Formal Verification of Hardware Peripheral with Security Property

Formell verifikation av extern hårdvara med säkerhetskraav

VT 2017

NIKLAS ROSENKRANTZ
JONATHAN Y HÅKANSSON
Formal Verification of Hardware Peripheral with Security Property

Jonathan Yao Håkansson, Niklas Rosencrantz

Supervisor: Roberto Guanciale

Examiner: Örjan Ekeberg

CSC, KTH 2017
Abstract

One problem with computers is that the operating system automatically trusts any externally connected peripheral. This can result in abuse when a peripheral technically can violate the security model because the peripheral is trusted. Because of that the security is an important issue to look at.

The aim of our project is to see in which cases hardware peripherals can be trusted. We built a model of the universal asynchronous transmitter/receiver (UART), a model of the main memory (RAM) and a model of a DMA controller. We analysed interaction between hardware peripherals, user processes and the main memory.

One of our results is that connections with hardware peripherals are secure if the hardware is properly configured. A threat scenario could be an eavesdropper or man-in-the-middle trying to steal data or change a cryptographic key.

We consider the use-cases of DMA and protecting a cryptographic key. We prove the well-behavior of the algorithm. Some error-traces resulted from incorrect modelling that was resolved by adjusting the models. Benchmarks were done for different memory sizes.

The result is that a peripheral can be trusted provided a configuration is done. Our models consist of finite state machines and their corresponding SMV modules. The models represent computer hardware with DMA. We verified the SMV models using the model checkers NuSMV and nuXmv.
Sammanfattning


Vi gör jämförelser mellan olika minnesstorlekar och mätte tidsåtgången för att verifiera olika system. Vi ser att tidsåtgången för verifikation är långsammare än linjärt beroende och att relativt små system tar relativt lång tid att verifiera.
Terminology and Definitions

**Cartesian product** - The set of all ordered pairs of two other sets.

**CPU** - Central Processing Unit. It is the center of the computer and performs calculations.

**DMA** - Direct Memory Access. It is a controller unit that can release load off the CPU by transmitting a memory address to a hardware peripheral for direct memory access.

**Formal verification** - A verification of a formalized system such as hardware or software.

**FPGA** - Field-Programmable Gate Array. It is a hardware prototyping device.

**FSM** - Finite State Machine. It is a model of a mechanism which can enter a finite number of states.

**Hardware peripheral** - Computer devices that are connected to the computer.

**HID** - Human Interface Device, e.g. keyboard, touchscreen and mouse.

**HOL4** - An interactive theorem prover.

**MMU** - Memory Management Unit. It is an internal computer hardware that handles memory protection and faster translations between TLB and virtual paging.

**Memory-mapped i/o** - A method of using hardware by reading and writing to a special location in the computer’s main memory.

**Model checker** - A tool useful for model checking

**Model checking** - A process of property checking given a model of a system, automatically check whether a specification is satisfied

**NuSMV** - A Model Checking language and a tool that can verify or make a counterexample of symbolic formulas

**RAM** - Random Access Memory. A data storage simply seen as a large array.

**RS-232** - Recommended standard 232 (as recommended by IEEE). It is a serial interface for UART.

**SDRAM** - Synchronous Dynamic Random Access Memory.

**SoC** - System on a Chip. It is an integrated circuit that integrates all components of a computer or other electronic systems.

**State space** - The set of values which a process can take.

**TLB** - Translation Lookahead Buffer. It is a cache memory to facilitate virtual paging.

**Theorem proving** - A theoretical means to do formal verification usually using a theorem proving tool

**UART** - Universal Asynchronous Receiver / Transmitter. It is a hardware for serial data transmission.

**Virtual Paging** - A mechanism to divide main memory into virtual page numbers that can be cached.
1. Introduction
   1.1. Scope and boundary 6
   1.2. Benefits and Goal-settings 6

2. Background
   2.1. Why peripherals are a security problem 7
   2.2. Model Checking tools 7
   2.3. State of the art 9
   2.4. Related work 11
   2.5. Description and Comparison of Model Checking Tools 11
   2.6. Description of the UART 12
   2.7. Virtual memory 16

3. Specification and Methodology 16
   3.1. Description of Scenario 17
   3.2. Use-cases 18
   3.3. Modelling, specification and verification 19
   3.4. Subtasks 20
   3.5. Simulation 22

4. Description of Modules 22
   4.1. CPU module 22
   4.2. UART module 23
   4.3. DMA module 23
   4.4. RAM module 24
   4.5. MMU module 24

5. Results and Discussion 26
   5.1. Security Properties which were verified 27
   5.2. Future work 27
   5.3. Benchmark results 28

6. Concluding Remarks 29

7. References 31

8. Appendix 32
   8.1. UART Specification 32
1. Introduction

In recent years, Model Checking has emerged as a means both to test and verify the correctness of hardware and software systems. Correctness in this context means that the model satisfies the requirements and that there is a formal proof of correctness. Model checkers can automatically verify whether a model satisfies the requirements [6]. We use NuSMV and nuXmv to conduct formal Model Checking of hardware peripherals. The hardware peripherals we build models for are the 16550 UART with DMA, a RAM and a MMU.

There are tests for software such as unit tests and integration tests, where unit tests are testing a single case but model verification are testing the whole model.

