

Devil-in-FPGA: Coherent Trojans in Reconfigurable Heterogeneous Systems

Cristiano Rodrigues¹, Shahin Roozkhosh², Sandro Pinto¹, and Renato Mancuso²

¹Universidade do Minho, ²Boston University

Abstract—Given the demands of the modern technological landscape, the rate of change in specialized hardware is ever-increasing. However, the manufacturing industry witnesses the capital-intensive demand for heterogeneous acceleration on top of an already complex quest for optimizing tradeoffs between high performance, security, and efficiency. The heterogeneous acceleration mandates support for various clusters, e.g., GPUs, NPUs, TPUs, and FPGAs, empowered with inter-cluster coherence. This introduces yet another dimension that promotes the flexible deployment of customized hardware models, whose impact on overall system security remains uncertain.

This paper exposes the potential threat of integrating latent trojans hidden within custom hardware into the coherence system. Notably, by acknowledging the necessity to maintain a consistent unified memory view between processing elements, we showcase how current coherence mechanisms lack a systematic design compatible with tightly integrated PEs and why security can be overlooked in favor of performance. We first recognize the gravity of the challenge arising from the need for multi-vendor integration in which hardware modules are often regarded as black boxes designed by third parties. Next, we categorize the fundamental operations that can be performed by a malicious hardware trojan with access to the coherence subsystem. We then introduce a flexible framework to define, implement, and demonstrate coherent trojans on commercially available platforms with cache-coherent FPGAs. We demonstrate that such trojans can carry out stealth attacks to break confidentiality, integrity, and availability.

Index Terms—Hardware Trojans, Coherence Trojans, Coherence Attack, Heterogeneous Computing, FPGAs.

1. Introduction

Heterogeneous computing architectures are witnessing a paradigm shift in response to a growing demand for specialized Processing Elements (PEs), dictated by the booming landscape of data-intensive systems and applications. Specialized hardware heterogeneity has proven its effectiveness, leading to the widespread integration of specialized units like GPUs, TPUs, NPUs, and FPGAs onto commercial platforms. The heterogeneous computing paradigm is now embraced by major players in general-purpose computing, like Intel and AMD, who have made strategic moves in FPGA company acquisitions (e.g., AMD acquired Xilinx in a bold \$50 billion move) and are racing toward chiplet architectures [1]. This

transition has already encountered legal [2] and security challenges, with hardware trojans becoming a reality [3]. Nonetheless, multi-vendor heterogeneous systems are poised to shape the near future of computing platforms [4–6].

A core issue in heterogeneous systems is inter-PE data movement, with manufacturers investing substantial efforts in high-performance microarchitectural support for data coherence to ensure a unified view of memory between PEs. Unfortunately, current system coherence approaches are not hardened against malicious actors, leaving a substantial security gap that can hinder the smooth adoption of multi-vendor heterogeneous systems. Co-located PEs are black boxes designed by third parties. Yet, they are expected to cooperate to maintain a consistent view of the broader system memory. Well-established coherence protocols with different trade-offs are already in use, including ACE [7], CHI [8], and CXL [9]. However, state-of-the-art hardware-assisted coherence subsystems were designed with the key assumption of a *single-vendor paradigm* with *in-house-only privileges*. We note that this assumption is increasingly inapplicable to the emerging heterogeneous systems landscape. Recent literature has hinted at the risks associated with coherent hardware trojans [10–12] and (timing) side-channel attacks [13], with some proof-of-concept attacks [14]. However, the research community has yet to fully understand the security implications of coherence in multi-vendor systems.

In this paper, we explore the gap left open by security-agnostic data coherence. Cache coherence is part of a processor’s critical inner workings and if compromised, can lead to severe consequences. Alarming, no existing security technology, such as Trusted Execution Environments (TEEs), IOMMU, or encryption, can fully protect against coherence attacks from misbehaving PEs. Given the ubiquitous need for coherent accelerators, we aim to stimulate a discussion to devise sound security-hardened coherence approaches. Motivated in this direction, we focus on answering the following research question: “*To what extent and level of effort can malicious PEs exploit coherence to undermine heterogeneous platforms’ security and safety guarantees?*”

We first introduce a Coherent Trojan model to capture the behavior of “any” malicious module that can stealthily interact with the coherence subsystem by mimicking the behavior of a cache. Next, we analyze how it can compromise the Confidentiality, Integrity, and Availability within the system, regardless of any specific platform, protocol, or computing technology. Then, we select a commercially available CPU+FPGA platform where support for inter-cluster

coherence is already available, noting that FPGAs can be seen as a generalization of application-specific accelerators. On said platform, we carry out a full implementation of a configurable framework, namely the *Trojan Tool*,¹ to demonstrate the practicality of mounting coherence-based attacks on mainstream OSES and systems. Using *Trojan Tool*, we demonstrate four proof-of-concept (PoC) attacks that compromise the CIA triad: (i) a physical address disclosure attack that bypasses ASLR, (ii) a privilege escalation attack that grants root access to an unprivileged application, (iii) a denial-of-service attack that degrades a safety-critical application’s performance, and (iv) a confidentiality attack that extracts encrypted secrets from an Arm TrustZone secure service. Finally, we provide a set of principles for building secure-by-design solutions and propose extensions to current coherence protocols to make them security-hardened.

In summary, this paper makes the following contributions:

- (i) We introduce the *Coherent Trojan* model, a formalization of coherence-based hardware trojans applicable to diverse heterogeneous computing technologies (§4);
- (ii) We design a specialized Instruction Set Architecture (ISA) to abstract the formulation of attacks from platform-specific coherence protocols;
- (iii) We implement *Trojan Tool*, a configurable framework based on our ISA, to facilitate the development of coherence attacks and the testing of countermeasures (§5);
- (iv) We demonstrate four proof-of-concept attacks that compromise the Confidentiality, Integrity, and Availability (CIA) triad on a commercial CPU+FPGA platform (§6);
- (v) We propose actionable countermeasures, outline key principles for designing security-hardened coherence protocols, and proposed a long-term solution (§8).

2. Background and Related Work

Scope. Hardware Trojans can be stealthily inserted into any coherent PE, such as a coherent FPGA or GPU. Due to their intrinsic flexibility, this paper focuses on the system model comprising a PE with a tightly-coupled FPGA (cf. Fig. 1). Our survey of such platforms in Table 4 (§A) reveals that the ACE protocol [7] is a prevalent choice for coherent FPGAs. Consequently, we narrow our scope to CPU+FPGA heterogeneous systems with a broadcast-based coherence protocol like ACE (§D).

2.1. Cache Coherence

If caches in two or more subsystems (e.g., CPU cluster and accelerator) are kept coherent, they are said to belong to the same coherence domain. Cache-coherent PEs within the same domain must expose interfaces to follow the same *coherence protocol*. Coherence protocols are typically implemented following either a broadcast-based or a directory-based coherence management philosophy [15, 16]. In the former, upon a cache miss, coherence transactions are broadcast system-wide to all caches, regardless of whether they

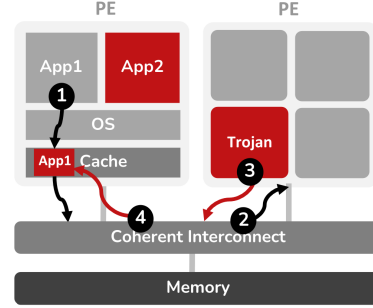


Figure 1: Illustration of the system model with a high-level example of malicious coherence traffic. Upon a cache miss (1), the coherent interconnect broadcasts a snoop request to all PEs (2). In this example, the Hardware Trojan fakes a cache hit and returns a tampered cache line (3), controlling *App1* cache line payload (4).

hold the requested line. In the latter, coherence transactions are forwarded only to the caches that may contain the requested line. Coherence requests, typically referred to as snoops, denote the process by which a PE probes other PEs to determine the state of a line in their caches. Snoops are orchestrated by a coherent interconnect (CI), which is responsible for coordinating the progress of all coherent PEs, facilitating direct master-to-master data transfer whenever possible. Regardless of the protocol used by the PEs, each needs to correctly implement an interface to interact with the system-wide coherence pathways at the hardware level so that they **can** be included in the same coherence domain.

Coherent Acceleration. Traditionally, FPGAs only had support for high-performance non-coherent bus interfaces for memory transfers, as seen in platforms like the Xilinx Zynq-7000 and Altera Arria 10 (cf. Table 4 §A). However, to support tighter integration, modern **coherent** FPGAs such as the Xilinx Zynq UltraScale+ and Altera Stratix 10 now include specialized coherent interfaces like AXI Coherence Extension (ACE) [7], Coherent Hub Interface (CHI) [8], and Compute Express Link (CXL) [9]. These interfaces maintain a consistent data view between the CPU cluster and accelerators on the FPGA by enabling bidirectional snooping. This allows the CPU cluster to snoop any FPGA-side cache(s) upon an LLC miss, and conversely, the FPGA can snoop the CPU cluster when it needs to access a physical address that might be cached in the CPU-side LLC.

Coherence Backstabbing. The bidirectional snooping capability of modern coherent FPGAs can be leveraged for purposes other than acceleration. A technique known as *coherence backstabbing* [17] used these coherent interfaces to intercept, profile, and redirect memory accesses. Implemented on Commercial Off-The-Shelf (COTS) platforms, it demonstrates that custom logic on an FPGA can interact with a CPU cluster via cache-coherence protocols to manipulate the addressing semantics of memory accesses. Originally introduced for benign purposes under the name Coherence-Aided Elective and Seamless Alternative Routing (CAESAR), this method was designed to profile applications and reduce memory access interference in real-time systems.

1. We open-source *Trojan Tool* and all associated artifacts at [link](#).

TABLE 1: ASIC-to-CPU and FPGA-to-CPU Hardware Trojans, compared by target (platform, interface), Trojan (coherent, configurable, reproducible), attack, and countermeasures (coherent-aware, address space-aware, implemented). Zynq-7000 (Z7), Zynq-7010 (Z7-10), Cyclone V (CV), and Zynq UltraScale+ (ZU+).

