Inconstant: Refining Assumptions in Formal Constant-Time CPU Verification

Flavien Solt
National University of Singapore
Department of Computer Science
Singapore, Singapore

Christopher W. Fletcher
University of California, Berkeley
Electrical Engineering and Computer Sciences
Berkeley, California, USA

Abstract

Constant-time (CT) programming prevents secret data from affecting a program's timing, but a CT program can still leak secrets if the CPU's microarchitecture introduces data-dependent timing variations. Recent tools attempt to formally verify, at the register-transfer level (RTL), that a CPU preserves the CT property. In practice, every such proof is conditional: it relies on assumptions about the implementation and the surrounding system.

This paper refines the assumptions behind three proof styles used by current automated CT verifiers: sampled-ISA self-composition, safe-instruction reasoning, and taint tracking with declassification. We show that these assumptions are often more demanding or more subtle than their high-level descriptions suggest, and that violating them can hide real timing leaks in practice. To demonstrate the practical impact, we construct Inconstant, a modified 2-stage Sodor RISC-V CPU that violates the CT contract while being accepted by the four publicly available tools that we evaluate (LeaVe, CSL, μCFI, and VeloCT). Based on these findings, we provide practical guidelines for interpreting and strengthening automated CT verification results.

CCS Concepts

• **Security and privacy** → **Side-channel analysis and countermeasures**; **Logic and verification**; • **Hardware** → *Functional verification.*

Keywords

constant-time execution, formal verification, microarchitectural timing side channels, RISC-V, information flow tracking

ACM Reference Format:
Flavien Solt and Christopher W. Fletcher. 2026. Inconstant: Refining Assumptions in Formal Constant-Time CPU Verification. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '26), November 08–12, 2026, San Jose, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3831252.3834262

1 Introduction

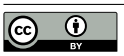
Software that processes secrets, such as cryptographic keys, must not leak those secrets through timing. Timing measurements of cryptographic code have repeatedly been turned into key-recovery attacks [6, 7, 26]. The established defense is *constant-time (CT) programming*, where the developer writes code whose control flow and memory-access pattern are independent of secret data. At the source level, this rule is well understood. However, a program that is CT at the instruction-set architecture (ISA) level may still leak secrets on a CPU, because microarchitectural mechanisms such as caching and resource contention can introduce data-dependent timing variations [25, 27, 47].

Hardware-level CT verification. To bridge this gap between ISA-level reasoning and CPU behavior, a recent line of work proposes automated formal verification at the register-transfer level (RTL) [9, 13, 14, 40, 45]. The goal is to prove that a given CPU design preserves the CT property: if a program is CT according to the ISA specification, then executing it on the CPU introduces no additional timing leaks. Such a proof would free the software developer from reasoning about microarchitectural internals.

The role of assumptions. In practice, every such proof is *conditional*. Each verification tool relies on assumptions about the CPU implementation and about the system surrounding it. When those assumptions hold, the proof is sound. When they do not, the proof may accept a design that actually leaks secrets, a situation known as a *false negative*. For a verification user, the critical question is therefore not just whether the tool says "pass," but what conditions must hold for that result to be meaningful.

Three proof styles. Existing automated CT verifiers fall into three families, each with a different proof strategy and a different set of assumptions. *(1) Sampled-ISA self-composition* (LeaVe [45], CSL [40]) runs two copies of the CPU on the same program with different secrets and checks whether the observable behavior differs. It extracts the ISA-level trace directly from the implementation rather than from a separate reference model, and is therefore commonly described as assuming functional correctness. *(2) Safe-instruction reasoning* (ConjunCT [13], VeloCT [14]) identifies instructions that are inherently CT (e.g., additions that always take the same number of cycles) and proves that programs composed only of such "safe" instructions remain CT. It is commonly described as assuming that only safe instructions execute. *(3) Taint tracking with declassification* (μCFI [9]) marks secret data as tainted, propagates taint through the hardware, and checks whether taint reaches a timing-relevant observation point (a "sink"). To avoid false alarms from legitimate data flow, it removes taint on *architectural* output paths such as register-file writes, under a stated condition on where those paths may lead.

This paper. We show that the assumptions behind all three styles are either more subtle or more demanding than these high-level descriptions suggest. For each style, we derive a more precise soundness condition and show how violations of that condition can hide real timing leaks. For sampled-ISA self-composition, full functional



This work is licensed under a Creative Commons Attribution 4.0 International License.
ICCAD '26, San Jose, CA, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2873-0/26/11
https://doi.org/10.1145/3831252.3834262

correctness is sufficient but stronger than necessary: only bugs that force every distinct timing class to also produce a distinct ISA trace cause false negatives. For safe-instruction reasoning, soundness requires that *all* signals on instruction-carrying paths represent safe encodings *at all times*, a system-wide invariant that goes well beyond restricting the source program. For taint tracking, the stated condition on declassified paths is necessary but not sufficient, because it credits an operand input with carrying taint even when the logic reading that input is never verified as an instruction.

We demonstrate implications of these assumptions individually, then construct Inconstant, a modified 2-stage Sodor RISC-V CPU [5] whose timing behavior violates the CT contract yet is accepted by the four publicly available tools we evaluate: LeaVe, CSL, μCFI, and VeloCT. Finally, we distill practical guidelines to strengthen the assurance that current tools provide.

In summary, our contributions are the following:

- We characterize the soundness assumptions behind the three proof styles used by current automated CT verification, providing more precise conditions than prior descriptions.
- We show that these assumptions can be difficult to satisfy in realistic systems and that violations of these assumptions can mask CT contract violations.
- We construct Inconstant, a vulnerable CPU that is accepted by all four evaluated CT verification tools.
- We distill practical guidelines for interpreting and strengthening automated CT verification results.

