Platform Timing Contracts: A Lightweight Instrumentation for Capturing SoC Timing Channels

***Abstract*—Variations in the time a computer requires to perform computations can leak confidential information. Constant-time programming techniques defend against such timing attacks, but make assumptions on the behavior of the underlying hardware. As a response, previous work has introduced several techniques for verifying CPU compliance with hardware-software contracts. However, selecting which hardware-software contract to verify is non-trivial and requires careful consideration of the system's architecture and the specific timing attacks it may face. If a CPU is integrated into a platform with more timing channels than expected, CPU verification alone may be insufficient to guarantee security. In this paper, we address the integration of a verified constant-time CPU into a larger system. We make three contributions: (1) we define platform timing contracts as a formal way to capture an upper bound of the platform-induced timing channels, (2) we design a low-overhead synthetizable instrumentation that materializes these contracts and is compatible with all existing CPU verification methods, and (3) we demonstrate that existing CPU verification techniques along fail under integration, whereas platform timing contracts enable stronger, system-wide security guarantees against timing attacks.**

I. Introduction

Timing side-channel attacks exploit variations in execution time to leak secrets through measurable side effects. By carefully choreographing benign-looking instruction sequences, an attacker can steer microarchitectural state, such as caches, TLBs, execution ports, branch predictors, or DRAM buffers, to make secret-dependent effects externally observable. Two broad families have emerged: transient/speculative-execution attacks [1]–[12], which leave secret-dependent footprints during mis-speculated execution, and contention-driven channels on shared resources [13]–[19], which measure interference to encode information. These attacks can often be mounted entirely in software, without special privileges or hardware access. Mitigation is notoriously difficult after fabrication. Software or microcode defenses are often partial [20] or impose significant overheads [21].

To address these challenges, recent work introduced hardware-software contracts, which inform developers about the side effects of their code on a CPU [22], [23]. As a result, several techniques have been proposed to formally verify that a CPU implementation complies with such contracts [24]–[29] to prevent confidentiality breaches before chip fabrication.

Platform timing channels. CPUs are often verified in isolation [24]–[29] against hardware-software contracts before integration into a larger system, both for scalability and because the exact specifications of the integrating platforms are usually unknown during CPU design and verification. However, hardware-software contracts inherently assume a specific system context. When verification ignores this context, the results may not hold once the CPU is integrated into a platform. For example, the constant-time contract observer mode [22] exposes the program counter and all memory-access addresses, capturing timing effects of caches and of the memory hierarchy [22], [23]. Other modes, such as the architectural observer, reveal even more detail by also disclosing values exchanged with memory. Such observer modes depend on platform-level features (e.g., SoC components), yet many verification techniques overlook these modes or cannot flexibly capture platform-induced channels.

Platform timing contracts. Platform timing contracts extend the idea of hardware-software contracts to encompass the broader system context in which a CPU operates. They aim to capture the timing behavior of the entire platform, including interactions with other components such as memory controllers, I/O devices, and interconnects. By considering these interactions, platform timing contracts can provide a more comprehensive understanding of potential timing channels and their implications for security. To verify the confidentiality guarantees of a CPU integrated into a specific platform using existing techniques that verify a CPU in isolation, we introduce an automatic instrumentation that captures the relevant platform-specific timing information while retaining compatibility with all existing hardware-software contract verification techniques.

We apply the instrumentation corresponding to each platform to the RISC-V Sodor CPU, which is commonly used in constant-time verification benchmarks. Not only does the verification time not increase significantly when applying the instrumentation if no violation is found, but we demonstrate the existence of platform-specific timing channels on typical platforms, that would otherwise be missed.

FS: TODO Add the Kronos results

In summary, our contributions in this paper are:

- We introduce platform timing contracts that summarize platform-specific timing behaviors.
- We implement an open-source lightweight CPU instrumentation that accounts for all platform-induced timing side-channels.
- We apply instrumentations corresponding to various platforms to the RISC-V Sodor CPU to capture platform-specific timing information and show that these platforms can introduce unique timing channels that may not be present in isolated CPU verification and that would be missed by some existing verification techniques.

All our experiments can be found at: https://placeholder/

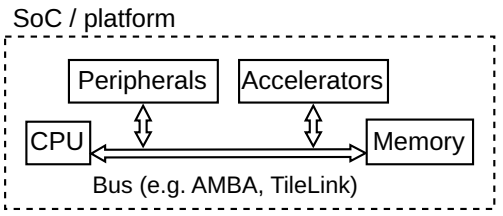

Fig. 1. Example SoC platform architecture.

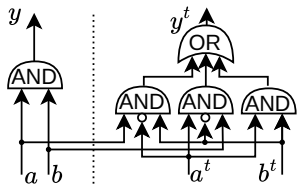

Fig. 2. Taint tracking at gate level [34]. Left: original circuit. Right: Added instrumentation for taint tracking. The original inputs are a and b. The taint bits corresponding to a and b are respectively a^t and b^t. The output taint bit y^t indicates whether the output is tainted, i.e., influenced by the tainted inputs.

II. Background

In this section, we provide background on SoC platforms, timing side-channels and non-interference properties.

