Formal Processor Modeling for Analyzing Safety and Security Properties

Benjamin Binder, Samira Ait Bensaid, Simon Tollec, Farhat Thabet, Mihail Asavoae, Mathieu Jan

Formal Processor Modeling for Analyzing Safety and Security Properties

Benjamin Binder, Samira Ait Bensaid, Simon Tollec, Farhat Thabet, Mihail Asavoae and Mathieu Jan
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

***Abstract*—Thanks to the emergence of open hardware initiatives, the exact behavior of the hardware design can be analyzed and combined with program representations to verify system-level safety and security properties. However, such formal verifications require the design of appropriate abstract models to scale with the complexity of the analyzed computational systems. In this paper, we compare the different needs when designing abstract processor models for the evaluation of timing predictability, a safety property, and for security assessments when considering fault injections. We also report how the process of building these abstract processor models could be automated.**

***Index Terms*—Formal Methods, Hardware Models, Safety and Security Properties**

I. Introduction

The design of complex computational systems, such as Cyber-Physical Systems (CPSs) or the Internet of Things (IoT), is facilitated by the emergence of open hardware initiatives [2]. Such initiatives propose software-like development workflows, from complex high-level Hardware Description Languages (HDLs) [4] down to circuits, while using sophisticated compilation chains. These approaches favor the availability of hardware designs, which can thus be used as (detailed) golden models, replacing the standard manual reference where only certain design details are provided.

These CPSs and the IoT are often subjected to safety and/or security requirements. Ensuring those requirements can be done with various degrees of confidence, from informal argumentation to formal verification of properties. When using the latter approach, the formal verification of software and hardware parts of a system is generally performed as separate activities and focuses mostly on functional correctness. For example, a binary representation of an application can be executed with formal Instruction Set Architecture (ISA) semantics [1], while hardware designs can be composed from formally proved modules [11]. However, in the former there is no hardware model while the latter ignores the software level. The availability of hardware designs, combined with ISA application representations, enables new possibilities in the formal verification of safety and security properties at system level, in particular by integrating non-functional characteristics. We restrict the notion of hardware design to that of processor design and we consider timing to be the non-functional characteristic of interest, when dealing with safety properties.

In this paper, we study how to design formal processor models so as to prove safety and security properties, such as timing predictability and assessments of Fault-Injection (FI). We present several instances of a general workflow, shown in Fig. 1, to address various needs as designing specialized abstractions. Due to the availability of a processor design, different abstract processor models can be derived while analyzing its code, both data and control paths (i.e., model generation in Fig. 1). In the case of security assessments under FI, such an abstraction requires a high temporal and spatial accuracy on certain parts of the design while non-important aspects of the system can be removed. Both control and data path are relevant, since security properties should follow how instructions (i.e., control) are correctly (or not) executed on the processor (i.e., data). In the case of timing predictability evaluations, the datapath is abstracted to black-box, but cycle-accurate, pipeline stages imposing timing constraints to the

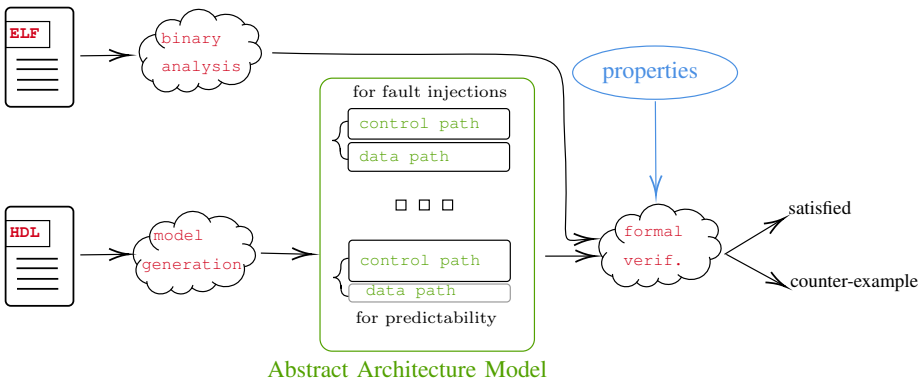


Figure 1: General workflow for the verification of system-level safety and security properties.

instruction flow. The actual system executions capture how a given binary code is executed on the processor design and hence, we extract a binary representation (e.g., a control-flow graph or explicit traces) to represent these sequences of instructions (i.e., binary analysis in Fig. 1) in the formal verification step.

The remainder of this paper is organized as follows. Section II describes related work on formal modeling of micro-architectures. In Section III, we show how to design abstract processor models to assess timing predictability and we apply this approach over an out-of-order pipeline to detect timing anomalies. In Section IV, we present how to design abstract processor models to identify exploitable fault-injection points and we illustrate this process over a RISC-V processor. In Section V, we look how to automate the construction of formal hardware models, using underlying hardware compilation chains. Finally, Section VI concludes the paper.

II. Related Work

The literature around modeling real-time systems often elaborates results using a pen and paper approach, without relying on formal and executable models—necessary in the automatic verification of properties. There are several exceptions, for example the framework PROSA [10], which focuses on on the correctness of schedulability analyses (and not microarchitectural modeling). The SIC pipeline is accompanied with a formal semantics based on a detailed transition system but that is not intended to execute in a verification methodology [18]. Even when formal models are assumed for the automatic verification, few modeling details are provided [15]. Model checking has been used in order to estimate the worst-case execution time (WCET), however on in-order architectures [14], [31]. A similar approach to ours for verifying a specific safety property by model checking has already been proposed [3]. In the present work, we nevertheless adopt a more generic viewpoint, e.g., letting the number of resources be provided as parameters. A similar architecture template (cf. Fig. 2) is used in [29], however together with an analytical approach.

Formally verifying FI-related security properties has been the subject of several works, in general at the ISA level. The modeling of the program execution has allowed to explore all the possible effects that certain FIs, such as register corruption, may have on program execution [7], [37], [16]. Nevertheless, recent results [17], [26] have shown the interest of considering the microarchitectural behavior to find FI points ignored by a strictly ISA-based study. This also helps to limit spurious vulnerabilities (i.e., false positives) that are not feasible in practice. However, to the best of our knowledge, no work has combined formal modeling of hardware and the verification of FI-related security properties. In the scope of general security properties, formally verifying hardware/software co-designs has been the subject of numerous works, such as [34]. However, they mainly focus on functional properties, such as the correctness of updating ISA-level states of processors [19] or detecting bugs in specific hardware components synthesized on FPGA [33]. Note that safety properties can also target functional failures [13], which share similar modeling requirements with our security-assessment approach.