Attack	Yr	Target		Trojan			Attack			Counter.		
		Plat	Int	Coh	Cf	Rp	C	I	A	CA	AS	IM
ASIC	Kumar et al. [18]	'18	N.A	N.A	○	○	○	○	●	○	○	●
	Khan et al. [19]	'20	Gem5	N.A	○	○	○	●	●	○	○	●
	Kumar et al. [20]	'23	Gem5	N.A	○	○	○	○	●	○	○	●
	Shikama et al. [21]	'23	Gem5	N.A	○ ¹	○	○	●	○	○	○	○
	Chacon et al. [12, 14]	'24	Gem5	N.A	●	○	○	○	●	○	○	●
FPGA	Jacob et al. [22]	'17	Z7	GP/HP	○	○	○	○	●	○	○	○
	Kim et al. [10]	'17	Z7	ACP	●	○	○	○	●	○	○	○
	Chaudhuri et al. [23]	'18	CV	ACP	●	○	○	○	●	○	○	○
	Benhani et al. [24]	'19	Z7-10	GP/HP	○	○	○	○	●	○	○	○
	Gross et al. [11]	'22	ZU+	ACP	●	○	○	○	●	○	○	○
	This paper	'25	ZU+	ACE	●	●	●	●	●	●	●	●

¹ Trojan intercepts coherent traffic but does not exploit coherence protocol itself.

2.2. Hardware Trojans.

Hardware Trojans are a malicious, intentional modification of circuitry designed to compromise a system’s security by leaking information, altering data, or causing denial of service when triggered [3]. A particularly concerning target is the coherence subsystem, where coherent PEs have unrestricted access to system memory through the coherence fabric. Despite extensive research on Hardware Trojans targeting plethora of different circuits [25], such as L1 data caches [19], NoC components [18, 20, 21], and specialized hardware [26, 27], the cache-coherence subsystem remains relatively underexplored. However, emerging evidence suggests that it poses a non-negligible threat to the security of heterogeneous systems [10–12, 14, 21].

Table 1 provides a comparative overview of prior work on Hardware Trojans where an external unit (ASIC or FPGA) targets a host CPU. Prior research has predominantly focused on Trojans that leverage non-coherent or half-coherent interfaces. Non-coherent Trojans (○) are typically inserted into the physical communication path between two endpoints to intercept or manipulate traffic. For instance, such Trojans have been designed to attack NoC interconnects to degrade performance or perform DoS attacks [18, 21], corrupt data packets [20], or eavesdrop on inter-PE communication [21]. Other examples include Hardware Trojans targeting the L1-L2 cache interface [19] or directly accessing boot memory [22]. Half-coherent Trojans (◐) have leveraged Arm’s Accelerator Coherency Port (ACP) to interact with the LLC, and degrade performance by injecting invalidation requests [10] or to subvert secure boot [11, 23]. In contrast, Trojans leveraging full-coherent interfaces (●) can reach any PE without being on a direct communication path. The first such Trojan was introduced by Chacon et al. [12, 14], who manipulated the MOESI Hammer protocol to tamper with data from other chiplets. However, the authors only validated simple attacks on a simulated environment assuming prior knowledge of target addresses.

In contrast to prior work, we present the first full-coherent

Trojan realized on COTS silicon and demonstrate end-to-end PoC attacks that simultaneously breach confidentiality, integrity, and availability. The Trojan is fully configurable and released as an open-source framework, enabling reproducibility, systematic exploration of coherence-exploitable vulnerabilities, and rapid evaluation of defenses. Unlike prior work on full-coherent interfaces [12, 14], our Trojan can be inserted post-manufacturing via bitstream manipulation (§2.3), a more immediate and realistic vector than ASIC supply-chain insertion, enabling retroactive compromise of deployed platforms. We are also the first to exploit the ACE protocol for fine-grained, intra-chiplet attacks, whereas prior work [12, 14] focuses on inter-chiplet attacks. Finally, while the countermeasures proposed in [12] protect coherence traffic and target inter-chiplet NoCs, they do not integrate with intra-chiplet mechanisms that define address spaces and security policies, as we propose in §8.

2.3. Trojan Insertion in FPGAs

Unlike Hardware Trojans in ASICs, only inserted *pre- or during-manufacturing*, FPGA-based Trojans can also be inserted *post-manufacturing* by modifying the bitstream. This distinction expands the attack surface, providing additional opportunities for an attacker to introduce malicious logic.

Design-time trojan insertion in FPGAs. In line with classic Hardware Trojans, design-time insertion can occur through three main vectors during the hardware development lifecycle [25, 28]: through malicious insiders, compromised third-party IP cores [29–31], or malicious EDA tools [32, 33]. A malicious insider could, for instance, embed covert logic that exfiltrates secrets or alters behavior without detection by the verification team. Similarly, the integration by a benign designer of untrusted third-party IP can introduce hidden backdoors or trigger circuits [3, 30, 34]. Finally, the design toolchain itself can be exploited as an attack surface. EDA tools can be tampered with [33] and covertly embed malicious logic into the final bitstream during synthesis. Moreover, recent work have demonstrated the complete compromise of IEEE 1735 IP encryption across major EDA vendors [32], allowing attackers to decrypt, modify, and re-encrypt protected IP cores, enabling the replacement of legitimate IP libraries with trojanized versions.

Post-manufacturing trojan insertion in bitstreams. Inserting hardware Trojans into FPGA bitstreams post-manufacturing has been demonstrated as a practical attack against commercial products [35, 36]. Although modern FPGAs employ bitstream encryption and authentication, these protections have been compromised in several widely deployed families, such as Xilinx 7-Series and UltraScale(+), enabling adversaries to extract, modify, re-encrypt bitstreams [37–39]. Such capabilities make it possible to embed malicious logic and redeploy trojanized configurations without vendor authorization. The feasibility of such attacks is amplified by open-source frameworks that facilitate bitstream reverse-engineering and modification [26, 40–42]. While these tools require device-specific expertise, they significantly lower the

barrier for adversaries to manipulate deployed configurations. Moreover, recent attacks [39, 42, 43] show that dynamic partial reconfiguration (DPR) [44, 45] can also be exploited to insert or remove Trojan on-demand. Although experimental, they highlight a realistic path for runtime modifications that may evade static detection mechanisms.

3. Problem Statement

This section outlines the threat model and highlights the difficulty of protecting coherence interfaces in the presence of hardware trojans in the coherence subsystem.

3.1. Threat Model

At the software level, we assume the attacker controls either a malicious user-space application (§6.1 to §6.3) or a malicious OS (§6.4). In contrast to prior work [12, 14], we make no assumptions about the attacker’s knowledge of the victim’s physical memory layout. However, the attacker is assumed to have enough information² to distinguish victim snoops from general traffic. The Hardware Trojan’s payload is triggered by specific memory access patterns [3, 19, 28, 29, 46]. At the hardware level, we assume that the Hardware Trojans can be inserted without initial administrator privileges, either at design time or at runtime via privilege escalation. Trojans inserted at design time will bypass secure-boot and authentication once they are embedded in the vetted bitstream. At runtime, trojans can be inserted through DPR [42] and even by unprivileged tenants [47]. This capability is consistent with state-of-the-art FPGA-to-CPU attacks [48–50] and supported by established bitstream manipulation techniques (§2.3). Physical attacks and directly exploiting privileged software are out of scope.

3.2. Unprotected but Trusted Coherence

Cache coherence is a fundamental aspect of a processor’s operations. Coherence protocols were originally designed for trusted, single-party environments, where security measures were deemed unnecessary. However, breaches in these protocols can have severe consequences. The core issue lies in the disconnect between access control policies enforced at the virtual address level and their absence at the physical address level. Since physical addresses are the sole “currency” of coherence subsystems, they remain agnostic to virtual address spaces. In a multi-party computing paradigm, this limitation becomes problematic, as current security technologies cannot prevent malicious players from crafting arbitrary coherence messages. Since coherence protocols were not designed with security in mind, they fail to account for untrusted players within the coherence domain. It follows that virtual memory isolation is violated if untrusted hardware players are present in the coherence domain.

2. The attacker can profile the source code offline or, if closed-source, monitor the firmware activity at runtime, e.g., its context switching.

Our motivation is to leverage the backstabbing technique to emphasize the potential consequences of this gap. Although the backstabbing technique has previously been implemented on a coherent FPGA for benign purposes [17, 51], in this work, we back-stab the coherence domain to highlight the aforementioned security gap. We argue that address-based virtual memory isolation can be compromised by an Hardware Trojan using the coherence backstabbing technique. We conclude that as addressing schemes evolve, revisiting the role of spatial isolation in modern heterogeneous systems becomes essential. Integrating spatial isolation into future coherence subsystems (as proposed in §8) is a critical step toward advancing vendor-agnostic heterogeneous platforms with security-hardened cache coherence.

4. Coherent Trojan

This section introduces the concept of *Coherent Trojan* (§4.1), which resonates with the paper’s threat model (§3.1).

4.1. Definition

We coin the term *Coherent Trojan* to specifically refer to any *Hardware Trojan* capable of *Coherence Backstabbing*, that is, mimicking coherence protocols “once” added to a coherence domain. It is important to emphasize that this definition remains platform-agnostic, encompassing various hardware modules such as accelerators capable of communication through supported coherence protocols upon inclusion. A *Coherent Trojan* can perform two main categories of operations. At the top level, we distinguish between passive and active operations (*PO* and *AO*). The term *Coherent Trojan* evokes that the custom logic is capable of inserting itself on the back of a coherence domain where (1) useful information about the behavior of all coherent elements is *passively* monitored, and (2) the state of the upstream caches, and the semantics, timing, and correctness of downstream memory accesses can be *actively* impacted.

4.2. Passive Operations (*PO*)

Passive operations are those where a *Coherent Trojan* only observes the flow of coherence messages with *zero* impact on its semantics. When performing a *PO*, a *Coherent Trojan* emulates an empty cache’s behavior via the **rapid** acknowledgment of received snoops with the proper response, as a discreet empty cache would. The snoop acknowledgment process (necessary to ensure coherent protocol correctness) is orthogonal to the passive nature of observing snoops.

