




# Refinement-Based Verification of Device-to-Device Information Flow

Ning Dong , Roberto Guanciale , Mads Dam   
KTH Royal Institute of Technology

**Abstract**—I/O devices are the critical components that allow a computing system to communicate with the external environment. From the perspective of a device, interactions can be divided into two parts, with the processor (mainly memory operations by the driver) and through the communication medium with external devices. In this paper, we present an abstract model of I/O devices and their drivers to describe the expected results of their execution, where the communication between devices is made explicit and the device-to-device information flow is analyzed. In order to handle general I/O functionalities, both half-duplex (transmission and reception) and full-duplex (sending and receiving simultaneously) data transmissions are considered. We propose a refinement-based approach that concretizes a correct-by-construction abstract model into an actual hardware device and its driver. As an example, we formalize the Serial Peripheral Interface (SPI) with a driver. In the HOL4 interactive theorem prover, we verified the refinement between these models by establishing a weak bisimulation. We show how this result can be used to establish both functional correctness and information flow security for both single devices and when devices are connected in an end-to-end fashion.

**Index Terms**—Formal verification, Refinement, Serial interface, Device driver, Interactive theorem prover, Information flow

## I. INTRODUCTION

I/O devices are indispensable components for interactions with the external environment (e.g., print documents, transmit data, and receive user’s commands). Their proper operation is critical for trustworthiness: Poorly written device drivers are the predominant reason for operating system crashes [1]–[3], and devices themselves can be vulnerable to side-channel attacks [4], [5].

Existing work [6]–[10] mostly focuses on the verification of functional properties of device drivers, by analyzing the interactions between the controlling software and the I/O device. In this paper, we present a verification approach that includes inter-device communication. This allows to establish end-to-end information flow properties, for example to guarantee the absence of side channels.

Our strategy is based on refinement. First we define a formal “concrete” model of a specific I/O device, which formalizes the device behavior that is observable by the controlling software and other external devices, and a model of its device driver. The combination of these two models provides a software/hardware subsystem that can interact with other software

This work has been supported by the TrustFull project funded by the Swedish Foundation for Strategic Research. Ning Dong is supported by the China Scholarship Council for his doctoral studies.

components and external devices. We then define an abstract model of this subsystem, which is independent of the actual device and provides a general blueprint of the subsystem’s desired behavior and information flows. The goal is that this abstract model should provide a functionality that is correct and secure by construction, similar to ideal models used in cryptography. Our refinement establishes a weak bisimulation between the concrete and abstract systems.

There are three main benefits of this approach:

- Bisimulation allows to transfer both functional properties and information flow properties (e.g., progress-sensitive noninterference [11]) of the abstract model to the concrete one.
- The same abstract model can be refined by models for different I/O devices.
- The compositionality of bisimulation allows to preserve the verified properties when we compose the subsystem with other components: e.g., we can compose the subsystem with the other software or subsystems to show inter-host properties.

We choose the Serial Peripheral Interface (SPI) as the demonstrating example, and we provide the formal model of a specific device, the Texas Instruments McSPI device used in the AM335x family of processors [12], and its driver. The Serial Peripheral Interface is a synchronous protocol for serial communication that is mainly used in embedded devices. The protocol was first introduced in the late 1970s by Motorola and has become popular because of its simplicity and speed [13]. SPI devices support both half-duplex and full-duplex data transmissions, where the latter is used to improve performance by simultaneously sending and receiving data with external devices. Although full-duplex is effective in practice, this is to our knowledge the first example of verification in the literature of a full-duplex communication device, cf. [6]–[10].

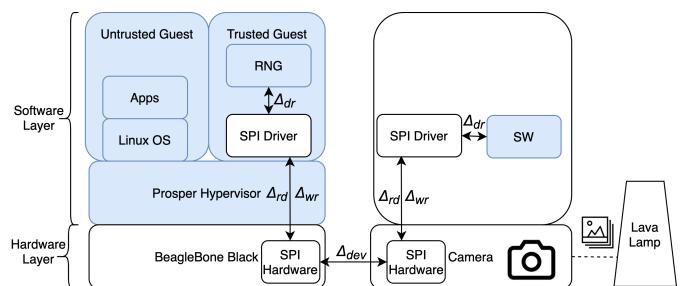


Fig. 1. The architecture of a random number generator

We use the refinement to establish several interesting properties of the system: (1) The driver never leads the device to enter a configuration that is undocumented by the hardware specification; (2) Two interconnected SPI subsystems correctly and securely exchange data when they are activated by their controlling software; (3) Communications (driver-to-device and device-to-device) provide progress-sensitive noninterference at both concrete and abstract levels. The latter is established by a notion of contextual indistinguishability derived from the weak bisimulation.

To demonstrate our results, we developed the demonstrator of Figure 1. We use a BeagleBone Black running the verified Prosper hypervisor [14] together with an Arducam Shield Mini 2MP Plus camera to capture a physical source of randomness for, in our case, the Verificatum e-voting system [15]. The two devices communicate using SPI. The verification allowed us to slim down the driver by removing some unnecessary device register operations. The driver model is a direct manual translation of the driver binary. Formalization of this step is left as future work. In section X, we discuss our approach to automate this step by establishing a bisimulation between the driver model and its binary.

All proofs and models have been formalized in the HOL4 interactive theorem prover [16], which supports specification and proof in classical higher-order logic. For full definitions and proofs, we refer the reader to <https://github.com/kth-step/sw-spi-cam-model/releases/tag/fmccad>.

## II. BACKGROUND

In this work, we model one of the devices of BeagleBone Black. This is a widely used development board with multiple peripherals, including SPI, I2C, UART, etc. The board has a TI AM335x processor [12] that uses the 32-bit ARMv7 instruction set architecture.

We focus on the SPI subsystem. Figure 2 shows the basic components involved in the SPI protocol: hardware connection, a controller, and a peripheral. In full-duplex mode, SPI permits to transmit and receive data simultaneously on separate data lines, SDI (Serial Data In) and SDO (Serial Data Out). The SPI controller uses the serial clock (SCK) line to maintain synchronization with the peripheral device. During each SPI clock cycle, from the controller’s perspective, one bit is transmitted from the controller to the peripheral on the SDO line, while the peripheral sends one bit to the controller on the SDI line. In half-duplex SPI transmissions, only one data line is used depending on the controller settings. In transmission-only mode, only the SDO data line is used, and vice versa for reception-only. The controller uses the chip select (CS) line to choose the desired communicating peripheral when multiple peripherals are connected. In this paper, we consider only the single peripheral case; extension to multiple peripherals is straightforward.

