# rtlv: push-button verification of software on hardware

Noah Moroze*, Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich
MIT CSAIL
moroze@csail.mit.edu*

## ABSTRACT

This paper presents rtlv, an approach for push-button formal verification of properties that involve software running on hardware for many cycles. For example, rtlv can be used to prove that executing boot code resets a processor's microarchitectural state to known deterministic values, while existing tools time out when attempting to verify such a property.

Two key ideas enable rtlv to handle reasoning about many cycles of circuit execution. First, rtlv uses hybrid symbolic execution to reason about a circuit with symbolic values while minimizing the complexity of symbolic expressions; this is achieved by compiling circuits to programs in the Rosette solver-aided language. Second, rtlv enables the development of reusable *circuit-agnostic property checkers* that have a *performance hint interface*, allowing developers to optimize verification performance while maintaining confidence that the proof is correct.

Using rtlv, formally verifying a state-clearing property for a small (1,300 flip-flop) RISC-V SoC takes only 1.3 seconds, while SymbiYosys, a popular open-source verification tool, is unable to finish within 12 hours. In another case study, rtlv scales to a larger 4,300 flip-flop RISC-V SoC where verifying this state-clearing property requires modeling over 20,000 cycles of software executing on hardware. Formal verification with rtlv helped us find and fix violations of the property in the baseline hardware, demonstrating that rtlv is useful for finding bugs.

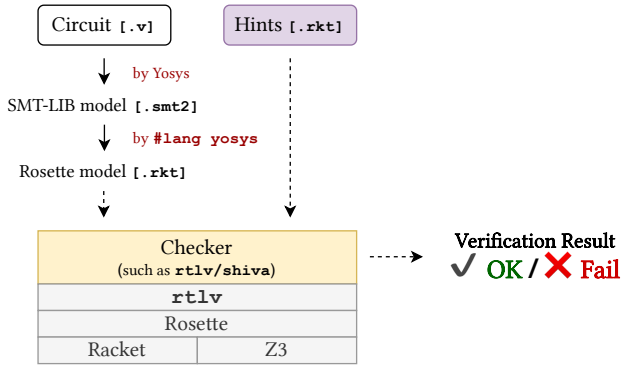


**Figure 1: The rtlv workflow. A user develops a circuit-agnostic property checker (yellow) that takes in a circuit model and performance hints that suggest optimizations. The user then inputs a particular circuit's Verilog code (white), mechanically transformed into a Rosette model by rtlv's compiler, along with circuit/property-specific performance hints (purple) that suggest optimizations. The checker returns "OK" if it is able to verify that the property holds.**

## 1 INTRODUCTION

Formally verifying digital hardware allows developers to increase their confidence in a system's security and correctness. Many example uses of popular hardware verification tools involve checking properties that can be verified after executing for a small number of cycles [16]. However, these tools are unable to efficiently verify properties that require executing for many cycles, making it difficult to use the tools for reasoning in a cycle-accurate manner about software running on the hardware.

For example, suppose a developer wants to verify that boot code executing after reset clears all microarchitectural state in a CPU. This requires modeling the complete execution of this boot code from an initially unconstrained circuit state and performing a solver query on the final state. For the PicoRV32 [17], a simple RISC-V CPU, verifying this property requires executing 104 cycles of boot code. Using SymbiYosys [14], a popular open-source hardware verification tool, the solver query resulting from unrolling 104 cycles of circuit execution is unable to finish within 12 hours.

This paper presents rtlv, an approach for verifying circuits that enables efficient cycle-accurate reasoning about software executing on hardware. With rtlv, modeling 104 cycles of boot code execution and verifying that all state is cleared in the PicoRV32 takes only 1.3 seconds.

rtlv efficiently handles many cycles of execution by symbolically executing the circuit while maximizing the amount of circuit state that remains concrete and minimizing the complexity of symbolic expressions in the circuit state. rtlv accomplishes this through a verification workflow, shown in Figure 1, based on two key ideas. First, the developer uses rtlv to compile the circuit to Rosette [15], a domain specific language for solver-aided programming embedded in Racket. This lets developers verify the circuit using Rosette and take advantage of Rosette's unique symbolic execution approach and rewrite rules. Second, the developer builds a *circuit-agnostic property checker* that has an interface for supplying *performance hints*. These hints suggest certain optimizing transformations of the circuit state. The property checker is responsible for ensuring that these optimizations do not affect the correctness of the proof, resulting in a smaller core of trusted code. To verify the circuit, the developer invokes the property checker with the extracted circuit model as well as a list of performance hints. Overall, Rosette's symbolic execution system and performance hint optimizations result in simpler solver queries compared to those produced by tools like SymbiYosys, which improves verification performance.