Generating formal models from hardware designs is an extensively studied problem [24], [22], [32], [28], [20]. However, all consider Verilog or VHDL as input hardware languages, while we start from a high-level design language that poses different problems when generating abstract processor models specific to the targeted properties. This is made possible thanks to the high-level hardware compilation frameworks that are currently being built (see [4], [39], [36] to cite a few).

III. Abstract Processor Models for Timing Predictability

In this section, we target the verification of non-functional, safety properties, namely relative to the processor timing behavior. We first address the general modeling needs for timing predictability, then we exemplify this modeling approach through an Out-of-Order (OoO) template, and finally we verify a particular safety property on the OoO model.

A. Abstract Modeling for Timing Predictability

A suitable model necessarily integrates both hardware and software features, whose combination characterizes the system and in particular its non-functional timing characteristics. The properties are not correlated to the functional complexity of architectures, which materializes into the datapath. More precisely, we do not need to consider the functional aspects beyond their impact on the pipeline-level timing behavior. Instead, we need to develop an abstract formal model of the processor focusing on the instruction progress—the software characteristics—through the successive pipeline stages of the processor—the hardware characteristics. On the contrary, pipeline-level models are required since pipeline stages are essential to the cycle-accurate timing behavior, which enables to observe external events, e.g., the full completion of an instruction.

Our abstraction thus needs to precisely delimit the pipeline stages from the datapath of the hardware architecture, and to extract the control signals that impact the timing behavior of the control path according to the pipeline stalling logic. We also need to map at any time instructions onto the identified stages that process them, from the input program—this is the combination of the hardware and software specifications. Finally, our model also requires explicitly capturing (absolute) time, in view of verifying properties.

B. Application to an Out-of-Order Pipeline Template

Hereafter, we exemplify the main features of a suitable model for the verification of timing-predictability properties, from a simplified standard OoO-pipeline template, shown in Fig. 2. We present an overview of this template designed for the detection of timing anomalies [6]. We intend to provide formal-modeling details below. This template consists of stages that process instructions in the same order as specified in the

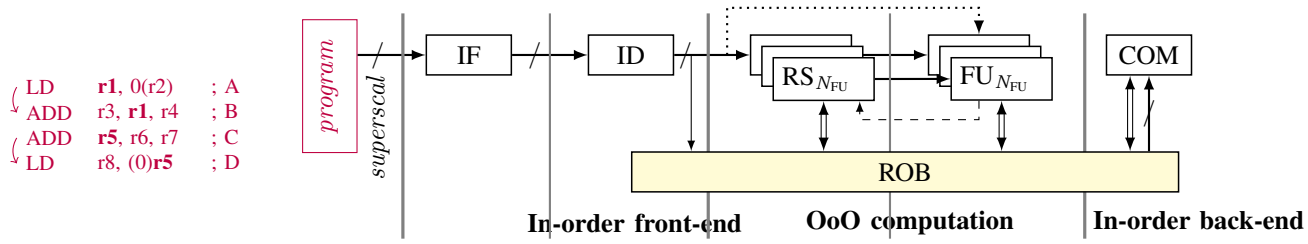


Figure 2: Representative hardware template of an OoO pipeline based on Tomasulo's algorithm. The pipeline has N_{FU} functional units and is able to fetch, decode and commit *superscal* instructions per cycle from a software specification (*program*) [6].

program (i.e., IF, ID and COM), namely the in-order front-end and the in-order back-end, as well as of Functional Units (FU) associated with Reservation Stations (RS) and a Reorder-Buffer (ROB) in order to implement Tomasulo's algorithm [40] for OoO computation. Instructions are dispatched to the ROB that keeps track of their status (pending/completed/committed) ($\longleftrightarrow$), which is used to schedule instructions to the FUs taking into account data dependencies and to later commit instructions in-order. Results from the FUs are bypassed to all RSs ($\dashrightarrow$), which essentially avoids waiting for the ROB update and allows back-to-back executions.

We encode the template introduced above into a formal specification (written in the TLA+ modeling and verification language [25]). In our abstraction, pipeline stages do not have side effects, such as a write to the memory or the register file. We consider multi-cycle instructions that thus may cause stalling. The pipeline timing behavior depends on the number of units for each stage (cf. *superscal* and N_{FU} in Fig. 2), on the program dependencies that clearly restrict OoO computations, and, when needed, on the mere information of the required computation clock cycles.

1) Abstract Datapath and Computations: The specification relies on hardware parameters, i.e., *superscal* and N_{FU}, which allow to represent a particular version of the pipeline template by fixing its abstract datapath. We define a state variable for each pipeline stage ($\_IF$, $\_ID$, $\_RS$, $\_FU$ and $\_COM$), which notably contains the instructions that are currently processed. The specification also relies on a software-execution parameter, i.e., *program*, which specifies the input instruction sequence with increasing addresses[1] associated with information about the mapping onto the hardware architecture. This information originates from the analysis of the concrete program: each instruction embeds the set of admissible functional units (as an abstraction of the functional instruction class), as well as the set of possible latencies related to timing-variable stages (as an abstraction of the intended computations). The variables related to a timing-variable stage also contain the current elapsed latency (i.e., a counter) and the total required latency in the stage (assigned from the *program* input parameter). The memory is not explicitly modeled, but the IF stage and the FUs instead feature a variable timing behavior, since for instance they both may perform memory accesses resulting either in an instruction/data-cache hit or miss. The register file is not modeled either, but only the (Read-After-Write) data dependencies ($\downarrow$), which are explicitly encoded in *program*. The resulting abstract specification allows for all the *behaviors*, i.e., series of states, that are concretely made possible by different initial hardware states (e.g., the initial cache content), considering the execution of the input instruction sequence on the target architecture. It remains to actually make instruction classes progress through the pipeline, i.e., to encode the control path from the established datapath and the execution information.

2) Timing-Oriented Modeling of the Control Path: The template specification is a *transition system* (TS) characterized by an *initial-state predicate* and a *next-state relation* built from actions, namely transition predicates relating the values of variables in the current state (e.g., x) to their values in the next state (x'). We consider that the pipeline is initially empty of any instruction (cf. the datapath state variables). Transitions model the control path, which entails changes in the datapath state. In order to get a cycle-accurate abstraction of the control path, a transition models one clock cycle, where each stage processing is modeled by an action involving a datapath variable. The additional state variable $currCycle$ is a counter modeling the absolute time ($currCycle' = currCycle + 1$ as long as the input sequence has not fully executed). Finally, the $prog$ state variable is a record monitoring the execution, with a field ($rest$) containing the remaining instructions (not fetched yet) from *program* and a field ($exec$) modeling the ROB. Each of these fields are sequences of instructions, where the instructions are nested records, e.g., with Booleans $completed$, $committed$ for the instruction status in the ROB.