Hardware peripherals and their external connections are usually not part of software unit tests and also not often part of security models or security protocols [2]. The scope of our project is to build and verify formal models of a common external hardware peripheral that has been connected to the computer such as a typical UART RS-232 configuration. The configuration could mean many different devices (terminals, modem, serial lines, ...) or even emulated devices such as a system emulating an HID (a serially connected component behaving just like a keyboard or a modem while actually being something else).

1.1. Scope and boundary

The scope of this project and report is verification with the Model Checking tools NuSMV, nuXmv and SMV and UART there exist more complicated systems to verify. We prove that we can have a secure memory address provided our configuration that we also specify.

1.2. Benefits and Goal-settings

Many external hardware peripherals have no security model when connected to a computer and vice versa. An operating system will in many cases automatically trust a connected hardware peripheral such as a keyboard, mouse or even a network interface that is connected to the computer.

We build a formal model of the UART including its security model, both written in NuSMV. We omit the most fine-grained details of UART and use a simplified model of the hardware [2]. A simplified model is preferable in this case to make it easy to understand for non-technical readers, to limit the time building the model, to fit our scope for an undergraduate thesis and to omit unnecessary details. If the model was too big it should take to much time too and the time for this thesis are very limited.

Since our goal is to guarantee that the UART is a secure communication channel we will identify which parameters and internal states affect which memory addresses that are accessed by the UART and model it.
2. Background

In general there are two techniques for verification and automated proof generation:

1. Model checking, in which a model of a system can be tested and verified automatically whether this model satisfies a given specification. “Model Checking” is a generic name for methods and techniques for automatically checking the compatibility between a model and its formal specification. It is used to verify the correctness of software and hardware systems[5].

2. Automated theorem proving, in which a system attempts to produce a formal proof given a description of the system, a set of logical axioms, and a set of inference rules [5].

We prove that certain security properties hold if the UART is properly configured. We limit the scope of our project to communication via 16550 UART as it is described in its specification. We build a simplified model that still is realistic without too many details. The RS-232 specification is relatively easy in comparison to modern advanced peripherals[2]. Therefore we choose to build a simplified model that can still be useful for different purposes such as prototyping a security property or serve as a template for adding more details later.

2.1. Why peripherals are a security problem

In this thesis, different approaches are discussed for verifying a computer system. USB is not similar to UART in practice, but there are similarities in principle and both are used for data in serial transmission.

A security problem with hardware peripherals is that an operating system usually trusts a connected hardware peripheral that in practice can be used for abuse such as data theft or computer hijacking [8].

There are also similar scopes for other kinds of external hardware such as network controllers (NIC) and hardware peripherals that have built-in processors (GPU, FPGA). We have limited our scope to the 16550 UART. Other external hardware security models will be different in details but there will still be similarities for several properties.

A security property that is checked is distinct from a liveness property that checks that something good will happen. Checking a security property means checking that nothing bad will happen.

2.2. Model Checking tools

Prior to automated Model Checking, mathematical models were difficult to verify by hand. Model checking is a verification technology that provides an algorithmic means of determining whether an abstract model representing, for example, a hardware or software design—satisfies a
specification expressed as a temporal logic (TL) formula. TL was originally developed by philosophers for investigating the way that time is used in natural language expressions [12]. Linear temporal logic (LTL) was later suggested by Amir Pnueli who proposed two systems to handle events that are not always true and called them eventualities that have or will have been true for some time only and distinct from invariants that are always true independent of time.

LTL notation today writes the eventuality is with the letter F (Future). The invariant is written with the letter G (Global) as for example in our formula which we will return to later in this report[5].

\[ G \text{proc.out} = \text{write2UART} \& \text{proc2UART address} = 0 \rightarrow \text{next(uart0.Tx}[0]) = \text{proc2UART value} \]

The “F” in the above formula means that the following statement will be true after a while and not necessarily immediately. In the above specification we have written (by using write2UART) non-deterministically to the first byte of the serial transmission wire and checking that the value actually appears.

There are several factors contributing to the success of Model Checking. Primarily, Model Checking is fully automated. Unlike deductive reasoning using theorem provers, this ‘push-button’ method requires neither proofs nor experts to check whether a finite-state model satisfies given system specifications [2]. Besides verification of correctness, it permits bug detection as well. If a property does not hold, a model checker can return a counterexample denoting an actual execution of the given system model leading to a state that contradicts the specification. Such counterexamples are bugs, provided that the property is a requirement. From a practical aspect, Model Checking also works with partial specifications, which allows the separation of system design and development from verification and debugging.

A Model Checking tool such as NuSMV and NuXMV is a software which automates Model Checking. Model checking is a formal verification that the model satisfies the requirements. If a formula is a true statement about all possible behaviors of the model, then the model checker confirms it. If a formula is a false statement about the model then the model checker constructs a counterexample to the formula that witnesses the falsehood of the formula.

A sequence of transition steps can be executed as an actual use-case. This means that we have a method for formally verifying finite-state systems. Specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not. Extremely large statecharts can often be traversed in minutes. This technique has been applied to on CERN to check PLC programs, for example. Students have shown that it can be applied on the PLC System [1].

A suggested disadvantage of Model Checking is that the technique is limited to handling finite state machines, while software systems are generally specified as infinite state machines and that hardware systems have properties that allow model checking to succeed, while software systems may not exhibit similar properties [1].
2.3. State of the art

Model checking tools like NuSMV and NuXMV use state machines where each node symbolises a reachable state. A graph is nodes with edges that connect each node. To know where to start there is a start node or a set of initial states. Let’s it be called S0. Every transition is an edge in the graph. This structures in model checking is called kripke structure. Transition is from a state to another state. To describe a state propositional calculus is needed. Propositional calculus is which properties that holds in the state[5].