This paper provides an overview of rtlv, and contributes the following:

- A description of rtlv's Verilog to Rosette compiler.
- An example circuit-agnostic property checker, rtlv/shiva, that can be used for verifying a security property similar to the microarchitectural state clearing property described above.

```
picorv32 cpu1(...);
picorv32 cpu2(...);

if (cycle_count == 104) begin
  assert(cpu1.reg1 == cpu2.reg1);
  assert(cpu2.reg2 == cpu2.reg2);
  // ... repeat for each register in picorv32
```

**Figure 2: Verification pseudocode using SymbiYosys. This tool requires an encoding with two copies of the CPU that are initially unrelated, along with an assertion that their state must be equal after the boot code runs.**

- A case study using `rtlv/shiva` to verify a real-world RISC-V SoC based on the OpenTitan, demonstrating that `rtlv`'s approach can be scaled to complex hardware and can be used for finding bugs in practice.

## 2 EXAMPLE: DETERMINISTIC START

*Deterministic start* is an example of a property where verification involves cycle-accurate reasoning over many cycles, often about software running on hardware. If a circuit satisfies deterministic start, then its internal state, including microarchitectural state, is fully cleared to deterministic values by boot code that runs on reset [2].

We use deterministic start as a motivating example for using `rtlv` over existing tools for verifying circuits. We performed an experiment where we set up both `rtlv` and SymbiYosys, a popular open-source hardware verification tool, to prove deterministic start for the PicoRV32 CPU. As shown in Figure 2, we encoded this property in SymbiYosys by instantiating two copies of the PicoRV32 and adding assertions that each register in the one copy is equal to the corresponding register in the other copy (i.e., there is only one possible value that the register can take on) after the boot code runs. We then configure SymbiYosys to use its Yosys-SMTBMC backend to model all 104 cycles of boot code execution and verify that these assertions hold.

Using `rtlv`, instead of writing verification code in Verilog, developers write code in Rosette, a solver-aided programming language embedded in Racket. `rtlv` provides a system for compiling circuits into a Rosette model. In order to verify deterministic start for the PicoRV32, a developer can instantiate a fully unconstrained PicoRV32 circuit state, call the circuit's `step` function 104 times, and then check the resulting circuit state to verify that it must be deterministic. Because Rosette provides solver-aided queries (effectively, direct access to the SMT solver), the `rtlv`-based approach can verify this property without instantiating two instances of the circuit and comparing them. As shown in Figure 3, the verification code can instead invoke the SMT solver once to find one concrete state that the circuit can be in after the boot code runs, and then invoke the solver again to prove that the state must be equal to that single concrete state. This more efficient check is enabled by the ability to make intermediate solver queries to build up a final solver query, which is not possible in SymbiYosys.

With these setups, the SymbiYosys proof does not finish within a 12-hour timeout, while the `rtlv` proof finishes within 1.3 seconds. To demonstrate how SymbiYosys's verification time scales, we proved a related state-clearing property for varying numbers

```
is_deterministic(state):
  # extract all symbolic variables in state
  sym_vars = symbolics(state)

  # generate mapping of sym_vars to one possible
  # set of concrete values
  solution = solve(sym_vars)

  # evaluate state under this mapping
  # to generate concrete state
  concrete_state = evaluate(state, solution)

  # assert that state must equal the concrete
  # solution, i.e. if state is deterministic, it
  # *must* equal the solution 'concrete_state'
  verify(assert(state == concrete_state))
```

**Figure 3: Verification pseudocode using `rtlv`, verifying the property without instantiating two circuit states. The code first calls the solver to generate some possible concrete state and then verifies that the symbolic state must always equal this concrete state.**