A. SoC platforms

In this paper, we use the words SoC and platform interchangeably to refer to the integrated circuit that integrates various components, including CPUs, memory, and peripherals, to provide a complete computing solution, as illustrated in Figure 1. The communication between these components is typically managed through standardized bus protocols, such as AMBA [30] and TileLink [31]. The bounds of components are not always clear-cut. For example, caches can be considered part of the CPU or of the platform, as shown in Figure 4.

B. Timing side-channels

Multiple hardware optimizations, such as caches and branch predictors, learn and exploit patterns in program execution to improve performance. In particular, the execution time of later instructions may depend on the outcome of earlier instructions. For example, accessing a cache line might be faster if the line has been brought into the cache recently by a prior memory access to a nearby location. Constant-time programming techniques, such as not using secret data as array indices, aim to ensure that secrets do not influence the execution time of a program through such channels [32], [33].

C. Information Flow Tracking

Hardware information flows capture properties such as confidentiality and integrity [35]. They are typically captured using one of two techniques. The first technique is called dynamic information flow tracking (DIFT). It captures information flows by augmenting the hardware design under verification with a synthesizable instrumentation that computes the propagation of taint bits [26], [34], [36]–[38] as shown in Figure 2. The second technique is called miter circuits. It captures information flows by comparing the outputs of two copies of the design under test, where all the inputs except the sources are unconstrained, while all the others are equal as illustrated in Figure 3.

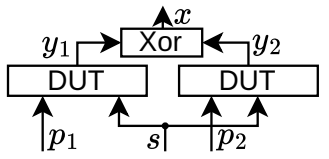

Fig. 3. Miter circuit. The two instances of the same design under test are supplied with the same secret data s and unconstrained p_1 and p_2 public data. The output bit y (i.e., instantiated for the instances in the miter as y_1 and y_2) reveals secret data if $x = 1$ is satisfiable.

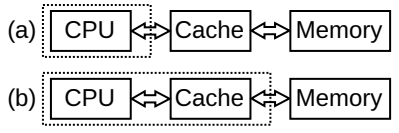

Fig. 4. Example verification scopes, represented as dotted boxes. (a: cache excluded) the cache is considered part of the platform. (b: cache included) the cache is considered part of the CPU.

D. Hardware-software contracts

Contract definition. Hardware-software contracts [22] split confidentiality responsibilities between the CPU and the software that it executes. For an initial microarchitectural state, a given program $\mathcal{P}$, and the contents of memory containing public information M_p and secret information M_s, the contract fixes an execution mode and an observation mode. The *execution mode* determines which instructions are executed (potentially transiently) and generate events. The *observation mode* determines which of those events are exposed to an observer (possibly transiently), such as the addresses or values involved in memory accesses. Under these two choices, the CPU's behavior is summarized as a sequence of observable events known as a hardware trace [22], [23].

Contract verification. Multiple independent verification efforts have been conducted to ensure that hardware-software contracts are upheld during program execution, in particular regarding constant-time behavior [22], [24]–[29]. Most techniques consider a fixed contract [24]–[26], [28]. Others leverage custom, non-trivial technique-specific insights to verify a wider variety contracts [27], [29] using the same technique.

Discussion. All these techniques make assumptions about the platform in which the CPU is integrated, yet there exists no language that allows to formally express and verify them.

III. Platform Timing Contracts

A. Motivational example

In this section, we first provide a motivational example showing that the integration of a CPU into a platform can introduce new timing channels that were not present in isolation. We then discuss and formalize reentrant information flows. Finally, we introduce platform integration contracts to capture these reentrant flows.

Methodology (cache excluded). Figure 4 (a) illustrates a small system in which a CPU is integrated in a platform that includes a cache and memory. We instantiate this system, using the Kronos CPU as a case study, and a direct-mapped cache. We use the state-of-the-art VeloCT [25] tool to perform constant-time verification of the CPU, which is delimited by the dotted box. This tool verifies if only instructions of a given "safe instruction set" are executed, then the CPU runtime is independent of their operands. We consider memory instructions as part of the safe instruction set that should execute in constant-time, in addition to arithmetic RISC-V instructions such as `add`, `sub`, `xor`, etc.

Results (cache excluded). The Kronos verification with VeloCT, which indicates that in isolation, the CPU is constant-time secure. In particular, this result suggests that the timing of any program made of arithmetic and memory instructions does not depend on the data that is processed, even if this data is used as memory addresses.

Methodology (cache included). Figure 4 (b) illustrates a scenario similar to Figure 4 (a), but with the cache included in the verification scope, i.e., considered part of the CPU under verification. Except for this point, we use the same methodology as in the previous cache excluded case.

Results (cache included). The Kronos verification with VeloCT, which indicates that this time, the CPU is not constant-time secure. In particular, this result implies that the program execution timing can depend on secret information used as addresses of memory instructions.

Take-aways. These results underline that while constant-time verification techniques typically operate at the level of a CPU in isolation (the platform's RTL might not yet be finalized at the CPU verification time), they may not account for all the potential interactions with other components in the system, such as caches or memory. Therefore, the integration of the CPU into a larger system can introduce new timing channels that were not present in isolation and that are not discovered by existing constant-time verification techniques.

B. Reentrant flows

We now describe and formalize reentrant information flows, which are at the core of the timing channels that can be introduced by platform integration.