Because the usage of NuSMV in this thesis this beginning of this section shall explain Symbolic model checkers only.

Symbolic or Implicit model checkers are using a logical representation for the states instead. This includes representations like Binary Decision Diagram to evaluate to give information about the sets of nodes that are being satisfied to. A example of BDD is a Binary Decision Diagram that acts like a truth table and Binary Decision Tree. In the Binary Decision Diagram 1 or 0 represent a way to reach f that are described as 0 or 1. Every transition is labeled 0 or 1.[5]

The process of Model Checking has been used successfully to verify complex computer hardware, communication protocols, and software. It is also used for checking and analyzing cyber-physical, biological, and financial systems [6]. Today Intel Corporation uses formal verification to verify the complex chips [12].

Typically, a model checker has a few basic components: a modeling formalism adopted to encode a state machine representing the system to be verified, a specification language based on Temporal Logics [5], and a verification algorithm which employs an exhaustive searching of the entire state space to determine whether the specification holds or not [2].

![Figure 1. A model checker with counterexamples](image-url)
SMV models consist of modules. Each module contains state variable declarations, variable initializations defining the initial states and assignments defining the transition relation.

NuSMV is able to process files written in an extension of the SMV language. In this language, it is possible to describe finite state machines by means of declaration and instantiation mechanisms for modules corresponding to synchronous composition, and to express a set of requirements in CTL and LTL[15]. NuSMV can work batch or interactively, with a textual interaction shell.

NuSMV takes as input a text consisting of a program describing a model and some specifications (temporal logic formulas) [7]. It produces as output either the word ‘true’ if the specifications hold, or a trace showing why the specification is false for the model represented by our program. One of the modules must be called main. Modules can declare variables and assign to them. Assignments usually give the initial value of a variable and its next value as an expression in terms of the current values of variables.

The user can organize the description of an FSM by dividing it into modules. Individual modules can be instantiated multiple times, and modules can reference variables declared in other modules. Standard visibility rules are used for naming variables in hierarchically structured designs. Modules can have parameters, which may be state components, expressions, or other modules.

Modules can be composed either synchronously or using advanced composition mechanisms. In a synchronous composition, a single step in the composition corresponds to a single step in each of the components. With interleaving, a step of the composition represents a step by exactly one component. If the keyword process precedes an instance of a module, interleaving is used.

Nondeterministic is important when making high-level models. The state transitions in a model may be either deterministic or nondeterministic. Nondeterminism can reflect randomness or choice in the states of the system being modeled. The ability to specify nondeterminism is missing from many hardware description languages.

Transition relations A transition relation is a constraint on the values that a variable can assume in the next state, given the value of variables in the current state. The transition relations of modules can be assigned either implicitly in terms of boolean relations on the current and next state values of state variables, or explicitly as a set of parallel assignment statements. The parallel assignment statements define the values of variables in the next state in terms of their values in the current state.

Model checkers are used to evaluate the satisfiability of various specifications and statements of different types.

- Basic (boolean expressions that must become TRUE).
- An end-state (To separate valid end-states from deadlocks).
- Progress-state (To tag states that did something good eligible).
- Accept-state
- Never claims (Behaviour to avoid)
Liveness properties (Proof that something good will happen)

The major problem for the process of Model Checking is a phenomenon called the state explosion problem where the number of states become very large and the system consequently becomes slow and impractically large [2].

2.4. Related work

Dam and Schwartz did formal verification of user processes in ARM with DMA [7]. They wrote that processes should avoid interact with each other. Processes that are suspicious should not have any influence towards processes that exist at higher levels and should not have any knowledge about the other processes at higher level about their execution. They explain that there exists protection rings with Memory Management Unit. There exists control flows of this information that could contain the information that an untrusted process could lay the hands on. Because these flows needs to be documented. A low order register could effect a high order register. These flows could appear without any attention. To make an attack, the lower process could just flip a configuration bit. They proved a non-infiltration in HOL4. It’s some interference between the processes and they try to prove it by HOL4 and that is somewhat similar to this work.

In a thesis written by Schwartz, he explains the importance of the hardware with isolation in today’s society, specifically about vulnerabilities with smartphones connected to internet. There are more electronic devices that are connected to internet than ever before. It says that isolation is an important aspect and when peripherals executes in parallel, the securities of those devices could be threatened. The thesis says that formal verification with model checking is important to verify the correctness of the hardware and it is related to the work because of the use to describing model-checking and HOL4.

A recent paper described correctness for embedded drivers using HOL4 [8]. HOL4 is a tool useful for theorem proving. Direct memory access was not covered in theorem proving of embedded drivers [7]. Our work is different from this work because we include DMA. We also use Model Checking instead of theorem proving.

Kupferman wrote that it is a significant difference between systems that are open and systems that are closed, meaning that it is important to consider the external systems and interconnected systems instead of isolated, closed systems [13]. They also write that model checking of open systems should perform the checking with respect to arbitrary environments and take into account uncertainty regarding the environment.

2.5. Description and Comparison of Model Checking Tools

We evaluated the tools for Model Checking, of which we finally decided to select NuSMV. The reason is that the NuSMV is well established and suitable for the needs. One can use Java Path Finder, NuSMV or the Spin model checker for Model Checking.