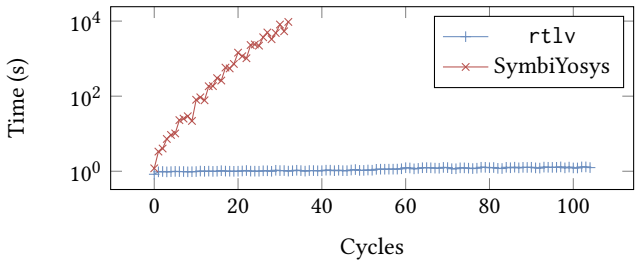


**Figure 4: Scalability of verifying deterministic start. SymbiYosys scales exponentially; results truncated after 32 cycles. `rtlv` scales linearly and takes 1.3 seconds to verify the property for 104 cycles.**

of cycles, showing that a subset of registers were cleared after the appropriate number of cycles. Figure 4 shows the runtime of verifying each property against the number of cycles it took to verify that property. This plot illustrates how SymbiYosys's runtime increases exponentially with the number of cycles, while `rtlv`'s runtime is relatively flat: in this particular case, `rtlv`'s time is linear in the number of cycles. Code for this experiment can be found at https://github.com/anishathalye/deterministic-start-benchmark.

This example shows that in order to verify properties such as deterministic start, `rtlv`'s approach doesn't suffer from the same scaling bottlenecks as SymbiYosys. The next section describes what these bottlenecks are and how `rtlv` eliminates them.

## 3 VERIFICATION APPROACH

`rtlv` verifies circuits using the Rosette solver-aided programming language, taking advantage of Rosette's symbolic execution approach in order to generate a more efficient low-level encoding of the property being verified as compared to existing tools like SymbiYosys.

### 3.1 SMT-based verification

Formal verification differs from traditional hardware verification: it involves *proving* that certain properties always hold, rather than checking that a circuit behaves as expected over a concrete (and often hand-designed) set of test vectors.

SMT-based formal verification, a popular approach for verifying hardware, works by encoding a model of the circuit and the property being proven into a boolean expression called an *SMT query*. Verification queries are expressed in the negative, meaning the query evaluates to true if the property is violated. In order to verify the property, the tool passes the query into an SMT solver like Z3 [5], which attempts to determine whether there is an assignment of values to the variables in the query that makes the query evaluate to true. If the solver determines the query is *unsatisfiable*, this *proves* the property, because there is no possible assignment that makes the query evaluate to true (i.e. violates the property).

Complex SMT queries result in bad performance or solver timeouts, so in order for a verification tool to efficiently prove a property, it must produce a "good" SMT encoding. In the case of properties that require executing for many cycles, `rtlv`'s approach, building on Rosette's symbolic execution, generally produces better SMT encodings than existing work.

### 3.2 Symbolic execution with Rosette

To enable reasoning about circuits using Rosette, `rtlv` includes a Verilog to Rosette compiler, which works as follows. First, a circuit's Verilog source is passed into the Yosys [18] synthesis tool, which generates an SMT-LIB representation of the circuit. A domain-specific language (DSL) provided by `rtlv`, called `#lang yosys`, then transforms Yosys's SMT-LIB output into Rosette code. Yosys's front-end only supports Verilog, so if the circuit is written in a different HDL, it must be converted into an equivalent Verilog representation. We use sv2v [13] for converting circuits written in SystemVerilog.

Both `rtlv` and SymbiYosys rely on Yosys's SMT-LIB backend for producing a circuit model for verification. The key distinction between the two approaches is *how* this model is used. As-is, Yosys's output describes the behavior of the circuit on each clock cycle as a transition relation. This is a function that takes in two circuit states—a current state and a next state—and returns a boolean indicating whether or not the next state can be reached by stepping the current state. Expressing the circuit transition as a relation allows SymbiYosys's Yosys-SMTBMC backend to encode execution into the solver query directly.

For example, suppose we want to prove that a circuit satisfying an initial predicate $I$, after $n$ cycles, satisfies some predicate $P$. Suppose that the transition relation between circuit states $s_i$ and $s_j$ is written as $T(s_i, s_j)$. For this task, SymbiYosys will produce an SMT query with variables $s_0, s_1, \ldots, s_n$ and the following assertion:

$$\text{assert}(I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \cdots \wedge T(s_{n-1}, s_n) \wedge \neg P(s_n))$$

Empirically, SMT solvers exhibit poor performance when reasoning about long chains of $T(s_0, s_1) \wedge \cdots \wedge T(s_{n-1}, s_n)$.