Open-sourcing. The sources of Inconstant and the code for our experiments are available at:
https://github.com/flaviens/inconstant-artifacts

2 Background

2.1 CPU implementation and integration

Modern CPUs rely on microarchitectural mechanisms that improve performance but complicate security reasoning. For example, pipelining overlaps different instruction stages. Speculative execution allows the processor to continue along predicted paths and later discard wrong-path work. Shared structures and contention further couple execution timing to implementation details. Systems generally follow either the Harvard model, which separates instruction and data memory, or the Von Neumann model, which shares memory structures between them.

Timing side channels. Timing side channels arise when data affects execution time. They can appear architecturally, for example through data-dependent loop bounds [26], or microarchitecturally, for example through cache behavior [6, 7]. CT programming aims to eliminate architecturally visible timing dependence, but is agnostic of the microarchitectural timing behavior of the CPU.

Threat model. We adopt the threat model assumed by the CT verifiers under study. An adversary observes the timing of an ISA-level CT program running on a target CPU and tries to infer secrets processed by that program. Internal CPU state is not directly visible. Physical side channels such as power and EM are out of scope.

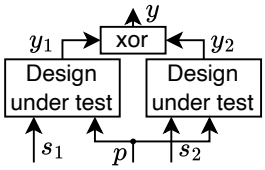


Figure 1: Miter circuit. Two design-under-test (DUT) instances are supplied with identical public data p and unconstrained secrets s_1 and s_2.

2.2 Information Flow Tracking

An information flow [16, 21–23, 34] is defined between a source, which provides information, and a sink, which may be affected by it. In confidentiality settings, the source provides secret data and the sink is observable by an adversary. Information propagates from a source to a sink if changing the source eventually affects the sink.

Hardware verification tools commonly model information propagation in one of two ways [8]. Taint tracking augments the design with taint tags propagated through instrumentation [2, 9, 37, 39, 41]. Self-composition, also known as miter circuit, duplicates the design and checks for secret-dependent differences between the two copies under a miter-style construction [13, 14, 40], as illustrated in Figure 1. Among the tools considered, μCFI [9] uses taint tracking; the others use self-composition.

2.3 Non-Interference Properties

Hardware-software contracts. A CPU, given a program and data, can be modeled as producing a sequence of observable events, called a *hardware trace* [31]. Hardware-software contracts [19] (we will refer to them as contracts in this paper) define both an execution mode and an observation mode. The execution mode determines which instructions execute in the sense of producing observable events, potentially including transient execution. The observation mode determines which events are visible to the adversary for those executed instructions. The CT contracts used by the tools in this paper generally assume sequential execution and a CT observer. Under that observer, the adversary sees the sequence of executed PCs, memory addresses accessed by executed instructions, and operands of variable-latency instructions. These observations capture common timing channels, including address-dependent memory latency [6, 7, 17, 18, 27, 29, 47, 48] and contention on shared execution units [1].

Non-interference. Non-interference is stated with respect to a fixed contract. It requires that observable behavior under that contract be independent of secret data. Let (P, M_p, M_s) denote a program P with public data M_p and secret data M_s. A CPU implementation satisfies the relevant non-interference property if, for all P, M_p, and all secret pairs M_s, M_s', Equation 1 implies Equation 2. Equation 1 says that the two executions are indistinguishable at the ISA level, while Equation 2 requires the same of the implementation-level trace. Intuitively, non-interference requires the implementation not to introduce secret-dependent observable differences beyond what the ISA already exposes.

$$\mathrm{ISA}(P, M_p, M_s) = \mathrm{ISA}(P, M_p, M_s') \tag{1}$$

$$\text{Micro}(P, M_p, M_s) = \text{Micro}(P, M_p, M'_s) \tag{2}$$

3 Assumptions

3.1 Proof Style 1: Sampled-ISA Self-Composition

LeaVe [45] and CSL [40] sample the ISA trace from the execution of the CPU under test, without relying on an ISA model.

Traditional sampled-ISA self-composition assumption formulation. LeaVe [45] and CSL [40] require the CPU to be functionally correct.

Known limitation and refinement. Prior work acknowledges that sampled-ISA self-composition assumes functional correctness. This statement is conservative for soundness, since it does not distinguish bugs that merely break functionality from bugs that also suppress CT counterexamples. Our refinement isolates the exact condition, which focuses checking on contract-relevant observation divergence.

Apparent ISA trace. We distinguish the *reference* ISA observations ISA^{ref} as defined by a golden reference ISA model, from the *apparent* ISA observations ISA^{app} as sampled by the retirement predicate ϕ. Equation 1 can hold for the reference ISA trace ISA^{ref}, yet the apparent trace ISA^{app} may still show a violation if the CPU has a functional bug, *apparently* invalidating Equation 1, precluding a check of Equation 2 and hence preventing the detection of a potential non-interference violation in this case. Hence, previous works [40, 45] formulate that the CPU must be correct, i.e., free of bugs that could cause the CPU implementation to violate the ISA. We show that while sound, there exists a less conservative assumption formulation that preserves soundness while allowing some functional bugs. In the rest of Section 3.1, we always assume that Equation 1 is satisfied for ISA^{ref}.

Unsoundness condition. A functional bug causes unsoundness when every pair of secret values that produces different microarchitectural observations (a violation of Equation 2) also produces different apparent ISA traces (a violation of Equation 1 for ISA^{app}). If any such pair violates Equation 2 without also violating the apparent Equation 1, then the microarchitectural difference is still visible in the apparent non-interference check, and sampled-ISA self-composition detects it.

Microarchitectural observation equivalence. For a given CPU, a program P and public memory M_p, we define an equivalence relation $\sim$, whose equivalence classes are referred to as *microarchitectural observation equivalence classes*. We say that $M_s \sim M'_s$ if both Equation 1 (instantiated with ISA^{app}) and Equation 2 hold for M_s and M'_s. If for (P, M_p), only a single microarchitectural observation equivalence class exists, then secrets cannot affect the microarchitectural observations when executing P on this CPU.

