Vyond: Flexible and Rapid WorldGuard-Based Security Prototyping using Chipyard

September 9, 2025

Abstract

Hardware isolation is a critical security feature in systems ranging from cloud computing to embedded devices. The WorldGuard security model offers a promising approach, yet few SoCs currently implement it. Fixed WorldGuard implementations in off-the-shelf SoCs, however, lack the flexibility needed for security designers to prototype and customize architectures. To address this, we open-sourced Vyond, a configurable WorldGuard implementation within Chipyard, enabling agile and adaptable security prototyping. We demonstrate how WorldGuard components integrate seamlessly into a Rocket SoC using Chipyard's parameterized framework. Additionally, Vyond provides three WorldGuard-enabled hardware implementations (Simulation, FPGA, and QEMU) and a baseline security monitor, allowing designers to rapidly implement a secure OS, test functionalities, and evaluate hardware costs. We anticipate that Vyond will accelerate RISC-V security research and drive innovation in hardware security architectures.

Motivation

Ensuring the confidentiality and integrity of code and data is fundamental to system security. Without a robust isolation mechanism, developers and platforms remain vulnerable to threats such as unauthorized memory access and device manipulation by malicious software or other bus initiators (e.g., DMAs). WorldGuard [1], a hardware-enhanced software isolation solution, enables a Trusted Execution Environment (TEE) on RISC-V platforms by providing an open, system-level approach to securing access to system resources (e.g., memory and peripherals). However, existing WorldGuard-enabled SoCs are limited in scope, protecting only a few hardware resources (e.g., DRAM and UART) and relying on fixed configurations (e.g., a predefined number of rule configuration slots). Due to these constraints, we have not found any existing TEE implementations leveraging WorldGuard.

Meanwhile, agile hardware design methodologies have emerged to address the growing cost of custom silicon architectures. Chipyard [2], one of the most widely used frameworks in research, provides configurable, composable, open-source, generator-based IP blocks that facilitate seamless integration across multiple hardware development stages.

Recognizing this opportunity, we propose integrating WorldGuard into Chipyard as a generator, allowing security designers and researchers to rapidly prototype and customize secure architectures by extending WorldGuard's protection to various peripherals.

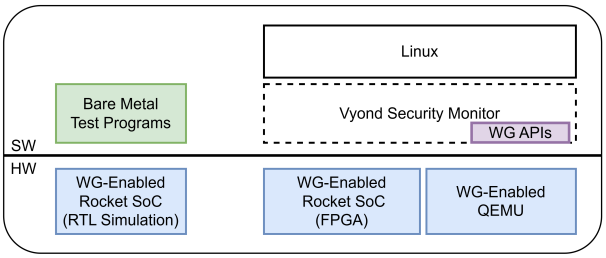

Figure 1: *Vyond provides unified framework to run security monitory on WG-Enabled QEMU, RTL simulation, , and FPGA, respectively.*

Contributions

As shown in Figure 1, we have implemented WorldGuard in Chipyard and open-sourced it as part of the Vyond project[1]. Vyond is a growing project aimed at developing a secure OS for RISC-V platforms. It provides a Security Monitor (SM) for a future Vyond-TEE, leveraging WorldGuard's hardware isolation features. We anticipate that Vyond will accelerate security research and foster the development of WorldGuard-based hardware security architectures. Vyond implements functionalities defined in the WorldGuard specification [1] while maintaining configurability, thanks to Chipyard's design principles.

1. Generators for WorldGuard Marker and Checker A WorldGuard Marker (WGM) is placed after the master, typically a non-WorldGuard-enabled core. The WGM holds the current world ID (WID) and marks transaction requests from the master with the corresponding WID. A WorldGuard Checker (WGC) is placed before a target (e.g., a peripheral, a con-