Instead of using the transition relation directly in the SMT encoding, `rtlv` transforms the transition relation into an imperative step function that allows for symbolic execution. This is always possible because we operate on a circuit with a single clock domain with all nondeterminism (such as don't-cares) resolved, so the behavior of the circuit is a total function. This transformation is performed by `rtlv`'s `#lang yosys` DSL, which transforms Yosys's SMT-LIB encoding of the circuit into Rosette code, producing a shallow embedding in Rosette through the use of Racket macros. The code defines:

- A Rosette struct that represents circuit state, which includes fields for all registers, memories, and current input values.
- The step function, which takes a circuit state and returns a new circuit state representing the result of running the circuit for one clock cycle.
- Functions for setting the inputs to a circuit and functions for getting the outputs from a circuit.

These definitions let us use Rosette to execute the circuit instead of encoding execution into the SMT query itself. At the very end of the execution, Rosette is able to construct a solver query that only considers the final circuit state, regardless of execution length, in contrast with the SymbiYosys encoding.

The example above would be encoded as follows. Let step($s$) be the step function. `rtlv` initializes $s_0$ to be a fresh symbolic variable, and then performs the following computation:

$$\begin{aligned} s_1 &= \text{step}(s_0) && \text{(Symbolic execution)} \\ s_2 &= \text{step}(s_1) \\ &\cdots \\ s_n &= \text{step}(s_{n-1}) \end{aligned}$$

$$\text{assert}(I(s_0) \wedge \neg P(s_n)) \qquad \text{(Solver query)}$$

In contrast to SymbiYosys's approach, the final query size does not inherently scale with the number of cycles of execution, and the query does not directly require the SMT solver to reason about the circuit's execution. Instead, the circuit's execution is computed in Racket/Rosette, which can take advantage of performing concrete computations directly in Racket as opposed to building up large terms for the solver to reason about. Thanks to Rosette's rewrite rules and `rtlv`'s performance hints (described in Section 4.1), such opportunities occur often and result in significant speedup.

`rtlv`'s approach is well suited to applications where much of the circuit state is concrete—for instance, applications modeling the execution of known software with concrete control flow. If the circuit state is primarily symbolic, each step will result in rapidly growing complexity in the circuit state's symbolic expressions, ultimately leading to a large final solver query (in this case, neither the SymbiYosys encoding nor the `rtlv` encoding will give good performance).

Deterministic start is a good example of a property well-suited to `rtlv`'s approach. The boot code is known, and its control flow is not dependent on unconstrained values, so the program counter is concrete on each cycle. In addition, by the nature of what the boot code does, the final circuit state will likely contain mostly or even solely concrete terms, because the entire circuit state becomes deterministic. Therefore, the verification runtime is dominated by symbolic execution time rather than the final solver query.

`rtlv` benefits from several design features in Rosette. The first is Rosette's hybrid symbolic execution model, which performs "type-driven state merging" [15] to merge values at control-flow joins, avoiding problems with path explosion found in traditional symbolic execution. In addition, Rosette has "rewrite rules", which are

heuristics to simplify symbolic expressions on the fly before they reach the solver.

As an example of how these two features can simplify symbolic execution, suppose a circuit has a 2-to-1 multiplexer, and consider a situation where each input is a concrete zero, but the select input is symbolic. A multiplexer represents the equivalent of a branch in software — a traditional symbolic execution system would separately evaluate a path for each possible value of the select input. However, Rosette's state-merging explores both branches and then merges their results into a single symbolic expression such as `(ite select 0 0)`. While SymbiYosys would have the solver reason directly about this term, Rosette's rewrite rules further simplifies this to `0`, since both branches are zero.

Another example of a rewrite rule simplification would be in instances where a circuit extracts a concrete bit from a concatenated concrete and symbolic value. For example, given a 5-bit symbolic `foo`, the expression `(extract 7 7 (concat 000 foo))` will be simplified by Rosette to `0`.

Rewrite rules give `rtlv` more efficient symbolic execution that results in smaller symbolic expressions in the final circuit state, resulting in smaller SMT queries.