For (P, M_p), suppose there exist at least two distinct equivalence classes M_s and M'_s (with $M_s \nsim M'_s$). Let us assume the existence of a functional bug in the CPU. The bug need not behave uniformly: for some values of M_s the bug may have no effect on the ISA trace ($ISA^{\text{app}} = ISA^{\text{ref}}$), for others it may alter it ($ISA^{\text{app}} \neq ISA^{\text{ref}}$), and for yet others it may alter it in a different way. What matters for soundness is the pattern of ISA^{app} across equivalence classes. To create a false negative for (P, M_p), the apparent ISA traces must differ across all pairs of equivalence classes: $ISA^{\text{app}}(P, M_p, M_s) \neq ISA^{\text{app}}(P, M_p, M'_s)$ for all (M_s, M'_s) with $M_s \nsim M'_s$. This can happen because the bug manifests for one class but not another, or because it produces different apparent traces for both. This requirement is captured in Equation 3.

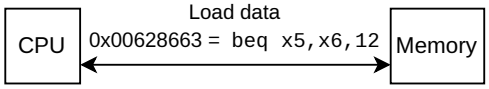


Figure 2: Example of a Von Neumann setting where data, which would decode as a branch if interpreted as an instruction, transits on the same bus as instructions.

$$\begin{aligned} &\forall P, M_p, M_s, M'_s;\ M_s \nsim M'_s \Rightarrow \\ &ISA^{\text{app}}(P, M_p, M_s) \neq ISA^{\text{app}}(P, M_p, M'_s) \end{aligned} \tag{3}$$

Temporal traces. $ISA^{\text{app}}(P, M_p, M_s)$ and $ISA^{\text{app}}(P, M_p, M'_s)$ denote temporal ISA architectural traces that are extended every time the retirement predicate ϕ holds. Equation 1 and Equation 2 operate over infinite traces (or up to program termination). Therefore, any divergence between $ISA^{\text{app}}(P, M_p, M_s)$ and $ISA^{\text{app}}(P, M_p, M'_s)$ will cause Equation 1 to be violated for the entire trace, regardless of when the divergence occurs in the trace.

Soundness-preserving functional bugs. This distinction matters in practice because not all functional bugs threaten soundness. Functional bugs that do not *always* introduce additional observations into ISA^{app}, or that manifest independently of data not already visible in the ISA trace, do not create the false-negative condition.

Soundness and unsoundness examples. Consider a bug that causes the CPU to crash after a store depending on the store *address*. Under the CT contract, the address is already part of the observation set, so the resulting timing classes are partitioned by already-visible information; this bug does not create a new false-negative condition. By contrast, a bug that behaves differently depending on the store *data* can threaten soundness because store data is not part of the CT observation set. Likewise, if an integer multiplication unit has data-dependent timing, a wrong multiplication result does not by itself mask the timing leak unless it also forces the sampled ISA trace to differ across all timing classes.

Sampled-ISA self-composition soundness refinement. The functional bugs that create false negatives are precisely those that force each distinct microarchitectural observation equivalence class to induce a distinct ISA^{app} trace.

3.2 Proof Style 2: Safe Instruction Reasoning

ConjunCT [13] and VeloCT [14] verify the safe instruction set (SIS) property, which guarantees non-interference for programs composed only of instructions from a predefined safe set.

Traditional safe-instruction reasoning assumption formulation. ConjunCT and VeloCT assume that the analyzed programs contain only instructions whose opcodes belong to the safe instruction set.

Known limitation and refinement. Prior work already states SIS only for programs drawn from a safe instruction set. This reads

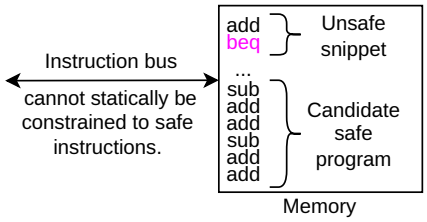


Figure 3: The SIS guarantee assumes that instruction-carrying paths never see unsafe encodings. The presence of an unsafe instruction elsewhere in time, for example in a bootloader, invalidates that guarantee even if the analyzed snippet itself contains only safe instructions.

as a source-program restriction, whereas soundness depends on a stronger invariant over instruction-carrying state over time. Our refinement shows that the condition is system-wide.

Soundness condition. In reality, these techniques assume more than source-level instruction safety. They require all signals that can feed instruction-processing logic, directly or indirectly, to contain only safe representations (i.e., words that decode as safe instructions) at all times [12]. The cited SIS formulation should therefore be read as imposing a proof obligation on instruction-carrying state, not merely as a source-program restriction.

Practicality in Von Neumann systems. Safe instruction reasoning techniques assume that the instruction domain remains restricted to safe instruction encodings for the whole proof context. In a Von Neumann system, instruction and data traffic share structures. This can make it difficult to guarantee that all words appearing on instruction-carrying paths are always decodable as safe instructions. Figure 2 illustrates such a case.

Practicality in Harvard systems. Even in Harvard-style systems, the assumption can remain difficult to justify in realistic deployments. A CT-relevant code fragment often executes within a larger lifecycle that includes boot code, operating-system code, or exception handlers containing unsafe encodings. Figure 3 illustrates that the guarantee can be invalidated by unsafe encodings that appear at a different time, even if the specific snippet of interest is architecturally composed only of safe instructions.

Implication. Restricting the source program is therefore not enough. Soundness requires a system-wide invariant over instruction-carrying paths and the memories feeding them over time, which is much stronger than the common informal statement that the program uses only safe instructions.

Temporal aspect. In particular, this also means that currently formulated safe-instruction reasoning cannot establish the correctness of defenses that may be enabled dynamically, such as Arm DIT [3], because those defenses do not imply that unsafe encodings are absent from the machine for its entire powered-on lifetime.

Safe-instruction reasoning soundness refinement. An unsafe representation appearing on any instruction-carrying path is sufficient to invalidate the SIS guarantee and enable false negatives.

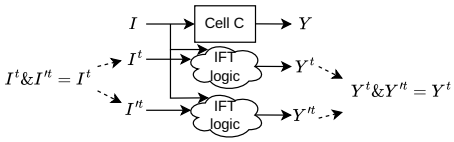


Figure 4: Adding taint to an input vector can only increase the bitwise taint of the output vector. The duplicated IFT logic compares the outputs produced from two different input taint vectors.