We chose between three frameworks.
Java Path Finder JPF, is used to verify executable Java ByteCode programs. JPF was created by NASA AMES Research Center [14]. Main focus of JPF uses and executes Java ByteCode and can store states, match restore program states[14]. Main usage for JPF is Model Checking of concurrent programs. You can use JPF as model check of distributed application, Model Checking of user interfaces, low level program inspection, program instrumentation[14].

SMV. It is a language for verification of finite state systems (FSM)[15]. The tool NuSMV conducts checking of finite state machines and checks if specifications are correct against CTL called temporal logic.NuSMV supports both CTL and LTL. NuSMV is a BDD-based (Binary Decision Diagram) model checker that allows to check finite state systems against specifications in the temporal logic CTL[15]. The software is freely available at http://nusmv.irst.itc.it/ where also a tutorial and manual is available. This program makes it able to have finite systems from completely synchronous to completely asynchronous. Data types are thought to be used as finite state machines and it have only datatypes as Boolean, scalar types, bit vectors and fixed arrays. NuSMV has following features Interaction, Analysis of invariants, Partitioning methods, LTL Model Checking, PSL Model Checking, SAT-Based Bounded Model Checking[15].

We chose SMV and NuSMV because they were readily available, well-established, free of charge for academic use and easy to learn and use. SMV is a specification language for model checking. NuSMV, the model checker, is an Open Source product, it is actively supported and has a substantial user community and it has symbolic model checkers that are making usage better [15].

Spin-Model Checking is well-suited for modeling of concurrent systems. It is a verification system that can be used as verification tool for asynchronous process systems[10]. The Main focus of Spin is process interactions and provide abstract from internal sequence computations[10]. Some of the features that Spin has are:

- An intuitive program-like notation for design choices.
- A concise notation for general correctness requirements.
- A methodology for establishing logical consistency.
- Spin accepts a verification language PROMELA specified in syntax of Linear Temporal Logic [10].

2.6. Description of the UART

UART enables serial character bitstreams between a computer system or an FPGA and an external peripheral. The UART implements the RS-232 protocol timing, and provides adjustable baud rate, parity, stop, and data bits. The feature set is configurable, allowing designers to
implement just the necessary functionality for a given system. The UART provides a memory-mapped interface that allows peripherals (such as a processor) to communicate with the UART by reading and writing control and data registers. Our UART doesn’t have a built-in security model in its current availability [2].

The UART transmitter consists of a 7-, 8-, or 9-bit txdatal register and a corresponding 7-, 8-, or 9-bit transmit shift register. The shift register writes the txdatal register via the Avalon-MM slave port. The transmit shift register is loaded from the txdatal register automatically when a serial transmit shift operation is not currently in progress. The transmit shift register directly feeds the TXD output. Data is shifted out to TXD LSB First. The two registers provide double buffering. A master peripheral can write a new value into the txdatal register while the previously written character is being shifted Out. The master peripheral can monitor the transmitter's status by reading the status register's transmitter ready (TRDY), transmitter Shift Register empty (tmtn), and transmitter overrun error (TOE) bits. The transmitter logic automatically inserts the correct number of start, stop, and parity bits in the serial TXD data stream as required by the RS-232 specification. It is important that the busy signal has no hazards [2].

The UART serial receiver consists of two registers: Bitwise shift register and bitwise holding register. In the case of altera_avalon_uart, the memory management master peripheral reads the holding register. The holding register is loaded from the shift register when a new byte is received. These two registers provide double buffering. The rxdata register can hold a previously received character while the subsequent character is being shifted into the receive shift Register. The receiver logic checks for four exceptional conditions, frame error, parity error, receive overrun error, and break, in the received data and sets corresponding status register bits [2].

The UART uses memory mapped i/o. It requires configuration before use, which is generally achieved by writing values into a memory mapped configuration register. The UART can be used to send and receive arbitrary data asynchronously over two signal wires, TX and RX, respectively [2].

The serial interface uses an asynchronous protocol, i.e. no clock signal is transmitted along the data. The receiver has to have a way to "time" itself to the incoming data bits [2].

In the case of RS-232, that's done this way:

1. Both side of the cable agree in advance on the communication parameters (e.g. speed and format). That's done manually before communication starts.
2. The transmitter sends "idle" ("1") when and as long as the line is idle.
3. The transmitter sends "start" ("0") before each byte transmitted, so that the receiver can figure out that a byte is coming.
4. The 8 bits of the byte data are sent.
5. The transmitter sends "stop" ("1") after each byte.

The specific UART we have chosen is the one that is used in Altera FPGA. At the output of a UART the bits are usually represented by logic level voltages [2].
Figure 1. FSM diagram of the UART shift register.

The state machine waits for the CPU or DMA to place an entry in the transmit FIFO. Once there is data present it transmits one Start bit, 8 data bits and finishes with one Stop bit. The bits are sent starting with bit 0. A design requirement is that two bytes can be sent back to back, so the STOP state must go directly to the START state if more data is available [2].

If the configuration is done properly and correctly, the UART connection cannot allow manipulation of any protected user processes such as web browser or a command-line interpreter. A simple overview of our model of hardware, its policy enforcement (the driver) and a possible intruder can be seen in the following figure [2].
Figure 2. Hardware model.

The figure shows a simplified model of hardware host and the connected external hardware. In the model the K represents the memory of a cryptographic key that should not get overwritten. Device drivers timing constraints are important and also embedded systems have important timing constraints, often deterministic, as part of their specifications. For example, the landing gears of a quadcopter should deterministically respond within a certain number of ms of receiving input [7].