Netlist graphs. We describe a digital hardware system (e.g., the one illustrated in Figure 4), as a netlist $G = (V, E)$, where V is the set of vertices representing logic gates such as AND, OR, NOT, flip-flops, adders, and E is the set of directed wires between these vertices. We denote the subgraph of G that corresponds to the CPU that is integrated in this system as $G_C = (V_C, E_C)$, where $V_C \subseteq V$ and $E_C \subseteq E$. In particular, for all edges $(u, v) \in E_C$, both u and v are vertices in V_C. We denote the complementary subgraph as $G_P = (V_P, E_P)$, where $V_P = V \setminus V_C$ and $E_P = E \setminus E_C$, corresponding to the platform except the CPU that it integrates.

Constant-time programs & secret data. For a given constant-time verification technique, let us denote by $\mathcal{P}$ the set of constant-time programs that are considered for verification. The elements of the set $\mathcal{P}$ are triples $\tau = (P, M_p, M_s)$, where P is a constant-time program, M_p is the set of public memory valuations, and M_s is the set of secret values that the program may depend on. Specifically, this set $\mathcal{P}$ depends on the specific constant-time verification technique being used. For example, for ConjunCT [24] and VeloCT [25], a triple (P, M_p, M_s) is considered constant-time (i.e., part of $\mathcal{P}$) if it contains no so-called "unsafe" instructions, i.e., instructions that can create an operand-dependent timing channel. For Tan et al. [28], a triple (P, M_p, M_s) is considered constant-time (i.e., part of $\mathcal{P}$) if it ensures that the program's control flow and all memory accesses are independent of secret data. The conditions for a program to be constant-time with respect to ConjunCT's and VeloCT's definition are stricter than those of Tan et al. [28], as the latter generally forbid branches, for instance. Inclusively more restrictive constraints imply an inclusively smaller set $\mathcal{P}$ of constant-time programs.

Secret propagation. Constant-time verification techniques generally proceed by analyzing whether the valuation of a state element that captures timing differences can be influenced by secret information. For example, Tan et al. [28], LeaVe [27], ConjunCT [24] and VeloCT [25] monitor whether secret information can influence the commit signal, while μCFI [26] monitors whether secret information can influence a microarchitectural program counter structure.

We define the *valuation sequence* ν of a vertex. For a vertex $v \in V$ and a triple $\tau = (P, M_p, M_s)$, the valuation sequence $\nu_\tau(v)$ is the clock-accurate infinite sequence of valuations that the vertex takes during the execution of a program P with respect to the memory valuations M_p and M_s, starting from the CPU's and platform's reset state, assumed deterministic. Note that we assume determinism for simplicity without loss of generality; to account for non-determinism, we could consider a set $\nu_\tau^{\text{non-det}}(v)$ of all possible valuation sequences instead of a single sequence $\nu_\tau(v)$, without change in the following reasoning.

We denote by V^s the set of vertices in V whose valuation sequence can be influenced by secret information when the system executes a program written in constant-time with respect to the secret data. An element $v \in V$ belongs to V^s if and only if there exist two triples $\tau = (P, M_p, M_s)$ and $\tau' = (P, M_p, M_s')$ that only differ in M_s', and such that $\tau \in \mathcal{P}$ and $\tau' \in \mathcal{P}$ but $\nu_\tau(v) \neq \nu_{\tau'}(v)$. We say that a vertex $v \in V$ is *secret-dependent* if it belongs to V^s.

For a secret-dependent vertex $v \in V^s$, we define the set P_v^s of *secret propagation paths* to be the paths in the graph G that connect the source of secrets (that can take the value M_s or M_s') to v, where all the vertices in the path are secret-dependent. In particular, P_v^s is non-empty. Indeed, if $v \in V^s$, then at least one predecessor $u \in V$ of v must also be secret-dependent,

which recursively constructs a path in P_u^s.

Let us consider a strict subgraph $G' = (V', E') \subsetneq G$ and a vertex $v \in V'^s$. A path p in P_v^s is said to be *reentrant* in G' if it contains at least one vertex $u \in V \setminus V'$, i.e., a vertex that is outside of G'. Note that this u also belongs to $V^s \setminus V'$ because all nodes in p are secret-dependent by definition of P_v^s. If all paths in P_v^s are reentrant in G', then we say that the vertex v is *fully reentrant* in G'.

Example. Let us consider again the system Figure 4 (a) where $\mathcal{P}$ describes programs where secret data, which is for example initially stored in a well-specified general-purpose register, can be used as the address of a memory instruction. Because the CPU itself does not contain caches or other components whose timing depends on memory addresses, for v designating the commit signal (for LeaVe [27], Tan et al. [28], ConjunCT [24] or VeloCT [25]), or a microarchitectural PC signal (μCFI [26]), v is fully-reentrant in the CPU, if we consider G being the whole platform, and G' being the CPU.

C. Contracts

The key idea of our work is to define *platform timing contracts* that capture the possible reentrant information flows in the CPU that are allowed by a given platform. Since the interface between the output signals of a CPU and the rest of a SoC is usually much simpler than the hardware-software interface, these contracts are expected to be simpler than existing hardware-software contracts.