Bit transmission on the SDO/SDI lines is governed by the controller clock signal SCK, depending on configuration (clock polarity and edge settings). The SPI protocol can transmit messages of normally up to 16 bits, and delegates all error

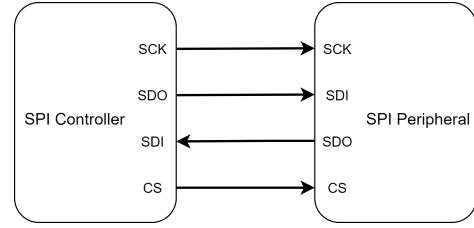


Fig. 2. Basic SPI connection: a controller and peripheral

detection, flow control, and application adaptation to higher-layer protocols. A driver can interact with the SPI hardware by register polling, interrupts, and Direct Memory Access (DMA). In this work, we rely on polling only. The following registers of the BeagleBone SPI controller are the ones used for polling:

- 1) The CP (controller/peripheral) bit of the MC (module control) register configures the SPI hardware as a controller (CP = 0) or a peripheral (CP = 1).
- 2) The channel configuration register (CCF) maintains the configuration of the communication channel. For instance, the TRM (transmit/receive modes) 2-bits of the CCF register controls the half and full-duplex modes: the values 0, 1, and 2 represent full-duplex, receive-only, and transmit-only respectively. The WL (word length) 5-bits configures the word length of the transmitted and received data. In our case, the driver fixes the WL bits to 7, which means the SPI word is 8-bits long, as all models transmit and receive bitwise data.
- 3) The TX0 (transmit buffer) register contains the data to transmit. The RX0 (receive buffer) register contains the received message bits.
- 4) The CST (channel 0 status) register is a read-only register and provides information about the status of TX0 and RX0 registers. The TXS (transmitter register status) bit of the CST register indicates if the TX0 register is empty: its value is 1 when the TX0 register is empty and can be written with the next word to transmit, and is 0 when the TX0 register is full and should not be overwritten. Analogously, the RXS (receiver register status) bit of the same register indicates the status of the RX0 register: its value is 1 when the RX0 register is full when data in the RX0 register is ready to be fetched and 0 when RX0 is empty.

## III. ARCHITECTURAL MODEL

We model devices and drivers as labelled transition systems (LTS) in the style of CCS [17], modelling the interaction between software and driver, driver and device, as well as between devices (through signals “on the wire”) by the simultaneous occurrence of an action  $\alpha$  and its dual  $\bar{\alpha}$ , where  $\alpha, \bar{\alpha} \in \Delta_{wr} \cup \Delta_{rd} \cup \Delta_{dev} \cup \Delta_{dr}$ . The top components of Figure 3 summarize the interfaces among models. Here,  $\Delta_{wr}$  is the set of write operations by the CPU, which is represented by the action  $wt\ a\ v$  for writing a byte  $v$  to the register with the memory-mapped address  $a$ , and the dual action  $\overline{wt\ a\ v}$

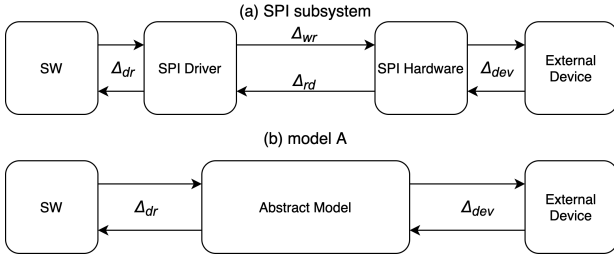


Fig. 3. The model architecture of SPI subsystem and abstract model

that is the corresponding action of the device. Similarly,  $\Delta_{rd}$  is the set of read operations by the CPU which is represented by the action  $rd\ a\ v$  for reading  $v$  from the register mapped at address  $a$ , and the dual action  $\overline{rd\ a\ v}$ . Representing this interaction as a CCS-style synchronous rendez-vous allows to reflect the potential side effects of register accesses on the SPI hardware. In the terminology of  $\pi$ -calculus [18], we use the “early” semantics. For instance, the reading of a memory-mapped register by the CPU non-deterministically spawns one transition for every possible resulting value.

The device model uses four additional types of action to model device-to-device interactions on the wire. The convention needs to take controller/peripheral asymmetry into account. For transmission-only mode the controller uses  $\overline{tx\ v}$  to send a byte  $v$  over the wire, and in reception-only mode  $tx\ v$  to receive a byte from the wire. For synchronous transfer of the (controller) byte  $v$  and (peripheral) byte  $v'$ , the controller uses  $xfer\ v\ v'$ . The peripheral uses the dual actions, i.e.,  $\overline{tx\ v}$  ( $\overline{tx\ v}$ ) for reception (transmission) and always  $xfer\ v\ v'$  for synchronous transfer. Let  $\Delta_{dev} = \{tx\ v, \overline{tx\ v}, xfer\ v\ v', \overline{xfer\ v\ v'} \mid bytes\ v, v'\}$ . Finally, the driver uses four additional actions to model invocations of the driver API by application SW and one additional action for returning control and result to SW (collected by  $\Delta_{dr}$ ).

The SPI subsystem consists of the SPI hardware running in parallel with its device driver with internal communication channels (e.g.,  $rd\ a\ v$ ), made inaccessible to the external world. In CCS parlance this is  $(d|s) \setminus (\Delta_{wt} \cup \Delta_{rd})$ , where  $d$  and  $s$  are states of the driver and hardware, respectively.

#### IV. SPI HARDWARE MODEL

The state of the SPI hardware is represented by a tuple  $s = (regs, sreg, c)$ . Here,  $regs$  is a function mapping addresses of memory-mapped registers to words, and  $sreg$  represents the internal hardware-controlled shift register for data transmission and reception. The component  $c$  captures the control state of the device and is used to track the progress of its four functionalities: initialization, transmission, reception, and full-duplex synchronous transfer.

With the exception of register RX0, register reads are side-effect free and simply communicate the current value of the register: i.e., for every state  $s$ ,  $s \xrightarrow{rd\ a\ s.reg(a)} s$ . Transitions that model register writes (i.e.,  $s \xrightarrow{wt\ a\ v} s'$ ) have side effects and are modeled by early instantiating all possible received values. Since many register updates are not atomic and require

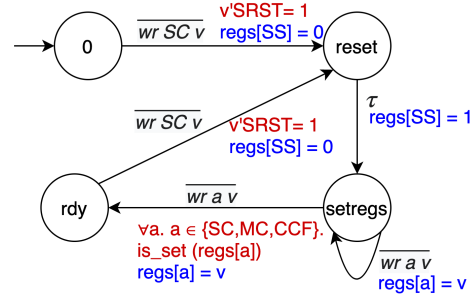


Fig. 4. SPI hardware initialization automaton

some time to take effect (e.g., writing into the transmission register does not automatically transfer the byte on the wire), transitions  $s \xrightarrow{wt\ a\ v} s'$  are usually followed by a silent transition  $s' \xrightarrow{\tau} s''$ , which is the system internal transition that applies the visible side effects.

