence of a timing difference.

However, if at least one of the traces proposed by UPPAAL is unreachable, which will be the case if the proposed traces are the ones from Figure 6.3 and Figure 6.4, we have only shown

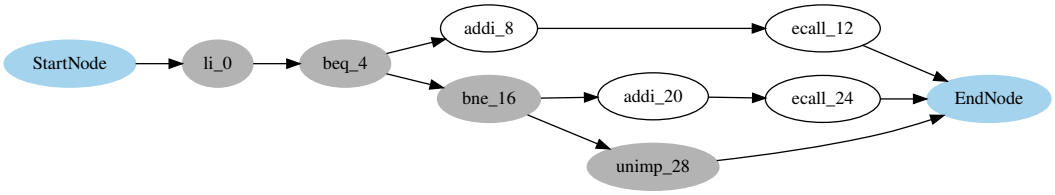


Figure 6.4: CFG for program at Listing 6.1, with a unreachable marked trace.

that the two specific traces does not create a potential timing attack vulnerability.
We, therefore, want to explore some other traces than the initially proposed by Uppaal. However, there is no built-in feature in Uppaal to get another counter-example to the query. In Chapter 9 we propose different possibilities to further explore the possible counter-examples for future work.

Chapter 7

Possible Timing Attack Vulnerabilities in *RSA 3072 verify*

In this chapter, we utilise the implemented tools to look for possible timing attack vulnerabilities in the *RSA 3072 verify* program available from the OpenTitan official GitHub-page [1], which are used in the ROM extension **ROM_EXT** stage of the secure boot process.
Most of the programs available on the GitHub-page consist of multiple files, which should be linked together, but the interpreter implemented as part of this work (mentioned in Section 2.2) only supports OTBN programs contained in a single file and the `.word` directive is the only supported directive. The *RSA 3072 verify* program considered in this chapter is, therefore, manually constructed by collecting all used sub-routines in a single file, and `.skip` and `.zero` directives are replaced by the appropriate number of `.word` directives. In the constructed program, we have also changed the syntax of the labels and annotated all sub-routines using `startf` and `endf`. The GitHub repository also contains a test program for the *RSA 3072 verify*, which includes concrete input values, namely the signature, modulus of test key, Montgomery constant, and squared Mongomery Radix. The program also contains the expected result written as a comment in the program. Before we perform any tests, we run the program with the concrete input listed above and get the expected result, indicating that the program is constructed correctly. Running the program used 142538 cycles.

7.1 Verifying Uppaal no-data

First, we apply the Uppaal no-data setup to check if all possible paths take the same amount of cycles, even if we disregard the data manipulation. If Uppaal can satisfy the query, it is proved that the program cannot vary in execution time. However, as mentioned earlier, this requires a finite number of paths through the constructed Uppaal no-data model, and all these paths should take the same amount of cycles.

When trying to verify the query `A<> Process0.EndNode && Process1.EndNode` described in Section 5.3, we get the result of `Property is not satisfied`, which means that one of the traces has reached the **EndNode** while the other has not.

7.2 Simulating Uppaal no-data-SMC

The two traces received as the counter-example to the Uppaal query used in the Uppaal no-data test do not indicate how big the possible time difference is. We construct a Uppaal no-

[1] `https://github.com/lowRISC/opentitan/tree/master/sw/otbn/crypto`

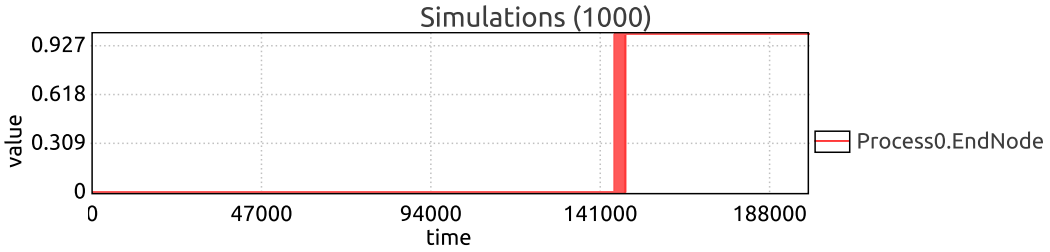


Figure 7.1: This graph shows the different amount of cycles used before the end node of the model representing the *RSA 3072 verify* was reach. The amount of cycles ranges between 144942 and 148062.

