FORTE: Fault-tOlerant RISC-V with Trusted Enclaves

Pranav Bellur
pbellur2@illinois.edu

Rohan Das
rohan24@illinois.edu

Swetha Karthikeyan
swethak2@illinois.edu

Siddharth Rau
sidrau2@illinois.edu

Saipranav Venkatakrishnan
sv34@illinois.edu

Ved Vyas
vedsv2@illinois.edu

Abstract

We present FORTE, an open-source Linux-capable RISC-V processor combining physical memory protection (PMP), a hardware root of trust (RoT), and hardware-level fault tolerance mechanisms to defend against bit flips, glitching, and other physical fault-injection attacks. FORTE provides hardware support for the Keystone framework, an open-source trusted execution environment (TEE) that leverages PMP and RoT to create secure memory enclaves for user programs running in potentially untrusted environments. By integrating hardware protection features alongside Keystone's enclave model, FORTE aims to deliver a secure processor that not only isolates sensitive computation via TEEs but also remains robust against malicious and fault-based attacks. To our knowledge, this represents the first open-source RISC-V processor that is both Linux-capable and specifically targeted at booting the Keystone framework, including adapting Keystone software to our platform, while simultaneously addressing hardware-level faults.

1 Problem Statement

The modern RISC-V landscape has a growing demand for open-source, Linux-capable hardware that can provide robust security in increasingly unsafe and physically exposed environments. Beyond software-level isolation, real-world deployments face threats from fault injection, voltage glitching, and naturally occurring bit flips (e.g., single-event upsets), making hardware-level fault tolerance an essential complement to trusted execution. This requirement is necessary to move RISC-V forward to be more broadly usable in secure applications. The Keystone framework offers an open-source path toward establishing Trusted Execution Environments (TEEs), but it has limited examples of working on real hardware beyond the original paper. In the original paper [4], Keystone was tested on a single closed-source SiFive processor, and the rest of the hardware platforms were on FPGAs; notably, none of these platforms publicized incorporating dedicated protections against fault-based attacks or transient hardware errors. We believe that an open-source silicon RISC-V core that is both OS-capable and hardened against such faults will be an interesting contribution, enabling Keystone to be tested on silicon-based processors that are resilient to the physical-layer threats present in unsafe deployment environments. We present FORTE, a **F**ault-t**O**lerant **R**ISC-V core with **T**rusted **E**nclaves, which integrates fault-tolerance mechanisms with the hardware root of trust and PMP required to boot a Keystone-based TEE. Achieving this in limited area will be a proof of concept for how edge-focused RISC-V microprocessors can become viable for secure edge applications operating under both adversarial and fault-prone conditions.

2 Design Proposal

2.1 Pre-built core Decisions

Based on area constraints, we initially considered implementing a minimal in-order pipeline from scratch supporting the RV64IMA instruction set and the RISC-V Privileged Architecture, with the goal of booting Linux kernel version 4.17 or later. To establish feasibility, we produced preliminary area estimates using the Rocket Chip generator as an approximation for an implemented RISC-V Core. Synthesis of the Rocket Chip RV64IMA configuration was performed using Yosys with the NangateOpenCellLibrary, a 45 nm standard-cell library [7], yielding an estimate of 69,643 μm^2 at 45 nm. Scaling by $65^2/45^2$ and applying a conservative routing factor of 1.7, we estimated 247,039 μm^2 for the standard-cell portion of a custom core. Private L1 cache SRAM arrays were estimated separately using CACTI [6]: at 65 nm with 64-byte cache lines and `itrs-hp` cell technology, a 1 KB direct-mapped instruction cache and a 1 KB 4-way set-associative data cache each came to approximately 0.044 mm^2, and a 2 KB boot "ROM" (implemented as SRAM, since a true ROM is not available on our process) was estimated at approximately 0.029 mm^2. Adding an additional 15% of total die area for I/O pad structures (JTAG, UART, and other peripheral interfaces), the combined estimate came to around 0.419 mm^2, comfortably within our $1mm^2$ area budget. These figures gave us confidence that a Linux-capable, Keystone-ready core was feasible within our area constraints.