[1] `https://github.com/samsung/vyond`

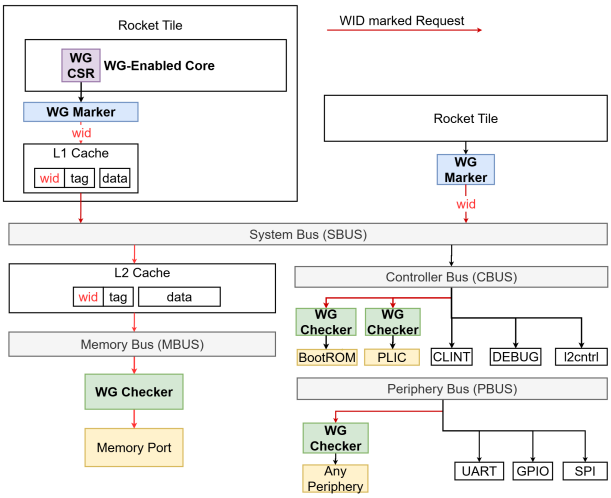

Figure 2: *WorldGuard-Enabled RISC-V SoC.*

troller, or memory) to enforce access control. The WGC includes PMP-like memory-mapped registers, allowing security policies to be configured for different WIDs. We have implemented both WGM and WGC as Chipyard generators, enabling reuse across various targets and configurations (e.g., the number of rule slots and WID width). Since they are independent of other hardware components, they can be configured flexibly, as demonstrated in Listing 1.

```
class WithWorldGuard(mwid: Int, widWidth: Int, nSlots: Int)
      extends Config((site, here, up) => {
  case WGPLICKey => {
    Some((PLICParams(),
      WGCheckerParams(
        postfix = "wgpplic",
        mwid     = mwid,
        widWidth = widWidth,
        nSlots   = nSlots,
        address  = 0x2040000,
        size     = 4096)
      ))
    // Other Configs such as DRAM, UART, etc..
  }
```

Listing 1: *Parameters of WG Checker for PLIC*

2. Traits for Masters and Slaves To integrate WGM and WGC into masters and slaves, we updated the existing master/slave traits. Only a few lines of code are required for integration, as shown in Listing 2. For example, in Listing 2, we insert a WGC for PLIC by modifying the TileLink bus connection. This approach allows seamless integration of WorldGuard-based security mechanisms into various SoC components.

```
trait CanHaveWGPPLICOrPlic { this: BaseSubsystem =>
  val (plicOpt, plicDomainOpt) = p(WGPLICKey) match {
    case Some((wgpParams, wgcParams)) => {
      val wgc = WGChAttachParams(wgcParams).attachTo(this)
      plic.node := wgc.node := tlbus.coupleTo("plic") {/*..*/}
      /*...*/ }
    case None => { /* Original code */ }}}
```

Listing 2: *Trait of PLIC to place WGC before PLIC.*

3. Extended Rocket Tile and Caches As shown in Figure 2, WorldGuard CSRs are tightly coupled with the RISC-V hart. We extended the Rocket Tile to support these CSRs and ensure that each request transaction includes a WID. Additionally, for DRAM protection using WGC, all caches in the memory hierarchy were modified to track WIDs for each cache block, ensuring correct tag matching and isolation.

4. Example Configurations and Test Programs We provide various configurable setups for security designers. In Figure 2, a WG-enabled core and a non-WG-enabled core (with WGM) are configured alongside peripherals protected by WGCs. Security designers can construct custom SoCs with minimal effort. To validate our WorldGuard implementation and assist security researchers, we developed five bare-metal test programs that run on a WG-enabled RTL simulation. These programs serve as reference implementations for security monitors.

Our test programs also identified a potential design issue in the WorldGuard specification. Specifically, we demonstrated a security vulnerability when the protected memory region is smaller than a cache block size (e.g., 64 bytes). We reported this issue to RISC-V International, and they acknowledged it, agreeing to include a documentation note[2].

5. WorldGuard-Based Security Monitor and Hardware Implementations To the best of our knowledge, no prior TEE has leveraged WorldGuard as a hardware security primitive. This is likely due to the lack of flexibility in off-the-shelf WG-enabled SoCs. As shown in Figure 1, Vyond provides various WorldGuard-enabled hardware implementations and APIs for configuring security policies. This allows security designers to: 1) implement TEE functionalities using QEMU [3][3] and RTL simulation, and 2) valuate the hardware costs of WG-enabled SoCs running on an FPGA board.

References

[1] SiFive. Worldguard specification. `https://lists.riscv.org/g/security/attachment/711/0/worldguard_rvia_spec-v0.4.pdf`, 2023. Accessed:2025-02-03.

[2] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4):10–21, 2020.

[3] Jim Shu. Worldguard technical paper. `https://patchwork.ozlabs.org/project/qemu-devel/cover/20240612081416.29704-1-jim.shu@sifive.com`, 2024. Accessed:2025-02-03.

[2] `https://lists.riscv.org/g/security/topic/worldguard_design_issue_with/110194739`

[3] WorldGuard on QEMU was implemented by Jim Shu at SiFive.