The UART has memory mapped registers (which are input/output and can affect the state of the UART). An additional input and output is the wire to which the UART is connected. Our model represents some features or the hardware such as the minimal set of necessary features of the 16550 UART [2].
2.7. Virtual memory

Virtual memory systems use a technique called paging with a page table that the operating system can change [3]. The page table translates virtual page numbers (VPN) to physical addresses.

![Virtual memory diagram](image)

**Figure 3. Virtual address translating.**

The address translation translates the virtual address to a physical address. If a process erroneously is trying to write to memory that the process does not have privilege to write to then it will be discovered already in the address translation and an error can be signalled.

3. Specification and Methodology

Provided that there is a memory unit “K” (for instance a cryptographic key) at a specific memory address which should be constant. We want to prove that it is always the case that K is not overwritten. We built one model for the main memory, one model for the UART and one main module for the process and the CPU. Then we prove that the following formulas are true. The number 0 means the first bit of the UART. Our first formula says that something good will happen (liveness): If we write to the UART the first bit then after a while (“F” in LTL) then the value will appear on the transmission wire (Tx[0]). The first letter “G” in the formula means that this formula is always true.

$$G ((\text{proc} = \text{write2UART} & \text{proc2UARTAddress} = 0) \rightarrow \text{next} (\text{uart0.Tx}[0]) = \text{proc2UARTvalue})$$

The above formula is checked and proves a functional property.
G (proc2memAddress != memory.secureAddress) -> G (memory.data[memory.secureAddress] = next (memory.data[memory.secureAddress]))

The above formula means that if the CPU doesn’t set the secure memory address for the DMA address then our secure memory will be constant.

G (proc = setDMAaddress & DMAmemAddress != secureAddress& proc2memAddress = secureAddress) -> G (memory.data[secureAddress] = next(memory.data[secureAddress]))

The property above is a functional property that guarantees that eventually something good will happen when we write to the UART at register 0. Because the memory can also be written using DMA, that variable must be included in the original formula. The SMV formulas are similar to modus ponens (logical implication) in computer logic. In SMV an array index must be constant and the state transitions are the only ways of defining changes. Loops are not explicitly coded whereas other specification languages have loops that the programmer can explicitly define.

\[ G \text{proc.out = write2UART & proc2UARTaddress = 0} \rightarrow F \text{next(uart0.Tx[0])} = \text{proc2UART.value} \]

\[ G \text{proc2memAddress != secureAddress} \rightarrow G \text{mem.k = next(mem.k)} \]

Our idea was to look for a counter-example. NuSMV had a library that provided use for the work. In LTL we can write the formulas and prove by counterexample or check all possible states. The reason why are verifying a safety property is to prevent abuse of hardware.

3.1. Description of Scenario

The scenario in the SMV model is that there is data in the memory which represents a cryptographic key, “K” that should not get overwritten. What we verify is that, it always is the case that the data is kept. The UART can transmit and receive and that there is a user process such as a CLI or an internet browser writing input and reading output to and from the UART[2]. The process can also read and write to the main memory of the computer. There are three models being built to represent the scenario: UART, PROC and MEM.

A well-known fact is that a computer mechanism that can be technically attacked (for example physically pulling out the RAM module and by such means extracting the data from the physical RAM memory) will not be 100% safe or completely protected in principle. The safest usage of the UART today is to not leave to connector open. RS-232 doesn’t specify data format or protocol. It can be a modem or something else e.g. X.25
If someone has physical access it is difficult to prevent abuse of the hardware. An attacker can for example take the physical memory modules and read the cryptographic keys, use JTAG, and many possible attacks.

3.2. Use-cases

A DMA use-case is included in the model. DMA is important because it allows the CPU to have less load than during non-DMA reads and writes[3]. It works in a way that the CPU sets a memory address where a peripheral can read or write and then the CPU no longer handles the data, it is written or read directly between the peripheral and the memory. The CPU programs the DMA controller by setting its registers so that the DMA controller knows what to transfer where.

There is an SMV module for the DMA, a module for UART, a module for an MMU and one module for the main memory.

An illustration of the actors and systems looks according to the following.

![Diagram](image)

**Figure 4. The subsystems, external network and process**

The FSM is the result of the synchronous composition of the subsystems the UART, the memory and the process. The new state space is the cartesian product of the ranges of the variables.

The use-cases we present are four:
3.3. Modelling, specification and verification

We built a behavioral model of hardware peripherals: The altera_avalon_uart, the main memory, and we specified the formulas to be verified. The SMV models correspond to the UART, the CPU, DMA and the main memory. The process and the CPU are variables.

The reasons we chose SMV are because of applicability, easy of use and availability. SMV is a language for describing the models we have been drawing as diagrams. SMV can check the validity of LTL (and also CTL) formulas on those models[15].

An SMV program is composed by a number of modules where each module contains state variable declarations, assignments defining the valid initial states and assignments defining the transition relation[15]. In the assignments the lhs can non-deterministically be assigned to any value in the set of values represented by the rhs, for example a finite set of possible states. Wordlength 64 took a considerable longer time to verify than wordlength 8.

The models were built in an assignment style with at least one initial state and all states having at least one next state, for example the memory data in the main module:

next(memory.data[memory.address])

When the reachable states are printed, the number is approximately infinite because of the large number of combinations between processes and memory addresses.