A special error state  $\perp$  is entered under the following conditions:

- 1) The hardware receives read or write requests that violate the SPI specification [12] (e.g., the RX0 register is read when its value is indeterminate);
- 2) The hardware attempts an operation that is not allowed by the specification (e.g., to update the shift register before the initialization is completed);
- 3) An operation is not supported by the formal model, for instance, accessing control registers beyond the single channel modelled here.

The behavior of transitions that have side effects can be represented by an automaton, which is split into four sub-automata for the four device functionalities.

1) *Initialization*: Figure 4 shows the hardware initialization automaton, where the black, red, and blue annotations describe the label, enabling conditions and side effects of transitions respectively. Note that we have omitted all transitions that lead to  $\perp$  in Figure 4, which applies to the following figures as well. The initialization is activated when the value 1 is written to the SRST (software reset) bit of the SC (system configuration) register. The  $\tau$  transition exiting state *reset* models the hardware completion of the reset operation and sets the SS (system status) register to 1. This register can be used by a driver to detect when the reset process is finished. In state *setregs*, the device awaits the set up of the hardware configuration, which is achieved by writing the SC, MC, and CCF registers. This step is necessary before starting data transmissions because the SPI hardware needs basic parameters, like the CP bit of the MC register and the WL bits of the CCF register. If one of these register updates sets a value that does not conform with the specification (e.g., the value of WL bits should no less than 3), then the model enters the state  $\perp$ . Once all required registers have been written, the model enters the ready state *rdy*. Now the SPI can be utilized for data transmissions or be reinitialized.

2) *Synchronous transfer*: Figure 5 depicts the synchronous transfer sub-automaton. From the ready state, the synchronous

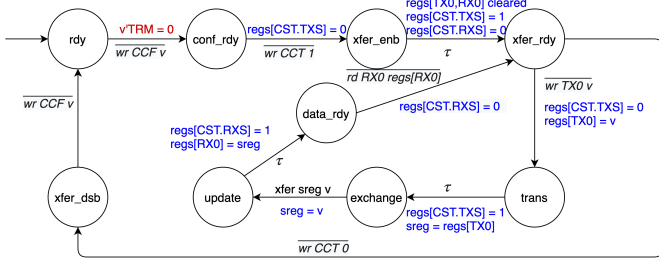


Fig. 5. SPI hardware synchronous transfer automaton

transfer is activated when the TRM bits of the CCF register are set to 0. Then, updating CCT with 1 activates the state *xfer\_enb* and clears the TXS bit. The following silent transition makes the side effect of enabling the channel visible: the registers TX0 and RX0 are cleared, and the TXS and RXS bits are set to 1 and 0 respectively. From *xfer\_rdy*, once the message  $v$  to transmit is written to TX0, the TXS bit is cleared. The following silent transition transfers the data from the TX0 register to the shift register and the TXS bit is set internally. The device will now synchronize with an external SPI device, simultaneously transmitting the shift register and receiving one byte  $v$ , which is copied into the shift register. The following silent transition makes the communication visible to the driver, by copying the shift register to RX0 and setting the RXS bit. Finally, from the state *data\_rdy*, the received data can be fetched by reading the RX0 register. This also resets the RXS bit. The transmission process is repeated until the channel is disabled by writing 0 to the CCT register in the state *xfer\_rdy* and then resetting the CCF register to its original value.

As mentioned before, from the diagram in Figure 5, we have omitted all transitions that lead to  $\perp$ . This happens, for instance, if TX0 is written before the TXS bit is set or when the model is in the state *data\_rdy*, or if RX0 is read while RXS is not set.

3) *Transmission and reception*: The structure of the half-duplex automata for transmission and reception is similar to the synchronous transfer automaton. However, there are some notable differences:

- 1) The transmission and reception automata are activated by setting the TRM bits to 1, resp. 2 for receive-only, resp. transmit-only mode.
- 2) In transmission mode, the transmission automaton will not receive data from the external device, which means the RXS bit remains unchanged. The EOT (end-of-transfer status) bit of the CST register is used to indicate the end of transmission. The EOT bit is cleared when *sreg* is updated with the output data, and it is set when the data is transmitted to the external device. In this way, a driver can check the EOT bit rather than the RXS bit when applying the transmit-only mode.
- 3) After the channel is enabled for the receive-only mode in the reception automaton, the hardware first receives the external data and then uploads it to the RX0 register. Therefore, unlike the synchronous transfer automaton,

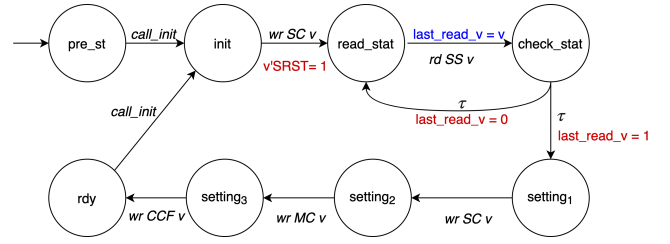


Fig. 6. Driver initialization automaton

the TX0 register should not be used. A correct driver should wait for the hardware until the received data is ready through reading the RXS bit. The TXS and EOT bits are not applied in the reception automaton.

## V. SPI DRIVER MODEL

The driver model is a direct manual translation of the real SPI driver binary and interacts with the hardware model using operations on the device registers. The model exposes all accesses to memory-mapped registers that are performed by the actual driver.

The driver state is a tuple  $d = (b_1, b_2, idx, last\_read\_v, c)$ . Here,  $b_1$  is the transmit, and  $b_2$  the receive buffer. The variable  $idx$  points to the next byte in  $b_1$  to be transmitted. The byte  $last\_read\_v$  is the last returned value from the hardware, used for the driver's internal operations. The last component  $c$  is the driver's control state. We define sub-automata corresponding to each of the four device functionalities.

1) *Driver initialization*: Figure 6 shows the driver initialization automaton. The automaton is invoked by an external call to the driver initialization function, represented here by the action *call\_init*. In state *init*, the automaton writes the SC register to reset the hardware. Then the automaton reads the SS register and updates the  $d.last\_read\_v$  with the returned value. In the state *check\_stat*, the automaton checks the fetched value to determine if the hardware finished the reset process. If the value is 1, the automaton enters the state *setting1*, otherwise it returns to the previous state and repeats this loop. Finally, the automaton enters the ready state by setting several registers in order (SC, MC, and CCF), indicating that the driver model is prepared to process function calls for data transmissions and reinitialization.