3.3 Proof Style 3: Taint Tracking

μCFI. μCFI [9] proves CT security through taint tracking: it taints the inputs of the instruction under verification (IUV) and checks whether taint reaches a microarchitectural sink, as required by Equation 2. It then declassifies the IUV's architectural outputs, such as register-file writes and forwarding paths, so that legitimate architectural propagation of the IUV's result does not violate Equation 1. Without this step, later transmitter instructions could receive tainted operands through ordinary architectural data flow and cause spurious violations. μCFI states a precondition on where a declassified signal may fan out, and a static analysis pass checks it on each verified design.

Traditional taint-tracking assumption formulation. "Declassification is sound if no unexpected information flows to the PC could have been missed when μCFI is proven. This is guaranteed iff the following precondition is proven on a design: all outgoing paths from a declassified signal converge either into another declassified signal or one of the SRC inputs to instructions before reaching the PC."

Known limitation and refinement. μCFI discharges this precondition by walking the fanout of each declassified signal. The remaining imprecision lies in what happens once that walk succeeds. Reaching an operand input is treated as safe because operand inputs are where taint gets injected, but that reasoning holds only for readers the proof models as an IUV. Since IUVs are enumerated from the ISA, a reader that is no ISA-defined instruction never has taint injected on its behalf, while the structural check still passes. Satisfying the precondition is therefore not sufficient.

Cause of unsoundness. A false negative means that no taint reaches a sink such as the program counter, causing the proof to underestimate secret-dependent observable events. Taint propagation is bitwise increasing. For a combinational cell C with input taint I^t and output taint Y^t, replacing I^t by I'^t such that $I^t \& I'^t = I^t$ yields an output Y'^t with $Y^t \& Y'^t = Y^t$, as shown in Figure 4. Thus, false negatives can only result from missing taint injection or excessive declassification. Since the IUV is always supplied with tainted operands, the relevant cause here is excessive declassification.

Shared architectural and microarchitectural paths. Declassification authorizes the IUV to transmit its output to the architectural state, but the same paths may also carry values to transient dependent instructions. The mechanism cannot tell the architectural

Table 1: Sources of unsoundness (false negatives). Here, "apparent Equation 1" refers to Equation 1 with ISA^{app}.

LeaVe/CSL	μCFI	ConjunCT/VeloCT
Apparent Equation 1	Equation 2	Equation 2

flows that enforce Equation 1 from microarchitectural flows that may affect Equation 2.

Example. μCFI taints only the IUV's inputs. Transient or internal microarchitectural operations are not ISA-defined instructions and are therefore never modeled as IUVs, even though they read the same operand inputs that the precondition treats as taint sources. If such an operation reads secret data from the register file through a declassified architectural path and uses it to influence branch prediction [42] or trigger a memory access [35], the resulting CT violation may be missed.

Taint-tracking soundness refinement. A declassified path that satisfies the fanout precondition can still hide a violation when one of its consumers is never modeled as an IUV.

3.4 Summary of non-interference violations

Table 1 summarizes the equations in the non-interference definition that can be violated when assumptions of the respective tools are violated. This also assumes no error in how a technique is mapped to the DUT. For LeaVe [45], for example, the sampled observations and the retirement predicate ϕ must be specified correctly.

4 Evaluation

We first confirm the characterizations of Section 3 by evaluating individual assumptions with example instances (Section 4.1). We then present Inconstant (Section 4.2), a vulnerable CPU accepted by the evaluated tools even though it violates the CT property, which serves as a stress-test benchmark for a non-trivial intersection of these assumption gaps.

Tools and setup. We evaluate the publicly available automated CT verification tools in our scope: μCFI [9] (commit `8e3cf1a`), VeloCT [14] (commit `b0b1af1`), LeaVe [45] (commit `48656b3`), and CSL [40] (commit `725445e`). ConjunCT [13] has no public implementation, and its repository points to VeloCT.

Target CPU. We consider the 2-stage Sodor RISC-V CPU [5]. This simple CPU appears in the CT verification benchmarks used by several of the works under study [40, 45]. It has an instruction fetch (IF) and an execution (EX) stage. It always speculatively fetches the instruction located at `PC+4`, and this fetch operation is killed if there is a change in control flow, i.e., if this simple static prediction was a misprediction. We interpret this kill as inserting a bubble (i.e., a no-operation instruction). The CPU stalls in two cases: (1) when the IF stage has been killed in the previous cycle, or (2) when it is waiting for data from memory.

Evaluation setup. We evaluate on a server with an Intel Xeon Gold 6258R CPU and 754 GB RAM, running JasperGold 2025.06 for μCFI and CSL. Times are medians of three runs.

Table 2: Individual-assumption verification results. Pass means no detected violation.

	CSL	μCFI	VeloCT
Assumption 1	Pass ✓	Fail ✗	Fail ✗
Assumption 2	Fail ✗	Fail ✗	Pass ✓
Assumption 3	Fail ✗	Pass ✓	Fail ✗

4.1 Individual assumptions

We now demonstrate individual assumptions of each automated CT verification technique under study.

For CSL, we fix its supplied Sodor wrapper: `csr_io_retire` replaces `exe_reg_valid`, which remains high during stalls. Repaired μCFI corrects register-address mapping and checks both add operands. For VeloCT our flow provides Sodor-specific input pairs with explicit SIS semantics and an invariant post-check.

4.1.1 Sampled-ISA Self-Composition assumption. We construct a functional bug that leads LeaVe and CSL to suffer from a false negative for the CT property. Specifically, we modify the Sodor implementation to include a functional bug that causes the CPU to raise an exception if it add's the value `0x12345678` to itself. This functional bug introduces two distinct microarchitectural observation equivalence classes per add instruction, corresponding to whether the instruction raises an exception or not. Because each microarchitectural observation equivalence class maps to a distinct ISA trace, no single ISA trace ever covers two different timings, which leads LeaVe and CSL to miss the violation as explained in Section 3.1.

```
  add x1, x2, x3
  exit # terminate the program
  ...
exception_handler:
  eret # return to normal execution
```

Listing 1: Example RISC-V snippet with an addition, supposed to be CT. No exception should be taken.