We now illustrate how the abstract datapath state is used in order to accurately model the control path, by focusing on one of the most critical, OoO-specific elements of the control path. Modeling the scheduling of instructions to FUs requires selecting the next pending instruction. It relies on the instruction status in the ROB.

a) Prerequisite Operator: Let us define an operator $Exec$ used in order to specify the next-state relation. It returns the set of the ROB indexes of the instructions that will have already completed in the next cycle.

$$NxtFUBusy(i) \triangleq \_FU[i].currLat < \_FU[i].baseLat$$

$$Exec \triangleq \{i \in 1\,..\,Len(prog.exec) :$$

[1]We exclude branch instructions, thus focusing on one program path.

$$\lor\ prog.exec[i].completed$$
$$\lor \exists j \in 1\,..\,N_{FU} : prog.exec[i].PC = \_FU[j].PC \land \neg NxtFUBusy(j)\}$$

The operator *Exec* returns the set of indexes in the range of the current ROB (first line of the definition) s.t. the relevant Boolean field (*completed*) of the corresponding instructions ($exec[i]$) is set (first disjunct) or a back-to-back execution is possible (second disjunct). In the latter case, the instruction itself (PC field) is currently handled by one of the FUs, i.e., it is the instruction of the j-th element of the $\_FU$ state variable (penultimate line), and the instruction in this FU is to leave the FU in the next cycle (last line). Indeed, the $NxtFUBusy(i)$ operator uses the information about latencies contained in the $\_FU$ variable to determine whether the instruction currently handled by a given FU should remain in the FU in the next cycle and, hence, cause a pipeline stalling. This operator compares the current latency $currLat$ of the i-th FU with the total required latency $baseLat$. The operator *Exec* is used to update the ROB field *exec* of *prog* in each cycle.

b) Scheduling on the FU*s:* Based on the operator *Exec*, we can now specify the very scheduling of instructions to the FUs. The operator $NxtFU(i)$ returns the instruction that is to be scheduled to the i-th FU in the next cycle, or a special instruction *empty* that models the absence of an instruction:

$$\begin{aligned} NxtFU(i) \triangleq\ & \text{IF } NxtFUBusy(i) \text{ THEN } empty \\ & \text{ELSE LET } minReady \triangleq Min(\{x.pc : x \in \\ & \quad \{y \in \_RS[i] \cup FURout(i) : \\ & \quad\ \forall z \in y.dep : z \in Exec\}\}) \text{IN} \\ & \quad \text{IF } minReady = 0 \text{ THEN } empty \\ & \text{ELSE CHOOSE } x \in \_RS[i] \cup FURout(i) : x.pc = minReady \end{aligned}$$

In the case that the i-th FU does not suffer stalling and thus may accept a new instruction in the next cycle (i.e., $NxtFUBusy(i)$ evaluates false line 1), we define a local operator *minReady* (lines 2-4) that determines the *address* (*pc* field from the *program* input parameter) of the relevant instruction among the candidate instructions. If this instruction exists (0 is the conventional address of the *empty* instruction; penultimate line), we select (TLA+ CHOOSE operator) the *instruction* itself whose address have been determined by the local operator *minReady* (last line). *minReady* implements an age-ordered policy that selects the oldest instruction whose all dependencies are satisfied (or will be in the next cycle). It is based on the assumption that older, preceding instructions in the program order, have smaller addresses (see above). It is also based on the operator $FURout(i)$ (not detailed) providing the set of the currently decoded instructions (in the ID stage) that have *actually* been assigned the FU under consideration. This is trivial when the decoded instructions have only one admissible FU and it lies on an arbitrary choice otherwise. Consequently, *minReady* selects the smallest address (line 2), from the instructions waiting in the associated RS or directly from the ID stage[2] (line 3), more precisely only those (line 4) whose all dependencies (*dep* field assigned from the *program* input parameter) will have been computed.

The issued instructions are removed from the RSs in accordance, while the non-issued decoded instructions are added for a later selection, the whole through simple set-theory operators. Each entry of the $\_RS$ variable (one per RS/FU) is updated under this consideration:

$$\_RS' = [i \in (1\,..\,N\_FU) \mapsto (\_RS[i] \cup FURouting(i)) \setminus \{NxtFU(i)\}]$$

Similarly, the $\_FU$ variable is updated using the $NxtFU(i)$ operator for each FU i.

C. Formal Verification of Timing Anomalies

As a case study, we introduce timing anomalies and show how to detect them in the context of a program execution on our OoO-pipeline template.

1) Timing Anomalies: Evaluating the timing behavior or predictability of CPSs is often based on the estimation of the worst-case execution time (WCET). Several methods allow to compute such a bound, e.g., testing-based methods [27], probabilistic methods [9] or static analysis [43]. In any case, these methods rely on specific assumptions addressing the absence of exhaustivity, which is impractical for complex designs. Certain execution phenomena, called Timing Anomalies (TAs), question those assumptions and may thus skew WCET analyses. As a consequence, it is essential to accurately detect the occurrences of TAs in the execution of an application on the target hardware architecture. Common OoO processors are known to present TAs [42]. We are to illustrate the detection of counter-intuitive TAs in our model, according to one family of formal definitions of TAs from the literature [15], [23], [8]. These definitions essentially consider that a TA occurs iff the (local) commit of a given instruction (A) is performed earlier in one execution behavior, say α (see Fig. 3), than in another behavior (β), whereas the commit of a later instruction in the program (D) is performed earlier in β than in α—in other words, a timing reversal in *commit* events.

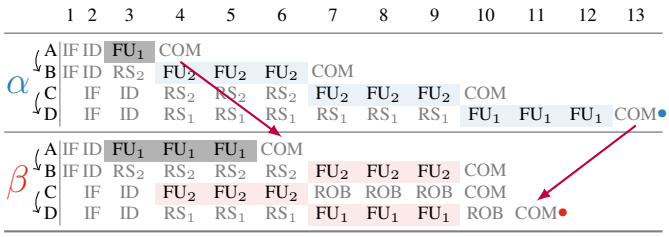


Figure 3: Execution traces showing the assignment to functional units (■/■/■) and a reversal of the order of commits (→) representing a counter-intuitive TA, obtained from the OoO template and the *program* of Fig. 2 (with $superscal = 2$ and $N_{FU} = 2$) [6].

[2]A decoded instruction is immediately issued to the FU if it is ready to execute and the related RS is empty (see ⋯► in Fig. 2).