2) *Driver synchronous transfer*: The driver synchronous transfer automaton is shown in Figure 7. With the driver in state *rdy*, the automaton is invoked by action *call\_xfer* with a buffer  $b_1$  copied to the driver's internal output buffer ( $d.b_1$ ). Before starting data transmission, the automaton first prepares the necessary settings for the hardware by writing the CCF and CCT registers. Notice that CCF is read prior to writing in order to maintain other channel configurations (e.g., transmission speed). At this point, the automaton loops reading the CST register and checking the TXS bit, as long as the value of TXS is 0. Once the value 1 is read, the automaton enters the state *write\_data*. The following step writes the TX0 register with one byte data that is sent to the external device, leading to the state *read\_rxs*. Hereafter, the automaton repeatedly reads the

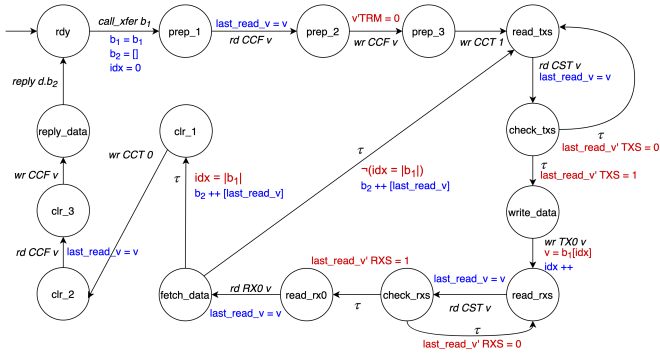


Fig. 7. Driver synchronous transfer automaton

CST register as before but checks the RXS bit rather than the TXS bit, which indicates the hardware transmission is finished and the received data is available in the RX0 register. If the RXS bit is 1, then the automaton in the state *read\_rxs* issues a read request to the RX0 register. Next, the automaton can fetch the received data and check if all bytes in the output buffer are transmitted. If there are more bytes to transmit, the automaton returns to the state *read\_txs* and repeats the process. Otherwise, the automaton clears the CCT register and the CCF register to their initial values. Finally, the driver replies the received data (*d.b2*) to the program that invoked the driver by using the label *reply* and returns to the ready state.

The driver's transmission and reception automata are similar and left out.

## VI. ABSTRACT SPI SUBSYSTEM SPECIFICATION

In this section, we present an abstract specification of the combined device and driver subsystem. The model has the same interface as the concrete SPI subsystem (see Figure 3 (b)) and describes the visible effects of the four functionalities (i.e., initialization, full-duplex synchronous transfer, transmission, and reception) while ignoring all internal states of the SPI hardware and the memory-mapped device registers. The state of the abstract model is a pair,  $a = (t, c)$ . The component  $t = (b_1, b_2, idx, v)$  is the data state, which contains the output and input buffers  $b_1$  and  $b_2$ , the index of the next byte to be transmitted  $idx$ , and the received byte  $v$ . The component  $c$  is the control state of the abstract model.

The abstract initialization and synchronous transfer automata in Figure 8 are largely self-explanatory. The control structure is the obvious one with bytes in the transmit buffer *a.t.b1* being sent one by one and received bytes getting stored in *a.t.b2*. Note also that once in the ready state reinitialization must remain enabled.

## VII. REFINEMENT

The refinement is established by exhibiting a weak bisimulation [19]. This approach is useful to allow multiple levels of concretizations and abstractions through transitivity and compositionality (under parallel) of the corresponding equivalence.

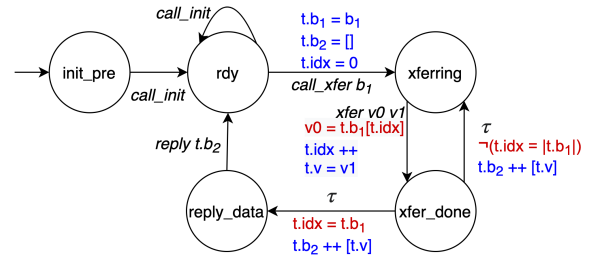


Fig. 8. Abstract initialization and synchronous transfer automata

Below we use  $p \xrightarrow{\tau^*(a)}_1 p'$  to indicate an arbitrary number of  $\tau$  transitions, optionally followed by an  $a$  transition.

**Definition VII.1 (Weak bisimulation).** Given two transition systems  $(S, \rightarrow_1)$  and  $(T, \rightarrow_2)$ , a binary relation  $R \subset S \times T$  is a weak simulation if for every  $(p, q) \in R$ :

- If  $p \xrightarrow{a}_1 p'$  then  $q \xrightarrow{\tau^* a}_2 q'$  for some  $q'$  s.t.  $(p', q') \in R$ .
- If  $p \xrightarrow{\tau}_1 p'$  then  $q \xrightarrow{\tau}_2 q'$  for some  $q'$  s.t.  $(p', q') \in R$ .

The relation  $R$  is a weak bisimulation if both  $R$  and  $R^{-1}$  are weak simulations. In the following, we write  $S \sim_R T$  when  $R$  is a weak bisimulation, and  $S \sim T$  if there exists  $R$  such that  $S \sim_R T$ .

Our weak bisimulation definition is slightly different from the standard definition that allows arbitrary  $\tau$  transitions after the observation  $a$  (e.g.,  $q \xrightarrow{\tau^* a \tau^*}_2 q'$ ). It is easy to show that our definition entails the standard one.

Weak bisimulation is transitive and compositional:

**Theorem VII.1.** If  $S \sim_{R_1} T$  and  $T \sim_{R_2} U$  then  $S \sim_{R_1 \circ R_2} U$ , where  $p (R_1 \circ R_2) q \Leftrightarrow \exists r. p R_1 r \wedge r R_2 q$

**Theorem VII.2.** If  $S \sim_R T$  then  $S|U \sim_{R'} T|U$ , where  $p|r R' q|r \Leftrightarrow p R q$ .

### A. An intermediate model

In order to show a weak bisimulation between the SPI subsystem and the abstract model  $A$ , we introduce an intermediate model  $B$ . The intermediate model, still abstracting from memory operations, has the states  $b = (t, sreg, c)$  with the control state  $c$  as in the abstract model, and with  $t$  of the shape  $(b_1, b_2, idx)$ , i.e., as  $t$ , but not including the received byte  $v$ , which is instead represented in an explicit shift register *sreg*, as in the SPI hardware model. Figure 9 shows on the top the full-duplex synchronous transfer automaton of the  $B$  model, and on the bottom demonstrates in part the weakly bisimilar control states in blue of the SPI subsystem under a relation  $R_1$ . For example, the control state *update* of the  $B$  model is weak bisimilar with two states of the SPI subsystem, (*check\_rxs|update*) and (*read\_rxs|update*) (driver and hardware's control states respectively). The control state (*check\_rxs|update*) is reached from the (*read\_rxs|update*) by reading the CST register, which is omitted in the  $B$  model. The  $\tau$  transitions between two control states that are weakly bisimilar with the same abstract state are also ignored. In our example, if the RXS bit is 0 when the SPI hardware

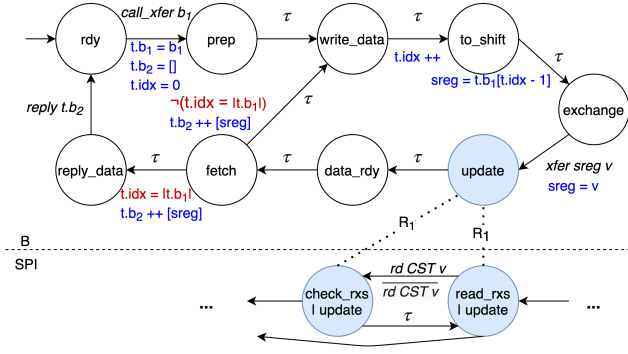


Fig. 9. Model  $B$  synchronous transfer automaton and part weak bisimulation

is in the control state  $update$ , the driver will return to the previous state by internally checking the fetched value. This stepwise approach makes it much easier to build the desired bisimulation relation.

