Kronos: Verifying leak-free reset for a system-on-chip with multiple clock domains

by
Noah Moroze
B.S., Massachusetts Institute of Technology (2020)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
February 2021

© Massachusetts Institute of Technology 2021. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
January 15, 2021

Certified by. . . . .
Anish Athalye
Doctoral Candidate
Thesis Supervisor

Certified by. . . . .
M. Frans Kaashoek
Charles Piper Professor
Thesis Supervisor

Certified by. . . . .
Nickolai Zeldovich
Professor
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Kronos: Verifying leak-free reset for a system-on-chip with multiple clock domains

by

Noah Moroze

Submitted to the Department of Electrical Engineering and Computer Science
on January 15, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Notary [3] uses formal verification to prove a hardware-level security property called deterministic start for a simple system-on-chip (SoC). Deterministic start requires that an SoC's state is fully reset by boot code to ensure that secrets cannot leak across reset boundaries. However, Notary's approach has several limitations. Its security property requires that all of the SoC's microarchitectural state can be reset to known values through software, and the property and proof technique apply only to SoCs with a single clock domain. These limitations prevent Notary's approach from being applied to more complex systems.

This thesis addresses these limitations through Kronos, a system consisting of a verified SoC that satisfies a new security property called output determinism. Output determinism provides the same security guarantees as Notary without requiring that all of an SoC's state be reset by software. The SoC used in Kronos, called MicroTitan, is based on the open-source OpenTitan [16] and includes multiple clock domains. This thesis evaluates Kronos and demonstrates that existing open-source hardware can be modified to satisfy output determinism with minimal changes, and that the process of proving output determinism reveals hardware issues that violate desired security guarantees.

Thesis Supervisor: Anish Athalye
Title: Doctoral Candidate

Thesis Supervisor: M. Frans Kaashoek
Title: Charles Piper Professor

Thesis Supervisor: Nickolai Zeldovich
Title: Professor

Acknowledgments

Thank you to Anish Athalye for his invaluable mentorship throughout this project. His research laid the groundwork for this thesis, and his guidance over the course of the project was fundamental to its success. Thank you to Professor Frans Kaashoek and Professor Nickolai Zeldovich, who, along with Anish, frequently provided insightful technical advice, feedback, and support.

Thank you to my friends, who were a source of learning, laughter, and support throughout my time in school. They made the experience truly special, and I look forward to these friendships continuing on beyond graduation. Finally, thank you to my family for their unconditional love and support, and particularly my parents who have always encouraged me to pursue my interests to the fullest extent.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Formal verification allows researchers and engineers to prove that a system's implementation maintains certain security properties. Formal verification of software typicallly uses an ideal instruction set architecture (ISA) model, but verifying security properties under an ISA model doesn't rule out vulnerabilities in hardware below the level of the ISA [19]. However, such vulnerabilities do exist. For example, the widely-publicized exploits Spectre [11] and Meltdown [14] are based on vulnerabilities at the microarchitectural level, the low-level implementation of a processor's ISA. Both attacks use side channels in microarchitectural state to leak confidential data. In order to make claims about such attacks, formal verification needs to encompass microarchitectural details too. Towards the goal of extending formal verification to include hardware, this thesis presents Kronos, a system with a hardware-level security property proven using formal verification. This chapter explains the security goals Kronos aims to achieve, the challenges this thesis addresses, a list of this thesis's contributions, and an outline for the rest of the thesis.

1.1 Background

This thesis builds off Notary [3], prior work that developed a system with a security property formally verified at the hardware level. Notary is a secure transaction approval device that supports running multiple software "agents" sequentially on the same SoC (system-on-a-chip, a system that integrates a CPU, memory, I/O, and other peripherals), while guaranteeing *noninterference*, meaning that the execution of one agent on the SoC cannot influence the execution of agents that run subsequently on the same SoC. Noninterference implies that confidential data from one agent cannot leak to another: if an agent operates on secret data, and an agent running later can access that data in some way, this would constitute the first agent influencing the execution of the second.

In order to guarantee noninterference, Notary uses formal verification to prove that the system satisfies a property called *deterministic start*. If a system (which is comprised of an SoC and its boot code) satisfies deterministic start, all of its microarchitectural state is cleared to known, deterministic values by that boot code running after reset. Since microarchitectural state encompasses all channels through which information could leak between agents, ensuring deterministic start before any agent runs also ensures noninterference.

Notary proves deterministic start for a small SoC built around the PicoRV32 [25] RISC-V processor core, a simple, unpipelined processor design with few peripherals and a simple interconnect system for the processor to interface with those peripherals. The PicoRV32 was adequate for running simple agents in the Notary prototype, but using a faster, more complex SoC could enable a Notary-like design to support more demanding applications, such as agents that require a higher performance CPU or more complex peripherals.

An example of an SoC that is a step up from the PicoRV32 in terms of complexity is OpenTitan [16], an open-source SoC being developed for use as a hardware root-of-trust, a security critical application. OpenTitan features a two-stage pipelined processor, yielding higher performance, and a large set of peripherals for I/O and encryption. Proving that this SoC can be used in a system that satisfies noninterference could enable a secure Notary-like architecture to be implemented on a more capable hardware platform, expanding the range of possible applications, while maintaining strong security guarantees.

1.2 Goal and challenges

The goal of this thesis is to prove a security property for MicroTitan, a custom subset of OpenTitan hardware, so that it can be used in a Notary-like system that guarantees noninterference. We assume a threat model similar to the one used in Notary, described in more detail in Section 2.2. Although the aim of this thesis is to prove a security property that achieves the same overall security goals as Notary, proving such a property for more complex hardware introduces new challenges that must be addressed.

While Notary relies on resetting all of the PicoRV32's state to guarantee noninterference, OpenTitan's microarchitectural state cannot be fully reset to known values, so an approach based solely on resetting all state will not suffice. Although resetting all microarchitectural state is sufficient to ensure that an SoC can provide noninterference, it is not necessary. For example, a hardware-implemented FIFO queue could store data in memory that is not initialized on reset, and this would be safe as long as the FIFO never outputs uninitialized data (e.g. if it outputs zero when empty). This thesis addresses the challenge of formally defining a security property that is strong enough such that an SoC that satisfies the property can be used in a system that provides noninterference, while not requiring that the SoC's state be entirely resettable.

A second challenge is that Notary's security property and its formal verification approach apply only to circuits with a single clock source, while OpenTitan's SPI and USB peripherals use separate clocks from the main processor. Therefore, the security property and the proof technique used in this thesis must be able to handle circuits with multiple clock domains. It's important to address this challenge since a Notary-like system gives untrusted software direct access to peripherals without going through a kernel, and therefore all peripherals must satisfy the security property required to achieve noninterference.

These are not the only challenges in proving security properties for an SoC like OpenTitan. However, this thesis focuses on addressing these ones in particular, which we believe are associated with common design patterns that would be found in other systems. For instance, it seems likely that other systems may implement I/O with FIFO memories left uninitialized after reset and written by an external source rather than software, requiring a relaxed property that allows for some uninitialized state. Peripherals that implement communication protocols like USB, which by specification requires a 48 MHz clock, most likely require their own separate clock domain.

In order to manage the scope of this thesis, Kronos uses a custom subset of the OpenTitan hardware we call MicroTitan. MicroTitan integrates the OpenTitan's Ibex processor with a ROM, RAM, and a subset of OpenTitan communication peripherals which are representative of these challenges: the UART, SPI, and USB peripherals.

1.3 Thesis contributions

This thesis makes several contributions:

1. A definition of a security property, *output determinism*, that provides the same security guarantees as Notary's deterministic start but does not require an SoC's state to be fully reset.

2. An approach, called *modular output determinism*, for proving output determinism for a digital circuit with multiple clock domains, which consists of:

   (a) A set of properties that are mechanically checked for a particular circuit using formal verification.

   (b) A sketch of a paper proof demonstrating that these mechanically checked properties imply that the circuit satisfies output determinism.

3. An implementation of modular output determinism for a subset of the OpenTitan hardware we call MicroTitan. This implementation comprises a system called Kronos, the components of which are illustrated in Figure 1-1. This implementation is open-source, and can be found on Github[1].

[1]https://github.com/nmoroze/kronos

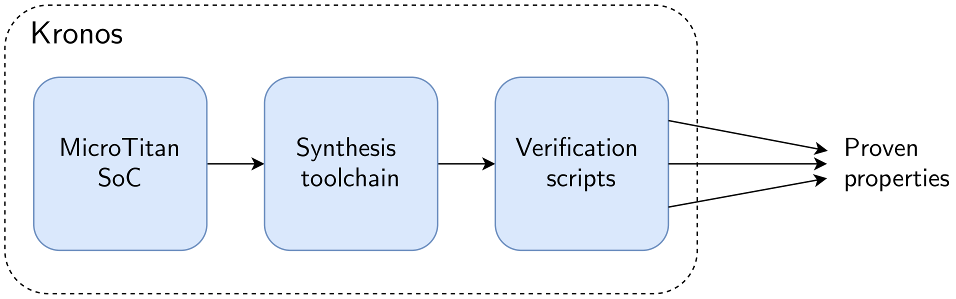


Figure 1-1: Overview of Kronos, the system presented by this thesis.

4. An evaluation of Kronos, which shows the following:

   (a) The performance of the formal verification implementation demonstrates that modular output determinism is computationally feasible.

   (b) OpenTitan hardware requires changes to satisfy output determinism, but the impact of these changes in terms of circuit size is minimal.

   (c) Proving output determinism catches potential security bugs.

1.4 Thesis outline

The rest of this thesis is outlined as follows. Chapter 2 discusses relevant background. Chapter 3 defines the output determinism security property. Chapters 4 and 5 discuss an approach for proving output determinism for a circuit with multiple clock domains. Chapters 6 and 7 describe how Kronos implements this approach for MicroTitan. Chapter 8 evaluates Kronos based on the questions above. Chapter 9 discusses related work. Finally, Chapter 10 concludes and presents ideas for future work.

Chapter 2

Background

2.1 Deterministic start

Notary's deterministic start resets a hardware system's complete microarchitectural state to known, deterministic values. This enables Notary's goal of allowing two software agents to run sequentially on the same hardware with complete isolation, guaranteeing noninterference.

The key idea behind noninterference, and the reason it is a desirable security property, is that two agents running in a hardware system may be mutually distrusting and have secrets that they want to remain confidential. For example, consider two agents A and B where agent A is meant to sign and approve cryptocurrency transactions with a secret key. A user may execute agent A to sign a transaction, then execute agent B to perform some other task. Agent A will perform the necessary cryptographic signature computations, which will likely leave the secret key in bits of the hardware's state such as RAM, or microarchitectural state such as a cache.

If this state is not cleared before agent B runs, then a malicous agent B might be able extract the secret key and send it to an adversary. This could occur either through obvious channels, such as reading it directly out of memory, or more subtle side channels, such as cache timing data. The possibility of side channels in particular motivates the use of formal verification to ensure that *all* microarchitectural state is cleared between executions. If all of this state is cleared, there is no way for agent B

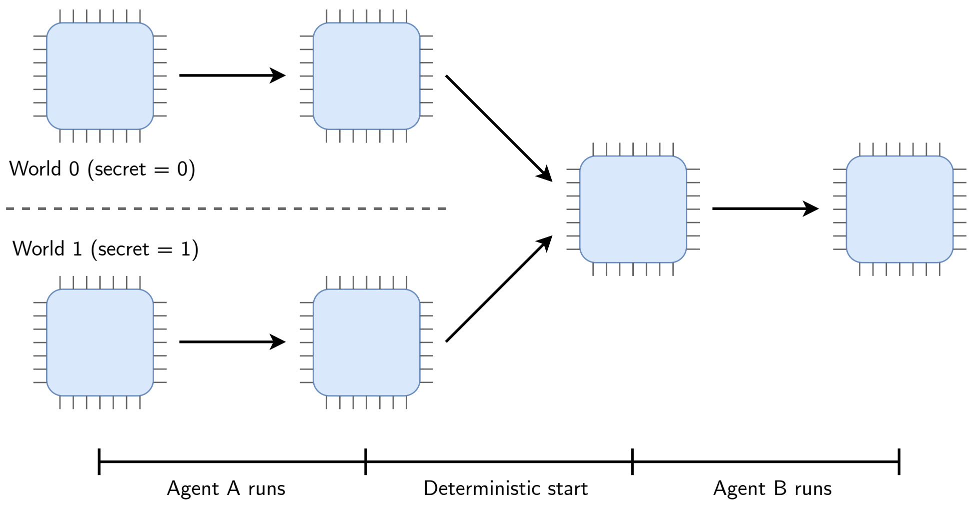


Figure 2-1: Deterministic start ensures noninterference between two agents (figure adapted from Notary [3]).

to extract any secrets that may be left over from agent A.

A strawman solution to this problem might be to reset the SoC between executing different agents using the hardware reset line. However, one of Notary's insights is that in real processor designs, simply asserting the hardware reset line does not reset all microarchitectural state. This is because running reset lines to every register is both expensive (in terms of silicon area), and unnecessary for functional correctness. Embedded systems often have boot code that runs after reset that ensures that "architectural state", i.e. state specified by a processor's ISA such as registers and memories, do get overwritten with initial values on reset. However, this does not ensure that microarchitectural state gets cleared.

In order to achieve a high degree of security, Notary extends this idea of initialization boot code through a concept called "software-assisted deterministic start." Software-assisted deterministic start involves writing boot code which is guaranteed to put the processor in a *fully* deterministic state, including architectural and microarchitectural state, when that code is executed after reset.

Proving deterministic start entails both writing this initialization code, and then using formal verification to prove that all microarchitectural state becomes deterministic after the initialization code runs, regardless of the system's starting state.

Although it's not possible to reset all microarchitectural state in a processor such as the OpenTitan, it's clear why this approach prevents secrets from leaking across reset boundaries, and the ideas behind deterministic start inform the approach used for proving output determinism.

2.2 Threat model

Kronos's goal is to prevent an attacker from exfiltrating any secret data that was part of the SoC state prior to the most recent reset. Kronos assumes an attacker can run arbitrary code on the SoC after reset, with full software access to all hardware peripherals. It also assumes the attacker has full access to the SoC's input and output pins, allowing the attacker to snoop on output or send arbitrary inputs to the SoC.

Except for microarchitectural side channels, Kronos's threat model does not include arbitrary side channels [29] such as electromagnetic radiation [2], power analysis [18], and acoustic analysis [10], the same limitations as Notary. Also like Notary, Kronos does not focus on physical attacks.

This threat model is strong enough for MicroTitan to be used in a transaction approval system as implemented in Notary, with the same security guarantees. Considering all I/O untrustworthy captures Notary's assumption that an externally connected device can be compromised or malicious, and allowing the attacker to run arbitrary code with hardware access captures Notary's assumption that software agents with full hardware access may be compromised or malicious.

2.3 Symbolic execution

Notary's formal verification technique, which this thesis adopts and extends for proving output determinism, is based on *symbolic execution*. Symbolic execution lets a developer reason about a system's execution efficiently while considering many possible states.

One key concept in symbolic execution is the *symbolic value*, which is a named

```verilog
module example(
  input        clk,
  input        rst_n,

  input [7:0]  in,
  input        in_valid,

  output [7:0] out
  );

  reg [7:0] data;
  reg       data_valid;

  always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      // set data_valid to zero on reset, but don't reset data
      data_valid <= 1'b0;
    end else if (in_valid) begin
      data_valid <= 1'b1;
      data <= in;
    end
  end

  assign out = data_valid ? data : 8'b0;

endmodule
```

Figure 2-2: Verilog code for symbolic execution example.

value that itself represents all possible values of an associated datatype. For example, a symbolic 2-bit value called `foo` simultaneously represents the values 0, 1, 2, or 3. *Symbolic expressions* are mathematical expressions based on symbolic values. For example, the symbolic expression `foo & 2`, which incorporates the previous value `foo`, could potentially be the value 0 or 2.

Kronos uses the Rosette [23] solver-aided programming library for symbolic execution in the Racket programming language. Rosette lifts many Racket constructs for symbolic execution and supports rich symbolic expressions including bitvectors, vectors of values, arithmetic, conditional operations (`ite` or "if-then-else"), bitvector concatenation/slicing, and more.

To illustrate how symbolic execution applies to an example circuit, consider the Verilog snippet in Figure 2-2. Suppose this circuit has just been reset, so the values stored in its registers are:

- `data_valid` = `1'b0`
- `data` = `data$0`

`data$0` is an 8-bit symbolic value, which represents that `data` is uninitialized on reset, and could contain any possible value from 0 to 255.