2) Formalization of a TA Property: In order to address the verification/detection of TAs, we expand the template specification with a $comTime$ field in the ROB entries that keeps track of the instant ($currCycle$) of each commit event occurring during the execution. Besides, TAs are defined as a relation between *two* different executions of the same program. One behavior of the template specification is related to a *single* (arbitrary) execution of the program. We thus need to consider two behaviors at a time. To do so, we define a self-composition through two instances of the template specification in a main, higher-order specification module [5]. All state variables are duplicated and each instance manipulates its own set. Both instances share the input parameters, which guarantees that we consider the same program and the same version of the architecture datapath. However, they may progress at their own pace, according to their actual choices of FUs and latencies.

We endow the main specification module with a safety property related to the *absence* of TAs. We use the simplest form of safety properties, i.e., an invariant, in order to stipulate that (counter-intuitive) TAs may never occur while executing the input program on the considered version of the architecture. We may observe a TA only when *both* executions have completed, at least up to a certain instruction. The operator $ProgDone(n)$ (not detailed here) returns a Boolean indicating whether both executions have completed (at least) up to the n-th instruction of the input sequence. The operator $ComTime(ex, n)$ returns the value of the $comTime$ field for the n-th instruction of the (first or second) execution ex. Based on these operators, we now specify the property expressing the absence of TAs:

$$
\begin{aligned}
NoTA \triangleq\ & \forall\, k \in 1\,..\,Len(Program) - 1 : \\
& \forall\, n \in k+1\,..\,Len(Program) : \\
& \wedge\ ProgDone(n) \\
& \wedge\ ComTime(1,\, k) < ComTime(2,\, k) \\
& \implies\ ComTime(1,\, n) \leq ComTime(2,\, n)
\end{aligned}
$$

The execution of the input sequence on the underlying architecture exhibits no TA iff, for any instruction k (line 1) and for any *subsequent* instruction n (line 2), it holds that if:
a) the execution is completed up to the considered instructions in both instances under consideration (line 3) and
b) the (local) commit ordering for instruction k is s.t. the first instance (e.g., α) precedes the second one (e.g., β) (line 4),
then the commit ordering is the same for the subsequent instruction n (line 5). Note that both instances are totally interchangeable. That justifies the fact that we fix a priori the roles of each in the property, namely their commit ordering.

3) Verification Example: We verify the property $NoTA$ by model checking. Consequently, the violation of the property indicates the *existence* of a TA, and the provided counter-example is a TA scenario. Let us consider the verification with $superscal = 2$, $N_{\text{FU}} = 2$ and $program$ describing the input sequence of Fig. 2 annotated with the data dependencies ($\downarrow$), the set of admissible FUs for the instructions (respectively, the singletons $\{1\}$, $\{2\}$, $\{2\}$ and $\{1\}$) and the possible latencies ($\{1\}$, i.e., no instruction cache miss,[3] for each instruction in the IF stage, and, respectively, $\{1, 3\}$, $\{3\}$, $\{3\}$ and $\{3\}$ in the FUs, where A experiences a variable latency). The model checker signals that the property is violated, thus indicating the *detection of a TA* while executing the input program on the OoO architecture. The reported counter-example is graphically represented here in Fig. 3.

IV. Abstract Processor Models for Fault Injections

In this section, we target the formal verification of security properties under Fault-Injection (FI) attacks.

A. Modeling Requirements for FI Security Assessment

FIs consist in applying an abnormal physical stress to the hardware to modify the behavior of the microelectronics. This leads to the appearance of incorrect values called *faults* in the micro-architecture as detailed by Yuce et al. [45]. These faulty values can be recovered or propagated through the processor circuit and lead to observable effects at the ISA or software level. For instance, this can result in the execution of a casual instruction, reading or writing to a wrong address in memory [30]. The observable effects of the faults can then be exploited by an adversary. To comprehensively locate and characterize FI-based vulnerabilities, developing a formal model of the processor helps to identify which FI attacks—targeting the hardware—defeat a given security requirement—often based on the software. Such a suitable processor model requires four elements that we now describe.
1) The *hardware model* of the processor provides a complete representation of both the combinatorial and sequential logics constituting a processor design.
2) The *software model* describes the sequence of instructions, represented in a binary form and performed on the hardware.
3) The *fault model* indicates how the physical attack interferes with the hardware model. A fault model has three dimensions: *temporal*, *spatial* and *effect*. The *temporal* dimension specifies the targeted clock cycles by the attack. The *spatial* dimension describes which signals can be modified by the fault. Finally, the *effect* dimension defines which values can be applied to the faulty signals.
4) The *security property* is necessary to identify if a FI attack can lead to new vulnerabilities by encoding the expected software behavior.
Compared to the modeling requirements described in Sec. III, the assessment of FI attacks requires not only a cycle-accurate but also a bit-accurate hardware model. This is necessary to accurately propagate the effects of the faulty signals into the hardware model. Note that various fault models can be assessed on formal models involving hardware, software and security properties. Hereafter, only transient faults that do not permanently damage the processor are considered—i.e., when the clock cycle leaves the temporal window specified in the

[3] Consider an instruction scratchpad for instance.

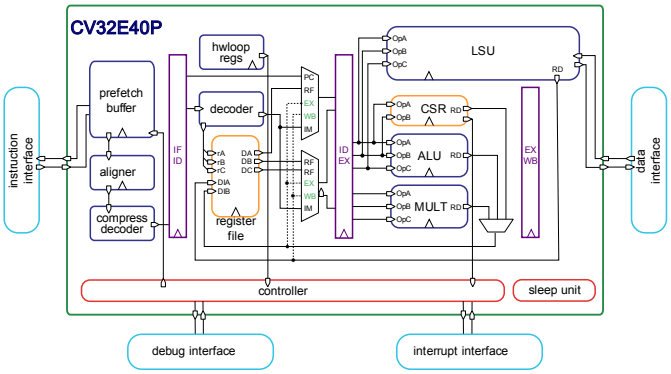
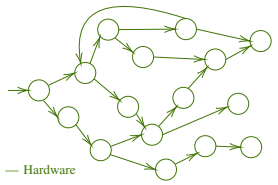
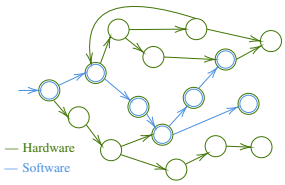


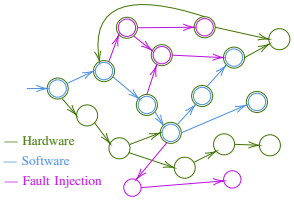
Figure 4: CV32E40P block diagram.



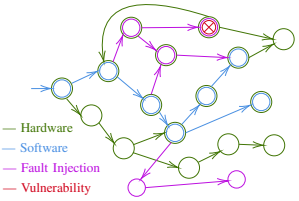
(a) Model of the hardware



(b) Restrict the model with SW execution



(c) Add new states and transitions due to FI



(d) Specify the vulnerability state

Figure 5: Modeling steps for FI vulnerability evaluation.

fault model, the targeted signals are no longer under the control of the fault model.