### B. Weak bisimilarity of the abstract and SPI models

The following two lemmas show the weak bisimilarity of  $B$  and SPI models,  $A$  and  $B$  models respectively.

#### 1) Weak bisimilarity of the intermediate and SPI models:

We define a relation  $R_1$  for the  $B$  and SPI models, which matches their control states as indicated in Figure 9 and requires the equivalence of data buffers and records, shift registers, etc. In addition, the relation  $R_1$  requires that if  $b$  is not in the error state then neither are the driver and hardware models, and vice versa.

**Lemma VII.1.**  $(d|s) \setminus \{\Delta_{wr} \cup \Delta_{rd}\} \sim_{R_1} b$

*Proof:* The two models have the same four functionalities, and the state transitions of the two models can be divided into the corresponding four sub-automata. We comment on the full-duplex synchronous transfer automaton, since the transmission and reception are similar and the initialization is straightforward. There are four kinds of transitions in this automaton for both models:  $call\_xfer\ buf$ ,  $xfer\ v\ v'$ ,  $\tau$  and  $reply\ buf'$ .

- $call\_xfer\ buf$ : The main point is to guarantee that the driver model performs the buffer copy and clears the internal received buffer as prescribed by the intermediate model.
- $xfer\ v\ v'$ : When the two models are in the control state  $exchange$ ,  $xfer\ v\ v'$  is used to exchange single bytes  $v$ ,  $v'$  with the external device. In order to guarantee weak bisimilarity, the driver must guarantee to write the value  $v$  to the TX0 register.
- $\tau$ : The major concern is to show the equivalence of data buffers, index and shift registers of the two models. There are three critical requirements that the driver should adhere to, otherwise the hardware model enters the error state and the weak bisimulation condition is violated.

- 1) The driver should delay writing the TX0 register until the TXS bit is 1, because the value 0 of TXS bit means the TX0 register is not ready to be written.

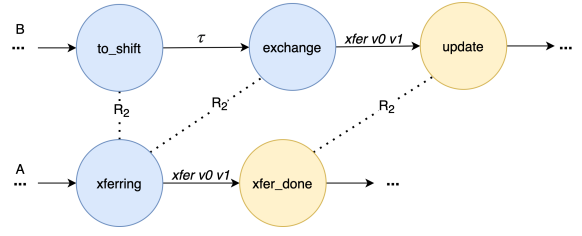


Fig. 10. Weak bisimulation example of the  $A$  and  $B$  models

This also means the driver should not immediately write the next byte after legally writing the TX0 register.

- 2) The driver should wait for the RXS bit to become 1 before reading the RX0 register. Otherwise, the RX0 register may not contain the received data.
- 3) To avoid error situations, the driver should read the CCF register before writing in order to keep the necessary channel configurations unchanged, such as WL bits.

- $reply\ buf'$ : When replying, the driver must ensure that the data in  $buf'$  is identical to the bytes read from the device.

2) *Weak bisimilarity of the abstract and intermediate models:* The relation  $R_2$  is defined in a similar way for the abstract and intermediate models. Figure 10 shows the relation for a part of the synchronous transfer automata of the two models, where weakly bisimilar control states are coloured identically. This relation basically matches control states under the requirement that buffers and records remain unchanged. The bisimulation condition forces input and output data of the two models to be the same.

**Lemma VII.2.**  $b \sim_{R_2} a$

*Proof:* Same methodology as for Lemma VII.1. ■

From Theorem VII.1, Lemma VII.1 and Lemma VII.2, it directly follows that there is a relation  $R_3$  for the abstract and SPI models:

**Theorem VII.3.**  $(d|s) \setminus \{\Delta_{wr} \cup \Delta_{rd}\} \sim_{R_3} a$  where  $R_3 = R_1 \circ R_2$

## VIII. SYSTEM PROPERTIES

In order to demonstrate the functional properties of the system, we verify three theorems for the abstract model. These theorems transfer easily to the concrete models using the bisimulation results of Section VII. Additionally, we show that the abstract (SPI subsystem) model never enters the error state.

The functional correctness of full-duplex synchronous transfer should show that buffers are exchanged correctly between two devices. To show this property, we define the process  $G(a_0, a_1) = (a_0 | (a_1 \{xfer\ v\ v' / xfer\ v' v\})) \setminus \Delta_{dev}$ , which composes the abstract model of an SPI subsystem with a “dual” paired device: if one controller device uses  $xfer\ v\ v'$  to transmit and receive data, the peripheral device uses the dual

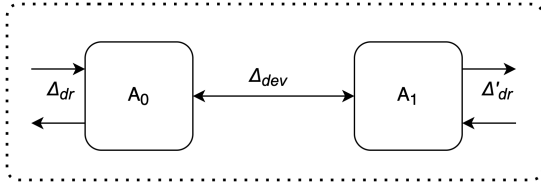


Fig. 11. Composition of two devices

label to synchronize. Figure 11 depicts the composition of two devices.

Theorem VIII.1 shows the functional correctness of the full-duplex synchronous transfer. Notice that buffers must have the same length, otherwise the larger buffer cannot be transmitted in its entirety.

**Theorem VIII.1.** *If  $0 < |b_0| = |b_1|$ ,  $(t_0, rdy) \xrightarrow{\text{call\_xfer } b_0} a_0$ , and  $(t_1, rdy) \xrightarrow{\text{call\_xfer } b_1} a_1$ , then  $\exists n a'_0 a'_1 a''_0 a''_1. G(a_0, a_1) (\xrightarrow{\tau})^n G(a'_0, a'_1) \wedge a'_0 \xrightarrow{\text{reply } b_1} a''_0 \wedge a'_1 \xrightarrow{\text{reply } b_0} a''_1$*

*Proof:* We show that the first byte can be exchanged correctly and then complete the proof by induction. ■

An analogous theorem shows the correctness of transmission/reception. In this case,  $l$ , the number of bytes to be received, should be greater than or equal to the length of the data buffer  $b_0$ , otherwise extra data of the buffer will be lost.