## 4 CASE STUDY

As a case study, we used `rtlv` to verify a security property called output determinism for a complex SoC based on a subset of Google and lowRISC's OpenTitan [8]. This case study demonstrates how we applied `rtlv` to a more complex verification task by creating a circuit-agnostic property checker called `rtlv/shiva` that allowed us to implement peephole optimizations in a disciplined way. In addition, this case study shows that `rtlv` can be used to find violations of security properties in practice.

If a circuit satisfies output determinism, its outputs must not depend on data present in the circuit state prior to reset. Output determinism is implied by deterministic start: one way to ensure a circuit satisfies output determinism is by clearing all circuit state on reset, so that no data that was present in the circuit prior to reset remains in the circuit anymore. Therefore, both properties can be proven by modeling the execution of boot code and verifying that uninitialized data has been cleared from the circuit state. However, output determinism does not require that all circuit state is reset to deterministic values: it allows state to depend on input data. This makes checking this property more complex, since it involves tracking a set of "allowed dependencies" (i.e. all inputs received over the course of execution), and verifying that the circuit's state post-boot code execution only relies on these allowed dependencies.

We prove this property for a RISC-V SoC we call MicroTitan, which is based on a subset of an existing project called OpenTitan. Figure 5 shows a block diagram of the circuit. It includes the Ibex CPU, 8KB of ROM, 8KB of RAM, and UART, SPI device, and USB device peripherals. MicroTitan includes multiple clock domains that we verified separately, but this case study focuses on the "core clock domain." The core clock domain contains the Ibex CPU, memories, UART, and slices of the SPI and USB peripherals. To satisfy output determinism, the hardware in this clock domain must execute boot code to clear certain state, and to verify this property, we must model this boot code execution.

MicroTitan's core clock domain is more complex than the PicoRV32 CPU, and its boot code takes a correspondingly longer time

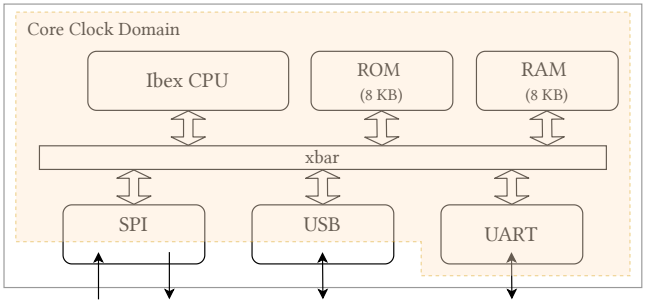


**Figure 5: A block diagram of MicroTitan, with its core clock domain highlighted.**

to execute. The bulk of this additional execution time comes from the uninitialized memories in MicroTitan that must be explicitly reset by loops in the boot code. Clearing the MicroTitan's uninitialized state takes 24,516 cycles of boot code execution, compared to only 104 for the PicoRV32. In addition, the MicroTitan circuit itself is more complex, consisting of around 4,300 flip-flops as compared to around 1,300 flip-flops in the PicoRV32. This means that solver queries about MicroTitan's state are likely to be more complex than queries about the PicoRV32.

### 4.1 Performance hints

In this case study, Rosette's symbolic execution system alone couldn't provide sufficient performance, since MicroTitan's state contains symbolic terms that grow rapidly over time and cannot be simplified by rewrite rules, which are heuristics and cannot be "complete." Therefore, we had to implement circuit-specific optimizations to make verification feasible. To do so in a disciplined way, we built a circuit-agnostic property checker called `rtlv/shiva` which takes in the Rosette circuit model (which includes components such as the step function and register names), executes the circuit based on the model provided, and returns whether or not the circuit satisfies output determinism. In addition, `rtlv/shiva` has a performance hint interface that allows a developer to suggest that the checker transform the circuit state in a certain way, generally to reduce the size of symbolic expressions and therefore simplify both symbolic execution and the final solver query.

A key idea is that `rtlv/shiva` is *sound*, and the hints are *untrusted*, meaning that no matter what hints a developer specifies, the tool will not erroneously say that a property holds when it does not. Supplying inadequate or incorrect hints can only harm performance or make the tool fail to prove the property. Performance hints cannot be misused to make the tool erroneously report "OK". This enables a workflow where the developer can freely experiment with performance hints without worrying about invalidating the proof.