Consider what happens when this circuit undergoes a single clock step. To model all possible inputs, we represent the current value of the inputs `in` and `in_valid` with the symbolic values `in$0` and `in_valid$0` respectively. After the step, the registers have the following values:

- `data_valid` = `(ite in_valid$0 1'b1 1'b0)` = `in_valid$0`
- `data` = `(ite in_valid$0 in$0 data$0)`

Both registers now contain symbolic expressions based on the conditional in the Verilog `always` block. Note that `data_valid`'s value can be simplified from a conditional down to just `in_valid$0`. Rosette is able to perform basic simplifications like this using a feature called rewrite rules.

On the other hand, `data` must remain as a conditional symbolic expression. The meaning of the expression is that if `in_valid$0` is `1'b1`, then `data` takes on the symbolic value `in$0`, otherwise it takes on the symbolic value `data$0`.

The value of the output `out` can now be expressed as the symbolic expression `(ite in_valid$0 (ite in_valid$0 in$0 data$0) 8'b0)`. This symbolic expression contains two nested conditionals, with three leaf values: `in$0`, `data$0`, and the constant `8'b0`. However, close analysis of the expression reveals that it can evaluate to only two possible values: `in$0` or `8'b0`. This is because for the second nested conditional to evaluate to `data$0`, `in_valid$0` would have to be equal to `1'b0`. However, if `in_valid$0` is equal to `1'b0`, then the whole expression would just evaluate to `8'b0` since the top-level conditional is based off `in_valid$0` as well. Therefore, `data$0` is not a possible value of this expression.

Although it's easy to determine the possible values of a simple expression like this just by looking at it, the expressions dealt with in real circuits may be much larger and more complex. Therefore, Rosette relies on SMT solvers like Z3 [7] to prove statements about the expressions. In the above example, a developer could use Rosette to verify a statement such as `assert out != data$0`. Rosette would call into an SMT solver, which would analyze the expression to determine that `out` can never be equal to `data$0`.

Notary uses this representation of circuit state as symbolic values and expressions to prove deterministic start. Just like in this example, the state of all registers uninitialized by the hardware reset line are initialized as symbolic values. This models all possible values that may have been left behind in the SoC. After executing boot code using symbolic execution, an SMT solver query proves that no register values may depend on these initial starting symbolic values. By considering all possible starting states in one execution, symbolic execution eliminates the need to exhaustively check each starting state one at a time, which would not be computationally feasible for anything but the simplest circuits.

Symbolic execution is often used for verifying software, but Notary and Kronos apply it to the execution of a cycle-accurate software model of the hardware circuit that implements an SoC.

2.4 OpenTitan

The OpenTitan [16] system-on-chip is an open source hardware system that's an appealing target for our work for several reasons:

- It's based on the Ibex [15] processor core, a design that's a step up from Notary's PicoRV32 in complexity with a two-stage pipeline and support for machine and user privilege modes. This enables better performance and the ability to use this core in a system that requires multiple privilege levels, such as allowing agents to take advantage of M/U mode separation or physical memory protection in a Notary-like setting.

- OpenTitan has a large suite of modular peripherals (such as co-processors for encryption and hardware support for multiple communication protocols) that make it more practical to integrate in a larger real-world system.

- OpenTitan is designed for use as a secure silicon root-of-trust in data centers, meaning that its intended use case inherently requires a high degree of security.

OpenTitan's Ibex processor implements the RISC-V ISA, just like the PicoRV32. Using RISC-V allows us to take advantage of its large ecosystem of tools and open-source processor designs.

2.4.1 Verification subset

The entire OpenTitan SoC is large—it consists of about 20,000 flip-flops compared to about 1,300 for the PicoRV32 SoC analyzed in Notary. Working with such a large device poses performance challenges for formal verification, and various components of the SoC introduce their own inherent challenges. This thesis focuses on proving properties about a more manageable subset called *MicroTitan*, which consists of about 4,300 flip-flops (around three times the size of PicoRV32, and around one-fifth the size of OpenTitan). Figure 2-3 shows a block diagram of this subset, with the clock domains used in each peripheral highlighted.

MicroTitan includes the Ibex processor, ROM, and RAM, which are the bare minimum required for executing code. Additionally, it includes three I/O peripherals: UART, device-side SPI, and device-side USB. The UART is the simplest of the three, and is clocked by the same clock as the processor. Verifying the UART acts as a simple baseline example of a single clock domain communication peripheral. The SPI and USB peripherals are both implemented in multiple clock domains, and serve as two different examples of how to apply the multiple clock domain verification technique developed by this thesis. OpenTitan is still under active development; this thesis uses the implementation as of commit `97b60c1` in the OpenTitan Git repository.

Since this thesis considers a subset of the OpenTitan's peripherals, we had to implement our own top-level Verilog module to wire them up into one SoC. We carefully constructed our top-level module implementation by copying the relevant sections of OpenTitan's top-level module to make MicroTitan as representative as possible. We use the same crossbar implementation as OpenTitan for connecting the

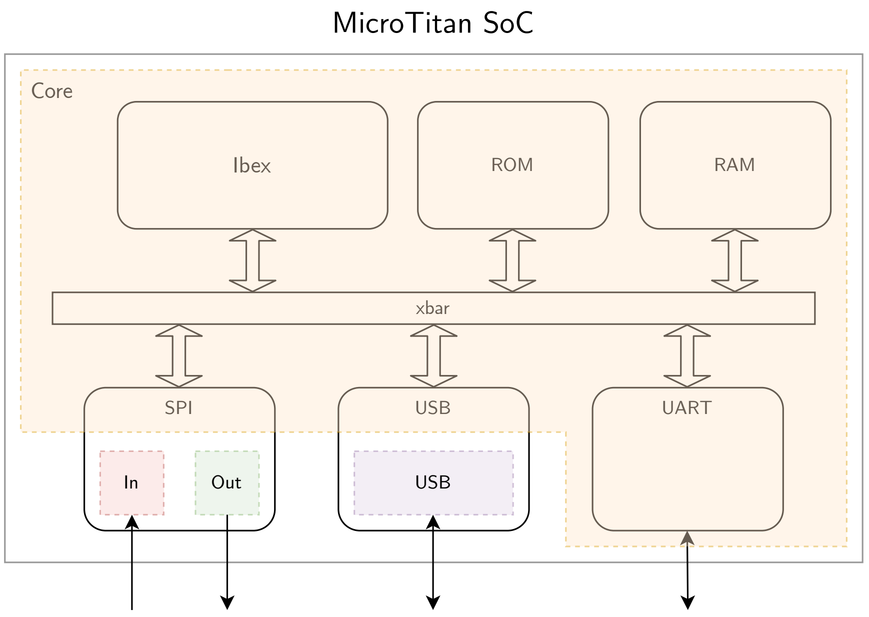


Figure 2-3: Block diagram of MicroTitan, Kronos's verified OpenTitan subset. Each of the four clock domains is highlighted. Note that each peripheral that uses a clock domain other than the core clock domain includes hardware in both. In addition, note that the SPI peripheral includes two independent clock domains, one for handling input and the other for output.

processor and peripherals.

During the process of verification, we discovered that some of the OpenTitan peripherals used in MicroTitan violate output determinism. In those cases, we implemented minimal patches in order to fix them. These patches are discussed in detail in Section 6.1.2, Section 8.2, and Section 8.3.

Chapter 3

Output determinism

This chapter formalizes the property *output determinism*, which can be used to prove a system provides noninterference without requiring an SoC's state be fully cleared. Section 3.1 begins by discussing the intuition behind achieving noninterference without resetting all state, and provides a prose definition of output determinism. Next, Section 3.2 defines concepts and notation that allow us to reason formally about the MicroTitan circuit. Finally, Section 3.3 uses these concepts to formally define output determinism.

3.1 Noninterference without state clearing

Notary guarantees noninterference between software agents by proving the deterministic start property. In order for an SoC to satisfy deterministic start, it must be possible to reset the entire state of the SoC to deterministic values by executing code on its processor. However, guaranteeing noninterference solely by clearing state does not work for OpenTitan, which has state that cannot be reset by code.

One example is OpenTitan's SPI input FIFO buffer, which stores data in memory that's not initialized on reset. There is no way of writing to this buffer from software—if an external SPI host does not write to it, then the buffer will contain the same data it did prior to reset. A general security property doesn't assume any particular behavior on the part of an external host, so we can't assume it must write data at any point. Therefore, we cannot guarantee that the data in this buffer is reset after executing any number of cycles.

However, just because this memory cannot be reset doesn't mean that it can leak uninitialized data. Considering this FIFO example, the hardware could be designed such that when the FIFO is empty, the FIFO data output is hardwired to zero. That way, no hardware modules that use this FIFO will ever be able to extract uninitialized data out of the FIFO memory. If the FIFO ever becomes non-empty, it should have had new data written to its memory that will become visible on the outputs. If the hardware is designed not to leak uninitialized data, it could effectively provide noninterference because a malicious agent will not be able to access any data left behind from a previous execution in a meaningful way.

This example begs the question of what exactly it means for uninitialized data to be "leaked". For example, does it count as leakage if uninitialized data is able to flow from one register to another? This thesis claims that in order to ensure security, the only thing that must not happen is for uninitialized data to influence *observable outputs* of the circuit. If uninitialized data cannot affect outputs, an attacker cannot exfiltrate secrets.

Output determinism is based off this claim: a circuit's state may safely contain secret data as long as externally observable behavior is not dependent on the secret data. If a circuit satisfies this property, it means that a circuit's outputs must be fully determined by its inputs. The definition of output determinism below formalizes this intuition.

Definition 3.1.1 (Output determinism)**.** If a circuit satisfies output determinism, then the output of the circuit at any step during execution may only depend on the inputs the circuit has received after reset up to that step. The outputs cannot depend on the circuit's state prior to reset.

A circuit that satisfies output determinism provides protection from directly leaking secrets and against timing side-channels, since any observable timing difference as a result of secret data would still mean that output values on any given cycle are

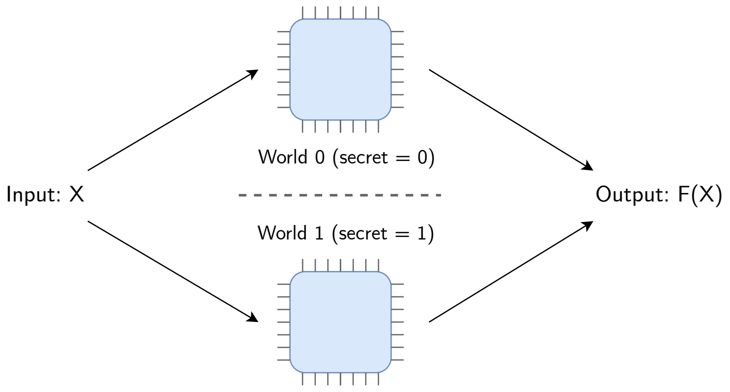


Figure 3-1: Circuits may safely contain secrets as long as the circuit functions as a "black box" where outputs are dependent solely on inputs, not secrets. Contrast this with the diagram of deterministic start, Figure 2-1.

not solely dependent on past inputs. This further means that a circuit that satisfies output determinism can be used in a system that guarantees noninterference, because state left over from the execution of a previous agent cannot influence the execution of subsequent agents in any observable way.

Although output determinism does not require resetting circuit state like deterministic start, it's useful to note that a circuit with its state entirely reset to deterministic values will satisfy output determinism.

This is the case since a circuit's outputs can always be determined by its input and its state. Therefore, the outputs of a circuit with deterministic state can be fully determined solely by its inputs. In addition, a circuit's new state after a clock step can be fully determined by its previous state and its inputs. Therefore, once a circuit has been put into a deterministic state, its state will be fully determined by its inputs from then on, meaning it satisfies output determinism from that point on.

As discussed in the FIFO example above, a circuit's state may not be fully resettable, and yet it may still satisfy output determinism. The Verilog code in Figure 3-2 shows a concrete example of a circuit where this is the case (this is the same code from the symbolic execution example in Section 2.3). The register `data` is not initialized on reset, and is written only when an external input `in_valid` is set to true. Therefore, we can't say for certain that `data` will be reset to a deterministic state after executing for a known number of cycles. However, this circuit does satisfy output determinism,

```verilog
module example(
  input        clk,
  input        rst_n,

  input  [7:0]  in,
  input        in_valid,

  output [7:0] out
  );

  reg [7:0] data;
  reg       data_valid;

  always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      // set data_valid to zero on reset, but don't reset data
      data_valid <= 1'b0;
    end else if (in_valid) begin
      data_valid <= 1'b1;
      data <= in;
    end
  end

  assign out = data_valid ? data : 8'b0;

endmodule
```

Figure 3-2: Verilog code for circuit with state that is uninitialized on reset, but does satisfy ouptut determinism.

since its single top-level output `out` is fully determined by past inputs on every step. To see that this is the case, consider the two possible values `out` may have depending on the value of `data_valid`. If `data_valid` is `1'b1`, `out` will be equal to `data`. Although `data` is uninitialized on reset, and thus starts out with an unknown value, its value will be overwritten by the `in` input at the same time `data_valid` becomes `1'b1` (since they both change in the same block). If `data_valid` is `1'b0`, then `out` is set to `8'b0`, which is a known value. Although `data` contains uninitialized data while `data_valid` is `0`, it does not matter since its value never reaches an output.

3.2 Formal reasoning about circuits

This section describes a mathematical model for reasoning about the digital circuit that implements the MicroTitan. This model consists of the circuit's state, top-level inputs and outputs, and step function.

MicroTitan is an SoC implemented by a stateful digital circuit comprised of combinational logic and registers. This thesis uses the term "register" to refer to all storage elements, including memories. Each register belongs to one or more *clock domains.*

The circuit's *state* is the set of all values stored in its registers. S is the set of all possible circuit states.

The circuit has some number of input and output wires, referred to as *top-level* inputs and outputs. I is the set of all possible top-level inputs, and O is the set of all possible top-level outputs.

A *step function* describes the behavior of the circuit. The step function takes in an initial circuit state, the values of all top-level inputs, a single boolean *reset* input, and a boolean *clock* input for each clock domain in the circuit. The function returns the new circuit state reached by stepping the initial state on the given input.

$\hat{C}$ is the set of all possible clock and reset inputs. The step function is written as `step`: $S \to I \to \hat{C} \to S$. $\hat{C}$ can also be decomposed into the set of all clock inputs C and a separate reset input like so: $\hat{C} = \{\text{rst=0}, \text{rst=1}\} \times C$.

The *output function* `out`: $S \to I \to O$ takes in a circuit state, the most recent input, and returns a set of outputs.

A register is "in a clock domain" if that register's value may change when the step function is called with the clock input corresponding to that domain set to true. Note that a register that's updated on the positive edge of a clock signal is in a different clock domain from a register that's updated on the negative edge of the same clock signal, since these two registers are not updated at the same timestep. This overapproximates the behavior of the circuit, since it does not consider the fact that these two clock domains are related. Also note that this model allows the circuit to have registers in multiple clock domains, which supports reasoning about memories with multiple write ports that are clocked in different domains.

The circuit has a single reset input that may reset any register in the circuit (as opposed to one reset per clock domain). Therefore, when the reset input is true, the step function will return a state where each register is set to its circuit-defined reset value if it has one. Any register that does not have a defined reset value will be stepped as usual based on the clock inputs. This reset is *asynchronous*, meaning that stepping with reset asserted will reset registers in all clock domains regardless of the clock inputs.