**Theorem VIII.2.** *If  $0 < |b_0| \leq l$ ,  $(t_0, rdy) \xrightarrow{\text{call\_tx } b_0} a_0$ , and  $(t_1, rdy) \xrightarrow{\text{call\_rx } l} a_1$ , then  $\exists n a'_0 a'_1 a''_1. G(a_0, a_1) (\xrightarrow{\tau})^n G(a'_0, a'_1) \wedge a'_1 \xrightarrow{\text{reply } b_0} a''_1$*

Finally, we show that the abstract model can never enter an erroneous state. The bisimulation transfers this property to the SPI hardware and the driver:

**Theorem VIII.3.** *If  $c \neq \perp$  and  $(t, c) \rightarrow (t', c')$ , then  $c' \neq \perp$*

## IX. INFORMATION FLOW SECURITY

Formal device and driver verification projects have generally focused on functional correctness [6]–[10]. However, the device driver can possibly leak sensitive information and therefore, for critical applications, information flow analysis is needed. One of the main benefits of establishing weak bisimulation instead of a simulation is that the former guarantees that two systems have the same information flows (up to channels that are not modeled here, like timing). We show that weak bisimilarity is sufficient to capture progress-sensitive noninterference (PSNI), in the sense of Hedin and Sabelfeld [11]. Let  $E$  be the set of transition labels of the system under consideration. In our case, we may consider a system as in Figure 11 with  $E = \Delta_{dr} \cup \Delta'_{dr}$ , where  $\Delta_{dr}$  and  $\Delta'_{dr}$  are distinct driver interfaces that are both high, since the interfaces are used to communicate sensitive data. We assume a context  $C$  that is allowed to interact with the system using any label in  $E$ . This context is additionally equipped with a public, distinguished interface of labels  $P$  that the context can use to receive and produce publicly observable stimuli. Then, any observations using labels in  $P$  that can cause the abstract and

concrete models to be distinguished must be due to  $C$  being able to bring the two systems to states that  $C$  can distinguish. Of course, if the two systems are weakly bisimilar, this is in fact not possible, motivating the following definition.

**Definition IX.1** (Contextual indistinguishability). *Two states  $s_1$  and  $s_2$  are contextually indistinguishable,  $s_1 \approx s_2$ , if for every context  $C$ ,  $(s_1 | C) \setminus E \sim (s_2 | C) \setminus E$ .*

We use the term contextual indistinguishability instead of contextual equivalence, as the former considers only contexts of very specific shapes. It is not the case that contextual indistinguishability implies contextual equivalence in general, as the latter is a congruence, specifically under CCS sum, which is former is not. However, weak bisimulation *is* a congruence under parallel composition and restriction. Thus, if  $s_1$  and  $s_2$  are weakly bisimilar, then they are also contextually indistinguishable. The converse implication, of course, does not hold. It also follows directly that  $\approx$  is transitive.

The concept of contextual indistinguishability is related to Focardi et al.'s nondeducibility of composition (NDC) [20], which in our setting would be the condition  $(s | C) \setminus H \sim s \setminus H$  on  $s$ , where  $H$  represents the high labels and  $C$  is restricted to interact using only  $H$ . However, it is not clear how to adapt the NDC condition to our refinement-based setting, and also, in contrast to contextual indistinguishability, the NDC condition is not able to accommodate systems such as ours that obtain low observability only through the use of the context.

For the definition of PSNI, a run  $\pi$  is any sequence of transitions starting from an initial state. Such a run is *complete* if it cannot be extended, i.e., it is either unbounded or ends in a final state. For a run  $\pi$ , we let  $O(\pi)$  be the list of public labels in  $\pi$ . We can now define PSNI adapted to our setting of reactive systems as follows:

**Definition IX.2** (PSNI). *Two states  $s_1$  and  $s_2$  are PSNI, if for every complete run  $\pi_1$  starting from  $s_1$ , there exists a complete run  $\pi_2$  starting from  $s_2$  such that  $O(\pi_1) = O(\pi_2)$ , and vice versa.*

The definition can be seen to be equivalent to the one in [11], or in terms of termination only, with the notion of weakly termination-sensitive noninterference of [21]<sup>1</sup>.

Contextual indistinguishability is a sufficient condition for PSNI, because it guarantees the existence of traces for two transition systems with the same observable labels.

**Theorem IX.1.** *If  $s \approx t$ , then  $s$  and  $t$  are PSNI.*

If  $s$  and  $t$  are not PSNI, then we find a complete run  $\pi_1$  from  $s$  such that all complete runs  $\pi_2$  starting from  $t$  have different low observations from  $\pi_1$ . Clearly, this allows a context  $c$  using labels in  $L \cup H$  to steer  $s$ , possibly nondeterministically, into a state  $s'$  that cannot be matched by  $t$ , in the sense of weak bisimilarity. Here  $L$  represents low labels.

<sup>1</sup>In fact, at our low level of modelling, with weak bisimulation, the adversary does not have any model-external means (such as exhausting the memory) at its disposal to prevent progress. Hence our account is also strongly termination-sensitive in the terminology of [21].

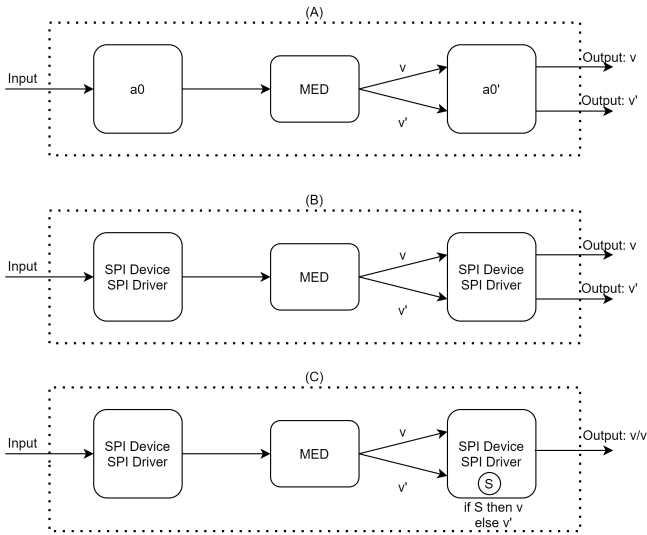


Fig. 12. Information flow security example

We can also show that PSNI transfers under  $\approx$ :

**Theorem IX.2.** *Suppose  $s \approx s'$  and  $s'$  and  $t$  are PSNI. Then  $s$  and  $t$  are PSNI.*

We cannot in general replace weak bisimulation by the corresponding notion of simulation in the definition of contextual indistinguishability. A device driver may leak a sensitive boolean  $s$  by either terminating execution conditionally on  $s$  or by entering a diverging loop (e.g., `while (s) {}`), but still be (weakly) simulated by the abstract model. In this case, an external attacker may discover the value of the secret boolean by observing the impossibility of transmission of a buffer.