data-SMC model to simulate a selected number of times, which provides information about some possible execution times. When testing the UPPAAL no-data-SMC model with the query, `simulate 1000 [200000]{Process0.EndNode}`, we get the graph on Figure 7.1, which shows that the observed execution times is between 144942 and 148062 cycles. The problem with this query is that we cannot guarantee that the maximum execution time is below 200000 cycles. However, the value is chosen based on execution with the concrete values provided in the official program, which used 142538 cycles. Additionally, the maximum difference between the number of cycles performed by the two instantiated processes is 148062 – 144942 = 3120

Some traces discovered with the UPPAAL no-data and UPPAAL no-data-SMC are possibly unreachable, making the time difference non-problematic. We could proceed with the analysis of the *RSA 3072 verify* by either using the TraceChecker to test whether the traces are executable or test whether we can use the models considering the data operations and still find a time difference. Instead of using either of these, we will, in the following section, locate the reason for the timing differences and remove it.

7.3 Removing Timing Difference

An approach to remove timing differences without altering the program's behaviour is inserting no-operation (`NOP`) instructions to the shorter paths. We examine the program for instructions leading to possible timing differences. The *RSA 3072 verify* program does not contain any `LOOP` constructed loops, backwards-jumping, and does not read the RND special purpose register. The remaining possibilities for time difference are `BEQ` and `BNE` instructions.
We find that the program contains a single `BEQ` and a single `BNE` instruction. The `BEQ` instruction is in the `mont_loop` sub-routine and the `BNE` instruction is in the `modexp_var_3072_f4` sub-routine.

Above the `mont_loop` sub-routine in the official program, a comment mentions that the sub-routine is a variable time sub-routine. We can see this by examining Listing 7.1 which is an excerpt of the `mont_loop` sub-routine. The `BEQ` instruction is shown at line 2. If the compared registers are distinct, the instructions from lines 3 to 15 will be executed. Four of these instructions are executed for 12 iterations in a loop, one of them being a `BN.LID` instruction, which takes two cycles. The instructions from lines 3 to 15 take together 67 cycles to complete. If the values stored at the compared registers used in `BEQ` are equal, only the instructions at lines 14 and 15 at Listing 7.1 will be executed, taking only two cycles.

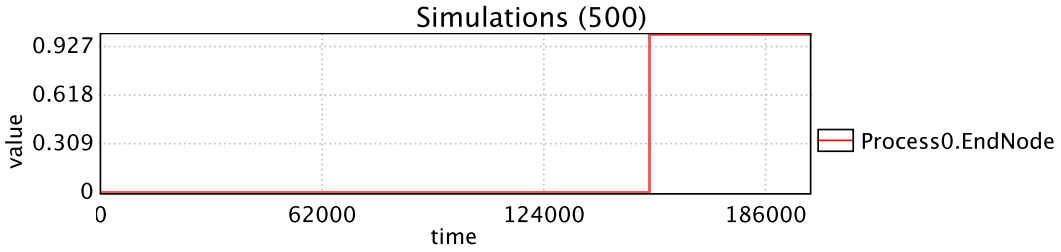


Figure 7.2: The graph shows the difference in cycles, after inserting the additional NOP instructions in Listing 7.1.

```
1  ...
2  beq x2, x0, lbmont_loop_no_sub
3   li          x12, 30
4   li          x13, 24
5   addi        x16, x22, 0
6   li          x8, 4
7   loopi       12, 4
8     bn.lid    x13, 0(x16++)
9     bn.movr   x12, x8
10    bn.subb   w24, w30, w24
11    bn.movr   x8++, x13
12
13  lb:mont_loop_no_sub
14  li          x8, 4
15  li          x10, 4
16
17 ret
```

Listing 7.1: Excerpt of the mont_loop sub-routine.