3.2.1 Example circuit

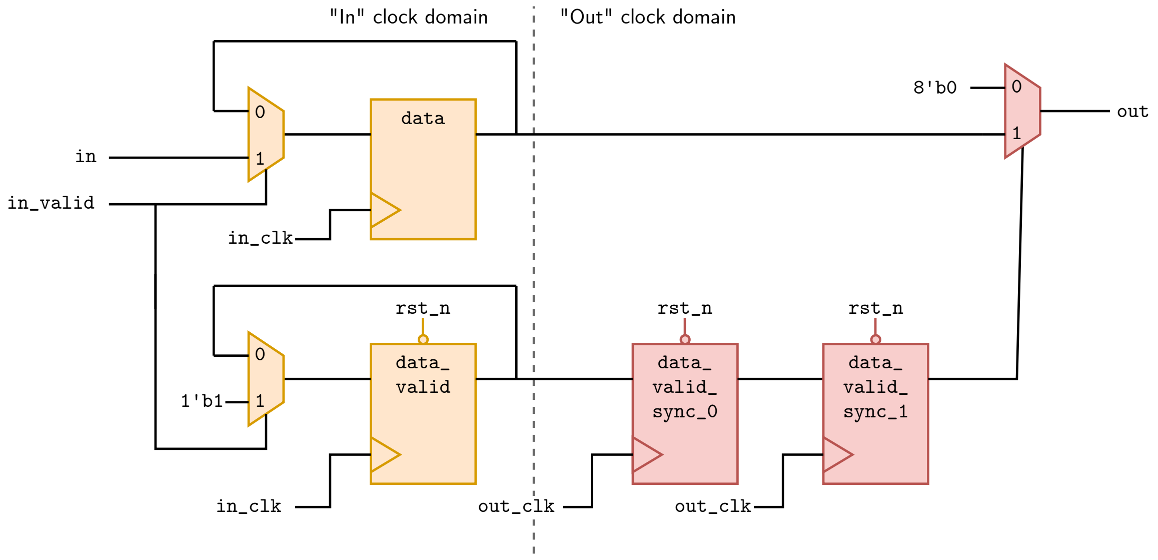
To show how this model maps to a concrete example of a simple circuit, Figure 3-3 provides an example circuit in Verilog that extends the circuit provided in Figure 3-2. In this version of the circuit, the `data` and `data_valid` registers are driven by a clock input `in_clk`, while `out` is intended to be read from a separate clock domain driven by `out_clk`. Therefore, the `data_valid` signal is synchronized through two registers in the output clock domain, `data_valid_sync_0` and `data_valid_sync_1`.

The model categorizes each component of this circuit:

- The circuit's state consists of the values of each of its registers: `data`, `data_valid`, `data_valid_sync_0`, and `data_valid_sync_1`.
- `in_clk` and `out_clk` correspond to the "in" and "out" clock domain inputs, respectively. `rst_n` is the global asynchronous reset.
- The top-level inputs to this circuit are `in` and `in_valid`.
- The sole top-level output of this circuit is `out`.
- `data` and `data_valid` are in the "in" clock domain, since their values are only updated on the positive edge of `in_clk`.
- `data_valid_sync_0` and `data_valid_sync_1` are in the "out" clock domain.

```verilog
module example_multiclk(
  input         in_clk,
  input         out_clk,
  input         rst_n,

  input [7:0]   in,
  input         in_valid,

  output [7:0]  out
);
  reg [7:0] data;
  reg       data_valid;
  reg       data_valid_sync_0;
  reg       data_valid_sync_1;

  always @(posedge in_clk or negedge rst_n) begin
    if (!rst_n) begin
      // set data_valid to zero on reset, but don't reset data
      data_valid <= 1'b0;
    end else if (in_valid) begin
      data_valid <= 1'b1;
      data <= in;
    end
  end

  always @(posedge out_clk or negedge rst_n) begin
    if (!rst_n) begin
      data_valid_sync_0 <= 1'b0;
      data_valid_sync_1 <= 1'b0;
    end else begin
      data_valid_sync_0 <= data_valid;
      data_valid_sync_1 <= data_valid_sync_0;
    end
  end

  assign out = data_valid_sync_1 ? data : 8'b0;

endmodule
```

(a) Verilog code



(b) Circuit diagram

Figure 3-3: Example clock domain crossing circuit.

3.3 Formal definition of output determinism

We want to prove that MicroTitan satisfies output determinism. Output determinism means that for any given sequence of inputs, the SoC output trace after executing an arbitrary number of cycles should be a deterministic function of its input trace after reset. Definition 3.3.1 below is a formal definition of this property.

First, we define the helper function $\texttt{run}\colon \forall n, S \to I^n \to C^n \to S$ which describes n cycles of execution with a sequence of inputs while reset is not held:

$$\begin{aligned}
\texttt{run}\Big(s,\ [],\ []\Big) &= s\\
\texttt{run}\Big(s,\ i :: \vec{\imath},\ c :: \vec{c}\Big) &= \texttt{run}\left(\texttt{step}(s, i, (c, \text{rst=0})), \vec{\imath}, \vec{c}\right)
\end{aligned}$$

This function is then used to formally define output determinism as follows:

Definition 3.3.1 (Output determinism)**.**

$$\begin{aligned}
&\forall i_0 \in I,\ c_0 \in C,\ \exists \Omega \in O,\ \forall s, s_0 \in S, && \text{(Base case)}\\
&s_0 = \texttt{step}(s, i_0, (c_0, \text{rst=1})) \implies \texttt{out}(s_0, i_0) = \Omega
\end{aligned}$$

$$\begin{aligned}
&\forall n \in \mathbb{N}, \vec{\imath} \in I^n,\ \vec{c} \in C^n, \exists \Omega \in O, \forall s, s_0, s_n \in S, && \text{(Inductive step)}\\
&s_0 = \texttt{step}(s, \vec{\imath}\,[0], (\vec{c}\,[0], \text{rst=1})) \implies\\
&s_n = \texttt{run}(s_0, \vec{\imath}\,[1:], \vec{c}\,[1:]) \implies\\
&\texttt{out}(s_n, \vec{\imath}\,[-1]) = \Omega
\end{aligned}$$

This property is expressed recursively. The base case shows that on the initial step where the hardware reset line is asserted, the circuit outputs must be equal to a value Ω that only depends on the circuit's input and clock signals on that step, i_0 and c_0. Importantly, Ω must not depend on the circuit's starting state s, which is denoted by the order of the quantifiers.

The recursive step shows that after executing the circuit for n cycles starting from its post-reset state s_0, its most recent output $\texttt{out}(s_n, i)$ must be equal to a value Ω that only depends on the input and clock signal trace for the past n cycles. Just like in the base case, the order of the quantifiers indicates that Ω may only be dependent on these inputs, and may *not* depend on the starting state s. Since this property holds for all n, and Ω may depend on n as well, this step illustrates that our output on *every* cycle after reset must be determined by the past input and clock signal trace.

Chapter 4

Modular output determinism

This chapter describes an approach for proving output determinism for a circuit with multiple clock domains. One challenge in coming up with such an approach is that the clocks do not step in a fixed, constant ratio (such as “clock domain A steps 2 times for every 1 step of clock domain B”). The OpenTitan SPI peripheral contains logic driven by a clock that can be started and stopped arbitrarily by an external host (i.e., it is not constant), and it is not required to run at a particular frequency (i.e., the ratio is not fixed). This makes it impossible to reason precisely about an execution of the entire circuit as a whole, since there are effectively infinite possible traces of clock inputs.

The approach described in this chapter, called modular output determinism, addresses this challenge by dividing up MicroTitan’s circuit into multiple logical “subcircuits” comprising of the registers in each clock domain, and then using formal verification to prove properties about each individual subcircuit separately. A proof sketch shows that the properties proven about each subcircuit together imply a top-level property about MicroTitan as a whole.

Figure 4-1 shows MicroTitan broken up into separate subcircuits for each clock domain. Note that each subcircuit has its own top-level inputs and/or outputs that communicate externally, labelled “top” in the figure. However, in addition, we can consider the inputs and outputs of each subcircuit to also include the signals that communicate between clock domains, which are labelled “CDC” (clock domain cross-

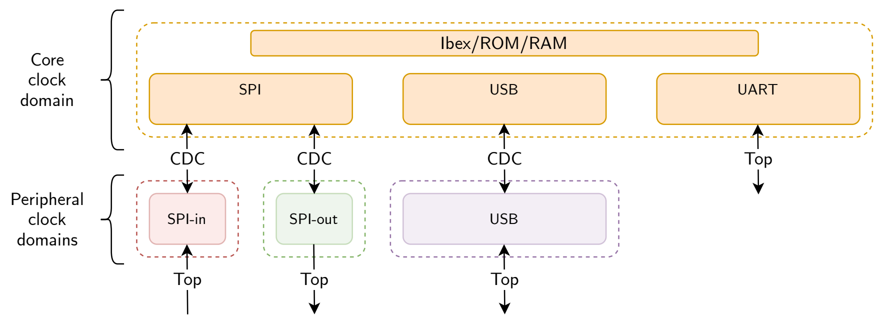


Figure 4-1: MicroTitan split into clock domain subcircuits.

ing) in the figure. The key idea behind modular output determinism is to prove that each subcircuit in MicroTitan satisfies an output determinism-like property, considering both CDC and top-level inputs and outputs. That is, for each cycle after reset, each clock domain's CDC and top-level outputs must be fully determined by its CDC and top-level inputs.

The intuition behind this technique is that if no secrets leak across clock domain crossing boundaries, then it's okay for top-level outputs to depend on data that's in clock domain crossing boundaries. Therefore, each subcircuit needs to have the data that flows between clock domain crossing boundaries verified along with the data that flows in and out of the circuit as a whole.

In order to express this idea more precisely, Section 4.1 defines concepts necessary to discuss the input and output boundaries of clock domain subcircuits. Section 4.2 defines the machine-verified modular output determinism subproperties that must be proven for each clock domain, and Section 4.3 provides a sketch of a proof that these subproperties imply top-level output determinism.

4.1 Model of multiple clock domains

Modular output determinism takes advantage of the fact that MicroTitan's circuit has a particular clock domain topology in order to categorize each clock domain in one of two categories. A single *core* clock domain communicates with three *peripheral* clock domains, with the key restriction that the peripheral clock domains cannot communicate with each other. The core clock domain is the SoC's primary clock domain, which includes the Ibex CPU, ROM, RAM, UART, and communication hardware between all peripherals. The peripheral clock domains are part of the SPI and USB peripherals. Figure 4-1 points out the two categories of clock domains.

4.1.1 Clock domain crossing

The concept of "clock domain crossing" (CDC) registers formalizes the input/output boundaries between clock domains.

Definition (CDC register)**.** Suppose we have a circuit with clock domains A and B. A *CDC register* from A to B is a register that:

- is in clock domain A but not in clock domain B.
- has an output that drives a register in clock domain B (directly or through combinational logic with other registers in any clock domain).

Note that this definition does not consider registers in multiple clock domains to be CDC registers. These cases use special logic that isn't captured by general reasoning about CDC registers. In the case of MicroTitan, the one instance of this is the dual-port memory in the USB peripheral. Section 7.2.1 and Section 7.3.2 discuss how this is handled in the implementation.

Also note that a register can drive registers in multiple clock domains and still be considered a CDC register. For example, a register in clock domain A that drives a register in clock domain A and clock domain B is still considered a CDC register from A to B.

To see how this definition applies to a concrete example, Figure 4-2 shows a snippet of the circuit diagram for Figure 3-3. In this circuit, `data_valid` is a CDC register from the "in" to "out" clock domains.

4.1.2 Outputs

Modular output determinism requires associating each top-level output with a par-

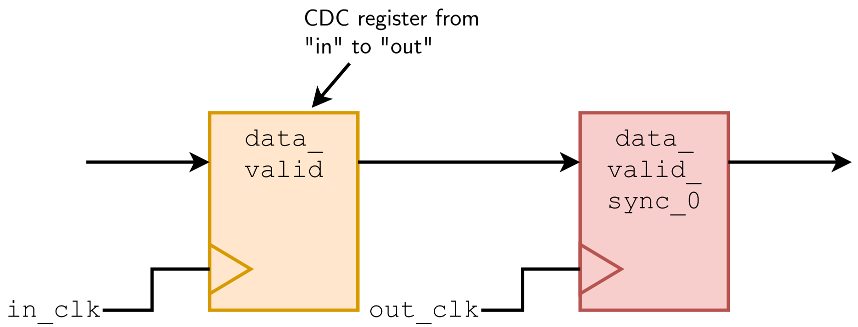


Figure 4-2: Diagram illustrating the definition of a CDC register.

ticular subcircuit. The most straightforward way to do this would be to associate a top-level output with a certain clock domain if that output is entirely derived from registers in that clock domain. For the most part, this is how MicroTitan's top-level outputs are categorized by Kronos.

However, this straightforward categorization does not always apply. Consider the circuit in Figure 3-3. The output `out` is combinationally dependent on registers in two different clock domains. This scenario also comes up in MicroTitan. The `miso_o` output from the SPI peripheral directly depends on registers from the SPI-out and core clock domains.

In order to handle these cases, modular output determinism does not consider the categorization of top-level outputs as inherently defined by the circuit. Rather, it is left to be chosen by a proof developer as a matter of proof strategy. In fact, subcircuit top-level outputs don't even have to correspond with the circuit's top-level outputs. Modular output determinism's only requirement is that the top-level outputs of each subcircuit (together with the circuit's top-level inputs) fully determine the top-level outputs of the circuit itself.

Suppose $\texttt{out}_{CD}$ represents the top-level outputs from each subcircuit CD for $CD \in \{\text{core}, \text{spi-in}, \text{spi-out}, \text{usb}\}$, given the current circuit state and inputs. Another way to describe this requirement is it must be possible to determine an output combination function ω such that for any given state s and set of inputs i:

$$\texttt{out}(s,i) = \omega(\texttt{out}_{\text{core}}(s,i), \texttt{out}_{\text{spi-in}}(s,i), \texttt{out}_{\text{spi-out}}(s,i), \texttt{out}_{\text{usb}}(s,i), i)$$

ω may rely directly on i since top-level outputs may be driven by combinational logic of the top-level inputs directly.

The distinction between each set of outputs relates solely to the verification approach. Modular output determinism requires the outputs from each clock domain subcircuit only depend on the CDC and top-level inputs to that clock domain. As long as it is possible to verify this, and an ω function can be found, then a given categorizaton of outputs is okay. If these requirements cannot be met, a new categorization of outputs must be chosen. If no output categorization meets these requirements, then modular output determinism cannot be applied to the circuit.

In Figure 3-3, since the output `out` is driven by two registers each in a different clock domain, `data_valid_sync_1` and `data`, each of those registers could be considered a top-level output of their respective clock domain (even though they don't correspond to circuit top-level outputs themselves). Then the value of `out` could be captured by the function ω like so (assuming Ω represents outputs from the "in" clock domain, and Θ represents outputs from the "out" clock domain):

$$\omega(\Omega, \Theta, i) = \begin{cases} 0, & \text{if } \Theta_{\texttt{data\_valid\_sync\_1}} = 0 \\ \Omega_{\texttt{data}}, & \text{otherwise} \end{cases}$$

The output categorization is the only concept described in this section that is decided by the developer. The clock domain subcircuits, the categorization of clock domains as *core* or *peripheral*, and which registers are CDC registers are inherently described by the circuit.

4.2 Machine-verified subproperties

In order to prove top-level output determinism, modular output determinism requires proving a subproperty called *core output determinism* for the circuit's core clock domain, and a subproperty called *peripheral output determinism* for each peripheral clock domain. These subproperties are similar to output determinism, but with the

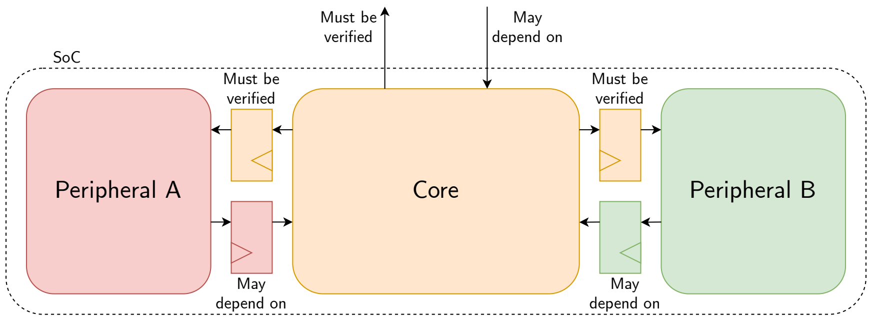


Figure 4-3: Core output determinism.

key difference that they include CDC registers as part of the input and output boundaries. This section defines these two subproperties.

4.2.1 Core output determinism

The first property is called core output determinism. It is defined as follows.

Definition 4.2.1 (Core output determinism)**.** If a circuit satisfies core output determinism, then on every step the top-level outputs of the core clock domain subcircuit and the CDC register values from the core clock domain to every peripheral clock domain may depend only on the circuit's top-level inputs and CDC register values from any peripheral clock domain. However, immediately after the step on which reset is asserted, the top-level outputs and CDC registers from the core clock domain may depend on top-level inputs *only*, not CDC register values.

Figure 4-3 provides an example of what proving core output determinism entails. All outputs of the core clock domain subcircuit, including top-level outputs and CDC register values flowing out of the subcircuit, must be verified, and are allowed to depend on top-level inputs and CDC regiser values flowing into the subcircuit. However, just like regular output determinism the outputs may not depend on uninitialized data left over in the core clock domain.

Note that Definition 4.2.1 specifies that outputs may not depend on CDC register values on reset—this is important for the soundness of modular output determinism.