Given the limited timeline of this project, rather than designing a pipeline from scratch, we chose to base FORTE on CORE-V Wally (cvw) [8], an open-source, configurable 5-stage RISC-V processor developed by the OpenHW Group. Wally already implements the full set of features our area estimates indicated we would need to build: support for the RV64IMA extensions, the Zicsr and Zifencei extensions, Sv39 virtual memory with a hardware page table walker and TLBs, physical memory protection (PMP), and the Machine/Supervisor-level privileged ISA required for Linux. Critically, Wally has already passed the RISC-V Architecture Test suite and successfully boots Linux on FPGA, giving us a verified, known-good starting point, allowing the focus on this paper to remain with creating a fault-tolerant processor for secure applications.

Wally's configurability is what makes it particularly well-suited to our goals. Because its feature set is parameterized, we can disable extensions that are unnecessary for our target (most notably the F/D/Q floating-point extensions) reducing area in line with our original estimates, while relying on software floating-point emulation (e.g., Berkeley SoftFloat or libgcc) if floating-point operations are required. This lets us recover much of the area budget that would otherwise be spent on an FPU and redirect it toward our primary contributions: hardware root-of-trust infrastructure, ECC and CSR hardening, and other fault-tolerance mechanisms needed to support Keystone and defend against bit-flip and glitching attacks. In short, starting from Wally lets us begin from a verified, Linux-capable, Keystone-compatible baseline and focus our limited time on the secure-design additions that differentiate FORTE, rather than on re-deriving a working pipeline and memory subsystem from scratch.

2.1.1 *Required Peripheral Hardware.*

Linux and the privileged spec require some peripherals. A timer is needed for preemption and scheduler ticks, provided by the `mtime` and `mtimecmp` registers implemented via a CLINT (Core Local Interrupter). An interrupt controller is required for routing external peripheral IRQs to the hart; we use the PLIC (Platform Level Interrupt Controller), which is the standard mechanism for this in RISC-V SoCs. A console UART is necessary for any meaningful boot debugging. The CLINT will be on-chip, but

further research needs to be done to determine if the PLIC will be on or off-chip due to concerns about pin count.

2.1.2 Firmware: OpenSBI and the Boot "ROM". We propose a two-stage boot approach where a small "ROM" loader copies the real firmware from flash into FPGA DRAM before handing off. This method is well established in the RISC-V ecosystem, in the lowRISC project [5]. Our 2 KB boot "ROM" is sized to hold this type of minimal first-stage loader: initialize basic CSRs, configure the memory map, copy the OpenSBI image from SPI flash to DRAM, and jump to it [9]. The full OpenSBI image lives in external flash and is loaded into DRAM at runtime.

2.2 Secure Hardware Features

2.2.1 RoT and Pin-out Lockdown.

Measured Boot and Remote Attestation. In an environment where the OS are untrusted, the manufacturer embeds a private cryptographic key into the hardware as part of the RoT, which is used by the first stage bootloader to hash the next software layer, typically the security monitor (see 2.3), before boot. The first stage bootloader (FSBL) is also part of the RoT, since it is usually an immutable ROM. If the hash matches the expected value, this can be used to attest to the software's authenticity and safety. Given a functioning SM, enclaves are guaranteed to be secure and a remote user can safely run sensitive user code in enclaves, even with a potentially malicious OS. On boot, the private key is directly loaded into memory and accessed by the FSBL to hash the SM. The private key is then erased from memory by the FSBL and control is passed off to the SM. This prevents attackers from manipulating the boot process or corrupting enclaves, as the software managing the enclaves is verified at boot time. If the verification fails and the hash is wrong, then the SM will be unable to sign attestation reports correctly. The remote user passes in a random number, and private performs cryptographic operations on the number with the correct hash, and asks the SM to do the same thing, and checks that the results match.