To make the above-described branches take the same amount of cycles without changing the operations, we can add 65 cycles as NOP instructions to the shorter path. Furthermore, we add an additional jump (JAL) instruction to the long path to skip the added NOP instructions such that they only extend the short path. Because of this, we now need to add 66 cycles as NOP instructions. The program excerpt at Listing 7.2 is the same as the one in Listing 7.1, but includes a loop (1 cycle) running a NOP for 65 cycles.

After inserting the additional NOP instructions to make the two branches from the BEQ instruction use a constant number of cycles, we create a UPPAAL no-data-SMC model, which we simulate 500 times. The result of the simulations at Figure 7.2 includes only two different execution times, 153478 and 153479 cycles, with a single cycle difference. Practically, it can be challenging for an adversary to observe such a small difference, but in the following, we remove it.

```
1   ...
2   beq        x2, x0, lbmont_loop_no_sub
3
4       /*   Start of false-branch    */
5   li         x12, 30
6   li         x13, 24
7   addi       x16, x22, 0
8   li         x8, 4
9   loopi      12, 4
10    bn.lid    x13, 0(x16++)
11    bn.movr   x12, x8
12    bn.subb   w24, w30, w24
13    bn.movr   x8++, x13
14
15  jal x0, lbskip
16
17      /*  End of false-branch    */
18      /*  Start of true-branch    */
19
20  lb:mont_loop_no_sub
21    loopi     65, 1
22      nop
23
24      /* End of true-branch */
25
26 lb:skip
27  li         x8, 4
28  li         x10, 4
29
30  ret
```

Listing 7.2: Excerpt of the `mont_loop` sub-routine with inserted `NOP` instructions.

We now examine Listing 7.3, which is an excerpt of the `modexp_var_3072_f4` sub-routine. If the compared registers in the `BNE` instruction store equal values, the instructions from lines 3 to 10 will be executed. Two of these instructions are executed for 12 iterations in a loop. The instructions from lines 3 to 10 take 27 cycles to complete. If, instead, the values stored at the compared registers are distinct, only the instructions at lines 7 and 10 at Listing 7.1 will be executed, taking only 26 cycles. As expected, due to the results on Figure 7.2 this only leaves a difference on a single cycle.

To make the two branches in Listing 7.3 take the same number of cycles, we can add an additionally `NOP` instruction. Firstly, we make the branches distinct by ensuring that the false-branch does not execute instructions from the true-branch, which we do by inserting an additional `JAL` instruction on line 7 in Listing 7.4. Since this jump takes an additional cycle, we add two `NOP` instructions to the true-branch.

```
1  ...
2  bne        x2, x0, lbf4_no_sub
3  li         x8, 16
4
5  lb:f4_no_sub
6
7  addi       x21, x24, 0
8  loopi      12, 2
9    bn.sid     x8, 0(x21++)
10   addi       x8, x8, 1
11
12 ret
```

Listing 7.3: Excerpt of the `modexp_var_3072_f4` sub-routine.

```
1  ...
2  bne        x2, x0, lbf4_no_sub
3
4      /*   Start of false-branch    */
5  li         x8, 16
6
7  jal x0, lbskip2
8  lb:f4_no_sub
9  nop
10 nop
11     /*   Start of false-branch    */
12     /*   Start of true-branch    */
13 lb:skip2
14
15 addi       x21, x24, 0
16 loopi      12, 2
17   bn.sid     x8, 0(x21++)
18   addi       x8, x8, 1
19
20 ret
```

Listing 7.4: Excerpt of the `modexp_var_3072_f4` sub-routine with inserted `NOP` instructions.

After including the `NOP` instructions, we verify that no timing difference exists in the program. The result of using the query `A<> Process0.EndNode && Process1.EndNode`, to check, is `Property is Satisfied`, which means that we successfully removed the time difference from the *RSA 3072 verify*. The running time of the constant time program is 153480 cycles.