Interface. We base our analysis on the widely-studied RISC-V ISA. The output signals of a RISC-V CPU are all through memory interfaces, which are made of data and control signals. Data signals transport the data to be read or written to or from the memory. Control signals transport the information about the memory transaction, such as the memory address, handshake signals, and depending on the memory bus protocol (e.g., AMBA AXI [30] or TileLink [31]), further control signals such as the type of transaction and the size of the data to be read or written. Input signals of a RISC-V CPU can be more diverse. They include memory input data and control signals, interrupt lines, and various other control signals such as clock gating signals.

Platform timing contracts. The goal of platform timing contracts is to ensure that state elements that capture timing differences, such as the commit signal or a microarchitectural program counter, *are not fully reentrant* in the CPU. To ensure this, a platform timing contract specifies an upper bound of the information flows between the output signals of the CPU and its input signals. We express a platform timing contract as a list of rules, listed per output signal of the CPU.

Example contracts. We provide some example contracts for a RISC-V CPU with a Von Neumann architecture, where we model the memory interface as an outbound address bus (`addr`), an inbound and an outbound memory data port (respectively `rdata` and `wdata`), and a simple valid-ready handshake protocol (inbound `resp` and outbound `req`), similar to the `req` and `gnt` handshake signals in the Hardware Processing Engines (HWPE) 2.0 interface protocol [39]. Equation 1 defines a simple integration contract for a platform with ideal memory that responds with a constant timing (i.e., that is address-independent). A change in the memory transaction control signals from the CPU can only change the returned value. Only the request signal (`req`) can affect the existence, and hence the timing, of a memory response. The diamond $\Diamond$ underlines that the right-hand side of the implication can happen at any time and for any number of times after the left-hand side. Equation 2 adds timing dependence on the memory address, which is typical of structures like caches. In this contract, `req` and `addr` have an identical clause. Indeed, a change in the address of a read, for example, can change a cache miss into a cache hit, changing the timing, and hence the value at some point in time, of the response, including the `gmt` control signal and the `rdata` data signals. Equation 3 models data-dependent interrupts and timing, which might occur, for example, if some peripheral can be controlled through memory-mapped registers such as an interrupt controller [40].

Conditional refinements. Equation 3 implies that writing tainted, i.e., secret, data to memory with a specific address that is not tainted might always trigger an interrupt or affect the timing of the memory response. In some settings, this might be true only for specific address ranges where peripherals are mapped, but parts of the memory address range can typically be used in a data-independent timing. To refine this contract, we introduce address-dependent contracts as in Equation 4, where the right-hand side of the implication depends on the memory address set `periph_range`, which is a fixed set of addresses. For example, this can be achieved with RISC-V Physical Memory Protection (PMP) [41] in a setting where the most privileged execution mode does not access secret data and protects the `periph_range` address range from being accessed from lower privileges, in the philosophy of trusted execution environments [42]–[51].

$$\begin{aligned} \text{req} &\rightarrow \Diamond\{\text{gnt}, \text{rdata}\} \\ \text{we} &\rightarrow \Diamond\{\text{rdata}\} \\ \text{addr} &\rightarrow \Diamond\{\text{rdata}\} \\ \text{wdata} &\rightarrow \Diamond\{\text{rdata}\} \end{aligned} \tag{1}$$

$$\begin{aligned} \text{req} &\rightarrow \Diamond\{\text{gnt}, \text{rdata}\} \\ \text{we} &\rightarrow \Diamond\{\text{gnt}, \text{rdata}\} \\ \text{addr} &\rightarrow \Diamond\{\text{gnt}, \text{rdata}\} \\ \text{wdata} &\rightarrow \Diamond\{\text{rdata}\} \end{aligned} \tag{2}$$

$$\begin{aligned} \text{req} &\rightarrow \Diamond\{\text{gnt}, \text{rdata}, \text{interrupt}\} \\ \text{we} &\rightarrow \Diamond\{\text{gnt}, \text{rdata}, \text{interrupt}\} \\ \text{addr} &\rightarrow \Diamond\{\text{gnt}, \text{rdata}, \text{interrupt}\} \\ \text{wdata} &\rightarrow \Diamond\{\text{gnt}, \text{rdata}, \text{interrupt}\} \end{aligned} \tag{3}$$

$$\begin{aligned} \text{req} &\rightarrow \Diamond\{\text{gnt}, \text{rdata}, \text{interrupt}\} \\ \text{we} &\rightarrow \Diamond\{\text{gnt}, \text{rdata}, \text{interrupt}\} \\ \text{addr} &\rightarrow \Diamond\{\text{gnt}, \text{rdata}, \text{interrupt}\} \\ \text{wdata} &\rightarrow \Diamond\{\text{rdata}, \\ &\quad \text{gnt if addr in periph\_range} \\ &\quad \text{interrupt if addr in periph\_range}\} \end{aligned} \tag{4}$$

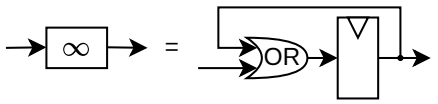

Fig. 5. Sticky-one operator. The sticky-one is reset to zero (not illustrated in the figure) and sticks to 1 once it is set.