B. FI-Assessment Process of an In-Order RISC-V Processor

We now illustrate how to implement these four elements over an in-order RISC-V processor and simplify them to formally verify FI-based vulnerability.

1) Hardware Design: The CV32E40P is a 32-bit processor intended for light and embedded use. It has a 4-stage pipeline (IF, ID, EX, WB) and the RTL implementation in the SystemVerilog language is provided by the OpenHW Group [35]. Fig. 4 shows the block diagram containing the main modules and their interconnections. From the hardware-design description, a formal Satisfiability Modulo Theories (SMT) model is produced by relying on the open-source synthesis tool Yosys [44]. This model is represented as a classical transition system and is illustrated (in green) in Fig. 5a. Each state of this transition system contains the values of the memory elements included in the micro-architecture while the transitions are governed by the combinatorial logic of the design.

2) Specified Software: Fig. 5b shows (in blue) how the specified program is used to restrict the model to some execution paths. Since the program behavior may depend on the input data, multiple execution paths can be explored in the model through the use of symbolic variables.

Listing 1: Example of a (bit-reset) fault model at RTL level.

```
1 state 10:90
2 assume [fw_mux] bit-reset
3 count 1
```

3) Fault Model: By injecting faults into the system, the program behaves differently over the micro-architecture. Such a fault injection is specified into the formal model by defining its temporal and spatial location and its effect. As an example, Listing 1 describes a bit-reset fault model, where the `fw_mux` bits are set to 0 (line 2) between cycles 10 and 90 (line 1). `count 1` (line 3) specifies that only one fault injection is allowed on this time interval. Fig. 5c illustrates the faulty behaviors by adding (purple) transitions into the model towards the (purple) states, which were previously not reachable by the program or not allowed by the hardware.

4) Security Property: Security requirements can be defined by expressing a property on the states of the model. Fig. 5d illustrates this additional information by considering a (red) state to be faulty. A model checker is then used to verify if the property holds or is violated, and, in this latter case, counter-example(s) are provided. Thereafter, the security property is intended to control the instructions passing through the pipeline. This analysis ensures that secure data are never accessed or that a given instruction, whose semantics indicate successful authentication at the program level for instance, is never executed.

Next, we present several simplifications to efficiently handle the model that we have previously described.

a) Hardware Reduction: The security property only checks the values taken by a restricted number of state variables. The hardware model can thus be reduced by keeping only the necessary ones [12]. Whether variables are useful is determined iteratively by observing if they have an influence on the property to be verified. When performed manually, this technique can also remove entire modules from a hardware design if they are not involved in the faulty behaviors of the program. Regarding the CV32E40P use case, the Control and Status Register (CSR) module—which manages privileged execution modes [41], performance statistics, and interruption and debugging mechanisms—can safely be removed as its behavior does not interfere with the specified security property. Table I shows the proportion of variables saved in the formal model of the CV32E40P processor when applying this optimization. Even if the CSR does not represent a large part of the netlist, its deletion considerably reduces the model size because of the memory features that it contains.

The model can be further simplified by adding constraints on the environment. The CV32E40P micro-architecture has memory elements (e.g., flip-flops providing a large range of

Variables	Netlist			Formal Model
	Wire bits	Logic cells	of which Flip-Flops	SMT-functions
Quantity	11903	374	18	110033
Proportion	6.6%	4.5%	10%	32%

Table I: Number of variables saved both at the RTL (Netlist) and formal model levels when removing the CSR module.

initial states) that we can constrain. External modules can also interact with the processor via interrupts or debugging signals. Since these demands are not part of the model but can influence the security property, they must be controlled.

b) Software Reduction: At the software level, some instructions are not relevant regarding the security property and unused functions are embedded within the program due to the library linking process. A static analysis allows us to considerably limit the size of the input program used for the software model by removing these extra details. We restricted our models to execute an assembly program with 200 instructions—possibly with loops and unconstrained data input. This helps to reduce the number of states to be explored and their size in the formal model.

C. Formal Verification of FI-based Vulnerability

We now illustrate the capability of our abstract formal model to detect hardware vulnerabilities when verifying a specific security property. Listing 2 shows a security property that forbids the execution of the instruction, `0xfd5ff06f` in hexadecimal, allowing a secure authentication. The program does not normally allow this behavior, but it can be enabled by FIs.

Listing 2: An example of a security property.

```
assert (distinct [instr_id] 0xfd5ff06f)
```

Our fault model consists of a single bit-flip (bit-inversion) attack during one clock cycle of the program execution. We arbitrarily restrict the exploration of FI locations to the signals contained in the ID stage.

Model checking identifies several injection points on the micro-architecture, all due to the forwarding mechanism. Laurent et al. already point out [26] that it is possible to modify the forwarding control signal in order to recover values from EX and WB stages. These values can then be used as new operands, allowing, for example, to fool the comparison instructions. By modifying the forwarding behavior of the CV32E40P (depicted in Fig. 4 by a multiplexer in the ID stage) with a simple bit flip, the program execution reaches the vulnerable instruction indicated in Listing 2.

V. Towards Automated Extraction of Abstract Processor Models

The formal models described in the previous sections are all based on the accurate representation of specific details of the processor microarchitecture. Ideally, these models should be automatically derived from HDL processor designs. In this section, we present how we can take advantage of hardware compilation frameworks to ease the generation of abstract pipeline-stage-level models.

A. Hardware Compilation Framework: Chisel/FIRRTL

High-level design languages and the associated compilation chains enable the use of highly parameterized generators, domain-specific language constructs and advance module systems to facilitate the hardware design [4], [39], [36]. These compilation chains rely on several transformation passes to optimize the HDL (Verilog or VHDL) code that they generate for a later use as input in classical commercial (FPGA or ASIC) hardware-design flows. As in software compilers, hardware designers can also insert specific transformation passes in those compilation chains to manipulate the designs. This enables to deploy, within these chains, analyses to automatically construct abstract processor models. For instance, when targeting timing predictability, a transformation would mainly focus on the sequential logic to generate pipeline-level models, while for FI the whole design would initially be extracted, before abstractions are applied over the generated models.

We select the Chisel/FIRRTL hardware-compilation toolchain [4], [21] to illustrate this idea.Chisel (Constructing Hardware In a Scala Embedded Language) [4] is an open-source hardware-construction Domain-Specific Language (DSL) embedded in the Scala programming language. Adding hardware construction primitives to the Scala language allows the designers to write parameterized circuit generators, while using object-oriented and functional programming features to design circuits. Chisel emits synthesizable Verilog through an intermediate representation called FIRRTL [21], which stands for Flexible Intermediate Representation for RTL. Chisel thus constitutes the frontend part of the toolchain, FIRRTL, the middle-end from which all Scala-related hardware generators have been executed, and finally Verilog as a backend.