4.2.1. Sniffing. The term “*Sniff*” conveys the nature of *PO*’s, in which a *Coherent Trojan* silently observes the system’s activity. Sniffing a snoop request alone provides valuable information, and it is crucial to differentiate between a snoop request and a snoop response, as the latter is always expected upon receiving the former. Table 2 exemplifies three different snoops issued at times T_1 , T_2 , and T_3 . The

TABLE 2: Example of snoops sniffed by a *Coherent Trojan*.

Time	Pseudo Command	Snoop Action	Snoop Attributes	Address
T_1	<i>SnpRdShd</i> (0x1234)	Read (Rd)	CL in Shared (<i>Shd</i>) state	0x1234
T_2	<i>SnpInv</i> (0x5678)	Invalidate (Inv)	-	0x5678
T_3	<i>SnpBrr</i> ()	Barrier (Brr)	-	-

TABLE 3: Examples of *Passive* and *Active* operations.

Mode	Pseudo Command	Type	Snoop Attributes	Address
<i>Passive</i>	<i>SniffRd</i> (0x5678, $\forall$)	Sniff	CL is any ($\forall$) state	0x5678
<i>Active</i>	<i>Claim</i> (0x5678, <i>Drt</i>)	Claim	CL is dirty (<i>Drt</i>)	0x5678
<i>Active</i>	<i>IssueRd</i> (0x5678, <i>Unq</i>)	Issue	CL is unique (<i>Unq</i>)	0x5678
<i>Active</i>	<i>TamperDly</i> (0x1234, ∞)	Tamper	Delaying indefinitely (∞)	0x1234

three last right table columns represent the three primary components capturing the nature of any coherence snoop request: (i) its action, (ii) attributes, and (iii) target address.

Sniff Example. Table 2 illustrates examples of sniffing operations. At T_1 , the *Coherent Trojan* receives a snoop targeting address 0x1234 to read data in *shared* state. This operation represents a data access request, revealing the address of interest and the desired CL state. At T_2 , another snoop targets address 0x5678 to invalidate it. Unlike the previous one, this snoop lacks a specific attribute, indicating an invalidation request with no further details. Finally, at T_3 , a snoop targets the entire cache to initiate a barrier request, which aims to synchronize all pending memory transactions across all CLs. From the sniffed snoops (cf. Table 2), one can capture not only (i) snoop action, (ii) address, and (iii) attributes, but also (iv) their order, and (v) timing of arrival. The chronological sequence of the snoops (T_1 , T_2 , T_3) and their inter-arrival timing help in understanding the temporal flow of activities done by other PEs on the system.

4.3. Active Operations (AO)

Active operations impact the flow of coherence traffic within the targeted coherence domain and are classified into three high-level categories: (i) Claim, (ii) Issue, or (iii) Tamper. From other coherent PEs’ perspective, active operations remain perceived as if a legitimate coherent component is trying to maintain system coherence.

4.3.1. Claiming. Any PE added to the coherence domain gets snooped regardless of whether it has any copy of the target CL. When a *Coherent Trojan* claims possession of a CL, it actively acknowledges a snoop and “fakes” a cache hit. This allows a *Coherent Trojan* to assert control over specific CL data and its attributes, maintaining control over the system even **after** passing the CL to other PEs. For instance, in MOESI-based protocols, when a dirty copy of data is passed, the new owner assumes responsibility for updating memory. Conversely, passing a CL in shared state allows the *Coherent Trojan* to fake maintaining a copy of it and receive notifications when the CL is modified.

Claim Example. Targeting a known address, e.g. 0x5678, the *Coherent Trojan* wants to cause another PE to overwrite its value in memory. By sniffing the snoop activity, when a read

request on the target address is detect, the *Coherent Trojan* fabricates a response faking a hit (*Claim*(0x5678, *Drt*) in Table 3) passing a cache line (CL) marked as “dirty”. This CL is then forwarded by the CI to the requesting PE. Not only **any** data can be passed by the *Coherent Trojan*, but the requesting PE is now responsible for updating the memory on address 0x5678 with the dirty CL, leaving no record of the *Coherent Trojan* behind.

4.3.2. Issue snoops Requests. Faking the behavior of a genuine cache, a *Coherent Trojan* can also issue specific types of snoops to be broadcasted. For instance, it can orchestrate cache invalidation operations and enforce synchronization barriers among threads.

Issue Example. Upon detecting an update to a shared dataset at address 0x5678, the *Coherent Trojan* issues a coherent read request with a unique flag (*IssueRd*(0x5678, *Unq*) in Table 3). The CI then identifies that the *Coherent Trojan* requires exclusive ownership of the CL and broadcasts an “Invalidate” request to all other PEs on that address, causing them to invalidate their local copies. The *Coherent Trojan* retrieves the data with exclusive ownership and responds to subsequent accesses to address 0x5678 until it decides to issue a write-back command.

4.3.3. Tampering. Certain attack scenarios require the *Coherent Trojan* to manipulate coherence traffic and alter the timing and correctness of the coherence protocol. In such cases, the *Coherent Trojan* can intentionally introduce protocol violations or delays to freeze the system. In snoop-based protocols, the CI broadcasts snoops to all PEs in the domain and expects a reply from each of them. Thus, by delaying its responses, the *Coherent Trojan* causes the coherence subsystem to slow down or stall.

Tamper Example. Targeting a known address, e.g., 0x1234, the *Coherent Trojan* is programmed to deliberately halting the system and delay the acknowledgment of snoops indefinitely (*TamperDly*(0x1234, ∞)). Suppose a PE broadcasts the first snoop as per Table 2 i.e., (*SnpRdShd*(0x1234)). When the *Coherent Trojan* detects the snoop with the target address, it never acknowledges its receipt, causing the coherence subsystem to wait indefinitely for a reply.

4.4. Coherence Attack Scenario

A *Coherent Trojan*’s attack complexity can be as advanced as required, allowing any combination of operations and conditions. Three simplified examples are theorized to show how a *Coherent Trojan* can compromise the principles of (i) confidentiality by exposing `private_secrets`, (ii) integrity by executing unauthorized services, and (iii) availability by introducing Denial-of-Service (DoS) attacks via arbitrary delays in the coherence protocol.

For simplicity, assume the victim (Alg. 1) executes with a cold cache, resulting in issuing a snoop request to all system coherent elements, including the *Coherent Trojan*. **Confidentiality Attack** (Alg. 2), assumes a known

Algorithm 1 Victim Pseudocode with Physical Addresses

```
0xC1A111: private_secret = 0xBAADBABE
0xC1A222: if (authenticated) { ... }
0xC1A333: void real_time_task() { ... }
```

Algorithm 2 Confidentiality Attack Scenario

- 1: SniffRd(0xC1A111, $\forall$)
 - 2: IssueRd(0xC1A111, *Shd*)
 - 3: **passively** sniffs read snoop addresses to detect when the secret enters the victim cache, **if** sought, **actively** issues a snoop to read on the `private_secret` address
-

physical address³ where `private_secret` is stored, e.g., 0xC1A111. The *Coherent Trojan* passively *Sniffs* read snoop addresses to detect when the secret enters the victim’s cache. Upon observation, it actively *Issues* a read on the `private_secret` address broadcasting a snoop request. This prompts the victim to reply with a hit on the requested CL and to relay the CL data, effectively leaking the secret.

In the **Integrity Attack** (Alg. 3), assuming knowledge of the authentication code’s physical address, e.g., 0xC1A222, the *Coherent Trojan* initially follows a similar passive phase. Once a target address is *Sniffed*, the *Coherent Trojan* detects the key instruction CL, thereby determining the authentication status. Once detected, the *Coherent Trojan* **Claims** CL ownership, followed by passing a tampered instruction CL. This could involve swapping a conditional branch instruction with a nop, resulting in the *if* statement being evaluated as *true*, thereby mounting a credential bypass attack.

Finally, in the **Availability Attack** (Alg. 4), assuming knowledge of the real-time code’s physical address, i.e., 0xC1A333, the *Coherent Trojan* seeks the snoop address marking the beginning of the time-critical phase to tamper by introducing programmable delays. In this case, the *Coherent Trojan* effectively replies to the snoop request, saying, “I don’t have the data,” but with a delay. This forces the CI to wait and, therefore, throttle the victim’s real-time execution.

5. Rapid Prototyping of Coherent Trojans.

We designed and implemented a *Trojan Tool* as a runtime-configurable tool, specifically designed with a dual purpose—advancing practical research and being a hackers’ playground. The *Coherent Trojan* (§4.1) encapsulates a concept that extends beyond any specific architecture, while the *Trojan Tool* is a practical implementation of it. It was developed to serve as a “Swiss army knife” for coherence attacks on CPU+FPGA systems, allowing us to demonstrate and explore coherence attacks in emerging hardware models. While reconfigurability is a fundamental aspect of the *Trojan Tool*, it remains optional for a *Coherent Trojan*. It might choose to operate autonomously without relying on the attacker’s configuration, with the attack scenario pre-programmed into the hardware. For scenarios where it is desired real-time control over the *Coherent Trojan*, a configuration port allows

3. Later (§6) we will relax this assumption to showcase more sophisticated attacks without specific knowledge of the victim’s physical mapping.

Algorithm 3 Integrity Attack Scenario

- 1: SniffRd(0xC1A222, $\forall$)
 - 2: Claim(0xC1A222, *Unq*)
 - 3: **passively** monitors snoop addresses seeking authentication CL, **if** sought, **actively** replies claiming CL ownership
-

Algorithm 4 Availability Attack Scenario

- 1: SniffRd(0xC1A333, $\forall$)
 - 2: TamperDly(0xC1A333, ∞)
 - 3: **passively** seeks the snoop address marking the beginning of the time-critical phase, **if** sought, **actively** introduces delays to throttle the victim’s real-time execution.
-

for online (re)configuration, providing maximum control during the online attack phase. This approach preserves stealthiness while enabling the test of what-if scenarios efficiently. Additionally, the *Coherent Trojan* can be hidden inside any larger coherent IP, providing covert implementation of *Coherent Trojans* within existing hardware such as a GPU. *Trojan Tool* has three main units: (i) control and perform (ii) active or (iii) passive operations.