The system diameter is the number of steps needed to get to any reachable state from some initial state. If we make our memory small then the number of states is reported being finite by nuXmv:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>System diameter</td>
<td>2</td>
</tr>
<tr>
<td>Reachable states</td>
<td>6.34587e+63</td>
</tr>
</tbody>
</table>

More metrics were according to the following table.
Applying Model Checking to a design consists of several tasks. The following subtasks were completed in order to complete our main tasks.

1) The first task is to convert a design into a formalism accepted by a Model Checking tool. This means model building in SMV and formalizing the modules.

2) Before verification it is necessary to specify the properties that should be verified. We specified our properties:

   a) Property A: If we write to the UART then after some time the data will appear on the Tx wire.

      \[ G (\text{proc = write2UART} \& \text{proc2UARTAddress} = 0) \rightarrow F \text{next(uart0.Tx}[0]) = \text{proc2UARTvalue} \]

   b) Property B: If the CPU does not explicitly write to the address of the secure data, the secure data will be constant.

      \[ G \text{proc2memAddress} \neq \text{memory.secureAddress} \rightarrow G \text{memory.data}[\text{memory.secureAddress}] = \text{next(memory.data}[\text{memory.secureAddress}]) \]

   c) Property C: If the CPU does not explicitly set the memory address of the secure data for UART to write to with DMA, the secure data will be constant.

      \[ G (\text{proc = setDMAaddress} \& \text{DMAmemAddress} \neq \text{memory.secureAddress}) \rightarrow G (\text{memory.data}[\text{memory.secureAddress}] = \text{next (memory.data}[\text{memory.secureAddress}])] \]
3) We verified virtual addressing with MMU using automatic Model Checking. The short way to verify is to run nuXmv with the model file as an argument. The verification is completely automatic. There were no counterexample found when checking the model. We also checked that the opposites of our specifications were false.

4) We extended our models to include a DMA controller. We included a third specification for DMA.

![Diagram](image)

**Figure 5. The subsystems UART, memory, external network, DMA and process**

This was done by defining a new module named DMA and proving a formula given that the CPU doesn’t set the memory address of the protected cryptographic key at the given memory address.

\[
\text{LTLSPEC } G \left( \text{proc} = \text{setDMAaddress} \land \text{DMAmemAddress} \neq 1 \land \text{proc} = \text{write} \land \text{proc2memAddress} = 1 \right) \rightarrow G \left( \text{memory.data}[1] = \text{next(memory.data}[1]) \right)
\]

5) First the memory model had a very small size (64-128 byte) and word length 8. We then increased both the word length and the array size in the memory of our models to a larger word length, up to 16 bits and 1024 rows of words i.e. 2 KB memory.

21
To measure the time of our commands in NuSMV and nuXmv, we used the „time“ command (the time utility is described in one of the POSIX standards). Our idea was to check that the model took different values between different steps. This was verified.

Doing „set autoexec time“ on the nuXmv shell executed time measurements automatically after each command. Processing the model took 16 seconds for the word length 64 after we increased the word length from 8 to 64. Using a word length of 16 combined with array size 1024 resulted in processing time of 5 minutes on a modern Linux computer with 16 GB RAM and Intel i7 CPU. This can be compared to the 55 minutes it took to verify an 8-bit adder while a 4-bit adder takes less than a second to verify. The different varying memory sizes are reported in the benchmark results at the end of this paper.

We also checked that the DMA formula is false for different memory addresses in the RHS and the LHS of the formula. It was sometimes also checked that the opposite of true formulas is false.

3.5. Simulation

Simulation means that we can apply values to SMV models. We simulated a random execution of our system and received output with all the reachable states. We also checked in nuXmv that the model wasn’t empty (zero) by using the interactive command `pick_state`.

4. Description of Modules

In this chapter the modules are described in more detail.

The module `main` is the top-level module. This module has several variables and instantiates the other modules. The `ASSIGN` statement is used to define the initial states and transitions of every module. To improve clarity, certain less important details have been omitted.

4.1. CPU module

The CPU module consists of registers that can contain a word. The word length is usually 8, 16, 32 or 64. The words are in their turn partitioned into page frames where a physical page usually has the same size as a virtual page. The CPU can also have different processing states. All of the states are not completely implemented in our model, for example the idle state is not implemented and virtual paging is only enabled for the first few pages for the sake of brevity. All states that are verified are implemented.
VAR
  registers : array 0 .. 64 of unsigned word [64];
  proc : {idle, read, write, read2UART, write2UART, setDMAaddress};

Figure 5. SMV code for CPU module

4.2. UART module

The UART module consists of variables and state transitions. The variables are Rx and Tx representing the receiving and transmitting wires. UART is an important part of the model because it is a connection to the outside where a malicious attacker could attempt abuse of our system. Reading from the UART is represented according to the following state transition.

next (uart2CPU) := RX

Figure 6. SMV code for UART module

The Rx and Tx data types were defined in different ways depending on their respective functionalities. Rx was defined as receiving a word from the outside and Tx was defined as an array of words that is transmitted. The reason for this choice of data types is that the state transitions for Tx must be defined for every element whereas Rx could be defined in a single state transition.

4.3. DMA module

The DMA module is also defined according to variables and state transitions. It is instantiated with the processor and the DMA address that the CPU sets. The DMA controller contains several registers that can be written and read by the CPU. These include a memory address register, a byte count register and one or more control registers.