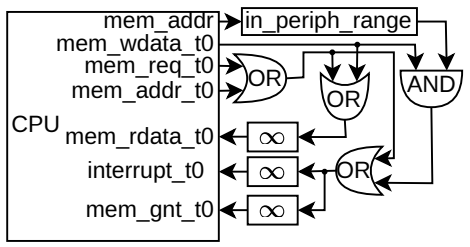

Fig. 6. Platform timing contract instrumentation for a design whose information flows are tracked with dynamic information flow tracking instrumentation. Signals that end with `t0` represent the taint signals [34], [37].

Ordering of platform timing contracts. Like for sinks, we can define a partial ordering of platform timing contracts. We say that a contract A is (inclusively) stronger than a contract B if for every output of the CPU (i.e., the left-hand side of the arrow in the contract), the set of possible CPU inputs tainted by this output in contract A is a subset of the set of possible CPU inputs tainted by this output in contract B. Said otherwise, contract A is stronger than contract B if it allows for fewer possible information flows. For example, Equation 4 is stronger than Equation 3, and Equation 2 is stronger than Equation 3. A CPU that is verified to have a constant-time behavior with respect to a stronger platform timing contract is also verified with respect to a weaker platform timing contract.

FS: TODO Continue here

IV. Instrumentation

We show that such contracts can be expressed as a synthesizable platform timing contract instrumentation (PTCI). Because it is an information flow property, the PTCI cannot be directly applied to the CPU under verification (in the original CPU before IFT instrumentation or miter construct, the notion of taint [26], [37] or two-trace property [24], [24], [27], [28] does not yet exist). Instead, the PTCI is applied to the construct that supports information flows, which is either a circuit with DIFT instrumentation, or a miter (see Section II-C).

Sticky-one operator. Before we construct the PTCI, we introduce the sticky-one operator in Figure 5, whose functionality is to stay at one as soon as it is set to one, and only comes back to zero when the system is reset. This operator will model the diamond ◊ operator in the instrumentations.

PTCI for DIFT. We first construct the PTCI for CPUs instrumented for DIFT [34], [37]. A contract information flow from a CPU output to a CPU input can be immediately expressed as a wire between the two signals' corresponding taint signals, separated by a sticky-one operator to accommodate for information flows that might return later (i.e., to accommodate for the ◊ operator). When several CPU outputs have a contract

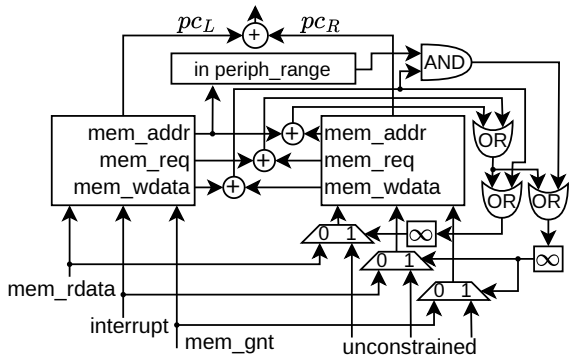

Fig. 7. Platform timing contract instrumentation for a design whose information flows are tracked with a miter.

information flow to a single CPU input, the corresponding taint signals can be combined using a logical OR operation, meaning that the input can be influenced by any of these outputs. The PTCI for the contract in Equation 4 is shown in Figure 6 when relying on DIFT instrumentation.

PTCI for miter. To construct the PTCI for Equation 4 for a CPU that relies on a miter construct, the PTCI must ensure that the CPU inputs that are influenced by the CPU outputs are distinct between the two copies of the CPU. This is achieved using `XOR` operations between the CPU outputs of the two copies. Ultimately, another `XOR` operation with the result of this difference is performed with the input, deciding whether for a given input, the two copies of the CPU will receive the same input, or one copy will receive a flipped input. Note that there is no conceptual difference between the PTCI for a miter construct and the PTCI for a DIFT instrumentation, yet the latter benefits from the abstraction built into the DIFT instrumentation, which allows for a more compact and arguably more intuitive PTCI.

V. Evaluation

In this section, we first present the PTCI implementation and evaluate its performance and the resulting soundness improvement (Section V-A). For comprehensiveness, we then also verify that not only the CPU, but also the rest of the platform complies with its own contract terms, i.e., that a platform will not propagate more taint than what the contract stipulates (Section V-B).

Evaluation setup. We execute the performance evaluations on a server equipped with an Intel Xeon E5-2667 CPU with 256 GB RAM. We use the 2-stage Sodor RISC-V CPU [52], which is a simple CPU commonly used in constant-time verification benchmarks [27], [28].

A. PTCI implementation

The limited interface between the CPU and the platform makes the PTCIs simple, as earlier illustrated in Figure 6 and Figure 7. We implement a Yosys synthesizer pass [53] that instruments a CPU that is either instrumented with taint tracking logic such as GLIFT [34] or CellIFT [37], or part of a miter construct. The synthesizer pass takes as input a

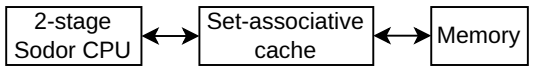

Fig. 8. Simple SoC architecture with a CPU, a cache, and memory.