5.1. Control Unit (CU)

The Control Unit (CU) orchestrates the operational units influencing the coherence subsystem. The CU has an input command which determines the specifics of the influence. Serving as an abstraction layer, the CU interfaces between the command and operational units, ensuring that only the latter is platform-specific. This separation improves portability across hardware platforms. The CU input commands can be viewed as the core functionalities supported by our implementation. Specifically, we (i) designed a specialized ISA that supports passive/active operations and (ii) implemented the CU to support these instructions as input commands.

Table 3 lists a selection of supported input commands, with their corresponding pseudo-code. Similar to how a high-level programming language is translated into a sequence of assembly instructions executed based on the underlying ISA, an attack scenario (see §4.4) is composed of a sequence of input commands which is served to the control unit. A comprehensive list of all supported functionalities with their corresponding pseudo-command is outlined in §C to showcase the full capabilities of the *Trojan Tool*.

Finally, *Coherent Trojan* is designed to be modular, allowing an extendable instruction set with an optional internal Scratchpad Memory (SPM), enabling *Trojan Tool* to operate autonomously without runtime configuration. The SPM can also serve for logging, pattern analysis (§6.1 and §B), and intermediate calculations. Results, such as completion flags or data streams, can be logged on the SPM or directed to an attacker-owned memory. This flexibility facilitates transmission to external systems in remote attacks.

5.2. Passive Unit (PU)

The Passive Unit (PU) implements all the passive operations defined in §4.2. The PU receives control signals

from the CU and monitors the coherence flow. Further, our design facilitates the bidirectional exchange of information between the CU and the PU. Specifically, the CU requests the snoop(s) of interest (first row in [Table 3](#)), while the PU transmits the sniffed metadata back to the CU.

Sniffing. The PU emulates an empty cache by consistently replying that it does not hold the CL requested by the snoop request. The PU can operate in two modes: always-on mode or selective mode. In always-on mode, the PU performs its intended operation for every snoop received. In selective mode, the PU operates based on a trigger condition or filter. This trigger condition/filter is flexible and can be configured at runtime by the CU. The trigger condition can be any combination of a specific address and a specific snoop type.

5.3. Active Unit (AU)

The Active Unit (AU) implements all active operations defined in [§4.3](#). Like the PU, the AU is configured by the CU and crafts ACE ([§D](#)) signals to (i) initiate or (ii) respond to snoops. The AU acts as a typical cache that provides data on a cache hit and fetches data on a cache miss. Hence, the CU may configure the AU with data to (i) inject into requesting PEs’ caches by *claiming* a cache hit ([§4.3.1](#)) or (ii) snatch CLs from other PEs by *issuing* a cache miss ([§4.3.2](#)).

Claim operations. The AU leverages the ACE AC, CR, and CD channels to **emulate a cache hit**. During a `Claim` operation, the AU receives snoops on ACE AC and replies based on the desired cache state (dirty, shared, unique) on ACE CR. When claiming ownership of a CL, the AU must pass it on ACE CD when snooped. To maintain coherence, the AU, having falsely claimed a hit, fetches the original CL from memory or injects intentionally corrupted data.

Issue operations. The AU **emulates a cache-miss** using the ACE AR/R and AW/W channels to request the allocation of a CL (`IssueRd` operation, see [§4.3.2](#)) or perform an invalidation (`IssueInv` operation, see [§4.3.2](#)). Both ACE AR/R and AW/W channels can be used to issue synchronization or barrier transactions (`IssueBrr` operation, see [§4.3.2](#)).

Tamper operations. The AU can use all ACE channels to perform tamper operations (`Tamper` operation [§4.3.3](#)) by misusing the ACE protocol. The AU is always dormant and operates only when commanded by the CU.

6. Proof-of-Concept Attacks

In this section, we showcase a fraction of the potential consequences a *Coherent Trojan* can have on a heterogeneous system. We demonstrate four distinct attacks that compromise the target system’s Confidentiality, Integrity, and Availability (CIA triad) using the *Trojan Tool* ([§5](#)). We will show ([§6.1](#)) that if the victim runs known software code, the attacker can (i) unveil its physical location and (ii) use the exposed physical addresses to mount more complex attacks ([§6.2](#), [§6.3](#), and [§6.4](#)). All attacks comprise two phases: a profiling phase, in which the attacker runs the victim

offline to derive unique fingerprints from its coherence traffic, and an exploitation phase on the live target, in which the Trojan delivers its payload whenever it detects the profiled fingerprints. All attacks were validated on AMD Xilinx Zynq Ultrascale+ ZCU102 and ZCU104.

6.1. Deanonimization Attack

To reveal the physical location of the target victim, we develop a *Deanonimization Trojan* (*D-Trojan*) that monitors the coherence traffic and detects its unique snoop patterns. We leverage *D-Trojan* to mount a *Deanonimization Attack*, which can both subvert first (EL1) and second-stage (EL2) virtual memory translation, as well as obfuscation techniques like address space layout randomization (ASLR). *D-Trojan* is a key piece of subsequent attacks ([§6.2](#), [§6.3](#), [§6.4](#)), allowing us to relax the assumptions made in [§4](#) and exploit the victim without pre-knowledge of its physical location.

Overview. The *D-Trojan*, shown in [Fig. 2a](#), sniffs coherence traffic from the CCI, searching for fingerprints of target code or data. This fingerprint, selected offline by the attacker, uniquely identifies the victim through specific data and instruction patterns. The attacker configures *D-Trojan* to intercept all DDR-fetched data before it reaches the CCI, while preserving protocol correctness and data integrity. To achieve this, *D-Trojan* emulates a cache that consistently acknowledges possession of snooped CLs, creating a window (between memory fetch and delivery to the CCI) where the *D-Trojan* can scan the CL for the target pattern. If the pattern of interest is detected, it logs the snoop address, revealing the physical location of the target data or code.

Attack flow. *Trojan Tool* is configured by the attacker with pattern of interest which will be searched for on the coherence traffic. The *Trojan Tool* sniffs the coherence traffic, and on each snoop, issues a `claim` operation ([§4.3](#), [Appendix §D](#)) faking the ownership of the requested CL. Then it issues a non-coherent memory read to fetch the target CL from DDR, and compares it with the pattern of interest. If equal, the *D-Trojan* registers the snoop address, revealing the physical address of the victim. Then, to keep integrity and correctness of the protocol, regardless of pattern recognition result, *D-Trojan* returns the CL to CCI.

Attack setup. We conducted two synthetic experiments to assess the *D-Trojan* capability to deanonymize code (*T#1*) and data (*T#2*). We ran the experiments in two scenarios: (i) the victim running in another OS process (*xPross* scenario) and (ii) the victim running in another virtual machine (*xVM* scenario). In each run, the victim’s code/data location changes. In *T#1*, the victim is randomly placed in memory. In *T#2*, the victim allocates a buffer at a random memory location, with buffer sizes ranging from 64 B (CL) to 1 MiB (LLC size). On each run, the victim’s physical address is compared with *D-Trojan*’s output for validity. We use Linux OS v6.1 and the Bao hypervisor [[52](#)] v1.0.0.

Results. We ran both *T#1* and *T#2* experiments 1000 times. In *T#1*, the *xPross* scenario achieved a 100% success rate, while

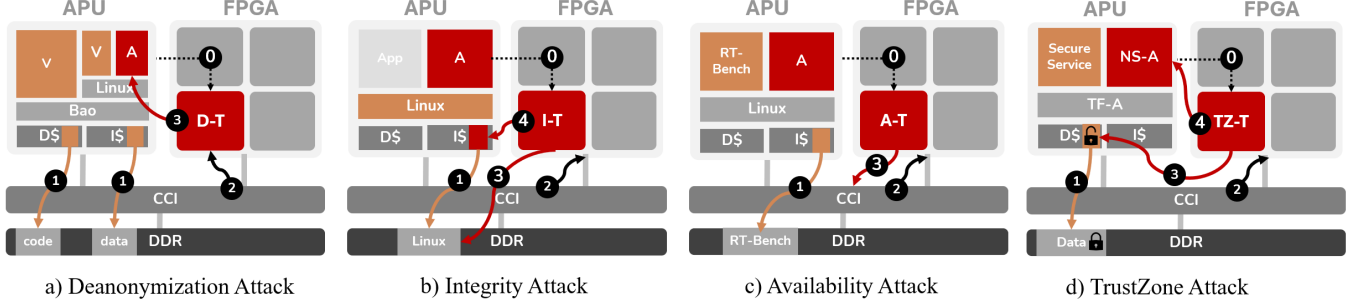


Figure 2: Illustration of four PoC Attacks. Orange shows the Victim (V), red shows attacker and Trojan activity. (0) Corresponds to the configuration of the *Trojan Tool* to create attack-specific *Coherent Trojan*. (1) Denotes a cache miss in either the data (D\$) or instruction (I\$) cache. (2) Snoop request from CCI. Steps (3) and (4) vary by attack: In *Deanonimization Attack*, (3) outputs the deanonimized physical address. In *Privilege Escalation Attack*, (3) reads a CL with target Linux code, (4) modifies the cached Linux code. In *Performance Degradation Attack*, (3) sends delayed snoop response. In *TrustZone Attack*, (3) reads plaintext from cache, (4) leaks plaintext to attacker.

the *xVM* scenario achieved 99.8%. In the *T#2* experiment, we configured the victim with buffer sizes of 64 B, 4 KiB, 64 KiB, 128 KiB, and 1 MiB, achieving success rates of 0%, 22.6%, 54.4%, 64.9%, and 100%, respectively. The discrepancy between *T#1* and *T#2* experiments are rooted in how code and data interact with the cache. Code consistently triggers a cache miss on first execution, allowing *D-Trojan* to detect snoops immediately. Data, however, does not generate coherence traffic until it is evicted and refetched. The *T#2* success rate variability is due to low cache evictions, once we ran it in worst-case scenario (standalone). While such low cache usage is unlikely in a real-world scenarios, attackers can artificially increase evictions if necessary [53].