MODULE DMA (proc, DMAmemAddress)
VAR
  address : 0..2047;
ASSIGN
  next (address) :=
    case
      proc = setDMAaddress : DMAmemAddress;
      TRUE : address;
    esac;
4.4. RAM module

The RAM module has memory arranged in array of bytes. We experimented with different word lengths and different array sizes and chose the following definition. The reasons why we didn’t make the memory even larger were that the principle is proven already for a small memory size and that larger memory sizes will take significant longer time for the model checker to verify.

```plaintext
data : array 0 .. 2047 of unsigned word [ 64 ];
```

Our main memory was gradually increased from a very small memory that initially was used to verify the model. The RAM module includes specified state transitions given the DMA address and the address from the CPU in the other case of transition. The elements of the data array start the secure data part at the array index number `secureAddress` (for example any byte number can be used to begin the secure data part).

```plaintext
next (data[0]) :=
  case
    proc = write & proc2memAddress = 0 : proc2memValue;
    TRUE : data[0];
  esac;
next (data[secureAddress]) :=
  case
    proc = write & proc2memAddress = secureAddress : proc2memValue;
    TRUE : data[secureAddress];
  esac;
```

4.5. MMU module

Virtual memory systems use a technique called paging which we have partially implemented. We added an MMU to the modules, since this component can prevent the CPU from writing into the critical region of memory where you stored our cryptographic key.

The virtual addresses consist of twice as many pages as the number of physical page frames. Virtual memory addresses do not go directly to the memory. Instead, they go to the MMU that maps the virtual addresses onto the physical memory addresses. TLB management and handling TLB faults are done by the MMU. Our MMU module includes a page table to enable virtual
memory and a translation lookaside buffer for cache. Instead of a direct memory address, virtual pages with offset are verified and the secure data is chosen at a fixed physical address as well as a fixed page number. We initiate paging between 3 physical and 3 virtual pages in the page table. The TLB works as a cache memory for address translations and contains data for validness with a valid bit and which virtual and physical memory that should be mapped. The purpose of the TLB is to make paging fast.

```
MODULE MMU (proc, pageNumber, offset)

VAR
  pagetable : array 0 .. 7 of 0..3; -- 8 pages of virtual memory points to 4 pages of physical memory
  tlb : array 0 .. 7 of array 0..4 of 0..3; -- valid, virtual page, modified, protection, page frame number
ASSIGN
  next (pagetable[0]) := -- initiate paging between virtual page 0 and physical page 2
     case
       TRUE : 2;
     esac;
```

**Figure 9. SMV code for MMU module**

The TLB performs lookups of physical page frame numbers from virtual page numbers. The TLB is a cache memory for the page table.

The functional verification of the system with virtual memory was done with the following formula which checks that if we write to a certain virtual page with a certain offset, the data will appear in the main memory at the specified location.

\[
G ((\text{cpu.proc} = \text{write} & \text{pageNumber} = 0) \Rightarrow \text{next(memory.data[mmu.pagetable[0]][0])} = \text{cpu.proc2memValue})
\]

The index 0 in the formula above means the first byte of the first virtual page that is 16 bytes. One could also write a whole page instead of a byte. We write non-deterministically to the virtual memory and find it appearing in the physical memory after a while. The mapping in the page table represents address translation between virtual and physical memory.

| 0                  | Page frame 0 of physical memory | Page table 3 7 5 2 4 |
Table 2. The page table of our address space

<table>
<thead>
<tr>
<th>16</th>
<th>Page frame 1</th>
<th>Unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>Page frame 2</td>
<td>VPN 3</td>
</tr>
<tr>
<td>48</td>
<td>Page frame 3</td>
<td>VPN 0</td>
</tr>
<tr>
<td>64</td>
<td>Page frame 4</td>
<td>VPN 4</td>
</tr>
<tr>
<td>80</td>
<td>Page frame 5</td>
<td>VPN 2</td>
</tr>
<tr>
<td>96</td>
<td>Page frame 6</td>
<td>Unused</td>
</tr>
<tr>
<td>112</td>
<td>Page frame 7</td>
<td>VPN 1</td>
</tr>
</tbody>
</table>

The page table is a data structure that is used to map virtual page numbers to physical addresses (physical frame numbers).

We verify the security property for the MMU:

$$G \text{mmu.pagetable}[\text{pageNumber}] \neq \text{memory.secureAddress} \Rightarrow G(\text{memory.data}[\text{memory.secureAddress}][0] = \text{next}(\text{memory.data}[\text{memory.secureAddress}][0]))$$

The above formula means that if the CPU doesn’t write to the secure address then the secure data will be constant. It is verified in our third system.

**5. Results and Discussion**

In this chapter the results are presented which we also discuss.

It is important to verify both security properties and liveness properties. A security property is a verification that nothing bad happens. A liveness property is a verification that eventually something good will happen.

In total 9 specifications were verified. The results of our verification look according to the following output. The first specification is satisfied meaning that writing to the UART after a while appears on the wire Tx. This verification is a liveness property i.e. we verify in temporal logic that eventually something good will happen and our data will appear on the Tx wire.

$$G(\text{proc} = \text{write2UART} \& \text{proc2UARTAddress} = 0) \Rightarrow F\text{next(uart0.Tx}[0]) = \text{proc2UARTvalue}$$
The second specification is that the secure data is constant when we write to the UART, provided that the memory address is not that of the secure data. This is a security property being verified i.e. a verification that nothing bad happens.

\[ G (\text{proc} = \text{write2UART} \& \text{proc2memAddress} \neq \text{memory.secureAddress}) \rightarrow \\
G (\text{memory.data}[\text{memory.secureAddress}] = \text{next}(\text{memory.data}[\text{memory.secureAddress}]) \]