TABLE I
VERIFICATION RESULTS FOR THE ORIGINAL SODOR (O) DESIGN, FOR PTCI-INSTRUMENTED (EQUATION 2) SODOR (P). RESULTS ARE "PASS" IF MEMORY INSTRUCTIONS ARE CONSIDERED SAFE WITH REGARD TO ALL THEIR OPERANDS (AND IN PARTICULAR WITH REGARD TO THE ADDRESS).
TODO ADD RESULTS FOR NO-VIOLATION SCENARIOS.

	μCFI	**VeloCT**
Time (O)	8 min	1 min
Result (O)	Pass ✓	Pass ✓
Time (P)	10 min	1 min
Result (P)	Fail ✗	Fail ✗

Verilog description of the CPU under test in one of these two configurations (instrumented for taint tracking or in a miter) and outputs a Verilog design instrumented with the PTCI. In addition to the CPU's Verilog description, the synthesizer pass takes a conditional contract as input, similar to the contracts presented in Section III, and takes a machine-readable description of the CPU's interface ports.

B. Soundness improvement with PTCI

We have shown that the security guarantees provided by μCFI and VeloCT may not hold when the CPU is integrated into a platform that contains caches or other elements that have an address-dependent timing, for example described by the contract defined in Equation 2. We first verified Sodor with μCFI and VeloCT (ConjunCT is not open-source but is superseded by VeloCT). We now instrument Sodor with the PTCI and verify it again with the same tools. We report the verification results, as well as the time taken by each tool to formulate a proof in Table I. We observe that the PTCI indeed can correct the unsoundness of μCFI and VeloCT with regard to the platform integration such as cache side channels. As a result, the PTCI automatically transforms the verification of a CPU in isolation into a verification of a CPU in a contract-compliant platform.

Platform verification. Finally, we verify the compliance of the platform with the platform timing contract. Because these contracts specify a form of non-interference, they can be verified conveniently with miter constructs, similar to how most techniques verify constant-time properties for CPUs [24], [25], [27], [28]. We verify three platforms. The `cache` platform is a simple SoC architecture composed of a CPU, a set-associative cache and memory, as depicted in Figure 8. The `nocache` platform is like the `cache` platform, but the cache is removed, connecting the CPU directly to the memory, which is modeled as having no address-dependent latency. The `interrupt` platform is like the `cache` platform, but also adds an uncached interrupt controller at the address `0x1000`, which generates an interrupt signal when a non-zero value is written to this address. For each of these platforms, we verify Contract A that we define as in Equation 2, Contract B that we define as in Equation 3, Contract C that we define as in Equation 4 where

TABLE II
VERIFICATION OF THE COMPLIANCE OF PLATFORMS TO PTCI.

Contract	`nocache`	`cache`	`interrupt`
A	Pass ✓	Fail ✗	Fail ✗
B	Pass ✓	Pass ✓	Fail ✗
C	Pass ✓	Pass ✓	Pass ✓
D	Pass ✓	Pass ✓	Fail ✗

we constrain `periph_range` to be `0x1000` to `0x1000`, and Contract D that is like Contract C but with a `periph_range` of `0x2000` to `0x2FFF`. We construct miters whose designs under test is the platform seen from the CPU's perspective, i.e., whose inputs are the outputs of the CPU, and whose outputs are the inputs of the CPU. Note that the CPU itself is not part of this miter. To verify, for example, the last arrow of Equation 4, we enforce all inputs of the miter to be identical between the two sides of the miter except for the `mem_wdata` input, and we additionally constrain the `mem_addr` input, identical between the two sides of the miter, not to be in the `periph_range` range. We then verify that only `mem_rdata` can be affected by changes to `mem_wdata` by `xor`'ing the outputs of `mem_gnt` for the two copies of the platform in the miter together and expressing a SAT formula on this signal, which should never be set. Table II summarizes the verification results obtained with Cadence Jaspergold for the different platforms. The total duration of the experiment does not exceed one minute.

FS: TODO report the simulation results for no violation found.

VI. DISCUSSION

ISAs like RISC-V [41] communicate with the rest of the system through memory interfaces. Some other ISAs like x86 [54] have special I/O instructions that are not memory-mapped. The reasoning about platform timing contracts for these ISAs would need to account for the different ways in which they interact with the system and might incur more clauses in the contracts to account for these two separate channels for communicating with the system. System-level instructions of ISAs like MIPS [55] (e.g., the `mfc0` instruction) or ARM [56] (e.g., the `SVC` instruction) also require additional ports to include into contracts to describe how the system might react.

FS: Takeaway box: Given a platform specification, PTCI includes reentrant information flows in the verification of constant-time properties, even if the underlying verification technique initially did not account for them

VII. RELATED WORK

Previous work has proposed means of mitigating some platform-related side channels such as cache timing attacks [57]–[61] or network on chip interference [62]–[69]. Our work is orthogonal to these works and models a platform that is tolerated to have a specific set of potential side channels. Hardware-software contracts and some constant-time techniques [22], [27], [28] typically consider the assumption that addresses of memory operations might eventually influence timing. But not all platforms will have such flows, and some platforms will also create reentrant timing flows from other CPU output signals as discussed in Section III. The PTCI that we introduce proposes fine-grained platform integration contracts, and makes them synthesizable so that they can

directly be used upon techniques that are unaware of reentrant information flows.