Chapter 8

Conclusion

In this work, we made a tool utilising techniques such as model checking, statistical model checking and symbolic execution to discover timing differences in OTBN assembly programs. In Section 2.1 we introduce the relevant features of OTBN, namely the communication with the host processor, the register files, the controller, the memory, the different types of errors, and the instructions which have a significant impact on the execution time of OTBN programs. We have implemented an interpreter for OTBN programs, described in Section 2.1. The interpreter implements all 58 different instructions in the OTBN ISA. Among other things, we use the interpreter to check the execution time of programs.

The models constructed throughout this work are based on control flow graphs, constructed by translating the OTBN program into a CFG, where the different instructions are represented as nodes, and the edges connecting the nodes describe the control flow in the program. To ease the implementation of the CFG construction, we have some assumptions regarding sub-routines, described in Section 3.1. Additionally, we present and discuss the loop guidelines in Chapter 4, which we assume are followed, to ease the construction of the Uppaal models presented in Section 5.3.

The desired property of having no executions varying in time relates multiple traces, making it a hyperproperty. Therefore, the constructed Uppaal models are based on two processes representing the program under test and a CPU process used to synchronise them. After constructing the Uppaal models, we use both model checking and statistical model checking (SMC) to find the possibility of a timing difference and the size of the possible differences using the Uppaal no-data and Uppaal no-data-SMC models, respectively. For standard model checking, we use the Uppaal no-data model to make Uppaal test whether the desired property is satisfied. The Uppaal no-data model does not consider the data operations of the program under test and is, therefore, an over-approximation. This means that if the property cannot be verified, the counter-examples cannot be guaranteed to be a problem. However, if the property can be verified, the program is guaranteed to have no timing difference. To approximate the size of the time difference, we use Uppaal SMC to simulate Uppaal no-data-SMC model.

However, the Uppaal no-data model disregards the data operations of the instructions, and we, therefore, made the Uppaal with-data model, which executes the instructions. To facilitate this, we have made an external library which supports all the instructions in the OTBN ISA.

Another approach to removing the unreachable traces, instead of using Uppaal with-data, is to use symbolic execution based on the two traces we received as the counter-example,

which UPPAAL provides whenever a property is unsatisfied. However, this implementation still needs to be extended with the ability to remove unreachable traces. Possible approaches to removing traces are explained in Chapter 9.

We used the developed tool to find timing differences in *RSA 3072 verify*, which originates from the official repository of OpenTitan. In this program, we found a timing difference and the potential difference to range between 144942 and 148062 cycles. We inserted no-operation (`NOP`) instructions into the program to make all paths constant time. We then constructed a new UPPAAL no-data model for this program and verified the program's execution time to be constant.

Chapter 9

Future Work

In this chapter, we propose different approaches to explore other traces than the originally proposed as the counter-example to the UPPAAL query. The common problem for all of these approaches is that the set of all possible counter-examples can be infinite, and we, therefore, have to implement some threshold for how many traces we will explore.

9.1 Adding a Guard

One strategy could be to apply the SMT solver each time UPPAAL took a choice at `BEQ`, `BNE`, restarting or exiting loops and then add a guard to the first edge leading to an unsatisfiable path condition, such that UPPAAL are unable to take the edge. However, most likely, other reachable traces, which we want to be proposed by UPPAAL, may also include this specific transition, making it miss potential traces that possibly differ in execution time and are reachable.

To make the approach exclude fewer potentially time-differing traces, the added guard should only disable the transition when the number of cycles is the same as when the initially proposed trace used the transition. With this addition, we will only miss traces that reach the same transition after the same amount of executed cycles, which is still not optimal.

We see this problem on Figure 9.1, which have the guard `cycles != 10` added on the edge between `addi_12` and `ecall_16` since the symbolic execution found the trace where `ecall_16` reached at time 10 unreachable. However, as mentioned above, adding this guard also disallow other combination of loop executions, where the edge is reached after ten cycles.