CT violation explanation. Listing 1 shows a RISC-V code snippet where the addition should execute in constant time: the CT observation mode only observes the PC value sequence, and the addition is not considered a variable-latency instruction in Sodor. The sequential execution mode only executes the addition and exits. Hence, according to the CT model, the program's execution time should be independent of the values in `x2` and `x3`. However, due to this functional bug, the addition will take more time if `x2` and `x3` contain the value `0x12345678`, which violates the contract.

RTL simulation of the trigger tuple and one ordinary operand tuple, with the trap vector set accordingly, confirms the variable timing. Table 2 (Assumption 1) reports the tool results.

4.1.2 Safe instruction reasoning assumption. We construct a functional bug that yields a false negative for SIS-style CT verification, and that we observe directly in VeloCT. Specifically, we modify our Sodor implementation so that once the instruction fetch buffer has been populated by `0x00000000`, an add instruction executes in 2 cycles instead of 1 cycle if an operand has the value `0x12345678`. We use Sodor's default simulation setup [5] and experimentally note that it occasionally supplies the instruction fetch buffer with

`0x00000000`. As described in Section 3.2, the SIS proof style used by ConjunCT and VeloCT requires instruction-carrying paths to remain free of unsafe encodings over time, which our bug violates when `0x00000000` reaches the fetch buffer.

```
add x1, x2, x3
exit # terminate the program
```

Listing 2: Example RISC-V assembly snippet with an addition, supposed to be CT.

CT violation explanation. Listing 2 shows a minimalistic RISC-V code snippet where the addition should execute in constant time, but due to the modification that we inserted and due to the natural presence of zero encodings on the default setup, the addition will expose its operands through a timing channel, which is precluded by the CT observation mode and hence violates the contract. In our setup, these zero encodings do not appear at the ISA level because the bus protocol marks them invalid.

RTL simulation of the trigger tuple and one ordinary operand tuple confirms the variable timing. Table 2 (Assumption 2) reports the tool results.

4.1.3 Taint tracking assumption. The functional bug that we introduced in Section 4.1.1 will be detected by μCFI because the architectural control flow is a function of an architectural state. This resembles CVE-2024-44927, which μCFI found in the Kronos CPU [9], where a jump or a branch can read the input register of the previous instruction when the memory does not acknowledge a request in the next cycle, making the architectural control flow depend on that data. Instead, we modify our Sodor implementation to extend the stall by one cycle if the `x1` register contains the value `0x12345678`. This leak rides on a stall rather than on an architectural transaction, so no IUV carries it and, as discussed in Section 3.3, μCFI cannot detect it. Microarchitectural activity that is not backed by a committed instruction has caused real bugs, from stale register-file contents leaking on AMD Zen 2 [32] to speculatively executed compressed floating-point instructions hanging VexRiscv (CVE-2023-34896) [36].

```
add x1, x2, x3
ld  x5, 0(x4) # Assume that address x4 is mapped
exit # terminate the program
```

Listing 3: Example RISC-V assembly snippet with an addition, supposed to be CT. The stall of the load instruction may be extended depending on the value of `x1`.

CT violation explanation. In Listing 3, the addition should again execute in constant time, since the CT observation mode only observes the PC value sequence. We assume that `x4` contains a properly mapped address, i.e., that it will not cause an exception according to the ISA, but will cause the load to stall for at least one cycle. The CSL A3 run adds a one-cycle memory response because the supplied scratchpad cannot otherwise activate this load-stall trigger. The sequential execution mode only executes the addition, the load, and exits. However, information about the operands of the addition leaks through the timing of the program (either the stall is extended or not), violating the contract.

RTL simulation of one trigger value and one ordinary value in `x1` confirms the variable timing. Table 2 (Assumption 3) reports the tool results.

4.2 Inconstant

We now introduce Inconstant, a vulnerable CPU used to test the combined limits of the evaluated proof setups. The individual examples validate each refined soundness condition in isolation, whereas Inconstant shows that these conditions can co-occur in one design and jointly evade the evaluated proof styles.

4.2.1 Design of Inconstant. We first express the requirements for an undetectable vulnerability. We then present one instance of such a vulnerability through Inconstant.

Requirements. From Section 3, an undetectable vulnerability must simultaneously exploit all three classes of assumptions. First, the CPU must have an architectural bug that makes the apparent ISA trace a function of the leaked information whenever the vulnerability occurs, typically through a different control flow decision. Second, the leakage must occur microarchitecturally yet transmit over an architectural path, typically by speculatively reading the architectural state. Third, the leakage must require an instruction path populated by signals interpretable as an unsafe instruction. Under these conditions, the vulnerability creates a data-dependent commit time, which the evaluated tools observe.

Vulnerability. We use Inconstant to denote the following modified 2-stage Sodor implementation.

Ghost Delay Vulnerability (GDV). If the instruction buffer has ever been populated with `0x00000000`, an EX stage stall will take two cycles instead of one cycle and skip two instructions if `x1` contains `0xdec0dedd`.

Functional bug. The GDV includes a functional bug that influences the program's control flow in different ways depending on the microarchitectural observation. Concretely, if the stall takes one cycle (which is the normal behavior in the absence of the GDV), the functional bug is not triggered, and the next instruction is located at `PC+4`. If the stall takes two cycles because of an activation of the GDV, implying in particular that `x1 = 0xdec0dedd`, which is the secret information that we intend to leak, the next instruction is located at `PC+12` instead of `PC+4`. Hence, by construction of the GDV, *any secret-dependent change in observable events (Equation 2) implies a change in the ISA trace (Equation 1)*, rendering the GDV undetectable by tools that extract the ISA trace from the implementation, like LeaVe [45] and CSL [40].