Also, establishing bisimulation allows to compose the system with non-deterministic components safely. For instance, we can introduce a faulty communication medium (*MED*) between two devices that can indeterminately deliver wrong values. Figure 12 (A) represents the abstract model where two abstract devices (our *A* model) are connected through the given medium. As a result of the medium, the final output of the abstract model is non-deterministically  $v$  or  $v'$ . The compositionality of the weak bisimulation guarantees that in the system where the two concrete SPI subsystems are interconnected by the same medium (see Figure 12 (B)), the final output is also non-deterministically  $v$  or  $v'$ : the system has the same information flows. On the other hand, the system (Figure 12 (C)), where the receiving device driver decides the value according to a secret value, leaks a secret value via the final output. This model cannot be validated using contextual indistinguishability, but it can be when weak bisimulation is replaced by a corresponding notion of weak simulation.

## X. APPLICATION: SECURING A RANDOM NUMBER GENERATOR USING SPI

As a demonstrating application, we developed a secure random number generator (RNG) that relies on the SPI hardware for sourcing entropy. The architecture of the system is depicted

in Figure 1. The blue components are the software components not including the SPI driver(s). The SPI driver interacts with the SPI hardware through operations on memory-mapped registers ( $\Delta_{rd}$  and  $\Delta_{wr}$ ). We use a BeagleBone Black to connect with an Arducam Shield Mini 2MP Plus camera through SPI. The RNG captures images of the floating material in a lava lamp. This has been shown to be a good source of physical randomness [22], [23].

In order to prevent vulnerabilities of other software affecting the RNG, we develop a bare-metal application that integrates the SPI driver and that is executed on top of the Prosper hypervisor [14]. This is a hypervisor for ARMv7-A processors that provides provable separation between different guests and can be configured to grant accesses to the SPI registers to a dedicated partition only, running our driver. This allows an untrusted partitioned Linux guest (such as in our case, the Verificatum e-voting application [15]) to harden the built-in Linux RNG with physical randomness through a hypercall interface provided by the hypervisor with strong end-to-end security guarantees. In this scenario, the SPI subsystem plays an important role. Additionally to failing to function, a faulty device driver may reduce the entropy of the system by simply returning predictable buffers or it could communicate, directly or indirectly, internal data to the external device. Formal verification of the driver model allows us to rule out these problems. Moreover, it helped to identify redundant operations of the driver. For example, the initial version (extracted from the u-boot library) sets up the WL bits of the CCF register whenever the transmission functions are used, however it is enough to set them once in the initialization function.

In order to guarantee the absence of vulnerabilities at the code level, the refinement should be pushed down to the binary code of the device driver. We extract the driver model by manual inspection of the driver binary. This step has yet to be formalized. We don't view this as a major weakness, however, given that the memory-mapped registers use uncached memory only. We have experimented with the usage of the binary analysis tool HolBA [24] for verifying weak bisimilarity of the driver's assembly code and the driver model. The weak bisimulation relates fragments of binary instructions (i.e., program counter addresses) to a state of the driver's automaton. Each fragment has a single entry point, and either (1) consists of one single instruction accessing a device register or (2) does not access the device. In the former case, the instruction directly corresponds to a transition of the driver model. In the latter case, the fragment corresponds to a finite sequence of silent transitions. We then translate the relation into pre/post conditions for the fragments, which can be analyzed via HolBA weakest precondition tool and a Satisfiability Modulo Theories (SMT) solver.

## XI. RELATED WORK

Some previous work has applied the bisimulation methodology for verification in a theorem prover context [25], [26]. For example, Röckl et al. [25] verified the correctness of several communication protocols by proving weak bisimilarity. We



prove the equivalence of the abstract and SPI models using the same approach.

Several projects of formal verification of low-level software have focused on the operating system (OS), like seL4 [27] and CertiKOS [28]. However, the functional correctness of device drivers usually is not considered. For example, the seL4 microkernel [27] only guarantees the isolation of device drivers located in the user space, where the correctness of drivers is ignored. CertiKOS [28] initially did not verify the drivers as well. Based on CertiKOS, Chen et al. [10] developed a verified interruptible operating system with device drivers. They proposed a general device model with several instantiations and a realistic formal model of device interrupts. Although their device model has similarities with the one presented here, there are notable differences:

- 1) Their device model only contains events that can be observed by the CPU and ignores events that the external environments can observe. Our models consider device-to-device operations and properties (e.g., data transmissions);
- 2) Their device model covers only half-duplex communication (e.g., sending and receiving data over the UART port), while we also model full-duplex data transmission in both the abstract and concrete models;
- 3) In their case, device drivers are implemented inside the OS kernel and each device driver is treated as running independently on its own logical CPU. This requires a different isolation property of the OS kernel to guarantee the separation between different devices and the kernel, which is not provided by most OS kernels. Here, we describe the device driver as a normal process that can be embedded either inside or outside of the OS kernel.

Other previous work on verifying the functional correctness of device drivers studied various I/O devices, like UART [7], hard disk [8], and USB OHCI [6]. In their work, there is no abstract I/O device model to represent the general behaviours of different I/O devices, and it is too restrictive to extend their work on other hardware devices. Duan et al. [9] proposed an abstract device model that is plugged into the formal model of ARMv4 instruction set architecture and later extended it to support interrupts with respect to the ARMv7 architecture [29]. However, the device state is merged into the machine state in their model, which requires to carefully handle the interleavings between the execution of the device and processor. Because of the complexity, it is difficult to apply their model to verify I/O devices.

## XII. CONCLUSION AND FUTURE WORK

We modeled and verified an SPI subsystem that consists of the device hardware and its driver. The verification establishes a weak bisimulation between this model and an abstract specification, which is used to transfer functional and information flow properties of the abstract model to the concrete one.

Our methodology can be reused to verify other SPI subsystems by establishing a refinement with the abstract model

presented in this paper. There are some valuable lessons we have learned from this project:

- 1) Reading the hardware technical reference manual is not sufficient to understand the usage of real hardware. For instance, the order of some operations is unclear. Since the concrete hardware design is usually unavailable, lots of experiments are needed to properly account for the actual functionalities of different I/O registers.
- 2) The abstract model must capture the intended information channels. For example, our initial driver model did not have the *reply* label. It prevents the indented leakage of the received bytes to the software invoking the driver and makes it impossible to establish a refinement with the actual implementation.
- 3) It is usually inconvenient to build an abstraction of the device without taking the driver into account. Indeed the very purpose of the driver is to provide a tractable and efficient abstraction of the generally highly configurable hardware. This turns out to be useful not only for programming but also for verification.