FIRRTL comes with different Intermediate Representations (IR), called forms. Each form uses a smaller, stricter and simpler subset of the Chisel language features and defines different transformations to generate the next (lower) form. Compiling FIRRTL to Verilog is implemented as a set of passes that implement optimizations, such as constant folding or dead-code elimination. A so-called *high form* supports the Chisel high-level constructs such as vector types, bundle types, and conditional statements. These constructs are replaced by a set of low-level features, resembling a structured netlist that simplifies its translation to Verilog, in the lowest form named *low form*. Which one of these forms is the most appropriate one to generate abstract processor models? A lower form ensures an easier equivalence to the actual pipeline circuits and is thus mandatory for fault-injection assessment (Sec. IV), while a higher-level form facilitates the integration of complex properties such as timing predictability (Sec. III).

We now illustrate these differences in the high and low forms, through a very simple example presented by Listing 3. This example defines two registers, `reg1` and `reg2`, at lines 1 and 2. Both registers are initialized on reset using the

RegInit construct. Note that the reset and clock signals are implicit in Chisel (but can be explicit in the FIRRTL forms). Finally, both registers are updated at lines 3 and 4, but for register reg2 this update depends on the value of the cond variable (actual value not defined to simplify) and is thus performed within a Chisel when construct (line 4).

Listing 3: A simple Chisel code.

```
1 val reg1 = RegInit(0.U(4.W))
2 val reg2 = RegInit(0.U(4.W))
3 reg1 := value
4 when (cond) { reg2 := reg1 }
```

Listings 4 and 5 describe, respectively, the FIRRTL high form and low form obtained from the Chisel Listing 3.

Listing 4: FIRRTL high form from Listing 3.

```
1 reg reg1 : UInt<4>, clock with :
2    reset => (reset, UInt<4>("h0"))
3 reg reg2 : UInt<4>, clock with :
4    reset => (reset, UInt<4>("h0"))
5 reg1 <=value
6 when cond :
7    reg2 <=reg1
```

Listing 5: FIRRTL low form from Listing 3.

```
1 reg reg1 : UInt<4>, clock with :
2    reset => (UInt<1>("h0"), reg1)
3 reg reg2 : UInt<4>, clock with :
4    reset => (UInt<1>("h0"), reg2)
5 node _GEN_0 = mux(cond, reg1, reg2)
6 reg1 <=mux(reset, UInt<4>("h0"), value)
7 reg2 <=mux(reset, UInt<4>("h0"), _GEN_0)
```

It can be noticed that the when statement remains in the high form (line 6, Listing 4), while it is translated into a multiplexer in the low form (line 5, Listing 5). Note that in Chisel, a condition can be translated into a set of multiplexers to implement a multi-variable condition. Directly translating a high form into a formal-specification language may thus discard these multiplexers, which can be source points for a fault-injection attack as presented in Sec. IV. Such a statement can however be safely translated into a formal statement when targeting a pipeline-stage abstract model, such as the one shown for the detection of TAs (Sec. III). This demonstrates the need to develop custom FIRRTL passes aimed at automatically generating abstract models, against the targeted properties.

B. Extracting Abstract Pipeline Models

The FIRRTL design is internally represented with an abstract-syntax-tree (AST) structure, where passes recursively visit nested elements to manipulate the AST. The FIRRTL AST consists of IR nodes represented by objects, each of which is a subclass of the following IR abstract classes: circuit, module, port, statement, expression or type. The registers and when-condition nodes, shown on the previous listings, are in the statement class. Each update of a register is represented by a connect node that is also part of the statement class.

We now present a custom pass that currently targets the automatic generation of abstract pipeline models specific to safety properties, as presented in Sec. III. This pass analyzes both the combinatorial and sequential logic of the pipeline datapaths in their high-form FIRRTL representations. It produces the pipeline stages, their order (thus forward edges between stages) and the backward edges between stages. Note that backward edges do not only correspond to the processor data-forwarding mechanisms between stages, e.g., classically from the write-back/memory to the execute stages, but also for instance simply the update of the program counter.

To achieve this, Chisel registers must first be identified at the FIRRTL level and their dependencies analyzed.Then, assigning a pipeline stage to the identified registers starts from an arbitrary specified[4] register, to be placed in the first pipeline stage. Two successive explorations of the set of identified registers are performed. The first exploration aims to assign pipeline stages based on two rules relying on register dependencies: 1) when only a single forward link between a source and a destination register exists, assign to the destination register the immediately following stage of the source register, and 2) when a destination register has several (already assigned) source registers, assign to the destination register the immediately following stage of the register having the minimal depth in the pipeline. This latter rule detects any forwarding mechanism within a pipeline. The second exploration relies on a heuristic based on the idea that a designer groups the update of register located in a same pipeline stage within a same conditional block.

C. Case Study: a RISC-V-based Processor

We now illustrate how our pass analyzes a RISC-V processor, called KyogenRV [38], so as to extract an abstract pipeline model. KyogenRV is an open-source 5-stage pipeline processor (IF, ID, EX, MEM, WB) targeting Intel FPGAs and developed for academic purposes. We focus on the top module of the pipeline, which has 60 registers. We specify the if_pc register to our pass as being located in the first stage of this pipeline. Our pass then automatically computes the dependencies between these registers, which are made of 46 edges, assigns a pipeline stage to each register, and outputs the abstract pipeline model. Fig. 6 shows a subset of the identified registers and their dependencies (omitting the combinatorial circuitry). Registers are represented by blue boxes, while red boxes represents Chisel instances of modules embedded within the currently analyzed Chisel module. Our pass correctly identifies the 5 stages, with forwarding mechanisms from the WB and the MEM to the EX stage. On the Fig. 6, the abstract pipeline model corresponds to the dashed boxes only, one for each stage, and associated edges are not shown for readability reason. It is thus similar to the one used for the detection of TAs (Sec. III) with: 1) a single forward edge kept between each stage, and 2) the green edges (potentially merged) implementing the forwarding mechanism corresponding to the

[4]By the hardware designer for instance.