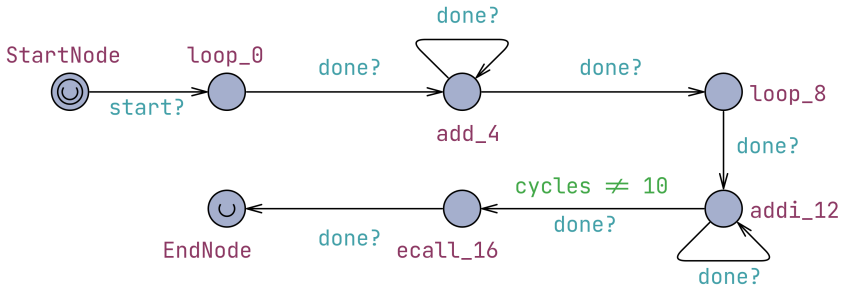


Figure 9.1: This shows a model with the guard `cycles != 10`, which disables the edge, since the TraceChecker found a problem when taken this transition at time 10.

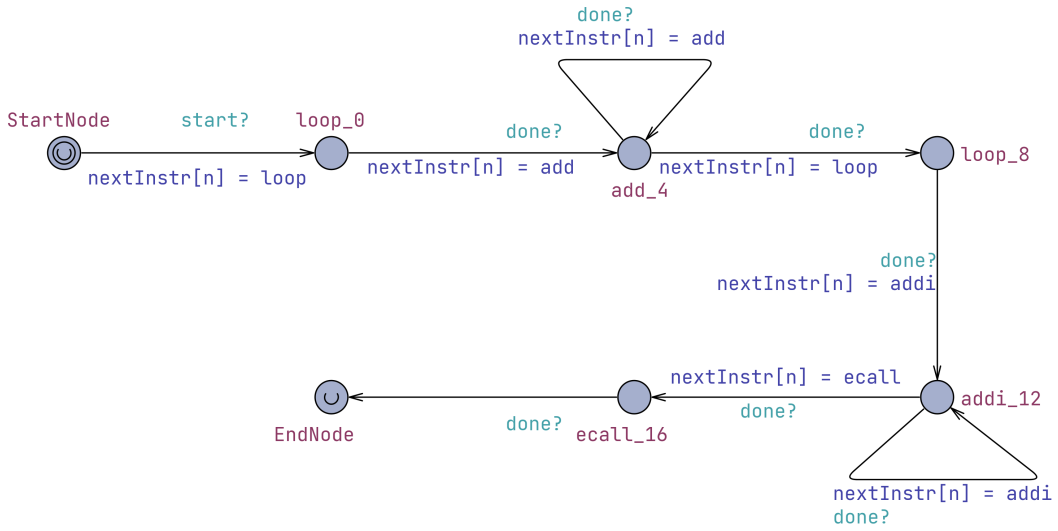


Figure 9.2: This model is similar to the model on Figure 9.1, but we encode the execution into the model.

9.2 Counterexample Trace Automata

Another approach could be to express that the counterexamples must not be some of the traces we already have shown to be unreachable. This can be done by encoding each unreachable trace as an automaton, which we refer to as an unSAT-automaton.

For each trace found unreachable, we construct two unSAT-automata, one for `Process0` and one for `Process1`. We synchronise the unSAT-automata with the `CPU` process in the same manner as `Process0` and `Process1`, through the `done` channel. In addition to the synchronisation, we add guards for each step in the unSAT-automata, ensuring that already seen traces will eventually enter the `EndNode`. Figure 9.3 shows how we construct such an automata.

Furthermore, `Process0` and `Process1` have an id (0 and 1). This id determines which index in the global array each of them writes to. The information these models write to the array is information about the instruction it will run next. This can be seen on the model on Figure 9.2, which have an update statement `nextInstr[n]`, where n is the id. The value at the corresponding index in the array is updated to the next instruction's opcode.

The values stored in the global array are read by the unSAT-automata, which takes a transition, either to `Done` or the next instruction depending on the value it read.

After the construction of these automata, we use the query `E<> (!Process0.EndNode ↙ ↪&& Process1.EndNode) && (!trace0.EndNode && !trace1.EndNode)`. If no trace satisfies the property, we have found all time-differing traces to be unreachable.