VIII. Conclusion

We introduced platform timing contracts as a natural extension of hardware-software contracts to reason about timing channels in full-system contexts rather than isolated CPUs. To make such reasoning feasible with existing verification frameworks, we proposed an automatic instrumentation that accounts for platform-specific timing effects while preserving compatibility with prior techniques. Our evaluation shows that this approach incurs minimal verification overhead while uncovering timing channels that would otherwise remain invisible in isolation.

References

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

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security*, 2018.

[3] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar *et al.*, "Fallout: Leaking data on meltdown-resistant cpus," in *ACM SIGSAC*, 2019.

[4] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross privilege boundary data sampling," in *ACM SIGSAC*, 2019.

[5] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *IEEE SP*, 2019.

[6] S. Van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "Cacheout: Leaking data on Intel cpus via cache evictions," in *IEEE SP*, 2021.

[7] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, "Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks," in *USENIX Security*, 2021.

[8] J. Wikner and K. Razavi, "Retbleed: Arbitrary speculative code execution with return instructions," in *USENIX Security*, 2022.

[9] D. Trujillo, J. Wikner, and K. Razavi, "Inception: Exposing new attack surfaces with training in transient execution," in *USENIX Security*, 2023.

[10] J. Wikner, D. Trujillo, and K. Razavi, "Phantom: Exploiting decoder-detectable mispredictions," in *MICRO*, 2023.

[11] J. Wikner and K. Razavi, "Breaking the barrier: Post-barrier spectre attacks," in *IEEE SP*, 2024.

[12] S. Ruegge, J. Wikner, and K. Razavi, "Branch privilege injection: Compromising spectre v2 hardware mitigations by exploiting branch predictor race conditions," in *USENIX Security*, 2025.

[13] D. J. Bernstein, "Cache-timing attacks on aes," 2005.

[14] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE SP*, 2015.

[15] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack," in *USENIX Security*, 2014.

[16] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: A timing attack on openssl constant time RSA," in *CHES 2016*. Springer, 2016.

[17] A. Moghimi, T. Eisenbarth, and B. Sunar, "Memjam: A false dependency attack against constant-time crypto implementations in SGX," in *Topics in Cryptology*. Springer, 2018.

[18] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR," in *ACM CCS*, 2016.

[19] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-CPU attacks," in *USENIX Security*, 2016.

[20] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Addendum to ridl: Rogue in-flight data load," 2019, https://mdsattacks.com/files/ridl-addendum.pdf.

[21] B. Herzog, S. Reif, J. Preis, W. Schröder-Preikschat, and T. Hönig, "The price of meltdown and spectre: Energy overhead of mitigations at operating system level," in *Proceedings of the 14th European Workshop on Systems Security*, 2021, pp. 8–14.

[22] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation," in *IEEE SP*, 2021.

[23] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein, "Revizor: Testing black-box cpus against speculation contracts," in *ASPLOS*, 2022.

[24] S. Dinesh, M. Parthasarathy, and C. W. Fletcher, "Conjunct: Learning inductive invariants to prove unbounded instruction safety against microarchitectural timing attacks," in *IEEE SP*, 2024.

[25] S. Dinesh, Y. Zhu, and C. W. Fletcher, "H-houdini: Scalable invariant learning," in *ASPLOS*, 2025.

[26] K. Ceesay-Seitz, F. Solt, and K. Razavi, "μcfi: Formal verification of microarchitectural control-flow integrity," in *CCS*, 2024.

[27] Z. Wang, G. Mohr, K. von Gleissenthall, J. Reineke, and M. Guarnieri, "Specification and verification of side-channel security for open-source processors via leakage contracts," in *CCS*, 2023.

[28] Q. Tan, Y. Yang, T. Bourgeat, S. Malik, and M. Yan, "Rtl verification for secure speculation using contract shadow logic," *ASPLOS*, 2025.

[29] Y. Hsiao, N. Nikoleris, A. Khyzha, D. P. Mulligan, G. Petri, C. W. Fletcher, and C. Trippel, "Rtl2mμpath: Multi-μpath synthesis with applications to hardware security verification," in *MICRO*, 2024.

[30] "Amba axi and ace protocol specification," Arm Ltd., Tech. Rep. ARM IHI 0022H, 2017. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/IHI0022H_amba_axi_protocol_spec.pdf

[31] CHIPS Alliance, *TileLink Specification*, 2020, available online. [Online]. Available: https://chipalliance.org/specs/tilelink/tilelink-spec-1.8.1.html

[32] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Topics in Cryptology*. Springer, 2006.

[33] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX, USA, 2016. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_almeida.pdf

[34] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *ASPLOS*, 2009.

[35] W. Hu, A. Ardeshiricham, and R. Kastner, "Hardware information flow tracking," *ACM CSUR*, 2021.

[36] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *DATE*, 2017.

[37] F. Solt, B. Gras, and K. Razavi, "Cellift: Leveraging cells for scalable and precise dynamic information flow tracking in rtl," in *USENIX Security*, 2022.

[38] F. Solt and K. Razavi, "Hybridift: Scalable memory-aware dynamic information flow tracking for hardware," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024.