Architectural Purging. Some architectural structures can be trained or primed by malicious code, or contain sensitive information. When switching in and out of enclaves, erasing the contents of caches, branch predictors, registers, etc. prevents execution environments from being able to access or influence each other. Statically partitioning these structures in hardware or flushing altogether allows isolation between enclaves and other potentially malicious code. This flushing can be triggered by the use of a custom purge instruction in the SM, or by some other hardware queue activated by enclave switch. Our implementation will require further research to determine the best way to trigger a flush.

Random Instruction Insertion. Side-channel timing attacks typically rely on comparing execution traces across multiple runs of a program. To mitigate against this, random dummy instructions will be inserted into the pipeline during execution [2]. These instructions will be inserted into the decode stage, freezing the fetch stage for one cycle. Rather than inserting different types of NOPs, the dummy instructions reuse opcodes, funct3, and funct7 fields from previously executed instructions to make them statistically similar to previous instructions. The timing of the instruction insertion is controlled by a variable clock divider operating at twice the target insertion frequency and gated by a random number generator.

To prevent dummy instructions from corrupting program state, a dynamic register bank that that maps the architectural register to physical registers is used, much like register renaming in out-of-order cores. Each physical register has a valid bit, and when a valid write occurs, the previously assigned physical register is first invalidated before a new one is randomly selected from the pool of free registers. When dummy instructions write to registers, it writes to the physical registers marked as invalid so that the dummy registers can change dynamically, making side-channel analysis more difficult. Memory accesses by dummy instructions are confined to a reserved unique address, and branch instructions are excluded from the dummy instruction set entirely to avoid pipeline flushing.

2.2.2 Fault Injections. In addition to the architectural and software protections discussed previously, FORTE intends to incorporates a set of hardware-level mitigations targeting both naturally occurring and intentionally induced fault injection attacks, including single-event upsets (SEUs) from radiation and clock/voltage glitching attacks from a malicious actor with physical access. Based on our research, we prioritized these features by their security impact relative to their area and implementation cost. The three highest-priority features (Error Correction Coding, Cacheline/CSR Scrubbing, and a Clock Integrity Monitor) form the core of our fault-injection defenses and are committed features of the design. The remaining features (an Inverter-Chain Glitch Detector, CSR Hardening/Verification, and a Brown-out Detection Circuit) are stretch goals to be implemented if time and area permit.

Error Correction Coding (ECC). Bit flips caused by background radiation (cosmic rays, or solar radiation in space datacenter environments) or by an intentional attacker (e.g., decapping a die and exposing it to a focused light source) can silently corrupt stored state. To address this, we implement Single Error Correction, Double Error Detection (SECDED) codes across our memory hierarchy.

The number of parity bits k required for a datum of size m is determined by the Hamming SECDED bound, $2^k \geq m + k + 1$, with an additional bit $c = k + 1$ added for double-error detection. For our target word sizes, this yields 7 check bits for 32-bit data, 8 check bits for 64-bit data, and 10 check bits for a 256-bit cacheline.

We prioritize ECC coverage as follows:

- **Registers and L1 I-Cache (highest priority):** Registers are touched by nearly every operation, making them the most exposed structure. Combined with a write-through policy for the I-cache, only parity checking is required on this path, simplifying the correction logic. Whether ECC propagation to lower cache levels is necessary for tape-out remains an open question to be examined further.
- **L1 D-Cache:** Because the D-cache uses a write-back policy, dirty cachelines cannot simply be re-fetched on error, so the full SECDED correction pipeline is required here rather than parity alone.
- **Page Table Walker and TLB (lowest priority):** Parity protection alone is sufficient. If tampering is detected beyond a tolerable threshold, the TLB can simply be disabled, falling back to a page-table walk.