6.2. Integrity Attack: Corrupt Arbitrary Memory

To showcase how a *Coherent Trojan* can tamper with the system’s Integrity, we developed *Poison Data Trojan (PD-Trojan)* and mounted a privileged escalation attack that corrupts memory in a controlled way and elevates the privilege level of user-space applications (EL0) to root (EL1).

Overview. Linux uses capabilities to allow user-space applications to perform privileged operations without granting full root permissions. Linux enforces capabilities through the `ns_capable_common` function (link), which verifies if the calling process has the permissions to execute the requested system calls. The *Privilege Escalation Attack* (Fig. 2b) manipulates `ns_capable_common` by modifying the if-statement that checks permissions, replacing it with a `nop` instruction, ensuring it never fails. By leveraging access to the Linux source code, the attacker identifies the `ns_capable_common` function fingerprint and configures the *D-Trojan* to monitor the coherent traffic in search of it. Upon detection, the *D-Trojan* triggers the *PD-Trojan* to deliver the payload tampering with the Linux target function.

Attack flow. The *Trojan Tool* is configured by the attacker with a static malicious payload, which is used to replace the victim’s CL of interest. The *PD-Trojan*, upon a *D-Trojan* trigger, swaps individual victim instructions, causing the victim to execute tampered code. Since the *PD-Trojan* never

changes the content of the DDR, it does not leave any trace of its operation, allowing its execution to go undetected.

Attack setup. We developed a user-space Linux application that exploits the `setresuid` system call to escalate privileges to root, bypassing `sudo` authentication. In Linux, all processes are assigned Unique Identifiers (UIDs) to enforce user permissions, with the root process having the UID 0 (unrestricted access). Any process that can change its UID to 0 gains root privileges. We use *PD-Trojan* to bypass `setresuid` permission checks, enabling an user-space process to set all UIDs to 0. After obtaining root privileges, the attacker modifies the `sudoers` file to ensure all subsequent applications launched by the user inherit root permissions.

Results. We ran the attack 1000 times and achieved a success rate of 87.1%. This success rate does not affect the attack’s effectiveness, just its speed. If the first run fails, the attacker just has to try until the privilege escalation succeeds.

6.3. Availability Attack: Degrade Performance

To showcase how a *Coherent Trojan* can tamper with the system’s Availability, we developed a *Performance Degradation Trojan (PD-Trojan)* that selectively interferes with the execution of a safety-critical application while keeping the rest of the system intact.

Overview. *Performance Degradation Attack* (Fig. 2c) keeps the correctness of the coherent protocol while inflicting arbitrary overheads on selected operations. Per the ACE protocol, the CCI must wait for all outstanding snoop requests to complete before further action. *PD-Trojan* exploits this feature and delays snoop responses of the target code to increase its overall latency. To selectively interfere with a specific operation, two distinct patterns are used: (i) one to identify the point where the attack begins, and (ii) another to mark when interference should stop. This technique enables targeted attacks on specific code structures at CL granularity, minimizing collateral impact on other applications.

Attack flow. In contrast to the other paper attacks, the *PD-Trojan* delivers its payload during a time window instead of in a single-shot operation. The *Performance Degradation*

Attack uses *D-Trojan* to find a start and end pattern which form the attack aperture where the performance will be degraded. Once the first start pattern is found, the *PD-Trojan* begins inflicting arbitrary delays on each subsequent victim snoop until the end pattern is found. At that time, the *PD-Trojan* stops interfering with the snoop timings and becomes dormant until the next start pattern is found.

Attack setup. To demonstrate this attack, we use RT-Bench [54] (commit 456161e1), to emulate a safety-critical application. RT-Bench provides several benchmarks with real-time-oriented metrics, such as deadline miss rate, job utilization, and execution time. For the purposes of this paper, we focus solely on total execution time. From the suite of RT-Bench benchmarks, we chose the IsolBench and Disparity benchmarks. The former is tailored to measure cross-core cache interference, while the latter is a data-intensive benchmark that generates high cache traffic.

Results. We ran each benchmark 20 times in five experiments: (i) a control run with no inflicted delay and (ii) four runs with delays ranging from 50 to 200 clock cycles. As expected, we observe a clear relationship between the inflicted delays and the total execution time. We design the *PD-Trojan* to selectively target specific safety-critical operations. However, we observed a degradation in performance across the entire system. In retrospect, this is an obvious outcome, since, per the ACE protocol, the CCI must wait for all snoop responses before proceeding. Although only a portion of the overall snoop responses were delayed, all were affected, which reveals a deeper performance degradation potential which may lead to a complete denial of services.

6.4. TrustZone Attack: Leak Secure World Secrets

In previous attacks (§6.1, §6.2, and §6.3), we demonstrated that isolation enforced by an operating system (EL1) and a hypervisor (EL2) can be subverted. Although these attacks can severely impact a plethora of systems, security-sensitive applications are often deployed in trusted execution environments (TEEs), such as Arm’s TrustZone [55]. TEEs provide strong protection by physically isolating security-critical applications from the rest of the (untrusted) system. Furthermore, to enhance the runtime security provided by TEEs, sensitive data is typically encrypted. While TEEs and encryption provide robust protection against strong adversaries, this duo is ineffective against a *Coherent Trojan*. To demonstrate the stealthiness of a *Coherent Trojan*, we develop an attack capable of extracting secrets from TrustZone, even when those secrets are encrypted in memory.

Overview. *TrustZone Attack* (Fig. 2d), leverages *TrustZone Trojan* (*TZ-Trojan*) to leak secure world data. At any moment data can be (i) at-rest, (ii) in-transit, or (iii) in-use. While data at-rest and in-transit can be effectively secured through encryption, data in-use relies on TEEs for protection. TEEs are effective at preventing direct accesses to secure memory. However, *Coherent Trojan* can bypass TEE isolation through coherent means. When the ciphertext is decrypted, the

resulting plaintext is cached. The *TZ-Trojan* fakes a cache miss, forcing the CCI to serve the cached plaintext instead of the ciphertext from main memory. We assume knowledge of the target code, allowing the attacker to derive fingerprints that reveal when decryption begins.

Attack flow. *TZ-Trojan* leaks the secret by issuing a coherent read snoop, `IssueRd(pAddr, Attr)`, which the CCI broadcasts to all coherent PEs. The attribute (`Attr`) of `IssueRd` is configured to accept a CL in any ($\forall$) state. Once *D-Trojan* deanonymizes the target ciphertext (`pAddr`), *TZ-Trojan* delivers its payload by faking a cache miss, leaking the plaintext from the caches where it resides.

Attack Setup. We emulate a scenario where the Non-Secure (NS) world receives an ciphertext from an external entity which is forward to secure service to perform authentication over it. The secure service decrypts it, checks if the plaintext meets the expected authentication condition, and returns authentication status. The attacker code runs at EL1, emulating a malicious OS, and invokes the victim through a secure monitor call (SMC). While the secure service decrypts the payload, *TZ-Trojan* extracts the plaintext from the cache.

Results. We ran the attack 1000 times with a success rate of 100%, defined by the leaked and original plaintext being equal. This high success rate is rooted in the level of control the attacker has over the secure service. Since the NS is the one invoking the victim, the attacker can precisely time its execution window, maximizing the likelihood of success.

7. Discussion

I/O Memory Management Units. IOMMUs, such as the Arm SMMU, cannot fully protect coherent ports. PEs typically use physically indexed and physically tagged (PIPT) caches, meaning only physical addresses are sent downstream. Thus IOMMUs, which enforce policies on virtual addresses, cannot protect coherence transactions between memory hierarchies (e.g., L2 and LLC/DDR) that exclusively operate on physical addresses. As a result, fully coherent interfaces, such as Arm CCI-400/500 ACE ports, are not protected. There is a distinction, however, between one-way coherent interfaces, like ACE-Lite, and fully coherent interfaces such as ACE. The former operate on virtual addresses and, in well-designed systems, are protected by an IOMMU. As an example of this duality, all traffic generated by AMD Zynq Ultrascale+ one-way coherent interfaces passes through the IOMMU, while ACE traffic does not. This lack of protection is harmless when all PEs are trusted, but it falls short when multi-party (potentially malicious) PEs are colocated on the same SoC.

I/O Memory Protection Units. IOMPUs provide an additional layer of protection by enforcing memory access control at the physical address level. IOMPUs are becoming readily available in modern platforms [56], e.g., all AMD Xilinx CPU+FPGA feature a Xilinx MPU. IOMPUs can be configured to allow or restrict PE access to specific memory regions. This suggests that even if the FPGA bypasses the IOMMU through coherent means, it could still be blocked

by an IOMPU. However, a *Coherent Trojan* can mount more advanced attacks to indirectly retrieve the target CL. For instance, a trojan could inject malicious code into the APU, forcing it to access data on behalf of the FPGA (cf. §6.2), or leak data allocated by the CCI (cf. §6.4). To accomplish the latter, the trojan would, upon receiving a snoop request, respond with a miss and immediately issue a read snoop for the same CL. The first snoop would be served from main memory and allocated in the cache, while the second snoop, issued by the trojan, would be served from that PE cache, thereby bypassing the IOMPU.

8. Countermeasures

The most effective countermeasure to protect a system against a *Coherent Trojan* is to disable the coherence support of PEs deployed in FPGA⁴. This approach offers a hotfix (via software patch) for security-critical systems, aiming at assuring full protection against coherence attacks. However, this solution is impractical for most systems, as it entails a complete lack of FPGA-side coherence capabilities, incurring in a performance hit for accelerators that would otherwise benefit from tight collaboration with CPU-side applications. In this section we present short-term actionable countermeasures (§8.1), outline key principles for building security-hardened coherence subsystems (§8.2), and articulate a vision for long-term secure-by-design solutions (§8.3).