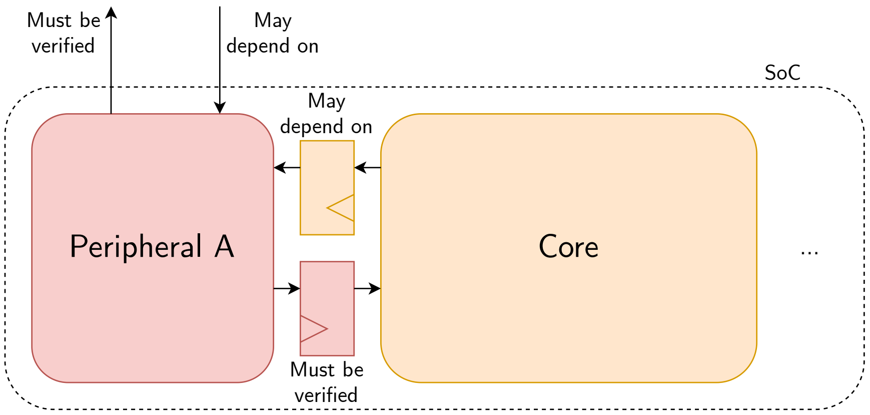


Figure 4-4: Peripheral output determinism.

4.2.2 Peripheral output determinism

The second property is peripheral output determinism. This property is similar to core output determinism. The main distinction is that this property refers to *individual* peripheral clock domains which communicate with the core clock domain, whereas core output determinism refers to CDC communication between the core and *all* peripheral clock domains.

Definition 4.2.2 (Peripheral output determinism for P)**.** If a circuit satisfies peripheral output determinism for some clock domain P, then on every step the top-level outputs of P's subcircuit and the CDC register values from P to the core clock domain may depend only on the circuit's top-level inputs and CDC register values from the core clock domain to P. However, immediately after the step on which reset is asserted, the top-level outputs and CDC registers from P may depend on top-level inputs *only*, not CDC register values.

Modular output determinism requires that peripheral output determinism be proven individually for each peripheral clock domain, hence the definition of the property being parameterized by P.

Figure 4-4 illustrates what proving peripheral output determinism entails. All outputs of the peripheral clock domain subcircuit, including top-level outputs and CDC register values must be verified, and are allowed to depend on top-level inputs and CDC regiser values flowing into the subcircuit. However, just like regular output determinism the outputs may not depend on uninitialized data left over in the core clock domain. Since peripheral output determinism only deals with one particular peripheral clock domain at a time, only one peripheral clock domain is depicted here.

4.3 Paper proof of top-level claim

Modular output determinism claims that proving core output determinism and proving peripheral output determinism for each peripheral clock domain implies output determinism for the entire circuit. This section presents a high-level argument for why this claim is true.

4.3.1 Notation

This subsection introduces additional notation to succinctly describe several concepts. $[C \underset{CDC}{\rightarrow} P]_n$ represents the values of CDC registers from the core clock domain to some peripheral clock domain P on cycle n. Analagously, $[P \underset{CDC}{\rightarrow} C]_n$ represents the values of CDC registers from some peripheral clock domain P to the core clock domain on cycle n. $[C \underset{CDC}{\rightarrow} P*]_n$ represents the values of CDC registers from the core clock domain to *all* peripheral clock domains on cycle n. Analagously, $[P* \underset{CDC}{\rightarrow} C]_n$ represents the values of CDC registers from *all* peripheral clock domains to the core clock domain on cycle n.

All of the above values are indexed assuming $n = 0$ represents the register values corresponding to the state immediately *after* the circuit is stepped with reset asserted.

Let i_n repesent the circuit's top level inputs on cycle n. In this case, treat the index such that $n = 0$ is the input to the circuit *when* it is stepped with reset asserted.

4.3.2 Proof sketch

One subtle point that makes it difficult to see why the two subproperties imply output determinism is that it may seem like there's a circular dependency between properties 4.2.1 and 4.2.2. Core output determinism assumes that CDC registers from peripheral clock domains are "safe" and allows CDC registers from the core clock domain to depend on them, while peripheral output determinism assumes that CDC registers from the core clock domains are "safe" and allows CDC registers from the peripheral clock domains to depend on them.

However, there is no circular dependency issue. On the very first cycle when the circuit is reset, the resulting CDC register values are not allowed to depend on any previous CDC register values, only top-level inputs. At any given cycle after that, the allowed CDC register dependencies from other clock domains are the values from the *previous* step, which have already been proven safe. This intuition is captured in the proof of this following lemma, which is ultimately used to prove that the two modular output determinism subproperties prove output determinism.

Lemma 1. *If a circuit satisfies property 4.2.1, and satisfies property 4.2.2 for all of its peripheral clock domains, then all of the circuit's CDC register values at every cycle n only depend on the top-level inputs the circuit receives between reset and cycle n, i.e. $i_0 \dots i_n$.*

Proof. We prove this lemma by strong induction.

Base case. First, consider the cycle when the circuit is reset (cycle 0). By the definition of property 4.2.1 we can see that $[C \underset{CDC}{\rightarrow} P*]_0$ only depends on i_0. By the definition of property 4.2.2, we can see that for every peripheral clock domain P, $[P \underset{CDC}{\rightarrow} C]_0$ only depends on i_0. Therefore, all CDC register values can be fully determined by the top-level inputs on cycle 0.

Inductive step. Assume lemma 1 holds on every cycle k such that $0 \leq k < n$. We must show that lemma 1 holds for cycle n.

By property 4.2.1 we know that $[C \underset{CDC}{\rightarrow} P*]_n$ only depends on $i_0 \dots i_n$ and $[P* \underset{CDC}{\rightarrow} C]_0 \dots [P* \underset{CDC}{\rightarrow} C]_{n-1}$. However, since we assume lemma 1 holds for all $k < n$, we know that $[P* \underset{CDC}{\rightarrow} C]_0 \dots [P* \underset{CDC}{\rightarrow} C]_{n-1}$ must only depend on top level inputs. Therefore, $[C \underset{CDC}{\rightarrow} P*]_n$ itself only depends on top-level inputs.

By property 4.2.2 we know that for each peripheral clock domain P, $[P \underset{CDC}{\rightarrow} C]_n$ only depends on $i_0 \ldots i_n$ and $[C \underset{CDC}{\rightarrow} P]_0 \ldots [C \underset{CDC}{\rightarrow} P]_{n-1}$. We can use a symmetrical argument to the one above: since we assume lemma 1 holds for all $k < n$, we know that for every peripheral clock domain P that $[P \underset{CDC}{\rightarrow} C]_0 \ldots [P \underset{CDC}{\rightarrow} C]_{n-1}$ must only depend on top-level inputs, and therefore $[P \underset{CDC}{\rightarrow} C]_n$ only depends on top-level inputs.

Since both $[C \underset{CDC}{\rightarrow} P*]_n$ and $[P \underset{CDC}{\rightarrow} C]_n$ for every P only depend on top-level inputs, all CDC register values only depend on top-level inputs for cycle n, and lemma 1 holds. □

Now, we use the result in lemma 1 to prove the following theorem.

Theorem 1. *If a circuit satisfies property 4.2.1, and satisfies property 4.2.2 for all of its peripheral clock domains, then that circuit satisfies output determinism as specified in definition 3.1.1.*

Proof. By property 4.2.1, we know that the top-level outputs of the core clock domain subcircuit may only depend on top-level inputs and CDC register values. By lemma 1, we know that CDC register values themselves may only depend on top-level inputs. Therefore, the top-level outputs of the core clock domain may only depend on top-level inputs.

Applying the same logic as above, property 4.2.2 and lemma 1 also show the top-level outputs of every peripheral clock domain only depend on top-level inputs.

Therefore, the top-level outputs of every subcircuit in our circuit only depend on top-level inputs.

A final observation is that the top-level outputs of a circuit can be determined by the top-level outputs of each subcircuit, as well as the top-level inputs to a circuit. This is true by definition, since as discussed in Section 4.1.2 it must be possible to find a function ω such that this is the case. This means that since the top-level outputs of each subcircuit only depend on top-level inputs, the top-level outputs of the circuit itself also only depend on top-level inputs. Therefore, the circuit itself must satisfy output determinism. □

Chapter 5

Machine verification approach

This chapter describes Kronos's approach for verifying property 4.2.1 and verifying property 4.2.2 for every peripheral clock domain in MicroTitan. Section 5.1 discusses a high-level method for reasoning about individual clock domains in the presence of other clock domains, and Section 5.2 and Section 5.3 go into detail about how core output determinism and peripheral output determinism are verified, respectively.

5.1 Modeling individual clock domains

Modular output determinism requires proving properties about individual clock domains. However, since clock domains interact through clock domain crossing, this requires a way to correctly model the execution of a particular clock domain in the presence of others.

Imagine reasoning about a particular clock domain, referred to as the "reference" clock. One way to reason about arbitrary clock signals is to consider that for every step of the reference clock, other clocks may step zero or more times. This captures all possible behaviors of the circuit with respect to its clock inputs, since it allows for any ratio of clock frequencies (including for either clock not to step at all), and it does not require the ratio of clock steps to be fixed throughout the circuit's execution.

Stepping a single reference clock domain on a given set of inputs can be thought of as calling the circuit step function with those same inputs but with only the corresponding clock input set to true. For example, a step function for the core clock domain, $\texttt{step}_C \colon S \to I \to \{0,1\} \to S$ can be expressed as:

$$\texttt{step}_C(s,i,r) = \texttt{step}(s,i,\{\text{core} = 1, \text{spi-in} = 0, \text{spi-out} = 0, \text{usb} = 0, \text{rst=r}\})$$

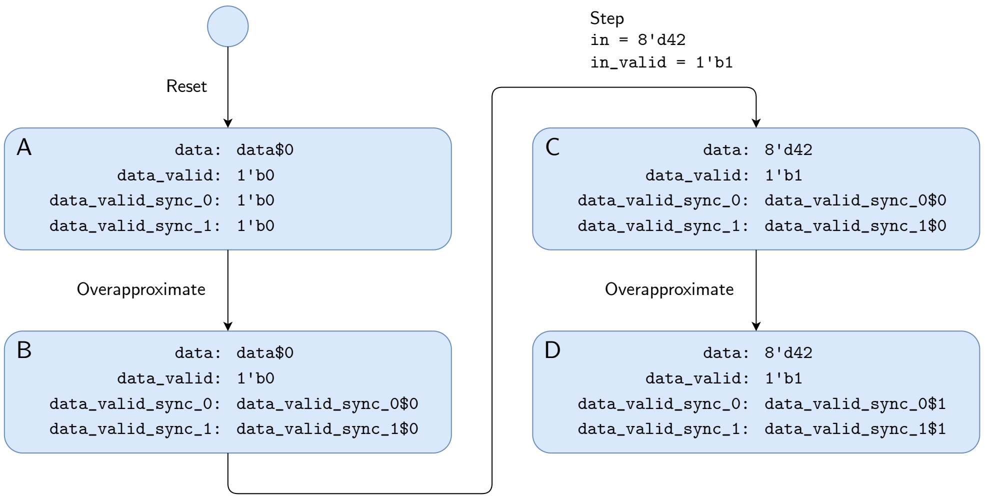
Symbolic execution can't directly model an unbounded number of clock steps in between steps of our reference clock. However, it can *overapproximate* the set of possible states after performing one reference clock step. Overapproximation entails setting all registers in non-reference clock domains to completely unconstrained symbolic values. This captures the full set of possible behaviors since other clock steps may only modify the state in non-reference clock domains, and overapproximating makes no attempt at constraining how these values are modified.

Therefore, the behavior of a single reference clock domain in the presence of multiple clock domains can be modelled by calling the corresponding step function, overapproximating the result, stepping the result of the overapproximation, overapproximating the new result, and so on for any number of steps.

Figure 5-1 provides an example of how the state of the circuit in Figure 3-3 changes when stepped from the perspective of the "in" clock domain. It begins with reset, after which all initialized registers have concrete values, and the uninitialized register `data` has some symbolic value (Figure 5-1 box A).

After reset, `out_clk` may step zero or more cycles before `in_clk` steps even once, since it may be arbitrarily faster or slower than `in_clk`. Therefore, the state of both `data_valid_sync_*` registers are unknown before `in_clk` steps. By analyzing the circuit, we can see that their value cannot be anything other than `1'b0`. However, this would not always be clear—for example, consider what would happen if this clock domain had a cycle counter register. This register could contain any possible value. Therefore, instead of trying to consider exactly what possible states could be in other clock domains, they are overapproximated entirely by setting the registers to fresh symbolic values (Figure 5-1 box B).

Figure 5-1 box C then shows the outcome of stepping the circuit with a pair of



Step
in = 8'd42
in_valid = 1'b1
Reset
A
data: data$0
data_valid: 1'b0
data_valid_sync_0: 1'b0
data_valid_sync_1: 1'b0
Overapproximate
B
data: data$0
data_valid: 1'b0
data_valid_sync_0: data_valid_sync_0$0
data_valid_sync_1: data_valid_sync_1$0
C
data: 8'd42
data_valid: 1'b1
data_valid_sync_0: data_valid_sync_0$0
data_valid_sync_1: data_valid_sync_1$0
Overapproximate
D
data: 8'd42
data_valid: 1'b1
data_valid_sync_0: data_valid_sync_0$1
data_valid_sync_1: data_valid_sync_1$1

Figure 5-1: Example of stepping a circuit in one clock domain while overapproximating the other.

concrete inputs, `in_valid` = `1'b1`, and `in` = `8'd42`. Despite the "in" clock domain now being full of concrete values, we still can't say anything about the possible values of the "out" clock domain, which may once again step for zero or more cycles. So, it must be overapproximated again with fresh values, giving the final state shown in Figure 5-1 box D.

By overapproximating in this manner, we end up considering a superset of possible circuit states (hence the term "overapproximation"). However, this is fine for the soundness of a proof: if a property holds for a superset of the possible circuit states, then it must also hold for the subset of states that are actually possible.

5.2 Core output determinism

To prove core output determinism, Kronos takes advantage of the fact that it's possible to reason about code executing in the core clock domain, since it includes the actual processor component of the SoC. This allows Kronos to use a "software-assisted" approach similar to Notary, where it reasons about boot code execting on the processor after reset.

One key insight is that if a circuit ends up in a deterministic state, its outputs will be determined solely by its inputs from that point on. A circuit's new state after a step is determined solely by its previous state and input. Therefore, after a circuit reaches a deterministic state, all future states that circuit reaches can be determined solely by its inputs. In addition, a circuit's output on a given cycle is determined by its state and input on that cycle. Putting this together, the circuit's outputs can be solely determined by its inputs on every cycle after the circuit reaches a deterministic state.

Kronos borrows Notary's software-assisted deterministic start approach to use boot code to put MicroTitan's core clock domain in a state where all registers only depend on past inputs and conclude that its output will only depend on its inputs for all future cycles. However, this is insufficient to fully imply core output determinism, since the property requires that a circuit's outputs be safe for *all* cycles after the reset line is deasserted (including the cycles during which boot code executes). To account for this, Kronos adds "output checking" to deterministic start, which entails verifying that the output on every cycle of boot code execution depends only on past input. This is a straightforward addition since deterministic start is implemented by symbolically executing boot code on the SoC, allowing Kronos to reason about all possible outputs during boot code execution.

With this extra output checking, deterministic start shows that outputs only depend on inputs on every cycle during boot code execution, and implies this is the case for every cycle after boot code execution, which together implies this property for all cycles. Therefore, deterministic start with output checking is sufficient to imply core output determinism.

Kronos primarily uses software assisted deterministic start with output checking to prove core output determinism. However, it's impossible to reset the entirety of MicroTitan's state in the core clock domain solely using boot code. Luckily, the remaining uninitialized state does not affect outputs. This is demonstrated using a separate proof, discussed further in Section 7.2.3. The rest of this section will focus on the algorithm used to verify that the boot code resets as much state as it can.

Pseudocode for software assisted deterministic start with output checking is shown in Figure 5-2. The algorithm symbolically executes the circuit model, initializing it

```
state = new_symbolic_state()

# allowed dependencies initially empty set
allowed_dependencies = {}

# step state once with hardware reset line asserted
state = step_core(state, reset=True)

while True:
  # new inputs include top-level and CDC registers from
  # peripherals to core
  inputs = new_symbolic_inputs()
  allowed_dependencies.add(inputs)

  state = step_core(state, inputs)
  state = overapprox_peripherals(state)

  # check outputs
  outputs = out_core(state, inputs)
  if not only_depends_on(outputs, allowed_dependencies):
    fail()

  # check state
  if only_depends_on(state, allowed_dependencies)
    # done!
    success()
  else
    # output example, helps determine what state needs to be reset
    ...
    continue
```

Figure 5-2: Pseudocode for verifying deterministic start with output checking, which implies output determinism for the core clock domain.

with an entirely symbolic state, and stepping it one clock cycle at a time. At each step, a set of symbolic top-level inputs and symbolic CDC register values are applied, in order to model the fact that the circuit may have arbitrary top-level or clock domain crossing inputs. Although symbolic, these values are added to a set of "allowed dependencies," since outputs may depend on them.