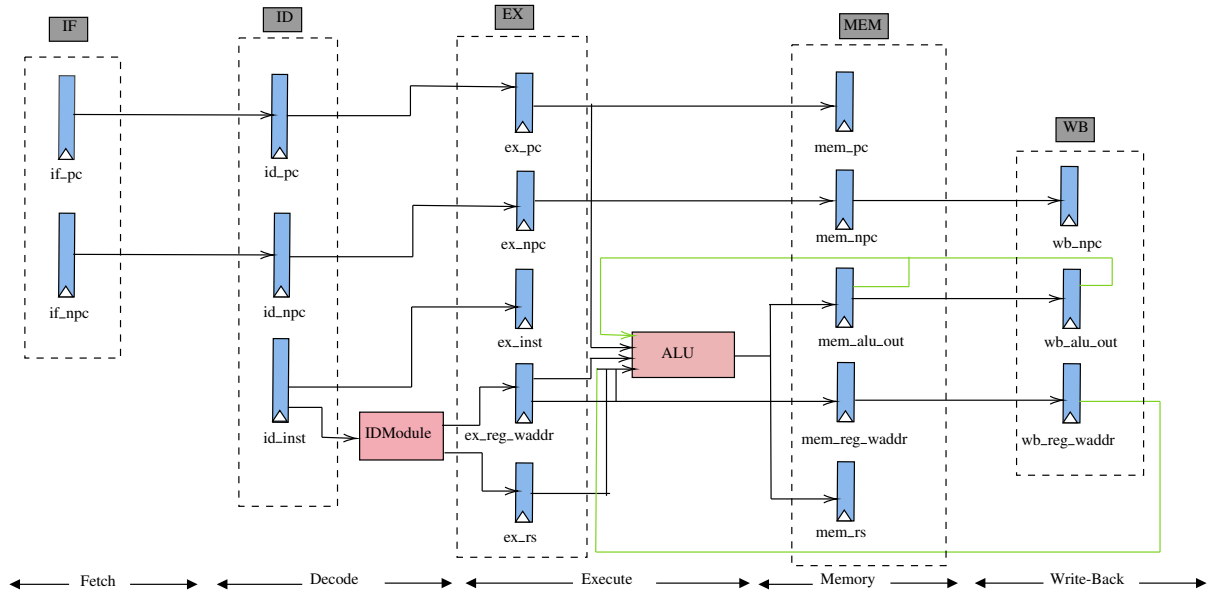


Figure 6: Representation of the extracted registers and (part) of the abstract pipeline model of the KyogenRV processor.

dashed edge (- -▸) of Fig. 2, which enables a back-to-back execution of instructions over the FUs.

Listing 6 presents a subset of the data forwarding implemented from the WB and MEM towards the EX stage. The Chisel wire `ex_reg_rs1_bypass` is updated from the output of the `mem_alu_out` register (line 5), located in the MEM stage, or from the `wb_alu_out` register (line 6), located in the WB stage, through the Chisel MuxCase construction. In the FIRRTL low form the MuxCase is translated into a cascade of multiplexers. This forwarding corresponds to the green arrows, shown in Fig. 6, from the registers `wb_alu_out` and `mem_alu_out` to the input of the ALU red box. The other green arrow, between the register `wb_reg_waddr` and the input of the ALU, is part of the check to detect the need for forwarding a value (not shown in Listing 6).

Listing 6: Forwarding in KyogenRV 5-stage.

```
1  val ex_reg_rs1_bypass = Wire(UInt(32.W))
2
3  /* Ci, i = 1..2 - conjuncts of write enable
       ↪ and selection signals */
4  ex_reg_rs1_bypass := MuxCase(ex_rs(0), Seq(
5    (ex_reg_raddr(0) === mem_reg_waddr && C1)
       ↪ → mem_alu_out
6    (ex_reg_raddr(0) === wb_reg_waddr && C2)
       ↪ → wb_alu_out))
7  ...
8  when (C4 /* no stalling condition */) {
9    mem_rs(0) := ex_reg_rs1_bypass
10 }
```

VI. Conclusion and Future Work

We have presented the various needs in the formal modeling of pipeline processors to verify both safety and security properties of CPS or IoT systems. The safety property that we consider (detection of timing anomalies) requires an abstract pipeline model where only stages are visible and whose timing behavior is accurate, while the security property (identification of fault-injection points) requires a cycle- and bit-accurate model where useless either software or hardware parts of the considered system have been removed. Finally, we have shown how a high-level hardware compilation toolchain can ease the adaptation of automatically generated formal abstract models to such safety and security properties. We have reported on a custom pass to generate an abstract pipeline model (and to be used for the detection of timing anomalies).

As future work, we are currently investigating how to expand the definition of timing anomalies to other hardware resources such as caches or speculation mechanisms and verify the needs in formal modeling of these elements. We also plan to implement various abstraction strategies of the formal models to speedup the fault-injection assessment. Finally, we plan to expand our pass to be able to generate the different formal models needed for both the safety and security properties that we consider.

Acknowledgments. The authors would like to thanks Claire Pagetti from ONERA, France and to the anonymous reviewers for providing valuable feedback which helped us improve this work.

References

[1] Armstrong, A., Bauereiss, T., Campbell, B., Reid, A., Gray, K.E., Norton, R.M., Mundkur, P., Wassell, M., French, J., Pulte, C., Flur, S., Stark, I., Krishnaswami, N., Sewell, P.: ISA semantics for armv8-a, risc-v, and CHERI-MIPS. ACM Program. Lang. **3**, 71:1–71:31 (2019)

[2] Asanović, K., Patterson, D.A.: Instruction sets should be free: The case for risc-v. Tech. Rep. UCB/EECS-2014-146 (Aug 2014)

[3] Asavoae, M., Jan, M., Ben Hedia, B.: Formal modeling and verification for timing predictability. In: ERTS (2020)

[4] Bachrach, J., Vo, H., Richards, B.C., Lee, Y., Waterman, A., Avizienis, R., Wawrzynek, J., Asanovic, K.: Chisel: constructing hardware in a scala embedded language. In: The 49th Annual Design Automation Conference, DAC '12. pp. 1216–1225 (2012)

[5] Barthe, G., D'Argenio, P., Rezk, T.: Secure information flow by self-composition. In: Proc of the 17th Computer Security Foundations Workshop, June, 2004, California, USA. pp. 100–114 (2004)

[6] Binder, B., Asavoae, M., Ben Hedia, B., Brandner, F., Jan, M.: Is this still normal? Putting definitions of timing anomalies to the test. In: Intl. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA) (2021)

[7] Bréjon, J.B., Heydemann, K., Encrenaz, E., Meunier, Q., Vu, S.T.: Fault attack vulnerability assessment of binary code. In: Proc. of the Workshop on Cryptography and Security in Computing Systems. pp. 13–18 (2019)

[8] Cassez, F., Hansen, R.R., Olesen, M.C.: What is a Timing Anomaly? In: Intl. Workshop on Worst-Case Execution Time Analysis. WCET'12, vol. 23, pp. 1–12. OASIcs (2012)

[9] Cazorla, F.J., Kosmidis, L., Mezzetti, E., Hernandez, C., Abella, J., Vardanega, T.: Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. ACM Comput. Surv. **52**(1) (Feb 2019)