Architectural path sharing. The GDV relies on a bubble during stall, reading `x1`'s value to determine whether to stall for a second cycle. Because this bubble is not an ISA-specified instruction, it will not be tested by μCFI, which has no means of knowing the existence of this instruction. Additionally, only IUVs will read tainted data through architectural paths. Hence, because it reads `x1` through an architectural path (the register read port), the ghost bubble instruction will see `x1`, and hence `x1`'s equality to `0xdec0dedd`,

Table 3: Original Sodor (S) and Inconstant results. μCFI/VeloCT entries count instructions reported safe. † marks a CSL domain excluding the GDV; ‡ marks a VeloCT domain excluding its triggering execution.

	LeaVe	CSL	μCFI	VeloCT
Time (S)	1h 28min	10s	1h 40min	68s
Result (S)	Pass	Pass	28	21
Time (Inconstant)	2h 16min	9s	33min	89s
Result (Inconstant)	Pass	Pass†	28	21‡

untainted, and hence its impact on the commit signal and on the PC will remain undetected.

Unsafe population of instruction paths. A condition for the GDV to be triggered is that the instruction buffer has at some point been populated with `0x00000000`. Establishing the absence of unsafe instructions (instructions, such as branches, whose encoding is not in VeloCT's safe set) in a general system requires strong global assumptions. As explained in Section 3.2, a Von Neumann architecture fetches instructions from memory, where data can take any value including `0x00000000`, so no static argument keeps that word out of the instruction buffer. Bus protocols also carry arbitrary data as long as control signals mark it invalid. In addition, besides the presence of unsafe instructions such as control-flow instructions, fetch address prediction like phantom jumps [46] or Sodor's next-address predictor, might fetch unexpected values outside of the strict program order.

Data-dependent trace. Once the instruction buffer has been populated at least once with `0x00000000`, the GDV causes the commit time to leak whether the `x1` register is `0xdec0dedd`. Stalls happen at every control-flow change, and such stalls are already permitted by the non-interference property. LeaVe proves that the original CPU satisfies a contract that does not expose load addresses [45], although loads can stall. The public VeloCT campaign covers 21 arithmetic and logic instructions, excluding both loads and the triggering control-flow instruction, hence its unchanged count does not test the GDV execution. The GDV only makes an already-permitted stall last one cycle longer in a secret-dependent way.

4.2.2 Implementation and Verification. Inconstant is based on the 2-stage Sodor [5] (commit `32d49f9`) and adds 49 lines of Chisel code. We have confirmed the vulnerability in RTL simulation by executing the snippet in Listing 4 with the trigger and a non-trigger value for `x1`, which violates the CT contract on Sodor's default simulation setup [5] where the instruction buffer has already been populated with `0x00000000` data.

```
  jal  zero, jump_target
  nop
jump_target:
  exit # terminate the program
```

Listing 4: Example RISC-V code creating a data-dependent timing variation on Inconstant.

Verification results. Table 3 summarizes the reported runs of the reference 2-stage Sodor and of Inconstant, including their wall times. The dagger marks CSL's published-domain result: its assumptions make the GDV trigger unreachable, so the Pass does not establish security for the vulnerable behavior. For μCFI, the reported time is the wall time of the complete 28-instruction run. The corrected-state LeaVe setup reports Pass in three co-scheduled runs (median 8,180s), after fixing both missing RTL connections and listing all GDV state. This establishes a false negative; its timing is not directly comparable with the sequential campaign. μCFI's unchanged 28-instruction result is also a false negative. VeloCT's 21-instruction count excludes the triggering execution. CSL's proof excludes the GDV as well; removing its illegal-instruction and same-ISA-trace assumptions makes the trigger reachable and produces a counterexample.

5 Discussion

Other capabilities. Besides verifying constant-time properties, μCFI can also detect functional bugs, either via its main mechanism or through CellDFT, a complementary data-flow tracker that monitors whether wrong data can populate the program counter [9].

Functional correctness. Bug detection is still an open challenge. LeaVe [45] assumes functional correctness, yet μCFI later uncovered bugs in Ibex [28], one of the CPUs on which LeaVe was evaluated. Nevertheless, we have shown in Section 3.1 that only a well-defined class of functional correctness bugs can lead to unsoundness for LeaVe and CSL.

Backdoors. It is unlikely that all of these conditions would accidentally align in the same design, so we encourage verification engineers to combine multiple techniques, even though Inconstant shows the combination is still imperfect. As shown in Section 4.2, an attacker with access to the design who tries to confuse verifiers into approving a vulnerable design as CT can exploit these assumptions. A long line of work studies how backdoors reach a design through untrusted third-party IP cores and how to detect or neutralize them [4, 30, 43, 44, 49, 50], and malicious commits to open-source hardware repositories offer the same opportunity.

Purpose of Inconstant. Inconstant is not a claim that this exact vulnerability is representative of deployed CPUs, nor merely a hardware Trojan example. It is a stress-test benchmark for intersecting assumption gaps, showing that multiple reasonable proof styles can fail on the same design at once. Future verifier authors can use it as a regression benchmark when adding stronger soundness checks.

6 Guidelines for verification users

None of the evaluated tools provides an unconditional end-to-end CT guarantee. However, each proof style can be strengthened with targeted checks. Below, we distill three guidelines that verification users can apply today to close the gaps identified in this paper.

Guideline 1: Check contract-field faithfulness, not full correctness. Sampled-ISA self-composition (LeaVe, CSL) does not need a bug-free CPU; it needs the ISA trace *projected onto the contract observation fields* to be faithful. As shown in Section 3.1, only bugs that create a distinct apparent ISA trace for every microarchitectural observation equivalence class can cause unsoundness, through a systematic divergence in the contract-relevant observations. Bugs that corrupt other state, such as a wrong computation result, without *always* affecting these observations are benign for soundness.