On each cycle, the algorithm performs two checks. First, the core clock domain state is checked to see if all elements of the state (not including the non-resettable state mentioned above) only depend on allowed dependencies. If so, the circuit has achieved deterministic start, and the algorithm returns success. If not, the script will continue stepping.

Second, the script queries an SMT solver on each cycle to ensure that all outputs (including core clock domain top-level and CDC output registers) are only dependent on values in the list of “allowed dependencies”. If they’re not, the script will terminate early, indicating a failure.

Although it does not cause the script to terminate, when the first check fails the script will output an example of a part of the state that differs and give two of its possible values. This allows a developer to go back and find a way to add or modify initialization code to try and reset this piece of state as well. They can then re-run the verification script and repeat this process to iteratively build up boot code until it resets all uninitialized state.

5.3 Peripheral output determinism

Kronos’s approach for proving peripheral output determinism is distinct from its approach for proving core output determinism. Since software does not execute directly within peripheral clock domains, peripheral output determinism must generally be satisfied *inherently* by peripheral hardware. In general, Kronos proves this property for each clock domain by proving a property we call *output equivalence* between the subcircuit in question and a version of the subcircuit initialized with fully deterministic state. If two circuits satisfy output equivalence, it means that starting from their initial states, they will have the same output on every cycle given the same trace of inputs. As discussed previously, the version of the circuit with deterministic starting state is guaranteed to have outputs that can be fully determined by its inputs. Therefore, proving output equivalence between these subcircuits is sufficent to prove that the original circuit’s outputs are also solely dependent on its inputs. If the proven set of outputs include both the subcircuit’s top-level outputs and its CDC register outputs, then this proves peripheral output determinism.

```
impl = step_peripheral(new_symbolic_state(), rst=True)
spec = make_deterministic_copy(impl)

assert(related(impl, spec))
assert(outputs(impl) == outputs(spec))
```

(a) Pseudocode for base case.

```
impl = new_symbolic_state()
spec = new_symbolic_state()
assume(related(impl, spec))
assume(outputs(impl) == outputs(spec))

inputs = new_symbolic_inputs()

impl' = step_peripheral(impl, inputs)
spec' = step_peripheral(spec, inputs)
assert(related(impl', spec'))
assert(outputs(impl) == outputs(spec))
```

(b) Pseudocode for inductive step.

Figure 5-3: Pseudocode for verifying output determinism for a peripheral clock domain.

Kronos uses an inductive technique called forward simulation to prove output equivalence. The algorithm used is shown in pseudocode in Figure 5-3. The two instances of the circuit to be compared are called the “impl” (implementation) and “spec” (specification) instances. The impl instance is initialized as usual—an initially fully symbolic state is stepped with the reset line asserted. The spec instance is copied from the impl instance but with all registers that are uninitialized on reset set to zero. Therefore, the spec state is entirely deterministic.

Proving output equivalence entails proving that given any two related impl and spec states, their top-level outputs are equivalent. What it means for two states to be “related” is up to the proof author, and is expressed using a refinement relation. A refinement relation is a function that takes in two state instances and returns a boolean stating whether the states are related or not. An example refinement relation between the implementation of a FIFO and a deterministic specification is the following: two FIFO states are related if all their register values are equal, and

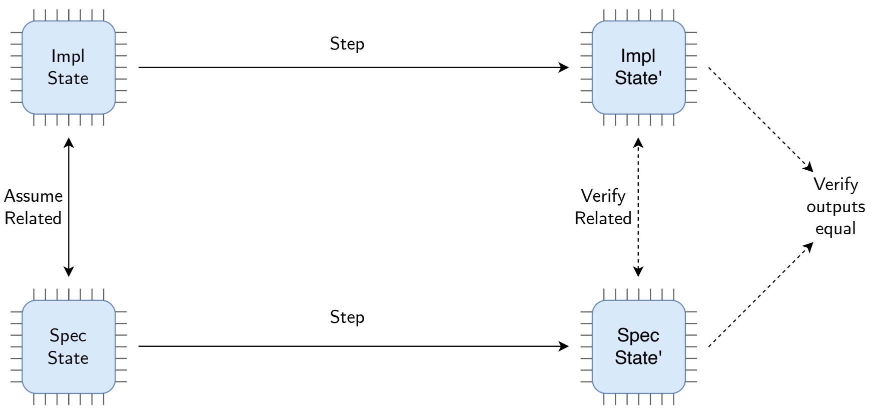


Figure 5-4: Relation diagram showing output equivalence inductive step. Note that in traditional forward simulation, the step functions applied to the impl and spec states are different. However, since the impl and step states are instances of the same system with no distinction besides their initial state, they use the same step function.

the *valid* memory values are equal (i.e., only memory values between the read and write pointer must be equal).

The machine-verified proof of output equivalence is based on the principle of induction. First, a base case is proven that shows the initial impl and spec states are related and that their outputs are equal. The inductive step initializes a fully symbolic spec and impl state, steps each on symbolic inputs, then proves that the post-step impl and spec states are related and have equivalent outputs, assuming that the initial symbolic states were also related and had equivalent outputs. This step is illustrated in Figure 5-4.

Chapter 6

SoC and toolchain implementation

This chapter describes the implementation of the MicroTitan SoC and the toolchain used for synthesizing it into executable Racket code. The other part of Kronos's implementation, the machine verification code, is discussed in Chapter 7.

6.1 MicroTitan

MicroTitan is the custom subset of OpenTitan hardware that we verify. This subset consists of the following components (refer back to Figure 2-3 for a block diagram):

- An Ibex CPU
- 8KB of ROM
- 8KB of RAM
- A UART peripheral
- A device-side SPI peripheral
- A device-side USB peripheral

In order to make MicroTitan representative of a realistic design, we carefully implemented it to be as similar to OpenTitan as possible.

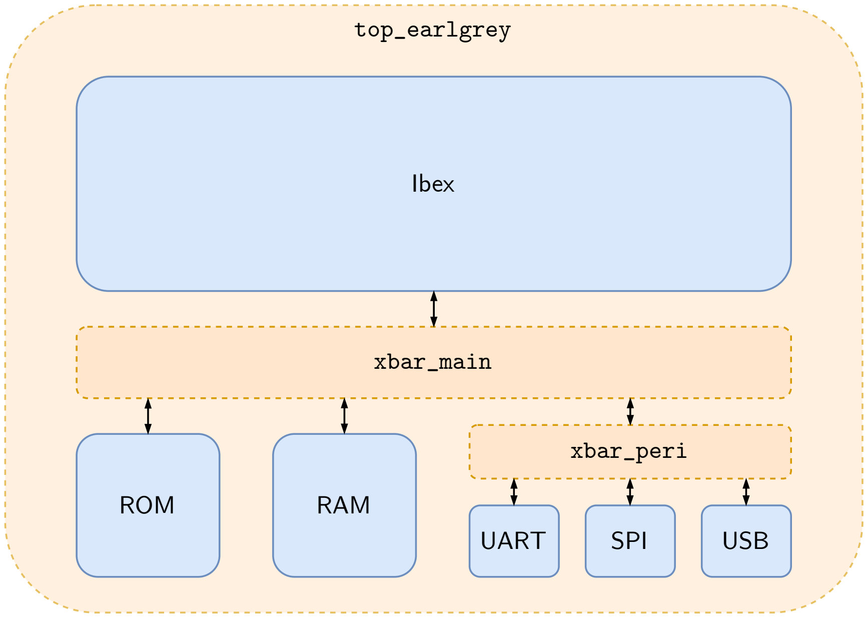


Figure 6-1: Implementation of MicroTitan. Custom components are shaded in orange with a dotted outline, parts which come directly from OpenTitan are shaded in blue with a solid outline.

Each OpenTitan component is modular, making it easy to implement a subset without modifying any of the individual components themselves. The only custom RTL in MicroTitan are three modules that replace equivalents in OpenTitan. These modules are called `top_earlgrey`, `xbar_main`, and `xbar_peri`, and consist of the top-level logic for integrating and wiring up all the components. When implementing these modules, we took care to make them representative of their OpenTitan equivalents—they were written by copying directly from OpenTitan, and then deleting and modifying lines as needed to wire up the subset of peripherals chosen.

OpenTitan and consequently MicroTitan are implemented in the SystemVerilog hardware description language. The code is based off OpenTitan commit `97b60c1`.

Figure 6-1 shows a block diagram of the MicroTitan implementation, with custom components distinguished from components directly taken from OpenTitan. Each custom component in this diagram corresponds to a single SystemVerilog modules, while the OpenTitan components are each comprised of multiple SystemVerilog modules.

Although MicroTitan is based directly off OpenTitan code, it does include sev-

	Component	LoC
Custom modules	`top_earlgrey`	317
	`xbar_main`	170
	`xbar_peri`	56
Modifications	Sync FIFO	+8
	Async FIFO	+13
	SPI RX order	+15
	USB memory	+30
	USB reset sync	+10

Table 6.1: Amount of custom code in MicroTitan.

eral small patches, which each fall into one of two categories. The first are changes necessary to make the hardware work with our toolchain. These are described in Section 6.1.1. The second set of changes are required to make MicroTitan satisfy output determinism, as this property is not met by the OpenTitan peripherals as is. These changes are described in Section 6.1.2.

Table 6.1 shows how many lines of SystemVerilog code are in each custom module, and how many new lines of code are needed for each modification needed to satisfy output determinism. The modifications for the toolchain are not shown—these were all simple modifications with no more than 10 lines changed for each of them.

6.1.1 Modifications for toolchain

Clock gating

OpenTitan includes a “clock gating” module that enables or disables a given clock based on a control signal. Clock gating is used in the Ibex CPU to turn off the core clock to the majority of its components in order to implement a low-power sleep state. Supporting clock gating would introduce additional complexity to Kronos’s toolchain and reasoning about multiple clock domains, and it is not important for functional correctness. Therefore, we choose to eliminate clock gating entirely in MicroTitan. This modification is made within the clock gating module by ignoring the clock enable input, and passing through the clock directly.

ROM hack

MicroTitan adds a hack to OpenTitan's ROM module that does not affect functionality but causes Kronos's toolchain to extract it in a way that's easier to work with.

Synchronous FIFO memory declaration

MicroTitan switches the SystemVerilog memory declaration style used in synchronous FIFOs from packed to unpacked. This change does not affect behavior but allows Kronos's synthesis toolchain to produce a more efficient representation of the circuit.

SPI latches

OpenTitan's SPI peripheral includes a few latches, which are unsupported by Kronos's toolchain. This was unintentional in OpenTitan, so MicroTitan includes a cherry-picked fix from a commit past its OpenTitan baseline.

6.1.2 Modifications for output determinism

MicroTitan includes several changes to OpenTitan hardware in order to ensure that it satisfies output determinism. This section describes those changes. The impact of these changes on hardware resource utilization is described in Section 8.2, and the security implications of *not* including these changes is discussed in Section 8.3. These modifications have been tested for correctness by incorporating them back into the OpenTitan, running an existing OpenTitan simulation on example software that uses the UART, SPI, and USB peripherals, and manually checking that the simulation behaves as expected.

Synchronous FIFO

The synchronous FIFO primitive was modified to set its output to zero when the FIFO is empty. Before making this change, the FIFO would output whatever element of its storage was being pointed at by its read pointer, even if that element was invalid. This meant, for example, that on reset a FIFO's data output would be set to some uninitialized value. Now, a FIFO will always output zero after reset until it receives new data.

This change is unnecessary for MicroTitan to achieve output determinism. However, it's included in MicroTitan for several reasons. First is that this change was upstreamed into OpenTitan itself, so it's now a canonical part of OpenTitan (although in a commit ahead of our baseline, requiring us to cherry-pick it into MicroTitan). The second reason is that this change simplifies reasoning in a few cases, since it allows us to make simplifying assumptions about the behavior of the synchronous FIFO (discussed in Section 7.1).

Asynchronous FIFO

The asynchronous FIFO was also modified to prevent uninitialized data from being leaked by its data output on reset. This is necessary to prevent the SPI peripheral from leaking uninitialized data and violating output determinism. However, this type of FIFO cannot be modified the same way as the synchronous FIFO—that would break the functionality of the SPI peripheral, which relies on being able to read possibly invalid data from an empty FIFO.

Instead, the asynchronous FIFO modification entails splitting up the underlying memory of a FIFO of depth N into a register that stores the first element, and a memory of size $N - 1$ that stores the remaining elements. This first register is initialized to zero on reset, which ensures that when the FIFO is reset its data output will be zero instead of the value of some uninitialized memory.

SPI configuration

The SPI peripheral has a configuration register that controls the order that bits are received via SPI (MSB or LSB first). If software toggles this register between every bit that is received, it can prevent a register that is uninitialized on reset from being filled before it becomes software-accessible. This is not intended behavior—a comment in the OpenTitan code say that this configuration register should not be changed during an SPI transaction. However, this is not enforced by hardware.

In order to fix this issue, MicroTitan's SPI peripheral has a modification that stores the value of the SPI RX order configuration register in a new register once it starts receiving a byte via SPI. This new register's value is not updated until the full byte is received, preventing any changes to the configuration register from taking effect mid-transaction.

USB memory read circuit

It's difficult to prove that the USB memory does not leak uninitialized data on an IN endpoint request, since the OpenTitan USB peripheral allows uninitialized data to leak from the memory into several intermediate registers before it is output.

It's possible that timing details of the USB protocol ensure that these intermediate registers are cleared before this data has a chance to reach outputs. However, this is difficult to verify using Kronos's verification approach. In order to ensure that MicroTitan satisfies output determinism, we modify the USB peripheral to prevent these intermediate registers from being updated with new values unless software explicitly writes to one of the configuration registers that indicates there is data available for response to an IN request.

USB reset synchronizer

OpenTitan's top-level module contains hardware that synchronizes the deassertion of the reset input into the USB clock domain, which is important for avoiding metastability issues. However, its design includes a 2-bit register that is uninitialized on reset, and the number of clock cycles the USB peripheral remains in reset for varies based on the initial value of this register. This varying reset behavior affects USB outputs, causing this synchronizer design to violate output determinism.

To fix this problem, MicroTitan switches the USB reset synchronizer to a design that does not include uninitialized registers. It's based on the best practices recommended in [6], and this design appears to be used in more recent iterations of OpenTitan as well.

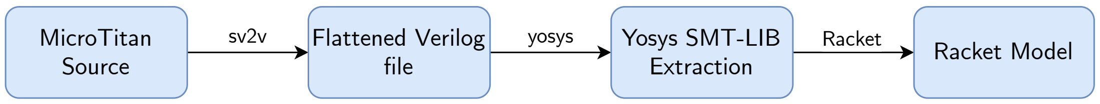


Figure 6-2: A toolchain for extracting Racket circuit model from SystemVerilog source.

6.2 Toolchain

In order to reason about the MicroTitan circuit using symbolic execution, it must be converted into a form that can be executed by the Rosette solver-aided programming language. This section describes the toolchain, shown in Figure 6-2, that's used by Kronos to convert MicroTitan into an executable form.

First, the MicroTitan source is fed into a program called sv2v [22], which translates SystemVerilog code into Verilog. This step is necessary because the next tool in the toolchain, Yosys, does not support SystemVerilog. Yosys [27] is responsible for synthesizing the generated Verilog code, performing optimizations and ultimately outputting it as an SMT-LIB model. SMT-LIB is an S-expression based language with similar syntax and semantics to Racket, making it easy to convert it into valid Racket code using an embedded domain specific language (originally developed for Notary).

The Racket model output by the toolchain is a representation of the circuit including a struct containing its full state combined with inputs, and a step function that encodes how this state transitions on a clock cycle given its inputs.

Additionally, this toolchain can be used with the Yosys `clk2fflogic` pass enabled. The `clk2fflogic` pass enables reasoning about stepping each of the circuit's clock domains individually, which Kronos uses for proving a property about the OpenTitan's asynchronous FIFO design. This proof is discussed further in Section 7.1.

One downside of `clk2fflogic` is that it transforms the circuit model and greatly expands the size of its state, which makes formal verification slower. Without `clk2fflogic`, the Racket model's step function steps all clock domains at once. Although on its own this doesn't accurately model the behavior of the circuit, it is still adequate for reasoning about a single reference clock domain since Kronos's verification code overapproximates all other clock domains on each step. Due to its simplicity and performance advantages, the extraction *without* `clk2fflogic` is used by default, while `clk2fflogic` extraction is used only for the async FIFO proof, where it's needed.