[39] PULP Platform Documentation, *HWPE-Mem Protocol*, PULP Platform, 2024, online; accessed July 4, 2025. [Online]. Available: https://hwpe-doc.readthedocs.io/en/latest/protocols.html

[40] RISC-V International Task Group, "Risc-v platform-level interrupt controller specification," RISC-V International, Technical Report Version 1.0.0 (ratified March 11, 2023), 2023, specifies PLIC architecture, interrupt prioritization, claims/completion. [Online]. Available: https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic-1.0.0.pdf

[41] RISC-V Foundation, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*, version 1.12-draft ed., RISC-V International, 2021, defines Physical Memory Protection (PMP) in Chapter 3. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/latest/download/riscv-privileged.pdf

[42] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *EuroSys*, 2020.

[43] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *USENIX Security*, 2016.

[44] F. McKeen, I. Alexandrovich, H. Berenzon, C. Rozas, J. Shafi, P. Shanbhogue, and V. Savagaonkar, "Innovative instructions and software model for isolated execution," *ACM HASP*, 2013.

[45] ARM Ltd., "Arm security technology—building a secure system using trustzone technology," ARM Ltd., Whitepaper, 2009. [Online]. Available: https://developer.arm.com/documentation/den0028/a/

[46] P. Nasahl, R. Schilling, M. Werner, and S. Mangard, "Hector-v: A heterogeneous cpu architecture for a secure risc-v execution environment," in *arXiv preprint arXiv:2009.05262*, 2020.

[47] B. McGillion, T. Dettenborn, T. Nyman, and N. Asokan, "Open-tee: An open virtual trusted execution environment," *arXiv preprint arXiv:1506.07367*, 2015.

[48] I. Lebedev, K. Hogan, J. Drean, D. Kohlbrenner, D. Lee, K. Asanović, D. Song, and S. Devadas, "Sanctorum: A lightweight security monitor for secure enclaves," in *arXiv preprint arXiv:1812.10605*, 2018.

[49] M. Schneider, R. J. Masti, S. Shinde, S. Capkun, and R. Perez, "Sok: Hardware-supported trusted execution environments," in *arXiv preprint arXiv:2205.12742*, 2022.

[50] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "Mi6: Secure enclaves in a speculative out-of-order processor," in *arXiv preprint arXiv:1812.09822*, 2018.

[51] F. Brasser, P. Jauernig, F. Pustelnik, A.-R. Sadeghi, and E. Stapf, "Trusted container extensions for container-based confidential computing," in *arXiv preprint arXiv:2205.05747*, 2022.

[52] B. A. Research, "riscv-sodor," https://github.com/ucb-bar/riscv-sodor, [Online; accessed 8-April-2025].

[53] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free verilog synthesis suite," in *Austrochip*, 2013.

[54] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2022, vol. 1-3, available at https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[55] C. Price, *MIPS IV Instruction Set (Revision 3.2)*, MIPS Technologies, Inc., Mountain View, CA, USA, Sep. 1995, mIPS IV ISA specification.

[56] Arm Limited, *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile (Issue G.a)*, Arm Limited, Cambridge, UK, 2021, dDI 0487G.a – includes Armv8.7 extensions.

[57] A. Kar, X. Liu, Y. Kim, G. Saileshwar, H. Kim, and T. Krishna, "Mitigating timing-based noc side-channel attacks with llc remapping," *IEEE Computer Architecture Letters*, vol. 22, no. 1, 2023.

[58] G. Saileshwar and M. Qureshi, "{MIRAGE}: Mitigating {Conflict-Based} cache attacks with a practical {Fully-Associative} design," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[59] L. Giner, S. Steinegger, A. Purnal, M. Eichlseder, T. Unterluggauer, S. Mangard, and D. Gruss, "Scatter and split securely: Defeating cache contention and occupancy attacks," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023.

[60] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "{ScatterCache}: thwarting cache attacks via cache set randomization," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[61] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018.

[62] H. M. Wassel, Y. Gao, J. K. Oberg, T. Huffmire, R. Kastner, F. T. Chong, and T. Sherwood, "Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, 2013.

[63] Y. Wang and G. E. Suh, "Efficient timing channel protection for on-chip networks," in *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*. IEEE, 2012.

[64] M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki, "A statically scheduled time-division-multiplexed network-on-chip for real-time systems," in *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*. IEEE, 2012.

[65] H. M. Wassel, Y. Gao, J. K. Oberg, T. Huffmire, R. Kastner, F. T. Chong, and T. Sherwood, "Networks on chip with provable security properties," *IEEE Micro*, vol. 34, no. 3, 2014.

[66] A. Psarras, J. Lee, I. Seitanidis, C. Nicopoulos, and G. Dimitrakopoulos, "Phasenoc: Versatile network traffic isolation through tdm-scheduled virtual channels," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 5, 2015.

[67] M. G. Alonso, J. Flich, M. Turki, and D. Bertozzi, "A low-latency and flexible tdm noc for strong isolation in security-critical systems," in *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 2019.

[68] M. S. Sadeghi, S. B. Sarmadi, and S. Hessabi, "Toward on-chip network security using runtime isolation mapping," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 3, 2019.

[69] A. Shalaby, Y. Tavva, T. E. Carlson, and L.-S. Peh, "Sentry-noc: A statically-scheduled noc for secure socs," in *Proceedings of the 15th IEEE/ACM International Symposium on Networks-on-Chip*, 2021.