Concretely, a verification user could:

(1) *Cross-check the contract interface.* Run a reference ISA model alongside the DUT and compare only the fields that the CT contract defines as observable (PC, memory addresses, timing-sensitive operands), which is cheaper than full functional equivalence checking yet sufficient.
(2) *Stress the retirement predicate.* Functional bugs that cause unsoundness must manifest through the retirement predicate ϕ. Fuzz or formally verify that ϕ fires at the correct points.
(3) *Prioritize data-dependent control-flow bugs.* Commercial CPU errata span many bug classes, as the trigger taxonomy of RemembERR [38] shows. Equation 3 isolates the subset that matters for soundness here, so targeted bug-finding campaigns might cover it efficiently.

Guideline 1. Cross-check the DUT's ISA trace on the contract observation fields only. This is strictly weaker than full functional correctness but sufficient for soundness.

Guideline 2: Check the system-level invariants behind SIS proofs. Safe instruction reasoning (ConjunCT, VeloCT) establishes the SIS property under the premise that only safe instruction encodings reach instruction-processing logic. As shown in Section 3.2, connecting that guarantee to end-to-end CT requires establishing this premise as a system-level invariant, not merely restricting the analyzed source program.

When the invariant is included in the proof or enforced by the system, the SIS result supports an end-to-end CT claim. When it has not been established, the result should remain scoped to the safe-instruction domain. It can still identify data-dependent timing behavior within that domain.

Verification users should observe the following constraints:

(1) *Interpret SIS results in a system context.* The result is valid only under the assumption that the entire instruction path is free of unsafe encodings at any point in time.
(2) *Account for dynamically toggled defenses.* For mechanisms such as Arm DIT [3], establish that instructions executed outside the enabled window cannot invalidate the instruction-domain invariant required by the SIS result.
(3) *Scope SIS to linear-control-flow subsystems.* For restricted-domain systems where only safe instructions are expected, SIS results may be combined with runtime monitors that check instruction-buffer contents.

Guideline 2. When using an SIS proof to support an end-to-end CT claim, prove or enforce that only safe encodings reach instruction-processing logic throughout the relevant execution scope.

Guideline 3: Check that every consumer of a declassified signal is verified as an instruction. Taint tracking with declassification (μCFI) clears taint on the architectural output paths of the instruction under verification, and already walks the fanout of each declassified signal to require that it converge on another declassified signal or on an operand input of an instruction. As shown in Section 3.3, that walk does not settle the question, because an operand input is only a taint source for readers that the proof enumerates as an IUV.

Verification users should observe the following constraints:

(1) *Match consumers against the IUV list.* For each operand input that terminates a fanout walk, identify every logic path that can drive a read of it. A path belonging to no ISA instruction, such as a pipeline bubble, makes the declassification unsound even though the fanout check passes.
(2) *Restore taint at unmodeled consumers.* Where such a consumer exists, re-inject taint there or delay the declassification until the transient window has closed. Automatically deciding which readers correspond to a modeled instruction remains an open research problem.

Guideline 3. A passing fanout check only shifts the obligation. Confirm that every reader reached by that check is an instruction the proof enumerates.

The check behind Guideline 3 extends the existing fanout walk with a lookup of which readers the proof enumerates. Its cost is expected to follow the number and reach of declassification points.

7 Related Work

Other CT verification tools. UPEC [15] is a manual technique that verifies similar properties. It requires information about the source of the speculation and requires interactive human judgments about whether partial information flows through the microarchitecture are legitimate. UPEC-DIT [11] verifies the SIS property like ConjunCT [13] and VeloCT [14], inherits the same limitations implied by the SIS property, and additionally requires the manual extraction of design conditions. Finally, SynthLC [20] synthesizes leakage contracts from SystemVerilog, relying on the functional correctness of the existing program counter registers and requiring significant manual effort, e.g., adding 23 dedicated registers and over 100 lines of new SystemVerilog code to the CVA6 codebase. Their soundness depends on expert manual intervention, and that of SynthLC also on the correctness of the hardware design itself.

Limitation analyses of formal verification. Rushby [33] discusses the strengths and weaknesses of formal methods for the certification of critical systems. Hurley et al. [24] study how formally specified temporal logic can be misused or misunderstood by humans, creating gaps between intended properties and what is actually validated. Dessouky et al. [10] underline, through a bug finding competition case study, that the comprehensiveness of formal methods is limited by the user-expressed properties.

8 Conclusion

This paper shows that the automated RTL CT verification techniques analyzed here prove security only under stronger assumptions than their high-level descriptions suggest. We identified the conditions needed for soundness and showed that violating them can hide CT-contract violations. We demonstrated this gap with Inconstant, a modified 2-stage Sodor CPU that violates CT behavior and separates established false negatives from vacuous or incomplete evaluations. These results clarify what each technique must still prove, and guide both its users and future work in this area.

Acknowledgments

We thank the anonymous reviewers for their insightful and constructive feedback. This research was funded by NSF grants CNS-1954521, CNS-1942888, CNS-2154183 and CCF-8191902.

References

[1] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port Contention for Fun and Profit. In *IEEE SP*.

[2] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. 2017. Register transfer level information flow tracking for provably secure hardware design. In *DATE*.

[3] Arm Limited. 2021. *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile (Issue G.a)*. Arm Limited, Cambridge, UK. DDI 0487G.a.

[4] Mainak Banga and Michael S. Hsiao. 2010. Trusted RTL: Trojan Detection Methodology in Pre-silicon Designs. In *HOST*.

[5] Berkeley Architecture Research. 2026. riscv-sodor. https://github.com/ucb-bar/riscv-sodor. Online; accessed 10-July-2026.

[6] Daniel J Bernstein. 2005. Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[7] Joseph Bonneau and Ilya Mironov. 2006. Cache-collision timing attacks against AES. In *CHES*.

[8] Katharina Ceesay-Seitz, Flavien Solt, Alexander Klukas, and Kaveh Razavi. 2025. Pathfinder: Constructing Cycle-accurate Taint Graphs for Analyzing Information Flow Traces. In *ICCAD*.