An auxiliary outcome of this project has been contributions back to the open-source tools this toolchain is comprised of. We've discovered and reported several sv2v bugs [1] through the process of trying to apply it to the entire OpenTitan codebase. We've also improved upon the library originally developed for Notary that's responsible for converting Yosy's SMT-LIB model into Racket. These improvements include fixing some performance issues related to scaling up the circuit size as well as usability improvements.

Chapter 7

Machine verification implementation

This chapter describes the implementation of Kronos's code for machine verification. Establishing modular output determinism for MicroTitan requires using machine verification to prove it satisfies four properties: core output determinism, and peripheral output determinism for the SPI-in, SPI-out, and USB clock domains. In addition, Kronos includes auxiliary proofs of output determinism specifically for the FIFO and asynchronous FIFO modules (with the modifications discussed above). The results of these auxiliary proofs are assumed when proving the four main properties in order to simplify verification. Table 7.1 gives the number of lines of Racket code used to implement verification of each property.

Property	Lines of Racket code
Core output determinism	464
Asynchronous FIFO aux. proof	329
Peripheral output determinism (USB)	303
Peripheral output determinism (SPI-in)	130
Peripheral output determinism (SPI-out)	128
Synchronous FIFO aux. proof	108

Table 7.1: Number of lines of Racket code used to verify each property. These counts do not include shared utility code or libraries such as the Racket DSL for executing SMT-LIB circuits.

The following sections describe how the machine verification of each of these properties is implemented.

7.1 FIFO auxiliary proofs

Due to the modifications described in Section 6.1.2, both the OpenTitan's synchronous and asynchronous FIFO designs satisfy output determinism if considered as a standalone circuit. Kronos proves this in separate auxilary proofs. These proofs work by showing output equivalence between a normal instantiation of a FIFO and an instance where its uninitialized memory is reset with all zeros.

By proving output equivalence between these two FIFO instances, Kronos is able to soundly perform a transformation in the MicroTitan circuit where it zeroes the storage of all verified FIFOs on reset. Since FIFOs are used frequently in MicroTitan and proving output determinism generally requires reasoning about the uninitialized data in a design, this transformation greatly simplifies reasoning about MicroTitan.

Since FIFOs are parameterized by their width and depth, this property is proven for each size of FIFO where it's assumed by other proofs. Table 7.2 lists each FIFO verified in MicroTitan.

FIFO	Type	Depth	Width
RAM response FIFO	Sync	2	33
SPI firmware mode arbiter request FIFO	Sync	4	2
Main to peripheral request crossbar FIFO	Async	3	100
USB available buffer FIFO	Async	4	5
USB received data FIFO	Async	4	17
SPI RX & TX FIFOs	Async	8	8

Table 7.2: Which FIFOs are verified by Kronos, including their type (synchronous vs asynchronous) and size parameters.

The async FIFO proof requires the use of `clk2fflogic` as discussed in Section 6.2. This allows Kronos to reason about stepping the FIFO's read and write clock domains individually or together. In order to reason about all possible executions, the inductive step for the async FIFO is verified three times, once for each of the three possible permutations where at least one clock steps.

7.2 Core output determinism

The core clock domain of MicroTitan consists of the Ibex processor, ROM/RAM, UART, and a portion of the SPI and USB peripherals. Verifying core output determinism entails implementing software-assisted deterministic start with output checking, as described in Section 5.2. There are two main implementation challenges that must be addressed based on the specific circuit being verified. The first is determining the boot code necessary to reset uninitialized state. Since this state includes microarchitecture that's not directly modifiable by software, the boot code incorporates some "tricks" to modify state indirectly through side effects. Second is dealing with performance issues that result from unbounded growth of symbolic state. Large symbolic expressions can make stepping the circuit and SMT solver queries very slow. If they get too large, solver queries may even time out without finding a solution. To address performance issues, Kronos uses a set of transformations that can be applied to the circuit that help performance while preserving soundness. These transformations are called "hints". The following two subsections describe how these challenges are addressed in the Kronos implementation.

The final subsection describes an additional machine verification step necessary to prove core output determinism, since the boot code is unable to reset all state in MicroTitan's core clock domain. This proof shows that the state remaining, which may contain uninitialized data, cannot leak to outputs.

7.2.1 Boot code

The MicroTitan boot code is responsible for resetting as much of the core clock domain state as possible. Note that the boot code is not part of our trusted computing base—it is verified to be correct. The final boot code, annotated with comments, is shown in Figure 7-1. Here, we discuss the tricks for each component to reset state.

```
## (constant definitions omitted for space) ##

## Reset SPI (1/2) ##
li x1, SPI_BASE
li x2, 0x30000 # rst rxfifo and txfifo
sw x2, SPI_CONTROL_OFFSET(x1)
li x2, 0x100
sw x2, SPI_CFG_OFFSET(x1)

## Reset UART ##
li x1, UART_BASE
# Set NCO for maximum possible baud rate
li x2, UART_NCO
slli x2, x2, UART_NCO_OFFSET
# Enable system loopback
li x3, 1
slli x3, x3, UART_CTRL_SLPBK_OFFSET
or x2, x2, x3
# Enable TX/RX
ori x2, x2, 0x3
# Write to control register
sw x2, UART_CTRL_OFFSET(x1)
# Write 32 bytes to TX
li x4, 32
li x2, 0x42
_tx_loop:
    sw x2, UART_WDATA_OFFSET(x1)
    addi x4, x4, -1
    bnez x4, _tx_loop

## Reset SPI (2/2) & USB ##
li x1, SPI_BASE
li x2, SPI_BUFFER_OFFSET
add x1, x1, x2
li x11, USB_BASE
li x12, USB_BUFFER_OFFSET
add x11, x11, x12

# fill SPI and USB memory (with unique values for debugging)
li x2, 511
li x3, 0
_periph_mem_loop:
  sw x3, 0(x1)
  sw x3, 0(x11)
  addi x1, x1, 4
  addi x11, x11, 4
  addi x2, x2, -1
  addi x3, x3, 1
  bge x2, x0, _periph_mem_loop
lw x0, -4(x1) # dummy load from SPI memory
lw x0, -4(x11) # dummy load from USB memory

## Reset Ibex ##
li x1, 0x8000
lw x0, 0(x1)

## Reset RAM ##
li x1, RAM_BASE
li x2, RAM_END
_ram_loop:
  sw x0, 0(x1)
  addi x1, x1, 4
  bne x1, x2, _ram_loop
lw x0, -4(x1) # dummy  load from RAM
```

Figure 7-1: Boot code for MicroTitan.

Ibex

The majorify of the registers in the Ibex CPU are either initialized on reset or become initialized after a small number of clock cycles spent executing any code. An exception is the register `stored_addr_q`, which must be reset explicitly by the boot code. This register stores the address currently being requested on the instruction memory bus, assumming the memory doesn't respond within the same cycle. It is uninitialized on reset, and without intervention will remain uninitialized since the MicroTitan boot ROM is hardwired to send a valid reply within a cycle.

To initialize this register, the boot code forces a delay by loading a word from the ROM using the data memory interface (Figure 7-1 lines 54-55). The request arbiter hardware in front of the ROM will then delay the instruction memory request to reply to the data memory request, causing `stored_addr_q` to be written with the CPU's current instruction fetch address.

UART

The UART peripheral is solely implemented in the core clock domain, and most of its registers are initialized on reset. However, it does contain two FIFOs with uninitialized memory: the receive ("RX") FIFO and the transmit ("TX") FIFO, that store data being received and transmitted by the UART, respectively.

The UART has a "loopback" control register, writable by software, that can be enabled to internally connect the output of the UART to the input. Although likely meant for testing, this inclusion makes it straightforward to reset the FIFO memories. In order to reset the UART FIFO memories, the boot code enables loopback and writes enough bytes to the transmit buffer to fill it up, and this data is then internally routed back to the RX FIFO as inputs, filling up the receive buffer. The section of boot code responsible for this can be found in Figure 7-1 on lines 11-29 .

One note is that we could use auxiliary FIFO proofs to allow us to assume that this memory is cleared. However, the UART uses FIFOs of width 8 and depth 32, which are relatively large compared to other FIFOs and slow to verify. In terms of verification time, it's actually faster to verify that this loop clears the memory fully. A designer could choose, however, to trade off smaller and faster boot code in exchange for more verification time to eliminate this portion of the boot code.

SPI

The SPI portions of the boot code reset just a few pieces of uninitialized state. The main part is a 2 KB memory used for buffering data to be sent and received by the SPI peripheral. The boot code loops over this memory and writes data to each entry in order to reset it (Figure 7-1 lines 32-49). In addition, there are two other registers left uninitialized until the core attempts a read from the SPI memory, so the boot code performs a dummy read after resetting it (Figure 7-1 line 50).

The primary complexity in handling the core clock domain portion of the SPI peripheral is the fact that symbolic CDC register values from the SPI-in clock domain leak into registers in the core clock domain. Left unchecked, these symbolic inputs cause registers in the core clock domain to eventually contain large symbolic expressions which negatively impact performance. Therefore, the first thing the boot code does is enable two registers that hold the asynchronous FIFOs involved in CDC communication in a reset state, which stops the flow of symbolic data (Figure 7-1 lines 4-8). However, since it takes several cycles for these configuration registers to be set, a bit of symbolic data still leaks into the core clock domain component of the SPI peripheral. A large part of the implementation complexity deals with handling this symbolic data in a performant manner. This is done using performance hints, which are discussed in Section 7.2.2.

The SPI peripheral has a few other registers that either cannot be initialized by executing boot code or are otherwise inconvenient to reason about in this way. However, it turns out that once the rest of the core clock domain is reset these registers will never leak uninitialized data, which we prove separately after verifying that the boot code resets all other core clock domain state (see Section 7.2.3).

USB

A minimal amount of the USB peripheral is implemented in the core clock domain, making it fairly easy to handle. The only state must be explicitly initialized is a 2 KB data buffer, which is cleared in the same loop as the SPI memory (Figure 7-1 lines 32-49).

One key difference between the USB peripheral memory and the SPI peripheral memory is that the USB memory may also be written directly from the USB clock domain portion of the peripheral via a second write port. This poses a challenge for reasoning from the perspective of the core clock domain: since we overapproximate the USB clock domain, it can perform arbitrary writes and effectively undo any progress made in initializing the USB memory.

To counter this, we add an additional check to the proof of peripheral output determinism for the USB clock domain where we prove that any data the USB clock domain may write to the USB memory is "safe", i.e. only dependent on the USB subcircuit's inputs. This means that once an entry of the memory is initialized by the core clock domain, it will never depend on uninitialized data from that point forward, no matter whether or not the USB clock domain writes to it as well. This overapproximation of the USB memory is implemented with the `abstract-or-overapprox-vector` performance hint, discussed in Section 7.2.2. With this transformation in place, the boot code is able to provably reset the core clock domain part of the USB peripheral simply by looping over and resetting each entry of the USB memory.

RAM

Since RAM is uninitialized on reset, boot code loops over the memory and writes zero to each entry (Figure 7-1, lines 58-64).

7.2.2 Performance hints

The following performance hints are general transformations that can be applied to any component of the circuit state while verifying output determinism, because they are guaranteed to be sound (i.e., they don't affect correctness). Below are a list of the performance hints used, along with a description of their behavior and what they're used for in the proof of core output determinism.

Overapproximate

The `overapproximate` hint overapproximates one or more specified registers with fresh symbolic values. This is useful since some symbolic expressions in the state may not encode any useful constraints but may continuously grow during execution, worsening performance. Overapproximating these expressions away ensures that they do not cause performance issues.

The SPI and USB memories are overapproximated in the implementation until the loop that resets them starts. Since they're being overwritten anyway, there's no need to keep track of the growing symbolic expressions that accumulate in these registers due to symbolic input from the corresponding peripheral clock domains. In addition, there's a register in the SPI peripheral that latches the output from the SPI memory on every cycle. Since we don't reset this register with boot code (it's handled by the proof discussed in Section 7.2.3), we simply overapproximate it.

Abstract

The `abstract` hint performs a solver query to determine if a given register relies only on allowed dependencies (i.e., the circuit's past inputs). If so, that register is set to a fresh symbolic value that is itself added to the set of allowed dependencies. If not, the register value is left as-is. The solver query ensures that this is a sound transformation.

Many registers in the SPI peripheral that interface closely with the SPI-in clock domain are abstracted. Since the values in the SPI-in clock domain that cross to the core clock domain are considered allowed dependencies, abstraction is able to replace these register values with symbolic values that themselves are considered allowed dependencies, making it quick for a solver query to determine that these registers do not depend on uninitialized data.

In addition, a handful of registers that are responsible for receiving clock domain crossing interrupts from the USB clock domain are abstracted.

Abstract-or-overapprox-vector

The `abstract-or-overapprox-vector` hint loops over every entry in a vector (the Racket representation of a memory), and checks if each one depends only on allowed dependencies. If so, it replaces that entry with a fresh symbolic value that itself is added to the set of allowed dependencies (like the `abstract` hint). If not, it still replaces the entry with a fresh symbolic value, but does not add that symbolic value to the set of allowed dependencies (like the `overapproximate` hint).

This hint is used to implement the transformation of the multiport USB memory. as described in Section 7.2.1. This hint captures the desired behavior since we know that the USB clock domain can overwrite values in the memory (hence always overwriting it with fresh symbolic value), but we know that if the old value was already solely dependent on allowed dependencies, the new value must be as well (since it could either be the same as the old value if no write is performed, or become a new, allowed value from the USB clock domain).

Concretize and run-and-replace

The `concretize` hint performs a solver query to determine if a given register containing a symbolic expression can have only a single concrete value, and, if so, then that single concrete value is swapped in to replace the symbolic expression.

The `run-and-replace` hint executes the circuit starting from reset once again for a certain number of cycles, but with a different set of hints. It then replaces the values of specified registers in the state of the main execution with the values of those registers in the state of the secondary execution.

The `run-and-replace` and `concretize` hints are used in combination to concretize the value of a register that stores the current state of the state machine that handles input from the SPI RX FIFO. In the main execution, almost all registers involved in implementing this state machine are abstracted, since they otherwise grow to large symbolic expressions that slow down the final state check solver query. After the boot code executes for 150 cycles, this state machine hangs on an idle state, due to the boot code disabling the RX FIFO at its start. However, by abstracting all the registers that implement this state machine, it is difficult to prove that this is the case. The `run-and-replace` hint is used to perform an auxiliary execution for 150 cycles, where none of this state is abstracted. The `concretize` hint is applied to the auxiliary execution to prove that the state machine must be in an idle state after those 150 cycles complete. The register that stores the state of the state machine is then overwritten in the main execution with the value for "idle", which prevents future updates to the rest of the state machine.

7.2.3 Additional output equivalence proof

Boot code cannot be used to reset all registers in the core clock domain. There is a register in the SPI peripheral, `sram_wdata`, that is initialized on reset but may end up containing uninitialized data that leaks from the SPI memory before it is reset by the boot code. Although boot code cannot reset `sram_wdata`, it turns out that any uninitialized data that ends up in this register will never influence outputs. Kronos proves this using an output equivalence proof between the circuit state after boot code is finished running, and an instance of the circuit state where this register's value has been made to fully depend on allowed dependencies. This lets us say that the register has effectively been initialized after boot code finishes running.

In order to show that this is the case, Kronos needs to prove an invariant relating the value of `sram_wdata` with another register, `byte_enable`, which is abstracted to a fresh symbolic value on each cycle. The invariant is that for any `i` between 0 and 3 (inclusive), if `byte_enable[i]` is 1, then `sram_wdata[8(i+1):8i]` is only dependent on allowed dependencies. In order to represent this in our circuit state, on each cycle `sram_wdata` is replaced with the expression shown below, before `byte_enable` is abstracted on that cycle. This overapproximation is proven to be sound using a solver query.

```
(concat
  (ite (= byte_enable[3] 1) fresh_wdata*[31:24] fresh_wdata[31:24])
  (ite (= byte_enable[2] 1) fresh_wdata*[23:16] fresh_wdata[23:16])
  (ite (= byte_enable[1] 1) fresh_wdata*[15:8]  fresh_wdata[15:8])
  (ite (= byte_enable[0] 1) fresh_wdata*[7:0]   fresh_wdata[7:0]))
```

Here, `fresh_wdata...` represent fresh symbolic values. The names with a star suffix are added to the allowed dependencies set, and the ones without are not. After boot code execution, this allows Kronos to easily construct a version of `sram_wdata` that only depends on allowed dependencies, where every 8-bit slice corresponding to a zero in `byte_enable` is replaced with zeros.