The verification tool itself is considered trusted since we assume it enforces this property. By encapsulating the trusted code for transforming circuit state in a non-circuit-specific tool, the developer can apply performance hints as needed while maintaining high confidence in the proof's correctness. In addition, this separation allowed us to reuse `rtlv/shiva` for multiple circuits: we used this checker to verify output determinism for the MicroTitan as well as the PicoRV32.

| Hint | Description |
|---|---|
| `abstract [field-name]` | Checks if field only depends on allowed dependencies—if so, replaces it with a fresh symbolic (added to allowed dependencies). |
| `overapproximate [field-name]` | Replaces field with a fresh symbolic. |
| `abstract-or-overappprox-vector [field-name]` | For each entry of a memory field, abstracts it if it only depends on allowed dependencies, and otherwise overapproximates it. |
| `collect-garbage` | Forces GC run. |
| `concretize [field-name]` | Determines using a solver query if field only evaluates to a single concrete value—if so, replaces it with this concrete term. |
| `run-and-replace [list-of-field-names list-of-hints]` | Re-runs verification from initial state to current cycle, using the provided list of hints. At the end of this sub-execution, replaces the given fields in the main execution with their values from the secondary execution. |

**Table 1: Performance hints [arguments in brackets] implemented by `rtlv/shiva`.**

One of `rtlv/shiva`'s performance hints is `concretize`. This hint takes in the name of a field in the circuit state, checks if the field can only evaluate to a single concrete value by issuing a solver query, and replaces it with that concrete value if so. The solver query ensures that this hint cannot be misused to transform the circuit state in an incorrect way. Relying on the solver allows a hint like `concretize` to be more powerful than rewrite rules. As an example, consider the symbolic expression `(ite a 0 a)` [1]. This expression is always equal to `0`, since it returns `0` when `a` equals `1`, and `a` when `a` equals `0` itself. However, Rosette has no built-in rewrite rule for this. If `rtlv/shiva` is told to apply `concretize` to a field containing this expression, it will set the field to `0`.

Performance hints are also useful since they can take advantage of the specific property being verified, and therefore be more specific than general optimizations such as Rosette's rewrite rules. For example, `rtlv/shiva` supports a hint called `abstract` that determines if a field only depends on allowed dependencies, and, if so, replaces that field with a fresh symbolic value that itself gets added to the set of allowed dependencies. This is sound since we can always overapproximate a value (i.e., set it to a completely unconstrained fresh symbolic) and the dependency check maintains the invariant that values in the allowed dependencies set only depend on previous inputs. However, a symbolic value being an "allowed dependency" is only meaningful in the context of output determinism, so this hint is inherently property-specific. In addition to `concretize` and `abstract`, `rtlv/shiva` supports several more hints, described in Table 1.

Without performance hints, verification of MicroTitan would time out. One example of a critical performance hint is the use of `concretize` on the state register controlling the SPI peripheral's RX FIFO state machine. This hardware state machine is responsible for gathering data the SoC receives via SPI and writing it into a memory where it can be consumed by software. Since output determinism doesn't assume anything about SoC inputs while boot code is executing, all SPI inputs are represented as symbolic data. These inputs affect state transitions in the SPI state machine, resulting in the state machine control register itself becoming a symbolic term. In addition, since the new state on each cycle depends on the previous state, the size of the symbolic term representing this state grows rapidly each cycle if left unchecked.

However, it turns out that independent of inputs, after 150 cycles of boot code execution, this state machine returns to an idle state, and it does not leave this state for the remainder of the execution. We take advantage of this fact by applying a concretize hint to the state machine control register on cycle 150, which causes `rtlv/shiva` to verify that the register must be idle and then replace the symbolic term with a concrete term. Once back to the concrete idle state, the register is able to remain concrete for the rest of the execution.

In addition to performance hints, `rtlv/shiva` supports a general interface called `unsafe-custom-hint` for user code to directly implement transformations of the circuit state. This was useful for this case study, since it allowed us to implement a complex transformation on each cycle based on the relationship between two registers. The exact transformation was circuit-specific and could not be easily generalized, so it was best implemented in user verification code. Like a performance hint, the code that implemented the transformation performed auxiliary solver queries to ensure that the transformation was correct.

By taking advantage of `rtlv/shiva`'s performance hints, verification of output determinism for MicroTitan finishes in under 100 minutes on a machine with an Intel Core i7-5930K. The verification code is single-threaded.