[9] Katharina Ceesay-Seitz, Flavien Solt, and Kaveh Razavi. 2024. μCFI: Formal Verification of Microarchitectural Control-flow Integrity. In *CCS*.

[10] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2019. HardFails: Insights into Software-Exploitable Hardware Bugs. In *USENIX Security*.

[11] Lucas Deutschmann, Johannes Müller, Mohammad Rahmani Fadiheh, Dominik Stoffel, and Wolfgang Kunz. 2024. A scalable formal verification methodology for data-oblivious hardware. *IEEE TCAD* (2024).

[12] Sushant Dinesh. 2025. VeloCT assumption on safe encodings on 32-bit state elements. Online; accessed 7-August-2026. https://github.com/FPSG-UIUC/veloct/blob/b0b1af10f9123acd9b00a9b1ed00210ea6d5cba7/learning/learn_inv_distributed.py#L258-L271

[13] Sushant Dinesh, Madhusudan Parthasarathy, and Christopher W Fletcher. 2024. ConjunCT: Learning inductive invariants to prove unbounded instruction safety against microarchitectural timing attacks. In *IEEE SP*.

[14] Sushant Dinesh, Yongye Zhu, and Christopher W Fletcher. 2025. H-Houdini: Scalable Invariant Learning. In *ASPLOS*.

[15] Mohammad Rahmani Fadiheh, Johannes Müller, Raik Brinkmann, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. 2020. A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors. In *DAC*.

[16] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C Myers, and G Edward Suh. 2017. Verification of a practical hardware security architecture through static information flow analysis. In *ASPLOS*.

[17] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security*.

[18] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS*.

[19] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-software contracts for secure speculation. In *IEEE SP*.

[20] Yao Hsiao, Nikos Nikoleris, Artem Khyzha, Dominic P Mulligan, Gustavo Petri, Christopher W Fletcher, and Caroline Trippel. 2024. RTL2MμPATH: Multi-μPATH Synthesis with Applications to Hardware Security Verification. In *MICRO*.

[21] Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. 2021. Hardware information flow tracking. *ACM CSUR* (2021).

[22] Wei Hu, Dejun Mu, Jason Oberg, Baolei Mao, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. 2014. Gate-level information flow tracking for security lattices. *TODAES* (2014).

[23] Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner. 2011. Theoretical fundamentals of gate level information flow tracking. *IEEE TCAD* (2011).

[24] Isabelle Hurley, Rohan Paleja, Ashley Suh, Jaime D Peña, and Ho Chit Siu. 2024. STL: Still Tricky Logic (for system validation, even when showing your work). *Advances in Neural Information Processing Systems* (2024).

[25] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *IEEE SP*.

[26] Paul C Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*.

[27] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE SP*.

[28] lowRISC C.I.C. 2026. Ibex: An Embedded 32 bit RISC-V CPU Core. https://github.com/lowRISC/ibex. Online; accessed 10-July-2026.

[29] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2018. MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations in SGX. In *CT-RSA*.

[30] Adib Nahiyan, Mehdi Sadi, Rahul Vittal, Gustavo Contreras, Domenic Forte, and Mark Tehranipoor. 2017. Hardware Trojan Detection through Information Flow Security Verification. In *2017 IEEE International Test Conference (ITC)*. 1–10.

[31] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. 2022. Revizor: Testing black-box CPUs against speculation contracts. In *ASPLOS*.

[32] Tavis Ormandy and Google Information Security Engineering Team. 2023. Zenbleed: CPU vulnerability in AMD Zen 2 (CVE-2023-20593). https://github.com/google/security-research/blob/master/pocs/cpus/zenbleed/README.md. Original disclosure; accessed August 1, 2025.

[33] John Rushby. 1993. *Formal Methods and the Certification of Critical Systems*. Technical Report SRI-CSL-93-7. Computer Science Laboratory, SRI International.

[34] Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *IEEE Journal on selected areas in communications* 21, 1 (2003), 5–19.

[35] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss. 2019. Netspectre: Read arbitrary memory over network. In *ESORICS*.

[36] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. 2024. Cascade: CPU Fuzzing via Intricate Program Generation. In *USENIX Security*.

[37] Flavien Solt, Ben Gras, and Kaveh Razavi. 2022. CellIFT: Leveraging Cells for Scalable and Precise Dynamic Information Flow Tracking in RTL. In *USENIX Security*.

[38] Flavien Solt, Patrick Jattke, and Kaveh Razavi. 2022. RemembERR: Leveraging Microprocessor Errata for Design Testing and Validation. In *MICRO*.

[39] Flavien Solt and Kaveh Razavi. 2024. HybriDIFT: Scalable memory-aware dynamic information flow tracking for hardware. In *ICCAD*.

[40] Qinhan Tan, Yuheng Yang, Thomas Bourgeat, Sharad Malik, and Mengjia Yan. 2025. RTL Verification for Secure Speculation Using Contract Shadow Logic. In *ASPLOS*.

[41] Mohit Tiwari, Hassan MG Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T Chong, and Timothy Sherwood. 2009. Complete information flow tracking from the gates up. In *ASPLOS*.

[42] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. 2023. Inception: Exposing new attack surfaces with training in transient execution. In *USENIX Security*.

[43] Adam Waksman and Simha Sethumadhavan. 2011. Silencing Hardware Backdoors. In *IEEE SP*.

[44] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. 2013. FANCI: Identification of Stealthy Malicious Logic Using Boolean Functional Analysis. In *CCS*.

[45] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. 2023. Specification and verification of side-channel security for open-source processors via leakage contracts. In *CCS*.

[46] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. 2023. Phantom: Exploiting decoder-detectable mispredictions. In *MICRO*.

[47] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*.

[48] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2016. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. In *CHES*.

[49] Jie Zhang and Qiang Xu. 2013. On Hardware Trojan Design and Implementation at Register-Transfer Level. In *HOST*.

[50] Jie Zhang, Feng Yuan, Lingxiao Wei, Zelong Sun, and Qiang Xu. 2013. VeriTrust: Verification for Hardware Trust. In *DAC*.