In addition, there are two other registers in the SPI peripheral that are not provably initialized by Kronos's boot code, but likely could be by different boot code. These two registers latch data that is in the SPI memory and is to be output over SPI, so they could be reset by having the boot code use the proper configuration registers to set up a transaction. However, since much of the state in the SPI peripheral ends up containing symbolic values during boot code execution, it is difficult to reason about a specific concrete execution in a performant manner. In order to avoid the pitfalls associated with this, the uninitialized data in these registers are also shown not to influence outputs as part of the proof described above.

7.3 Peripheral output determinism

Kronos verifies peripheral output determinism for each peripheral clock domain. As mentioned in Section 5.3, our general strategy for proving this property is by proving output equivalence between an instance of the peripheral clock domain subcircuit initialized normally and one with an entirely deterministic starting state. This section discusses interesting aspects of the implementations of each peripheral clock domain's associated proof.

7.3.1 SPI-in and SPI-out

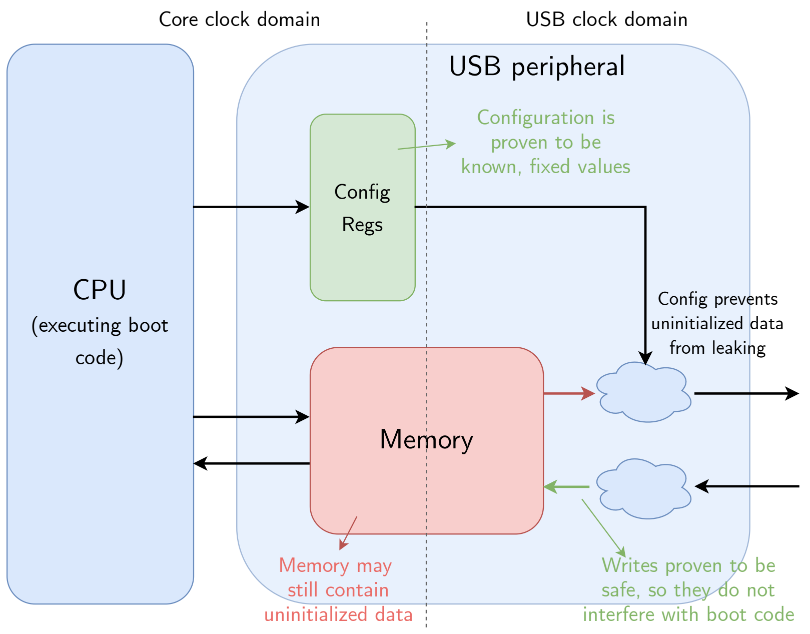
The SPI-in and SPI-out clock domains each have a single register that is left uninitialized on reset. In both cases, a straightforward output equivalence proof is enough to show that the uninitialized data in these registers never leak to outputs.

Although both of these proofs end up being quite simple, this is in part due to the FIFO auxiliary proofs. Without the FIFO proofs, the SPI proofs would have to handle the uninitialized data left in the async FIFO storage themselves.

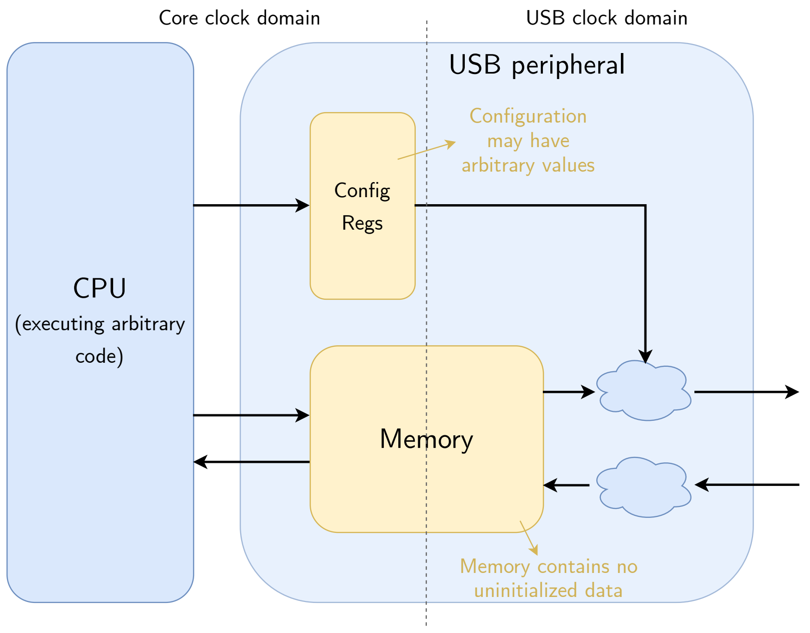
7.3.2 USB

The USB clock domain poses a unique implementation challenge. Since this clock domain also contains the multiport USB memory, it includes a large piece of state that is uninitialized on reset. With this uninitialized state and completely unconstrained inputs, it turns out that the USB clock domain subcircuit on its own does not actually satisfy peripheral output determinism, since values from this uninitialized memory may leak to outputs.

Proving peripheral output determinism for USB requires more specialized reasoning, with a hybrid approach that fuses the software-assisted strategy for core output determinism with the inductive strategy for proving hardware-inherent properties. Kronos divides up the USB proof into two separate proofs called “phase 1” and “phase 2”. Figure 7-2 is a block diagram of the USB peripheral showing the distinction between these two phases. The phase 1 proof corresponds to the USB peripheral’s operation while boot code is running. During this time, all entries of the USB memory are considered to potentially contain uninitialized data, since there is no way to reason about when any given entry would be reset by the boot code in terms of USB clock cycles. This phase, however, does allow for more specific reasoning about the values of CDC registers from the core clock domain. Since the boot code that executes during this time is known, it’s possible to verify that these CDC register values are not just safe, but that they also have specific, fixed values. It’s also important to note that these fixed values must be the *reset* values of these CDC


Core clock domain
USB clock domain
USB peripheral
Configuration is proven to be known, fixed values
Config Regs
CPU
(executing boot code)
Config prevents uninitialized data from leaking
Memory
Memory may still contain uninitialized data
Writes proven to be safe, so they do not interfere with boot code

(a) Phase 1


Core clock domain
USB clock domain
USB peripheral
Configuration may have arbitrary values
Config Regs
CPU
(executing arbitrary code)
Memory
Memory contains no uninitialized data

(b) Phase 2

Figure 7-2: The two phases of the USB clock domain proof.

registers, since otherwise additional reasoning would be required to determine what happens in the time between reset and the CDC registers being set to their desired values, which would defeat the purpose of using fixed inputs in the first place.

Luckily, the default values for the core to USB CDC registers ensure that no uninitialized data can leak from the USB memory to outputs. Kronos verifies in the core output determinism proof that these CDC registers maintain their reset values throughout boot code execution, allowing the proof of phase 1 to make this assumption.

In addition, the phase 1 proof includes logic to ensure that data from uninitialized state is not written to the USB memory itself. This enables the assumption discussed in the proof of core output determinism, that “safe” memory entries remain safe, even though the memory may be written to in otherwise arbitrary ways by the USB clock domain.

It’s not possible to make assumptions about specific CDC register values after boot code execution concludes. At this point, however, it is possible to assume that the USB memory is fully reset, since this is proven by core output determinism. Therefore, phase 2 is able to use the output equivalence technique to show that the USB peripheral satisfies output determinism.

Chapter 8

Evaluation

This chapter provides an evaluation of Kronos answering the following questions:

- How long does it take to verify MicroTitan? (Section 8.1)
- Does MicroTitan require hardware changes in order to satisfy output determinism? If so, how substantial are those changes? (Section 8.2)
- Does proving output determinism help find potential security issues? (Section 8.3)

8.1 Performance

Table 8.1 presents the runtime of the machine verification implementation for each property we prove. These runtimes were benchmarked on a machine with a quad CPU Intel Xeon E7-8870 with 40 total cores (or 80 hyperthreads) and a 2.40GHz clock speed. The machine has 256 GB RAM.

All these properties can be proven in parallel, so on a machine with at least five cores the overall verification time is dominated by the slowest property. That would be core output determinism, with a runtime of **12,526 seconds** or about **3 hours and 29 minutes**. Single core performance can be calculated as the sum of times shown, dominated by core output determinism and the FIFO auxilary proofs, for a total of about **3 hours and 42 minutes**.

Property	Runtime (seconds)
Core output determinism	12526.5
FIFO auxiliary proofs	727.0
Peripheral output determinism (USB)	41.1
Peripheral output determinism (SPI-in)	18.4
Peripheral output determinism (SPI-out)	17.8

Table 8.1: Runtime of each verification script.

These runtimes show that the implementation of modular output determinism for MicroTitan is computationally feasible.

Scaling verification

Evaluating the runtime of verifying each property reveals that verification time is heavily dependent on the size of the circuit being verified. This is evident with regards to the synchronous and asynchronous FIFO verification, since Kronos verifies multiple instantiations of the FIFO with various size parameters.

Figure 8-1 shows a graph of FIFO verification time versus FIFO width and depth. The specific FIFO sizes verified for this project are distinguished on the graph, and summarized in table 8.2. The sum of these times determines the FIFO auxilary proof runtime presented in table 8.1.

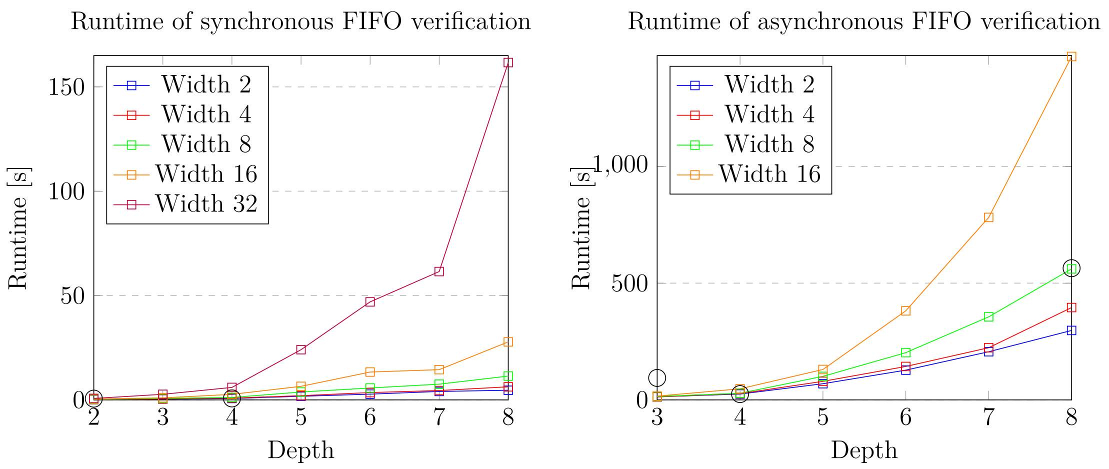


Figure 8-1: Graphs of FIFO runtime vs. FIFO size parameters for synchronous and asynchronous FIFOs. The black circles represent the parameters of FIFOs verified in MicroTitan.

Type	Depth	Width	Runtime (seconds)
Sync	2	33	0.6
Sync	4	2	0.7
Async	3	100	94.0
Async	4	5	24.9
Async	4	17	42.7
Async	8	8	564.2

Table 8.2: Runtime of verifying output determinism for FIFOs, parameterized by type (sync or async), width, and depth.

Figure 8-1 shows that the FIFO verification runtime scales exponentially with the depth of the FIFO, with the difference between FIFO width becoming more apparent at larger depths. The FIFOs verified in Kronos were all in a reasonable performance domain, but if they were deeper then verification of the FIFOs could become infeasible. This could be problematic since FIFOs contain uninitialized data on important boundaries, as they are often used to facilitate CDC communication and external I/O. If a circuit contains FIFOs deep enough that it becomes intractable to verify that they do not leak data, it could be infeasible to verify output determinism for the circuit overall.

Figure 8-1 also shows that asynchronous FIFO verification is an order of magnitude slower than synchronous FIFO verification for the same size parameters. This can likely be attributed to several factors. One is the increase in state size due to `clk2fflogic` extraction, which also requires more complicated logic to handle. Another is that the asynchronous FIFO requires proving three inductive steps, one for each permutation of clock steps, while the synchronous FIFO only requires proving a single inductive step.

This behavior of verification time scaling with state size is common in SMT-based proofs. Lee and Sakallah show this and present a method for verifying hardware properties that scales with state size [12]. Adopting techniques from this work could help if scaling were a major performance botttleneck.

8.2 OpenTitan hardware changes

In order to ensure that MicroTitan satisfies output determinism, Kronos includes boot code to reset state, as well as hardware changes to fix OpenTitan peripherals that make satisfying the property impossible or too difficult to verify. This section evaluates the impact of both the boot code addition and RTL changes, which is shown to be minor. This indicates that off-the-shelf hardware can be modified in order to maintain security properties with minimal undesirable consequences.

8.2.1 Boot code

MicroTitan includes boot code that resets as much uninitialized state as possible. This boot code consists of 44 RISC-V instructions and takes up 320 bytes, which is about 4% of the MicroTitan ROM. It takes $24,516$ cycles to run, which translates to 0.49 additional milliseconds on boot when run on a Nexys Video FPGA using a 50 MHz clock speed. The time to execute boot code primarily comes from looping over memory to reset entries. If MicroTitan were modified to include a larger RAM, for example, the boot time would increase proportionally.

8.2.2 RTL changes

Several OpenTitan components had to be patched in MicroTitan in order to make it possible to satisfy output determinism. The complexity of these changes in terms of lines of code, as well as the impact of these changes on hardware utilization are shown in table 8.3.

Change	LoC	LUTs used	Registers used
Sync FIFO	+8	+1.64%	+0.00%
Async FIFO	+13	+0.22%	+0.09%
SPI RX order	+15	+0.00%	+0.00%
USB memory	+30	+0.01%	+0.00%
USB reset sync	+10	−0.02%	+0.00%

Table 8.3: Lines of code required and FPGA resource use for each hardware change necessary to ensure MicroTitan satisfies output determinism.

These changes were benchmarked by synthesizing them for the Nexys Video FPGA dev board (natively supported by build scripts provided in the OpenTitan repository), using Vivado 2018.3. Note that these benchmarks are for the OpenTitan itself, not MicroTitan. This is to measure the impact these changes would have if upstreamed into OpenTitan. The measurement baseline includes the toolchain hacks described in Section 6.1.1.

For the purposes of benchmarking, these changes were applied indvidually to the baseline in order to clearly illustrate the individual impact of each one. All measurements are reported relative to the baseline, and resource usage is reported in terms of percentage of that resource available in the FPGA.

The FIFO patches are the highest-impact changes, likely because these structures are used many times throughout the design. The impact is still minor, however, using less than 2% of available hardware resources. Most other changes make a negligible difference, with the USB reset synchronizer replacement using slightly fewer hardware resources. None of these changes prevent OpenTitan from meeting timing constraints for the FPGA target.

Overall, these changes had a minimal impact on utilization and timing, and didn't require much code to implement.

8.3 Impact of violating output determinism

While verifying MicroTitan, we found multiple parts of the design that make output determinism impossible to satisfy. These issues were patched in the implementation of MicroTitan (see Section 6.1.2).

The USB reset synchronizer violation likely doesn't have a security impact in any real-world scenario—the extent of what an attacker can observe is whether or not the SoC's reset line was high or low two cycles before reset was last deasserted, and this information only leaks on the cycle immediately after reset is deasserted.

Otherwise, we believe each other output determinism violation may have some security impact. This section discusses the implications of each one.

8.3.1 UART RX data leak

In the baseline OpenTitan implementation, we discovered that a byte of uninitialized data can be read by software directly from the UART RX FIFO. This is because after reset, the baseline synchronous FIFO implementation exposes the first entry of its uninitialized memory directly to a software-accessible register. Although software can determine that this entry is invalid (by checking a separate to register to determine if the FIFO is empty), in the context of a Notary-like system that aims to guarantee non-interference across reset boundaries, malicious software could skip this check and extract a byte of data that was received over UART by a previously executing agent.

This issue was reported to the OpenTitan team. Although information sent over I/O channels is not considered confidential according to their security model, they accepted a patch[1] to prevent the synchronous FIFO from outputting invalid data (in all instances, not just the UART) in order to provide defense-in-depth.

Note that this issue does not actually lead to a violation of output determinism in MicroTitan, since the boot code resets data in the UART RX FIFO before it can leak. However, we discuss it here for two reasons. First, it would have been impossible to reset this state without the somewhat lucky inclusion of the UART loopback register. Second, discovering this issue led to a hardware change that hardened OpenTitan upstream.