The third specification is that the cryptographic key is constant when writing with DMA.

\[ G (\text{proc} = \text{write2UART} \& \text{proc2memAddress} \neq \text{memory.secureAddress}) \rightarrow \\
G \text{memory.data}[\text{memory.secureAddress}] = \text{next}(\text{memory.data}[\text{memory.secureAddress}]) \]

The fourth security property that we verify is that the key is constant also with an MMU included. The 0 in the formula below means the first byte of the page.

\[ G \text{mmu.pagetable}[\text{pageNumber}] \neq \text{memory.secureAddress} \rightarrow G \\
(\text{memory.data}[\text{memory.secureAddress}][0] = \text{next}(\text{memory.data}[\text{memory.secureAddress}][0])) \]

The above specifications were verified as true for different memory sizes where the different memory sizes had effect on running time of the model checker. We found that the difference is rather large in running time of the Model Checker between a small (64*64 bits) and a larger (2 kilobyte) main memory: Checking a small memory takes less than a second while a larger memory take several minutes for the model checker to complete.

We also checked for emptiness of the model and verified that our states were not always empty or zero states. This was done using the nuXmv interactive mode.

5.1. Security Properties which were verified

Verification in Model Checking means that we can verify models with a method. NuSMV includes property verification where each property is separately verified and the result is either “TRUE” or “FALSE” + counterexample.

The verification has not found a counterexample to the formulas. This means that the security property holds. The property that the system must satisfy is that the data of the cryptographic key “K” remains the same in the next state. It is verified by Model Checking that a cryptographic key in the MEM module will have the same data between two states. Since a counter-example was not found the model checker that our formulas are true.

In total there were four modules built in one file. The main module is mandatory, The models we built also include the UART module, a DMA module and a memory module.
5.2. Future work

Memory can be increased more and also using more parameters in the benchmark. We expect model checking to be used more in the industry and increasingly because verification of systems can improve and control quality.

Further extensions to our models can be to make separate memories for instruction and data and try the states for actual programs that are larger than the physical memory.

We expect functional properties to be verified in the industries i.e. formal verification used for the functional properties of a system and not only for the security properties.

5.3. Benchmark results

To get an idea of the time complexity for our method, we compared a 4-bit system to an 8-bit system where the 4-bit system took less than a second to verify and the 8-bit system took 54 minutes to verify. We then compared different memory sizes and adjusted parameters so that the system is not too large and thus too slow to traverse for the model checker. Our benchmark results suggest that the verification time of the model checker at least doubles or even quadruples every time we double a length.

After varying the memory size both by array length (memory size) and word length, the following benchmark results were found for our second system that included DMA. The model checker appears to take exponentially increasing time with increased memory. We used an Intel i7 CPU with 16 GB RAM to run the benchmarks. The following were the times it took to verify the different systems. The times are approximate and should be taken as approximations that show the increased time of calculations when verifying larger memory sizes.

<table>
<thead>
<tr>
<th>Physical memory size</th>
<th>With UART</th>
<th>With DMA</th>
<th>With MMU and 5 virtual pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 bytes</td>
<td>&lt; 1 s</td>
<td>&lt; 1 s</td>
<td>30 s</td>
</tr>
<tr>
<td>512 bytes</td>
<td>1 s</td>
<td>1 s</td>
<td>150 s</td>
</tr>
<tr>
<td>1 KB</td>
<td>30 s</td>
<td>30 s</td>
<td>Segfault after 7 minutes</td>
</tr>
<tr>
<td>2 KB</td>
<td>150 s</td>
<td>150 s</td>
<td>Not attempted</td>
</tr>
</tbody>
</table>

Table 3. Benchmark results
We also verified a 4-bit adder and an 8-bit adder to compare the verification times. The 8-bit adder verified in 55 minutes which doesn’t permit quick changes to the model.

6. Concluding Remarks

In this project verification with NuSMV and nuXmv did not take a very long time because our memory models were small. We experienced increased running times due to state explosion, a problem that is commonly reported in Model Checking [4]. The state explosion occurred when we increased the memory model from 2 kilobyte to 4 kilobyte where the model checker started to take a significant longer time to check the models. Therefore we chose to keep a memory of 2 kilobyte and not increase it more so that the verifications would not take a very long time. Approximation of the number of states for our systems becomes very large and therefore certain numbers were output as infinite because the numbers are very large and there was number overflow in the testing environment.
We think that Spin/Promela would have worked too for modelling our systems. We realize that verifying entire modern computer systems using model checking takes a long time due to the state explosion problem and that abstract models of the systems should be verified to practically finish the verification in reasonable time.
7. References

7. Dam, M., Schwarz, O. (2014). Formal verification of secure user mode device execution with DMA. In Haifa Verification Conference (pp. 236-251). Springer International Publishing.
14. NASA Java Path Finder Home Page babelfish.arc.nasa.gov/trac/jpf/wiki
15. NuSMV Model Checker Home Page, nusmv.fbk.eu
### 8. Appendix

#### 8.1. UART Specification

<table>
<thead>
<tr>
<th>Name</th>
<th>16550 UART</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group</td>
<td>Serial</td>
</tr>
<tr>
<td>Data bits</td>
<td>Determines the widths of the txdata, rxdata, and endopacket registers.</td>
</tr>
<tr>
<td>Stop bits</td>
<td>Use to terminated a receive transaction at the first stop bit.</td>
</tr>
</tbody>
</table>