8.1. Short-term Countermeasures

Homomorphic encryption. Homomorphic encryption offers a robust security foundation for security-critical applications that require full protection against *Coherent Trojans* while maintaining the ability to perform computation on unpatched hardware. Unlike traditional encryption, homomorphic encryption allows direct computation over ciphertext, ensuring data remains protected *at rest*, *in transit*, and *in use*, protecting data against data exfiltration attacks. However, its high computational overhead makes it a niche solution, practical only for specific, highly critical computations.

Coherence sandboxing. In systems where a trusted entity manages the deployment of third-party IPs (e.g., multi-tenant cloud-based FPGA services like Amazon AWS EC2 F1 [57]), the untrusted hardware can be sandboxed with its ability to issue and receive coherence messages restricted. The trusted entity would only forward snoops to allowed address ranges and reply with a miss to snoops on forbidden ranges. Cloud providers offering FPGA sharing services can implement coherence sandboxing in their trusted shell to securely integrate third-party IPs into the coherence domain.

Coherence sanitizer. The CI can be augmented to police coherence traffic, ensuring compliance with coherence protocol specifications and system security policies. For instance, the CI can enforce the timeliness of snoop requests by setting

4. On AMD Xilinx CPU+FPGA devices, the firmware can configure a register on the CCI to exclude the FPGA from the coherency domain.

programmable maximum response thresholds. If the snooped PEs fail to respond within this timeframe, the CI treats it as a miss and fetches the CL from main memory, preventing attacks such as described in §6.3. To prevent malicious snoops, the CI could enforce memory access rights defined in a programmable access control table. Similar to Coherence Sandboxing, snoops would be forwarded to PEs only if their attributes (e.g., snoop type and physical address) comply with the access criteria encoded in the table.

8.2. Principles of Security-Hardened Coherence

We now provide a perspective on the future of security-hardened coherence, reasoning about the principles for building secure-by-design long-term solution.

Key principles. We hereby provide three principles, in addition to typical requirements regarding scalability and throughput, that constitute the hallmarks of a good solution for security-hardened coherence.

P1 - Flexibility: An ideal solution shall allow cache-coherent PEs to execute entities with varying privilege levels. The privilege level of the coherence messages generated by the PE must dynamically follow the privilege level of the associated software entities. In this sense, solutions that partition PEs in more-or-less static coherence domains are in direct violation of the flexibility principle.

P2 - Address space awareness: Identifying data items by their physical address alone is insufficient to determine the security policy to apply to coherence exchanges. Security-hardened coherence should be able to associate data items with the security policies in effect for said item. Data items to be kept coherent shall be identified by their physical address, the address space(s) they belong to, and the security policies in effect for said address space.

P3 - Implicit configuration & transparency: OS and hypervisors already carry the burden of correctly configuring complex security and memory access policies. Security-hardened coherence shall not further aggravate this burden. The coherence subsystem shall work in tandem with the facilities that already exist in the system to define address spaces and security policies. Just like traditional hardware-aided coherence is transparent with regard to the platform software, so should security-hardened coherence. This conflicts with solutions involving explicitly managed configuration tables to encode coherence filtering policies.

Future of security-hardened coherence. Early solutions, such as those proposed in §8.1 and the *Coherence Message Checkers* (CMC) [12], hold great potential in the short term, but they are ill-suited to fully meet principles **P1** to **P3**. The CMC effectively behaves as an IOMPU, deployed in a trusted CI, that blocks coherence traffic from malicious PEs attempting to access other PEs' memory regions. A trusted OS running on a secure processor configures this IOMPU-like structure and, upon a PE OS request, allocates physical memory and sets access policies. However, this static, per-region partitioning with ad-hoc policy tables is

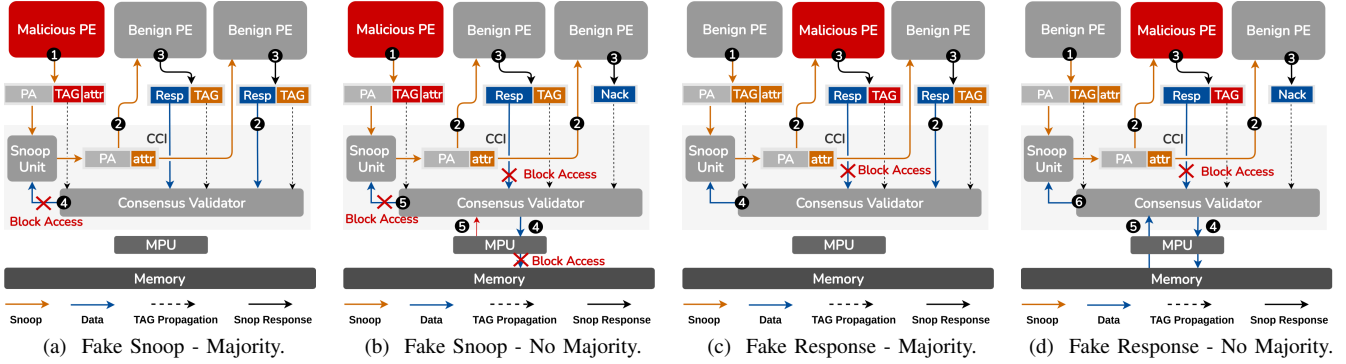


Figure 3: Coherence Extension handling malicious traffic. (a): Malicious PE snoops; majority consensus blocks invalid TAGs. (b): No majority triggers MPU to block invalid responses. (c,d): Malicious PE sends incorrect responses; majority or MPU ensures validity.

yet another privileged software layer to be managed, both error-prone [58] and unlikely to serve as a long-term solution. Instead, we advocate for native hardware extensions (§8.3) that enable seamless enforcement of coherence privileges, which work in tandem with existing address-space metadata rather than duplicating it in bespoke tables.

8.3. Address-Space-Aware Coherence Extension

Following principles **P1** to **P3**, we now propose a coherence extension, which precludes *Coherent Trojans* to craft fake coherence messages preventing coherence attacks (§6). This security extension is orthogonal to existing coherence features. As a design constraint, we assume that only the CI is trusted and all PEs are potentially malicious. This resembles a Byzantine Generals Problem [59], where mutual distrusted PEs must cooperate without being to detect misbehavior. To address this, we propose a “proof-of-ownership” consensus mechanism, where PEs must prove CL ownership to participate in its coherence data exchange.

Solution overview. Proof-of-ownership is established via a distributed ledger, where PEs with access rights to a CL hold a copy of the corresponding entry. We envision this ledger as an augmented LLC, where each CL includes extra bits for (i) for ownership identification (*TAG*) and (ii) security attributes (e.g., R/W/X) derived from existing address-space metadata. Every PE involved in a data exchange must prove ownership to a *Consensus Validator*, implemented in the trusted CI, by presenting the correct ledger entry.

Consensus Rule: PEs must present the same ledger entry to engage in coherence communication.

Consensus protocol. An effective proof-based consensus mechanism requires an asymmetrical cost: (i) enforcing the consensus must be cheap and (ii) violating it expensive. The proposed mechanism ensures this asymmetry by only requiring a *TAG* comparison to check consensus, while exploiting it requires a random guess of the CL *TAG*. The difficulty of guessing grows exponentially with the size of the *TAG*. A malicious PE has only a 1 in $(2^{\text{TAG_SIZE}} - 1)$ chance

of success. A single wrong guess results in the misbehaving PE being blacklisted and cut off from the coherence domain.

Misbehavior Cost: A misbehaving PE is starved from critical systems resources, e.g., memory, power, or clock.

Consensus validator. The *Consensus Validator* checks for consensus upon each coherence transaction. If a majority of PEs provide a matching *TAG*, any PE with a different *TAG* is flagged as malicious (cf. Fig. 3a and Fig. 3c). If there is no majority (e.g., an equal split across different *TAG*s), the *Consensus Validator* (which does not keep a copy of the ledger) queries a ground truth source, such as the MPU, to determine which PE is misbehaving (cf. Fig. 3b and Fig. 3d). The CI can perform this check by initiating a memory access on behalf of the PE under scrutiny; if the MPU blocks the access, the PE is confirmed to be malicious.

Architectural details. To define the *TAG*, the consensus mechanism can reuse existing identifiers such as Address Space Identifier (ASID) and Virtual Machine Identifier (VMID), integrating seamlessly with address-space metadata and memory controllers (MMU and MPU). A 16-bit *TAG* plus 2 bits for access attributes (R/W/X) adds 18 bits per CL (3.52% overhead for 64-byte CLs), providing a reasonable cost-to-security tradeoff and modest to negligible performance impact (hardware-only implementation). This offers a structural fix for coherence attacks, though thorough evaluation is needed and deferred to future work.

9. Conclusion

This paper thoroughly studies the underlooked security issues currently exposed in coherent heterogeneous computing systems. We systematically theorize on coherence-based attacks by introducing the *Coherent Trojan* model. Then, we highlight the gravity and practicality of the threat by implementing an (open-source) *Trojan Tool*, which we leverage to mount four attacks capable of undermining the CIA guarantees on mainstream systems deployed on modern CPU+FPGA SoCs. We envision this paper will represent a *call for action* for the research community to investigate fundamental approaches towards security-hardened coherence.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback and constructive suggestions. Cristiano Rodrigues was supported by FCT grant 2020.08729.BD and partial supported by FLAD, Fundação Luso-Americana para o Desenvolvimento (grant Proj. 2023/0221). This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Unit Project Scope UID/00319/2025 - Centro ALGORITMI (ALGORITMI/UM) <https://doi.org/10.54499/UID/00319/2025>, and by the National Science Foundation (NSF) under grant number CSR-2238476.