## 4.2 Bugs found

The verification process helped us find and fix four violations of output determinism in MicroTitan across all its clock domains:

(1) The possibility of leaking data that was previously sent via SPI prior to reset.
(2) The possibility of leaking data that was previously received via SPI prior to reset.
(3) The possibility of leaking memory that was stored in the USB transmit/receive buffer prior to reset (it is unclear if this issue is exploitable, but we patched it anyways to simplify verification).
(4) The possibility of leaking data about the reset line's previous values.

This demonstrates that `rtlv` can successfully be used for bug-finding. In addition, this work resulted in an upstream contribution to OpenTitan [10].

Overall, this case study shows that `rtlv` can be used to write performant verification code for complex circuits and successfully catch violations of a sophisticated security property. More information about this case study can be found in [11], and the verification code can be found at https://github.com/nmoroze/kronos.

# 5 RELATED WORK

Much prior work has been done on hardware formal verification tools, and a lot of this work uses similar ideas to the ones used in `rtlv`. However, none of this prior work has fully tackled the problem of performantly verifying software execution on hardware,

[1] In actual Rosette code: `(if (bveq x (bv #b1 1)) (bv #b0 1) x)`

which requires reasoning about many cycles of execution, using a push-button approach.

### 5.1 End-to-end software/hardware verification

There is a long line of work in the end-to-end verification of systems, beginning in 1989 with the CLI Stack [3]. Other notable works include Verisoft [1] and the CakeML project's end-to-end verified system [9]. These works prove deep end-to-end correctness theorems (i.e. full functional correctness) about software running on hardware, but they rely on heavyweight techniques using interactive theorem provers. In contrast, `rtlv` proves simpler properties, but in a push-button style.

### 5.2 SymbiYosys

SymbiYosys [14] is a popular open-source hardware verification tool that can verify safety and liveness properties, but it has several limitations. Like many commercial tools, SymbiYosys verifies properties written using SystemVerilog Assertions, making it difficult to express complex properties.

SymbiYosys supports many solver backends, each with their own capabilities and trade-offs. Several of the backends, including the built-in Yosys-SMTBMC, verify properties using bounded model checking (BMC) [4], which involves unrolling a bounded number of circuit steps into one large SMT query. This query is then passed into an SMT solver such as Z3 to determine satisfiability.

Like `rtlv`, SymbiYosys uses the Yosys synthesis tool. However, as discussed in Section 3, SymbiYosys encodes execution directly into the solver query, making it less effective for reasoning about many cycles of execution than our Rosette-based symbolic execution approach.

### 5.3 Symbolic execution for finding exploits

Zhang and Sturton [19] describes a system for finding exploits of security vulnerabilities in hardware. Their system generates input traces that will take a processor from its reset state to an error state that violates a developer-defined security property. It recursively searches from a given error state backwards to reset, generating the input trace in reverse.

Like `rtlv`, this tool is based on symbolic execution. However, a downside of traditional symbolic execution is that it results in path explosion, since it involves forking execution at every possible branching point. In order to narrow the search space, Zhang and Sturton employ application-specific heuristics to help the search converge on the reset state. This works in this case, but Rosette's symbolic execution uses "type-driven state merging" to prevent path explosion in general [15], making it useful for efficiently verifying a wide range of properties. In addition, this work is limited to bug-hunting for violations of safety properties, and cannot be used to prove that properties hold.

### 5.4 Self-equivalence with don't-cares

Lee and Sakallah [7] presents a hardware verification framework called Averroes. As a case study, this work verifies a property called "self-equivalence with don't-cares" (or "SEQX") for a Cortex M0+ processor. SEQX is equivalent to output determinism as defined in Section 4, showing that Averroes can be used to verify such a property. However, our case study's target, the MicroTitan SoC, is more complex than the Cortex M0+ CPU, which contains only 41 bits of un-reset state. Proving the property in our case requires modeling execution of state-clearing boot code, while their work did not address code execution. Therefore, their framework is insufficient for systems such as MicroTitan that cannot satisfy output determinism or SEQX without boot code.

Goel and Sakallah [6] presents another formal verification framework called AVR, which is descended from prior work on Averroes. However, Averroes shares AVR's limitation of not supporting the modeling of code execution.