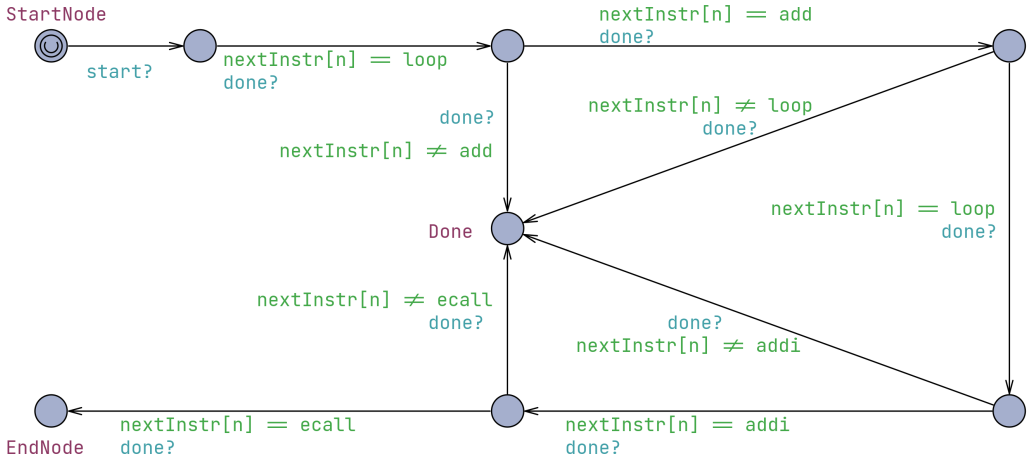


Figure 9.3: This unSAT-automata represent a specific trace through the model on Figure 9.2.

Bibliography

[1] OpenTitan, *OpenTitan Use Cases.* `https://docs.opentitan.org/doc/security/use_cases/`, Retrieved: 01/03/2022.

[2] OpenTitan, *OpenTitan Big Number Accelerator (OTBN) Technical Specification.* `https://docs.opentitan.org/hw/ip/otbn/doc/`, Retrieved: 01/02/2022.

[3] R. N. Fjeldsø and S. S. Nielsen, *Formalising the Execution of OpenTitan Big Number Accelerator Programs*, 2022. `https://projekter.aau.dk/projekter/files/459466352/Formalising_the_Execution_of_OpenTitan_Big_Number_Accelerator_Programs.pdf`.

[4] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Annual International Cryptology Conference*, pp. 104–113, Springer, 1996.

[5] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal 4.0," *Department of computer science, Aalborg university*, 2006.

[6] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen, "Uppaal smc tutorial," *International journal on software tools for technology transfer*, vol. 17, no. 4, pp. 397–415, 2015.

[7] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *International Conference on Computer Aided Verification*, pp. 171–177, Springer, 2011.

[8] OpenTitan, *Introduction to OpenTitan.* `https://docs.opentitan.org/`, Retrieved: 01/04/2022.

[9] OpenTitan, *OpenTitan Big Number Accelerator (OTBN) Instruction Set Architecture.* `https://docs.opentitan.org/hw/ip/otbn/doc/isa`, Retrieved: 01/02/2022.

[10] F. E. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[11] A. Møller and M. I. Schwartzbach, "Static program analysis," *Notes. Feb*, 2012.

[12] F.-X. Standaert, "Introduction to side-channel attacks," in *Secure integrated circuits and systems*, pp. 27–42, Springer, 2010.

[13] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.

[14] T. Roche, V. Lomné, C. Mutschler, and L. Imbert, "A side journey to titan," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 231–248, 2021.

[15] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy*, pp. 11–11, IEEE, 1982.

[16] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010.

[17] E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem, *et al.*, *Handbook of model checking*, vol. 10. Springer, 2018.

[18] A. Legay, A. Lukina, L. M. Traonouez, J. Yang, S. A. Smolka, and R. Grosu, "Statistical model checking," in *Computing and Software Science*, pp. 478–504, Springer, 2019.

[19] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[20] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE symposium on Security and privacy*, pp. 317–331, IEEE, 2010.

[21] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[22] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, pp. 1–38, 2008.

[23] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, vol. 8, pp. 209–224, 2008.