References

- [1] Samuel Naffziger et al. “2.2 AMD Chiplet Architecture for High-Performance Server and Desktop Products”. In: *Proc. of ISSCC*. 2020.
- [2] Bloomberg. *Arm-Qualcomm Lawsuit Could Muddle US Chip Design Leadership*. <https://news.bloomberglaw.com/us-law-week/arm-qualcomm-lawsuit-could-muddle-us-chip-design-leadership>. Accessed: 2024-12-28.
- [3] Mark Mohammad Tehranipoor and Farinaz Koushanfar. “A Survey of Hardware Trojan Taxonomy and Detection”. In: *Journ. IEEE Design & Test of Comp.* 2010.
- [4] Jianping Zhu et al. “Enabling Rack-scale Confidential Computing using Heterogeneous Trusted Execution Environment”. In: *Proc. of IEEE S&P*. 2020.
- [5] Kapil Vaswani et al. “Confidential Computing within an AI Accelerator”. In: *Proc. of USENIX ATC*. 2023.
- [6] Supraja Sridhara et al. “ACAI: Protecting Accelerator Execution with Arm Confidential Computing Architecture”. In: *Proc. of USENIX Sec*. 2024.
- [7] Arm. *AMBA AXI and ACE Protocol Specification*. Tech. rep. Jan. 2021.
- [8] Arm. *AMBA CHI Architecture Specification*. Tech. rep. Mar. 2024.
- [9] CXL Consortium. *Compute Express Link (CXL)*, www.computeexpresslink.org. Accessed: 2024-06-01.
- [10] Minsu Kim et al. “Evaluating coherence-exploiting hardware Trojan”. In: *Proc. of DATE*. 2017.
- [11] Mathieu Gross et al. “Breaking TrustZone Memory Isolation through Malicious Hardware on a Modern FPGA-SoC”. In: *Journal of Cryptographic Engineering*. 2022.
- [12] Gino A Chacon et al. “Coherence Attacks and Countermeasures in Interposer-based Chiplet Systems”. In: *Journ. ACM Trans. on Arch. and Code Optim.* 2024.
- [13] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. “Are Coherence Protocol States Vulnerable to Information Leakage?” In: *Prof. of HPCA*. 2018.
- [14] Gino A. Chacon et al. “Hardware Trojan Threats to Cache Coherence in Modern 2.5D Chiplet Systems”. In: *arXiv*. 2022.
- [15] James Archibald and Jean-Loup Baer. “Cache coherence protocols: Evaluation using a multiprocessor simulation model”. In: 1986.
- [16] Daniel Hackenberg, Daniel Molka, and Wolfgang E Nagel. “Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems”. In: *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on microarchitecture*. 2009, pp. 413–422.
- [17] Shahin Roozkhosh, Denis Hoornaert, and Renato Mancuso. “CAESAR: Coherence-Aided Elective and Seamless Alternative Routing via on-chip FPGA”. In: *Proc. of RTSS*. 2022.
- [18] Manoj Kumar J.Y.V. et al. “Run Time Mitigation of Performance Degradation Hardware Trojan Attacks in Network on Chip”. In: *Proc. of ISVLSI*. 2018.
- [19] Mohammad Nasim Imtaiz Khan, Asmit De, and Swaroop Ghosh. “Cache-Out: Leaking Cache Memory Using Hardware Trojan”. In: *Journ. IEEE Transact on VLSI Systems*. 2020.
- [20] Atul Kumar et al. “edAttack: Hardware Trojan Attack on On-Chip Packet Compression”. In: *Journ. IEEE Design & Test*. 2023.
- [21] Yoshiya Shikama, Michihiro Koibuchi, and Hideharu Amano. “A Hardware Trojan Exploiting Coherence Protocol on NoCs”. In: *Prof. of PDCAT*. 2023.
- [22] Nisha Jacob et al. “How to Break Secure Boot on FPGA SoCs Through Malicious Hardware”. In: *Proc. of CHES*. 2017.
- [23] Sumanta Chaudhuri. “A security vulnerability analysis of SoCFPGA architectures”. In: *Proc. of DAC*. 2018.
- [24] E. M. Benhani, L. Bossuet, and A. Aubert. “The Security of ARM TrustZone in a FPGA-Based SoC”. In: *IEEE Trans. Comput.* 2019.
- [25] He Li, Qiang Liu, and Jiliang Zhang. “A survey of hardware Trojan threat and defense”. In: *Journ. Integration*. 2016.
- [26] Rajat Subhra Chakraborty et al. “Hardware Trojan insertion by direct modification of FPGA configuration bitstream”. In: *Journ. IEEE Design & Test*. 2013.
- [27] Chen Dong et al. “Hardware Trojans in Chips: A Survey for Detection and Prevention”. In: *Journ. Sensors*. 2020.
- [28] Swarup Bhunia et al. “Hardware Trojan Attacks: Threat Analysis and Countermeasures”. In: *Proc. of the IEEE*. 2014.
- [29] Sally Adee. “The Hunt For The Kill Switch”. In: *IEEE Spectrum*. 2008.
- [30] Mohammad Tehranipoor et al. “Trustworthy Hardware: Identifying and Classifying Hardware Trojans”. In: *IEEE Computer*. 2010.
- [31] Swarup Bhunia et al. “Protection Against Hardware Trojan Attacks: Towards a Comprehensive Solution”. In: *Journ. IEEE Design & Test*. 2013.
- [32] Julian Speith et al. “How Not to Protect Your IP – An Industry-Wide Break of IEEE 1735 Implementations”. In: *Proc. of IEEE S&P*. 2022.
- [33] Christian Krieg, Clifford Wolf, and Axel Jantsch. “Malicious LUT: A stealthy FPGA Trojan injected and

- triggered by the design flow”. In: *Proc. of ICCAD*. 2016.
- [34] Xuehui Zhang and Mohammad Tehranipoor. “Case study: Detecting hardware Trojans in third-party digital IP cores”. In: *Proc. of IEEE HST*. 2011.
- [35] Jatin Kataria et al. “Defeating Cisco Trust Anchor: A Case-Study of Recent Advancements in Direct FPGA Bitstream Manipulation”. In: *Proc. of WOOT*. 2019.
- [36] Pawel Swierczynski et al. “Interdiction in practice—Hardware Trojan against a high-security USB flash drive”. In: *Journ. of Cryptographic Engineering*. 2017.
- [37] Maik Ender, Amir Moradi, and Christof Paar. “The Unpatchable Silicon: A Full Break of the Bitstream Encryption of Xilinx 7-Series FPGAs”. In: *Proc. of USENIX Sec*. 2020.
- [38] Maik Ender et al. “A Cautionary Note on Protecting Xilinx’ UltraScale(+) Bitstream Encryption and Authentication Engine”. In: *Proc. of FCCM*. 2022.
- [39] Maik Ender et al. “JustSTART: How to Find an RSA Authentication Bypass on Xilinx UltraScale(+) with Fuzzing”. In: *Journ. TCHES*. 2024.
- [40] Endres Puschner et al. “Patching FPGAs: The Security Implications of Bitstream Modifications”. In: *Proc. of ASHES*. 2024.
- [41] Maik Ender et al. “Insights into the Mind of a Trojan Designer: The Challenge to Integrate a Trojan into the Bitstream”. In: *Proc. of ASP-DAC*. 2019.
- [42] Nils Albartus et al. “On the Malicious Potential of Xilinx’s Internal Configuration Access Port (ICAP)”. In: *ACM Trans. Reconfigurable Technol. Syst*. 2024.
- [43] Jayeeta Chaudhuri et al. “Hacking the Fabric: Targeting Partial Reconfiguration for Fault Injection in FPGA Fabrics”. In: *Proc. of ATS*. 2024.
- [44] Intel. *Partial Reconfiguration*. <https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/partial-reconfiguration.html>. Accessed: 2024-12-20.
- [45] AMD. *Partial Reconfiguration Controller*. <https://www.xilinx.com/products/intellectual-property/prc.html>. Accessed: 2024-12-20.
- [46] Dina G. Mahmoud et al. “X-Attack 2.0: The Risk of Power Wasters and Satisfiability Don’t-Care Hardware Trojans to Shared Cloud FPGAs”. In: *IEEE Access*. 2024.
- [47] Simon Klix et al. “Silicon Heist: (Ransom) Attacks for Cloud FPGAs via Privilege Escalation”. In: *Proc. of USENIX Sec*. 2026.
- [48] Zane Weissman et al. “JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms”. In: *Journ. CHES*. 2020.
- [49] Mark Zhao and G. Edward Suh. “FPGA-Based Remote Power Side-Channel Attacks”. In: *Proc. of S&P*. 2018.
- [50] Ilias Giechaskiel, Kasper B. Rasmussen, and Ken Eguro. “Leaky Wires: Information Leakage and Covert Communication Between FPGA Long Wires”. In: *Proc. of ASIACCS*. 2018.
- [51] Irina Calciu et al. “Project PBerry: FPGA Acceleration for Remote Memory”. In: *Proc. of HoTOS*. 2019.
- [52] José Martins et al. “Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems”. In: *Proc. of NG-RES*. 2020.
- [53] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *Proc. of USENIX Sec*. 2015.
- [54] Mattia Nicolella et al. “RT-Bench: An Extensible Benchmark Framework for the Analysis and Management of Real-Time Applications”. In: *Proc. of RTNS*. 2022.
- [55] Sandro Pinto and Nuno Santos. “Demystifying Arm TrustZone: A Comprehensive Survey”. In: *Journ. ACM Computing Surveys*. 2019.
- [56] David Cerdeira et al. “ReZone: Disarming TrustZone with TEE Privilege Reduction”. In: *Proc. of USENIX Sec*. 2022.
- [57] Amazon. *Amazon EC2 F1 Instances*. [www.aws.amazon.com/ec2/instance-types/f1/](https://aws.amazon.com/ec2/instance-types/f1/). Accessed: 2024-12-21.
- [58] Jo Van Bulck et al. “A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes”. In: *Proc. of CCS*. 2019.
- [59] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine Generals Problem”. In: *Journ. ACM Trans. Program. Lang. Syst*. 1982.
- [60] Intel. *Intel® Stratix® 10 Hard Processor System Technical Reference Manual*. Tech. rep. Feb. 2024.
- [61] AMD. *Zynq UltraScale+ Device Technical Reference Manuals*. Tech. rep. Feb. 2024.
- [62] Microsemi. *PolarFire SoC MSS Technical Reference Manual*. Tech. rep. June 2024.