8.3.2 SPI TX data leak

The SPI peripheral is designed intentionally such that it will transmit data from its TX FIFO even when it reads as empty. This means that if the device has been reset, and no new data has been written to the SPI TX FIFO, a byte of uninitialized data left in the FIFO memory may leak over SPI if a host begins a transaction. This violates output determinism, which states that outputs may not be dependent on data left uninitialized at reset.

Since the CPU cannot read this uninitialized data directly, this can only be ex-

[1]https://github.com/lowRISC/opentitan/pull/2420

ploited by a malicious SPI host. It would be problematic in a scenario where data transmitted over SPI prior to the last reset should be confidential to the SPI host currently communicating with the device.

This problem was fixed by patching all asynchronous FIFOs to initialize the first entry of their storage to zero on reset. Only the SPI TX FIFO needed to be patched, but patching all FIFOs simplified reasoning about the circuit as a whole and had minimal impact.

8.3.3 SPI RX data leak

Another issue with the SPI peripheral is that data previously received over SPI prior to reset can be read by the processor. The problem is that there's a configuration register that can be changed by software that determines the order of bits received by the SPI (MSB or LSB first). This register can be changed during an SPI transaction, so in theory with precise enough timing software could toggle it such that the order is changed after each bit is received. The way the hardware is designed, this could prevent the 8-bit shift register that gets filled by the RX hardware from being initialized beyond a single bit. Therefore, 7 bits of uninitialized data can be read by software.

This is tricky to exploit, due to how precisely this configuration register must be toggled (in fact, it may be impossible to exploit if the memory interface is too slow to toggle the register fast enough). In addition, writing to this register during an SPI transaction can result in glitches due to metastability, which could result in unspecified behavior. For this reason, the OpenTitan documentation states that software must not change this register during SPI operation. Of course, that does not prevent malicious software from attempting to use this interface to exploit a vulnerability, since the hardware does not enforce this.

In order to resolve this issue, the SPI peripheral was patched to prevent the RX order from being changed while a byte is being transmitted.

8.3.4 Possible USB data leak

The USB peripheral starts out with an unitialized SRAM buffer, and we believe it may be possible for some of the data in this buffer to leak via USB. Since the USB peripheral implements the USB device protocol, data is transmitted only in response to an "IN" endpoint request from a USB host. Software is responsible for setting a configuration register to indicate when there is data available for a response. Regardless of whether this configuration register is set or not, data from the USB memory flows through a chain of two registers before it reaches a point where it may be transmitted as a response to an IN request. However, a problem may arise when uninitialized data is latched in those two intermediate registers, and software indicates that data is ready to be transmitted over USB while the peripheral is currently processing an IN request. Although the valid data that software wishes to transmit may not yet be latched in the two intermediate registers, the hardware will note that there is valid data ready to be sent and the uninitialized values in these registers will be leaked via USB.

It's unclear if the exact issue described here is exploitable. It's likely that these intermediate registers may actually be cleared with the correct valid data before an IN request is replied to. However, the complexity of the USB protocol has made it difficult to verify this. In order to ensure that MicroTitan satisfies output determinism, the USB peripheral is patched to not write to the intermediate registers unless software has already indicated that it has written valid data to the USB memory that it is ready to send.

Chapter 9

Related Work

9.1 Verifying systems software

The Serval verification framework [20] allows systems code to be verified against an idealized model of the RISC-V ISA, but any security properties proven with these verification methods could be violated due to microarchitectural bugs.

Murray et al. [19] use formal verification to prove information flow control properties, which provide integrity and confidentiality, on top of the C implementation of the seL4 microkernel. This is another example of low-level systems software with verified security properties, but this work assumes correct hardware, which doesn't capture vulnerabilities such as microarchitectural side channels.

9.2 Verifying hardware

9.2.1 Self-equivalence with don't-cares

Lee and Sakallah [12] apply formal verification to prove that 41 bits of state can be left uninitialized in a Cortex-M0+ ARM processor core without the uninitialized data propagating to observable outputs. They call this property "self-equivalence with don't-cares" or "SEQX".

The primary contribution of their work is a new formal verification technique, and the SEQX property is mentioned very briefly in the context of a benchmark for this technique—a formal definition of SEQX is not presented in the paper. However, this property sounds intuitively equivalent to the output determinism property presented in this thesis, and thus should be sufficient to provide noninterference. Despite the similarities, there are several ways in which their work varies from this thesis:

- SEQX appears to be an inherent property of the circuit, and their work does not discuss reasoning about boot code resetting state. Therefore, a circuit like MicroTitan does not satisfy SEQX, because some uninitialized state can leak to outputs unless it is initialized by boot code.
- Their target is only a CPU, as opposed to an entire SoC including peripherals, which have specific design patterns associated with their own challenges (all data leakage problems we found came from peripherals).
- Given that it's only a CPU, their Cortex-M0+ target is quite small compared to MicroTitan, with only 41 bits of uninitialized state. MicroTitan has 420 bits of uninitialized state in registers (not including memories, which comprise thousands of bits of uninitialized state).
- Their work does not discuss hardware with multiple clock domains.

Despite these differences, Lee and Sakallah's technique could be useful in conjunction with the techniques presented in this thesis, for example, their method could replace Kronos's method for proving remaining uninitialized state left over after boot code execution does not propagate to outputs.

9.2.2 CDC verification

Existing academic work focusing on formal verification of circuits with multiple clock domains seems to primarily focus on verifying clock domain crossing boundaries themselves [13], rather than properties of entire circuits. Clock domain crossing circuits are hard to design in part due to issues with metastability, which relates to analog behaviors of circuit components such as flip-flops breaking the digital abstraction. Although important to get right, this is beyond the scope of our work—we assume ideal digital circuits, and assume that the circuits we analyze do not have issues such as metastability.

9.2.3 RISC-V formal

One popular hardware verification project, `riscv-formal` [26], provides a framework for verifying that a particular processor correctly implements the RISC-V ISA. This is distinct from our work since it focuses on correctness with respect to an ISA, which makes no security guarantees.

9.2.4 Symbiyosys tool

Symbiyosys [8] is a tool for formally verifying digital circuits with support for formally verifying circuits with multiple clocks. However, many examples we have found use this tool to verify simple properties about small modules, not system-wide properties such as output determinism for an entire SoC.

9.3 Hardware state clearing

Several other works have also explored the concept of microarchitectural state clearing on context switches in order to enforce security guarantees.

Ge et al.[9] argue that hardware support for microarchitectural state flushing is necessary to ensure operating systems are safe from timing channels. They describe how flushing the L1 cache on an x86 machine requires a "brittle" sequence of instructions based on assumptions about undocumented hardware implementation details, due to a lack of a dedicated instruction. A related project [24] modifies an existing RISC-V processor to add a special "temporal fence" instruction that provides the miroarchitectural state clearing necessary to enforce safety from timing channels.

MI6 [4] is another project that argues for hardware state-clearing support. MI6 implements a prototype of an out-of-order processor with a `purge` instruction added to its ISA which clears microarchitectural state. In particular, it clears structures that track in-flight instructions, branch predictor structures, and cache state. This instruction is used for safe context-switching between security domains on a single core.

All of this work makes a similar observation to ours that state clearing can be useful for noninterference across context-switch boundaries, but they apply a hardware-based approach as opposed to Kronos's software-based approach, and do not employ formal verification to ensure that all state is reset. An advantage of Kronos's software-based approach is that it allows use of off-the-shelf hardware with minimal modifications, whereas a hardware-based approach requires drastic changes to the hardware.

9.4 Taint tracking

Taint tracking is a general approach for enforcing security policies around information flow. It can be applied to high-level application software or low-level hardware [21]. Taint tracking works by "tainting" data based on application-specific security properties (for example, data can be tainted to mark that it comes from a high security domain, and thus may not be leaked into a lower security domain). Calculations incorporating tainted data carry forward the taint to the result, allowing the flow of data throughout a system to be tracked. Similar to taint tracking, Kronos has a notion of "safe" and "unsafe" data, depending on whether or not it comes from outside inputs or uninitialized data respectively.

9.5 Hardware symbolic execution

Zhang and Sturton [28] present a strategy for applying symbolic execution to digital hardware, similar to this thesis. However, instead of using it to prove that hardware satisfies a specific security property, they use symbolic execution to search for the presense of a variety of general vulnberability classes in given hardware. They use a toolchain based on the Verilator simulation framework for converting SystemVerilog to C++, and use the KLEE [5] engine for symbolic execution.

Chapter 10

Conclusion

This thesis presents Kronos, which consists of the MicroTitan SoC and a security property called output determinism that has been formally verified at the hardware level. This thesis develops a methodology for proving this property, provides an argument that it is sound, and implements it for MicroTitan. This thesis further shows that hardware more complex than the PicoRV32 can be used to implement a system that guarantees noninterference as defined by Notary, and provides insight into challenges around proving security properties for realistic embedded hardware systems.

10.1 Future work

10.1.1 Modular reasoning

Reasoning about large systems as a whole is difficult, and reasoning about them modularly helps. This thesis already uses modular reasoning in a few ways:

- The FIFO auxiliary proofs entail modular reasoning about individual Verilog modules.
- Modular output determinism allows reasoning about each clock domain separately.

An additional level of modular reasoning that would be helpful is modular reasoning at the level of peripherals. OpenTitan peripherals seem well suited for this: they all use a standard interface designed to work with the same interconnects.

Having a formal way to reason about each peripheral individually and describe how they compose would make it possible to reason about security properites on a peripheral-by-peripheral basis, and then think about how they can be combined to build new custom subsets and draw conclusions about the security guarantees those subsets provide. Note that although our work lets us find potential security issues that affect the OpenTitan, we can only firmly state that MicroTitan has been verified, since that's the circuit we actually verify. We can't state any precise conclusions about OpenTitan itself, or about its peripherals as individual units.

10.1.2 Scalable verification

One pitfall of our verification approach based on symbolic execution and SMT solvers is that sizes of various structures can greatly affect verification runtime.

One example is the time it takes to reset memories. Our boot code loops over each entry of peripheral memories as well as the SoC's RAM in order to reset them, and the time to verify boot code execution scales linearly with its size. If MicroTitan's RAM was the same size as Notary's SoC, core output determinism would take hundreds of hours to verify, since symbolically executing a cycle of MicroTitan's more complex circuit is much slower.

However, the effects of a memory-clearing loop are regular and predictable no matter the size of the RAM. A way to reason about the effects of these loops that operates at a higher level of abstraction would be helpful for performance, and prevent artificial size constraints from having to be imposed on verified systems.

Another example is FIFO verification time scaling with size parameters. Although FIFO verification time wasn't a bottleneck, we wouldn't able to rely on using those proofs if we had any FIFOs that were prohibitively large. Lee and Sakallah [12] present a scalable verification technique that operates on hardware at the word-level, which could be interesting to apply here.

10.1.3 Persistent storage

One common component of SoCs used in real systems is persistent storage such as flash. However, it's not clear how persistent storage would fit into a system that aims to provide noninteference. Coming up with a security spec that takes into account persistent storage, as well as proof strategies to model it, would be an interesting direction for future work. OpenTitan includes a flash peripheral, so this would be necessary for verifying OpenTitan in its entirety.

10.1.4 Design-for-verification

This thesis focuses primarily on verifying hardware based on an existing open source design with minimal modifications. However, an interesting direction for future work could be to use insights gained from this work to either modify or design hardware from scratch that is well-suited to verification.

One change in particular that could have made verification simpler in the context of this project would be to add an extra control register to the SPI and USB peripherals that lets each of them be disabled by software (held in their reset state). This register would be set such that they are in the disabled state on reset.

This would be helpful since a lot of effort was dedicated to compensating for symbolic state that can leak while the boot code resets memories and sets up various control registers. For example, this would allow us to eliminate the need for using various performance hints to optimize the SPI RX state machine control logic, and it would eliminate the need for the 2-phase USB proof (since the peripheral would not be active during the first phase).

This change would likely not have a huge impact on hardware in terms of timing or resource utilization, so it seems plausible that it would be a worthwhile tradeoff for a multiclock design where a verified security property is desired. In fact, more recent versions of OpenTitan incorporate a "reset manager" module, which supports software controlled resets of the SPI and USB peripherals [17]. If incorporated into MicroTitan, this module may give this desired behavior.

10.1.5 Formalizing metatheory

Kronos proves multiple different properties using Rosette, but it does not formalize the way we use one proof to inform another (such as using the FIFO proofs in peripheral output determinism proofs), nor does it formally prove that the core and peripheral output determinism properties imply top-level output determinism. This constitutes a formality gap that could be fixed by formalizing the metatheory. This would reduce our TCB, and give us more confidence in our overall claim that MicroTitan satisfies output determinism.

Bibliography

[1] Issues · zachjs/zv2v. `https://github.com/zachjs/sv2v/issues?q=is:issue+is:closed+author:nmoroze`, 2020.

[2] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM side-channel(s). In *Proceedings of the 2002 IACR Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Redwood City, CA, August 2002.

[3] Anish Athalye, Adam Belay, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Notary: A device for secure transaction approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Huntsville, Ontario, Canada, October 2019.

[4] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. Mi6: Secure enclaves in a speculative out-of-order processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 42–56, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, CA, December 2008.

[6] Clifford E Cummings, Don Mills, and Steve Golson. Asynchronous & synchronous reset design techniques-part deux. *SNUG Boston*, 9, 2003.

[7] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, Budapest, Hungary, March–April 2008.

[8] Symbiotic EDA. Symbiyosys (sby) documentation. `https://symbiyosys.readthedocs.io/en/latest/index.html`, 2020.

[9] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: the missing OS abstraction. In *Proceedings of the 14th ACM EuroSys Conference*, pages 1–17, Dresden, Germany, March 2019.

[10] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *Proceedings of the 34th Annual International Cryptology Conference (CRYPTO)*, pages 444–461, Santa Barbara, CA, August 2014.

[11] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 27th USENIX Security Symposium*, pages 19–37, Baltimore, MD, August 2018.

[12] Suho Lee and Karem Sakallah. Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction. In *International Conference on Computer Aided Verification*, pages 849–865, July 2014.

[13] Bing Li and Chris Ka-Kei Kwok. Automatic formal verification of clock domain crossing signals. In *2009 Asia and South Pacific Design Automation Conference*, pages 654–659. IEEE, 2009.

[14] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium*, pages 973–990, Baltimore, MD, August 2018.

[15] lowRISC contributors. Ibex RISC-V core. `https://github.com/lowRISC/ibex`, 2019.

[16] lowRISC contributors. Open source silicon root of trust (RoT) | OpenTitan. `https://opentitan.org/`, 2019.

[17] lowRISC contributors. Reset manager HWIP technical specification. `https://docs.opentitan.org/hw/ip/rstmgr/doc/`, 2021.

[18] Rita Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In *Proceedings of the 2000 IACR Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 78–92, Worcester, MA, August 2000.

[19] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, pages 415–429, San Francisco, CA, May 2013.

[20] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Huntsville, Ontario, Canada, October 2019.

[21] Daniel Schoepe, Musard Balliu, Benjamin C. Pierce, and Andrei Sabelfeld. Explicit secrecy: A policy for taint tracking. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 15–30, 2016.

[22] Zachary Snow. sv2v: SystemVerilog to Verilog. `https://github.com/zachjs/sv2v`, 2020.

[23] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 530–541, Edinburgh, United Kingdom, June 2014.

[24] Nils Wistoff, Moritz Schneider, Frank Gürkaynak, Luca Benini, and Gernot Heiser. Prevention of microarchitectural covert channels on an open-source 64-bit RISC-V core. In *Workshop on Computer Architecture Research with RISC-V (CARRV)*. ACM, 2020.

[25] Claire Wolf. PicoRV32 – a size-optimized RISC-V CPU. `https://github.com/cliffordwolf/picorv32`, 2020.

[26] Claire Wolf. RISC-V formal verification framework. `https://github.com/SymbioticEDA/riscv-formal`, 2020.

[27] Claire Wolf. Yosys Open SYnthesis Suite. `http://www.clifford.at/yosys/`, 2020.

[28] Rui Zhang and Cynthia Sturton. A recursive strategy for symbolic execution to find exploits in hardware designs. In *Proceedings of the 2018 ACM SIGPLAN International Workshop on Formal Methods and Security*, FMS 2018, page 1–9, New York, NY, USA, 2018. Association for Computing Machinery.

[29] Yongbin Zhou and Dengguo Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. Cryptology ePrint Archive, Report 2005/388, October 2005.