In order to complete the binary verification of the device driver, we plan to follow the strategy of Section X, which establishes a bisimulation between the SPI driver model and its binary code using contract-based verification of the HolBA platform [24]. Moreover, we are planning to address two limitations of the current models: The absence of DMA and interrupts. While these can be encoded via explicit synchronizations processor/device-memory or processor-device, we think that explicit treatment of these features can simplify models and proofs [30]. Currently, our models are shallowly embedded in HOL4. This allows us to partially automate our proof via the HOL4 standard tactics. For example, large parts of the proof search are fully automated using METIS\_TAC. Our work can give insight for deeply embedding the models in HOL4. This can provide a general framework for modeling multiple types of I/O devices and increase automation by implementing decision procedures for checking bisimilarity.

Finally, our information flow analysis does not deal properly with side channels. How to do this is an open challenge, even for uncached memory, as here. For instance, precisely modelling timing is infeasible for real systems since we do not have accurate timing information of the underlying hardware. A more successful strategy consists in defining abstract leakage models in the form of observations (e.g., accessed memory addresses affect caches that in turn affect the timing) and preventing timing side channels by proving observational equivalence. We are currently working on validating [31] such models and defining methodologies to handle different side channels at each refinement step [32].

## REFERENCES

- [1] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001, pp. 73–88.
- [2] A. Ganapathi, V. Ganapathi, and D. A. Patterson, "Windows XP kernel crash analysis," in *LISA*, vol. 6, 2006, pp. 49–159.

- [3] V. Orgovan and M. Tricker, "An introduction to driver quality," in *Microsoft Windows Hardware Engineering Conf*, 2003.
- [4] J.-M. Schmidt, T. Plos, M. Kirschbaum, M. Hutter, M. Medwed, and C. Herbst, "Side-channel leakage across borders," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2010, pp. 36–48.
- [5] M. Li, Y. Zhang, Z. Lin, and Y. Solihin, "Exploiting unprotected I/O operations in AMD's secure encrypted virtualization," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1257–1272.
- [6] D. Monniaux, "Verification of device drivers and intelligent controllers: a case study," in *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, 2007, pp. 30–36.
- [7] E. Alkassar, M. Hillebrand, S. Knapp, R. Rusev, and S. Tverdyshev, "Formal device and programming model for a serial interface," in *Proceedings, 4th International Verification Workshop (VERIFY), Bremen, Germany*, vol. 259, 2007, pp. 4–20.
- [8] E. Alkassar and M. A. Hillebrand, "Formal functional verification of device drivers," in *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 2008, pp. 225–239.
- [9] J. Duan and J. Regehr, "Correctness proofs for device drivers in embedded systems," in *SSV*, 2010.
- [10] H. Chen, X. Wu, Z. Shao, J. Lockerman, and R. Gu, "Toward compositional verification of interruptible OS kernels and device drivers," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 431–447.
- [11] D. Hedin and A. Sabelfeld, "A perspective on information-flow control," in *Software safety and security*. IOS Press, 2012, pp. 319–347.
- [12] *AM335x and AMIC110 Sitara Processors Technical Reference Manual*. Texas Instruments, 2019. [Online]. Available: <https://www.ti.com/lit/ug/spruh73q/spruh73q.pdf>
- [13] S. Choudhury, G. Singh, and R. Mehra, "Design and verification serial peripheral interface (SPI) protocol for low power applications," *International Journal of Innovative Research in Science, Engineering and Tegnology*, pp. 16 750–16 758, 2014.
- [14] R. Guanciale, H. Nemati, M. Dam, and C. Baumann, "Provably secure memory isolation for Linux on ARM," *Journal of Computer Security*, vol. 24, no. 6, pp. 793–837, 2016.
- [15] "Open Verificatum project." [Online]. Available: <http://verificatum.org/>
- [16] K. Slind and M. Norrish, "A brief overview of HOL4," in *International Conference on Theorem Proving in Higher Order Logics*. Springer, 2008, pp. 28–32.
- [17] R. Milner, *A Calculus of Communicating Systems*, ser. Lecture Notes in Computer Science. Springer, 1980, vol. 92. [Online]. Available: <https://doi.org/10.1007/3-540-10235-3>
- [18] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, I," *Inf. Comput.*, vol. 100, no. 1, pp. 1–40, 1992. [Online]. Available: [https://doi.org/10.1016/0890-5401\(92\)90008-4](https://doi.org/10.1016/0890-5401(92)90008-4)
- [19] R. Milner, *Communication and concurrency*, ser. PHI Series in computer science. Prentice Hall, 1989.
- [20] R. Focardi, R. Gorrieri, and F. Martinelli, "Non interference for the analysis of cryptographic protocols," in *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000*, ser. Lecture Notes in Computer Science, vol. 1853. Springer, 2000, pp. 354–372.
- [21] V. Kashyap, B. Wiedermann, and B. Hardekopf, "Timing-and termination-sensitive secure information flow: Exploring a new approach," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 413–428.
- [22] L. C. Noll, R. G. Mende, and S. Sisodiya, "Method for seeding a pseudo-random number generator with a cryptographic hash of a digitization of a chaotic system," Mar. 24 1998, US Patent 5,732,138.
- [23] J. Liebow-Feeser, "Lavarand in production: The nitty-gritty technical details," Apr 2021. [Online]. Available: <https://blog.cloudflare.com/lavarand-in-production-the-nitty-gritty-technical-details/>
- [24] A. Lindner, R. Guanciale, and R. Metere, "Trabin: trustworthy analyses of binaries," *Science of Computer Programming*, vol. 174, pp. 72–89, 2019.
- [25] C. Röckl and J. Esparza, "Proof-checking protocols using bisimulations," in *International Conference on Concurrency Theory*. Springer, 1999, pp. 525–540.
- [26] P. Manolios, K. Namjoshi, and R. Sumners, "Linking theorem proving and model-checking with well-founded bisimulation," in *International Conference on Computer Aided Verification*. Springer, 1999, pp. 369–379.
- [27] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "seL4: Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 207–220.
- [28] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, and Y. Guo, "Deep specifications and certified abstraction layers," *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 595–608, 2015.
- [29] J. Duan, *Formal verification of device drivers in embedded systems*. The University of Utah, 2013.
- [30] O. Schwarz and M. Dam, "Formal verification of secure user mode device execution with DMA," in *Haifa Verification Conference*. Springer, 2014, pp. 236–251.
- [31] H. Nemati, P. Buiras, A. Lindner, R. Guanciale, and S. Jacobs, "Validation of abstract side-channel models for computer architectures," in *International Conference on Computer Aided Verification*. Springer, 2020, pp. 225–248.
- [32] C. Baumann, M. Dam, R. Guanciale, and H. Nemati, "On compositional information flow aware refinement," in *IEEE Computer Security Foundations Symposium*, 2021.