[10] Cerqueira, F., Stutz, F., Brandenburg, B.B.: PROSA: A case for readable mechanized schedulability analysis. In: 28th Euromicro Conference on Real-Time Systems, ECRTS, Toulouse, France, July. pp. 273–284 (2016)

[11] Choi, J., Vijayaraghavan, M., Sherman, B., Chlipala, A., Arvind: Kami: a platform for high-level parametric hardware specification and its modular verification. ACM Program. Lang. **1**, 24:1–24:30 (2017)

[12] Clarke, E.M., Kurshan, R.P., Veith, H.: The localization reduction and counterexample-guided abstraction refinement. In: Time for verification, pp. 61–71. Springer (2010)

[13] Cuenot, P., Delmas, K., Pagetti, C.: Multi-core processor: Stepping inside the box. In: ESREL 2021. Angers, France (2021)

[14] Dalsgaard, A., Olesen, M., Toft, M., Hansen, R., Larsen, K.: Metamoc: Modular execution time analysis using model checking. In: 10th Intl. Workshop on WCET. vol. 15, pp. 113–123 (2010)

[15] Eisinger, J., Polian, I., Becker, B., Thesing, S., Wilhelm, R., Metzner, A.: Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In: Design and Diagnostics of Electronic Circuits and systems. pp. 13–18. DDECS'06 (2006)

[16] Given-Wilson, T., Jafri, N., Lanet, J.L., Legay, A.: An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries and Case Study on PRESENT. In: Trustcom. pp. 293–300 (2017)

[17] Given-Wilson, T., Jafri, N., Legay, A.: Combined software and hardware fault injection vulnerability detection. Innovations in Systems and Software Engineering **16**(2), 101–120 (Jun 2020)

[18] Hahn, S., Reineke, J.: Design and analysis of sic: A provably timing-predictable pipelined processor core. In: 2018 IEEE Real-Time Systems Symposium (RTSS). pp. 469–481 (Dec 2018)

[19] Hicks, M., Sturton, C., King, S.T., Smith, J.M.: Specs: A lightweight runtime mechanism for protecting software from security-critical processor bugs. SIGPLAN Not. **50**(4), 517–529

[20] Irfan, A., Cimatti, A., Griggio, A., Roveri, M., Sebastiani, R.: Verilog2SMV: A tool for word-level verification. In: Design, Automation Test in Europe Conference (DATE). pp. 1156–1159 (2016)

[21] Izraelevitz, A., Koenig, J., Li, P., Lin, R., Wang, A., Magyar, A., Kim, D., Schmidt, C., Markley, C., Lawson, J., Bachrach, J.: Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In: Proc. Intl. Conf. on Computer-Aided Design. p. 209–216. ICCAD '17 (2017)

[22] Jain, H., Kroening, D., Sharygina, N., Clarke, E.: VCEGAR: Verilog CounterExample Guided Abstraction Refinement. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 583–586 (2007)

[23] Kirner, R., Kadlec, A., Puschner, P.: Worst-case execution time analysis for processors showing timing anomalies. Tech. rep., Technische Universität Wien (01 2009)

[24] Kroening, D., Purandare, M.: Ebmc. http://www.cprover.org/ebmc/

[25] Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., USA (2002)

[26] Laurent, J., Beroulle, V., Deleuze, C., Pebay-Peyroula, F.: Fault Injection on Hidden Registers in a RISC-V Rocket Processor and Software Countermeasures. In: Design, Automation & Test in Europe Conference (DATE). pp. 252–255. Florence, Italy (March 2019)

[27] Law, S., Bate, I.: Achieving appropriate test coverage for reliable measurement-based timing analysis. In: Euromicro Conference on Real-Time Systems. pp. 189–199. ECRTS'16 (2016)

[28] Lee, S., Sakallah, K.A.: Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction. In: Computer Aided Verification. pp. 849–865 (2014)

[29] Li, X., Roychoudhury, A., Mitra, T.: Modeling out-of-order processors for wcet analysis. Real-Time Systems **34**, 195–227 (11 2006)

[30] Library, J.I.: Application of Attack Potential to Smartcards and Similar Devices. Tech. rep. (2013)

[31] Mangean, A., Béchennec, J.L., Briday, M., Faucou, S.: Wcet analysis by model checking for a processor with dynamic branch prediction. In: Verification and Evaluation of Computer and Communication Systems. pp. 64–78. Springer International Publishing (2017)

[32] McMillan, K.: Cadence smv. Cadence Berkeley Labs, CA, 2000

[33] Mukherjee, R., Purandare, M., Polig, R., Kroening, D.: Formal techniques for effective co-verification of hardware/software co-designs. In: Proc. of the 54th Annual Design Automation Conference 2017. pp. 1–6

[34] Nienhuis, K., Joannou, A., Bauereiss, T., Fox, A., Roe, M., Campbell, B., Naylor, M., Norton, R.M., Moore, S.W., Neumann, P.G., Stark, I., Watson, R.N.M., Sewell, P.: Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In: 2020 IEEE Symposium on Security and Privacy

[35] OpenHW Group: Core-v cv32e40p risc-v ip. https://github.com/openhwgroup/cv32e40p

[36] Papon, C.: SpinalHDL. https://github.com/SpinalHDL

[37] Pattabiraman, K., Nakka, N., Kalbarczyk, Z., Iyer, R.: SymPLFIED: Symbolic program-level fault injection and error detection framework. In: Intl. Conf. on DSN. pp. 472–481 (2008)

[38] Saitoh, A.: Kyogenrv: simple 5-staged pipeline RISC-V. https://github.com/panda5mt/KyogenRV

[39] Schuiki, F., Kurth, A., Grosser, T., Benini, L.: LLHD: a multi-level intermediate representation for hardware description languages. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI) London, UK, June. pp. 258–271 (2020)

[40] Tomasulo, R.M.: An efficient algorithm for exploiting multiple arithmetic units. IBM J. of Research and Development **11**(1), 25–33 (1967)

[41] Waterman, A., Asanovic, K., Hauser, J., Division, C.: Volume II: Privileged Architecture. Tech. rep. (2021)

[42] Wenzel, I., Kirner, R., Puschner, P., Rieder, B.: Principles of timing anomalies in superscalar processors. In: Fifth International Conference on Quality Software (QSIC'05). pp. 295–303 (Sep 2005)

[43] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. **7**(3) (May 2008)

[44] Wolf, C.: Yosys open synthesis suite. http://www.clifford.at/yosys/

[45] Yuce, B., Schaumont, P., Witteman, M.: Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation. Journal of Hardware and Systems Security **2**(2), 111–130 (2018)