## 6 LIMITATIONS

`rtlv` is an effective approach for verifying hardware, but it has several limitations. `rtlv` has no way to bridge results proven with Rosette to verification tools like Coq for end-to-end proofs that require proving metatheory. For example, the case study described in Section 4 proves a property about an individual clock domain in MicroTitan, and while we have verified MicroTitan's other clock domains, we do not have a machine-checked, end-to-end proof that these individual properties imply a top-level output determinism property. Although we show this was adequate for catching violations, we cannot claim that the property was fully machine-verified.

There are also restrictions on the types of circuits that can be verified using `rtlv`. The first restriction is that the circuit may not contain combinational latches. We find this is not limiting for verifying designs that target FPGAs, since latches are generally considered bad practice in such designs and are therefore avoided [12]. In addition, `rtlv` does not support asynchronous resets or clocks that are derived from logic in Verilog—these features can either be removed or transformed into equivalent representations, or the circuit can be preprocessed using the Yosys `clk2fflogic` pass (which makes the circuit's Rosette model, and hence the resulting verification code, more complex).

A final limitation is that while `rtlv` can be used to verify properties by modeling software execution on hardware, this is limited to simple embedded applications. We expect `rtlv` can verify properties that require executing several hundred lines of C on microcontroller-level hardware, but we do not expect that the current design can verify code running on a full operating system with complex hardware.

## 7 CONCLUSION

This paper describes an approach called `rtlv` for reasoning about hardware using the Rosette solver-aided programming language. Our design choices and Rosette's capabilities lend `rtlv` the ability to performantly verify properties that require executing software on hardware for many cycles.

In addition, `rtlv`'s approach encourages the development of reusable circuit-agnostic property checkers that implement performance hints, which are critical for scaling verification to more complicated systems. Existing tools such as SymbiYosys do not allow for the use of this technique, since they do not provide direct access to the underlying SMT solver.

`rtlv`'s symbolic execution-based verification and performance hints enabled us to verify that a large hardware system, MicroTitan, satisfies a sophisticated security property, output determinism. This verification case study successfully found bugs, and demonstrates that `rtlv` is an effective approach for verifying such properties.

## REFERENCES


[1] E. Alkassar, M. A. Hillebrand, D. Leinenbach, N. W. Schirmer, and A. Starostin. The Verisoft approach to systems verification. In *Proceedings of the 2nd International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, Toronto, Ontario, Canada, Oct. 2008.

[2] A. Athalye, A. Belay, M. F. Kaashoek, R. Morris, and N. Zeldovich. Notary: A device for secure transaction approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, page 97–113, Huntsville, Ontario, Canada, Oct. 2019.

[3] W. R. Bevier, W. A. Hunt Jr., J. S. Moore, and W. D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, Dec. 1989.

[4] A. Biere. Bounded model checking. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 457–481. 2009.

[5] L. de Moura and N. 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, Mar.–Apr. 2008.

[6] A. Goel and K. Sakallah. AVR: Abstractly verifying reachability. In *Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 413–422, Apr. 2020.

[7] S. Lee and K. Sakallah. Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, pages 849–865, Vienna, Austria, July 2014.

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

[9] A. Lööw, R. Kumar, Y. K. Tan, M. O. Myreen, M. Norrish, O. Abrahamsson, and A. Fox. Verified compilation on a verified processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 1041–1053, Phoenix, AZ, June 2019.

[10] N. Moroze. [prim_fifo_sync] Make FIFO output zero when empty. https://github.com/lowRISC/opentitan/pull/2420.

[11] N. Moroze. Kronos: Verifying leak-free reset for a system-on-chip with multiple clock domains. Master's thesis, Massachusetts Institute of Technology, Jan. 2021.

[12] Nandland. Tutorial - what is a latch in an FPGA? https://www.nandland.com/articles/what-is-a-latch-fpga.html.

[13] Z. Snow. sv2v: SystemVerilog to Verilog. https://github.com/zachjs/sv2v, 2020.

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

[15] E. Torlak and R. 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.

[16] C. Wolf. Formal verification with SymbiYosys and Yosys-SMTBMC. URL http://www.clifford.at/papers/2017/smtbmc-sby/slides.pdf.

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

[18] C. Wolf. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/, 2020.

[19] R. Zhang and C. 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)*, page 1–9, Philadelphia, PA, 2018.