Appendix A. CPU+FPGA Platforms Landscape

There is an ever-growing offer of CPU+FPGA commercial systems from multiple vendors such as Intel [60], AMD Xilinx [61], and Microchip [62] tailored for different sectors and applications, like mainstream consumer applications, cloud providers, and even military. We surveyed and aggregated the characteristics of the most prominent CPU+FPGA platforms in Table 4.

Appendix B. Confidentiality Attack: Read Arbitrary Memory

To showcase how a *Coherent Trojan* can tamper with the system’s Confidentiality⁵ and read arbitrary memory, we developed *Leak Data Trojan (LD-Trojan)* and mounted an attack that extracts a Rivest-Shamir-Adleman (RSA) private key by reading it from memory.

5. The Confidential Attack is detailed here, in Appendix, due to space constraints. The *TrustZone Attack* §6.4 it demonstrates analogous findings.

Vendor	Platform	APU	Interconnect	SMMU	Coh. FPGA	ACE-Full	ACE-Lite	CHI	CXL	DVM	REF
AMD	Versal Adaptive	2xCortex-A72	Arm CCI-500	MMU-500	●	●	●	●	○	●	TRM
AMD	Zynq UltraScale+	4xCortex-A53	Arm CCI-400	MMU-500	●	●	●	○	○	●	TRM
AMD	Zynq-7000	2xCortex-A9	Arm NIC-301	—	○	○	○	○	○	○	TRM
Intel	Agilex 7	4xCortex-A53	Arteris FlexNoC	MMU-500	●	○	●	○	●	●	TRM
Intel	Stratix 10	4xCortex-A53	Arteris FlexNoC	MMU-500	●	○	●	○	○	●	TRM
Intel	Arria 10	2xCortex-A9	Arteris FlexNoC	—	○	○	○	○	○	○	TRM
Intel	Cyclone 10	2xCortex-A9	Arm NIC-301	—	○	○	○	○	○	○	TRM
Microchip	PolarFire	4xRV64GC	N.A	—	○	○	○	○	○	○	TRM

TABLE 4: Overview of the characteristics of most prominent CPU+FPGA Platforms, Including Vendor, Platform, APU, CCI, SMMU, Coherence Capabilities (ACE-Full, ACE-Lite, CHI, CXL), and DVM Support.

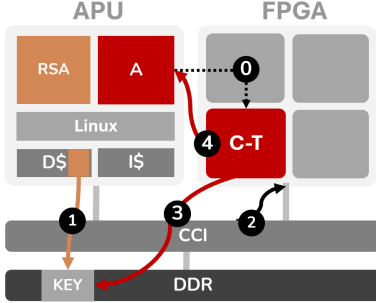


Figure 4: Illustration of the Confidential Attack PoC. Orange indicates the Victim (V), and red indicates the attacker and Trojan’s activity. (0) Configuration of the *Trojan Tool* to create the *LD-Trojan*. (1) Denotes a cache miss on data cache (D\$), and (2) a snoop request issued by the CCI. (3) Represents the read of the private key from memory and (4) the leak of this key to the attacker.

Overview. *RSA Attack*, depicted in Fig. 4, uses the *LD-Trojan* to read arbitrary memory and steal a private key. First, to find where the RSA key is stored in memory, the attacker uses *D-Trojan* to find a pattern that unequivocally identifies the target data to deanonymize. Typically, cryptographic keys are stored in pem files, which have specific headers and footers fixed regardless of the key and are specific to each cryptographic algorithm. We can target an algorithm by looking at its pem unique header. When the *D-Trojan* finds the physical location of the RSA key pem file, *LD-Trojan* leaks it.

Attack flow. The *Trojan Tool* is configured to receive the physical address ($pAddr$) and the memory size in bytes ($leakSize$) of the secret we want to leak, i.e., the size of the pem file. The physical address is provided by the *D-Trojan*, and the memory size is configured at runtime by the attacker code. When the target data is deanonymized by *D-Trojan*, the *LD-Trojan* leaks the secret by issuing a non-coherent read snoop, using the active operation `IssueNoCohNoCacheRd` (see §4.3 and Appendix D). The *LD-Trojan* issues snoops until all CLs containing the data of interest are read out, being the total number defined by $leakSize/CLSize$.

Attack setup. To demonstrate this attack, we implemented the classic Alice-Bob secret message exchange using asymmetric encryption and tried to leak Bob’s private keys. We used OpenSSL version 3.3.0 and targeted a private key generated

with the RSA algorithm. We implemented three applications that emulated Bob, Alice, and Eve and deployed them as user-space applications on Linux. Eve has access to a *LD-Trojan* that can monitor all coherence traffic on the system. Bob uses the OpenSSL `genpkey` function to generate a private key from which he derives the public key. Then he shares the public key with Alice, which she uses to encrypt the secret message using `OpenSSL pkeyutl -encrypt`. When Bob receives the message from Alice, he decrypts it with his private key using `OpenSSL pkeyutl -decrypt`. Eve is eavesdropping on all communications between Alice and Bob, getting the encrypted message, as well as monitoring all coherent activity on the system. When Bob uses his private key, Eve extracts it using the *LD-Trojan*. Having the private key, she can use it to decrypt the secret message or impersonate Bob.

Results. We conducted the attack 1,000 times, achieving a 99.6% success rate in recovering the RSA private key. The recovery took an average of 1.6 seconds since *LD-Trojan* detected the code until a full-out read of the RSA private key. The exploit’s high reliability and low execution time make it feasible for an attacker to scale the attack to steal multiple private keys. Although we focused on a specific victim and a specific type of pem file (RSA), the method could potentially target all pem files (cryptographic keys) processed by the system. An attacker could then use these keys to decrypt communications, impersonate software, or falsify authentication.

Appendix C. Coherent Attack Description Pseudo Language

In this appendix, we present a comprehensive list of all supported functionalities of *Trojan Tool*, demonstrating the tool’s flexibility and programmability. As detailed in §4, the operations are categorized into four types, *sniff*, *claim*, *issue*, and *tamper*. The commands supported by *Trojan Tool* are structured accordingly and summarized in the following tables: *sniff* in Table 5, *claim* in Table 6, *issue* Table 7, and *tamper* in Table 8.

TABLE 5: *Passive mode Sniff* operations.

Pseudo Command	Description
SniffRd(Addr,Attr)	<i>Sniff</i> and (Watch for) combination of <Addr,Attr> if snooped to be read.
SniffInv(Addr,Attr)	<i>Sniff</i> and (Watch for) an invalidation snoop request (i.e., discard even if it is dirty).
SniffCln(Addr,Attr)	<i>Sniff</i> and (Watch for) <Addr> if snooped to be cleaned, i.e., write back to memory.
SniffBrr(Addr,Attr)	<i>Sniff</i> and (Watch for) a barrier snoop request.
SniffLog()	<i>Sniff</i> and log at internal SPM.
SniffMirror(Addr)	<i>Sniff</i> and stream-mirror to <Addr>.

TABLE 6: *Active mode Claim* operations.

Pseudo Command	Description
Claim(Addr,Attr)	If snooped on <Addr>, <i>Claim</i> a hit in a <Attr> attribute CL
FeedClaim(Addr,Data)	If already <i>Claimed</i> on <Addr>, <i>Feed</i> <data> with promised (<i>Claimed</i>) attribute

TABLE 7: *Active mode Issue* operations.

Pseudo Command	Description
IssueBrr(Addr,Attr)	<i>Issue</i> coherent Barrier request on <Addr> ask for CL with <Attr> attribute.
IssueCln(Addr,Attr)	<i>Issue</i> coherent Clean request on <Addr> ask for CL with <Attr> attribute.
IssueInv(Addr,Attr)	<i>Issue</i> coherent Invalidate request on <Addr> ask for CL with <Attr> attribute.
IssueRd(Addr,Attr)	<i>Issue</i> coherent Read request on <Addr> ask for CL with <Attr> attribute.
IssueWr(Addr,Attr)	<i>Issue</i> coherent Write request on <Addr> ask for CL with <Attr> attribute.
IssueNoCohNoCacheRd(Addr)	<i>Issue</i> non-coherent Read request on <Addr>
IssueNoCohNoCacheWr(Addr)	<i>Issue</i> non-coherent Write request on <Addr>

TABLE 8: *Active mode Tamper* operations.

Pseudo Command	Description
TamperDly(Addr,Attr)	If snooped on <Addr,Attr>, delay in acknowledging snoops for some time.
TamperCond(Addr,Attr,Cnd)	If snooped on <Addr,Attr>, do not acknowledge until condition <Cnd> is met.

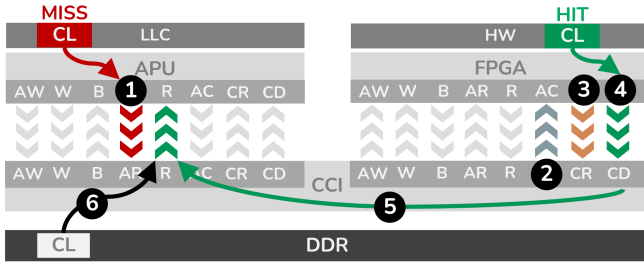


Figure 5: Example of the ACE coherence flow upon a cache miss on a CPU+FPGA platform.

main memory, i.e., fetching it from the DDR ⑥.

Appendix D.

AXI Coherency Extension (ACE)

ACE extends the AXI protocol to enable hardware-assisted cache coherence (Fig. 5). Alongside standard read (AR and R) and write channels (AW, W, and B), ACE introduces three additional channels (AC, CR, and CD) to facilitate coherence operations. In ACE protocol, upon a cache-miss at the LLC ①, the CCI broadcasts the event to all the coherent mains ②. It uses the snoop address (AC) channel to provide the snoop transactions' address—i.e., the physical address of the snooped CL—and any associated control information. Every snoop transaction has a single response associated with it, which can be used by a snooped main to provide the data for the requested CL. The snoop response (CR) channel is used to provide one such response ③. In case the snooped request can be served by another coherent main, the snoop data (CD) channel ④ is used to transfer the data payload ⑤. If the LLC cache miss is not resolved via coherence, the CCI satisfies the CL fill from