The encode/decode hardware consists of an XOR-tree encoder (with maximum depth $\log_2(\#\text{ check bits})$) generating the check bits on write, and a corresponding decoder on read that produces two flags: a single-error flag, corrected automatically via XOR-based syndrome decoding, and a double-error flag, which is unrecoverable and must raise an exception to preserve data integrity, since a silent double-bit error could otherwise corrupt program execution undebatably.

Cacheline and CSR Scrubbing. While ECC corrects errors at the point of access, errors in infrequently-accessed cachelines or CSRs can accumulate over time, increasing the probability that a second bit flip occurs in the same word before the first is corrected and producing an uncorrectable double-bit error. To bound this risk, we implement a

scrubbing FSM that proactively walks through cache memory during idle cycles, reads each line, applies SECDED correction if a single-bit error is found, and writes the corrected line back.

This approach drains single-bit errors continuously at very low area cost, since it requires only FSM control logic rather than additional storage. We additionally consider a banked cache organization, which would allow the scrubber to use a simple linear access pattern and isolate sections of the cache for scrubbing without contending with normal cache traffic. Given its minimal area overhead and direct mitigation of error accumulation, cacheline scrubbing is considered an essential feature of our fault-tolerance design.

Clock Integrity Monitor. Clock glitching, or introducing extra edges, removing edges, or manipulating the duty cycle of the clock signal, is a common fault injection technique that primarily disrupts the fetch stage and IF/ID pipeline register, potentially causing instructions to be skipped or corrupted. To detect this class of attack, we implement a digital clock monitor that compares the system clock against a reference signal, such as a reference oscillator or a digital pulse counter that expects N pulses within an N-cycle window.

If the observed clock deviates from the reference by more than a configurable margin (e.g., ±5%), the monitor asserts an alarm signal that can trigger a pipeline flush or full system reset, preventing the processor from continuing execution on a glitched clock edge. One open design question we intend to address is how to handle corruption of the reference signal itself (e.g., via a bit flip), which could otherwise produce a false sense of security; we are considering hardening the reference signal path itself.

Additional Features (Time/Area Permitting). If area and schedule allow, we additionally plan to investigate the following lower-priority mitigations:

- **Inverter-Chain Glitch Detector:** A delayed copy of the clock (or supply voltage) signal is generated via a chain of inverters. The original and delayed signals are passed through an AND gate; a glitch on either signal produces a detectable discrepancy at the gate output, which can be used to raise an alarm. This structure is attractive due to its very low area cost, but is complementary to, rather than a replacement for, the digital clock monitor.
- **CSR Hardening/Verification:** Beyond the SECDED protection applied to general memory, the most security-critical CSRs (`mstatus`, `mepc`, `mtvec`, and the PMP configuration/address registers) are candidates for Triple Modular Redundancy (TMR) with majority voting, given that a single corrupted bit in one of these registers can escalate privilege or redirect execution. As an additional check, we are considering a privilege-guarding scheme that latches CSR values across privilege transitions (e.g., on `mret`) and re-executes the transition if a mismatch is detected on readback.
- **Brown-out Detection Circuit:** An analog comparator monitors V_{dd} and asserts an alarm if supply voltage drops below a defined threshold, guarding against voltage glitching attacks. This mitigation has very low area overhead, but its response latency may be too slow to catch short-duration glitches, which is why it is deprioritized relative to the clock monitor and inverter-chain approaches.

2.2.3 Shadow Pipeline Implementation. In ASICs exposed to radiation, faults are more likely to manifest in logic that is larger in area. Hence, to cover the most likely victim logic/area of the core that could be susceptible to faults, we create redundant functional units for the execution stage [1]. Two identical functional units calculate the same operation on the same operands. If there is ever a mismatch, then a transient fault is detected. If 3 such transient faults are detected in a row, then a permanent fault is detected. Otherwise, this transient fault has gone away and regular computing is resumed. If a permanent fault is detected, the processor must find the faulty functional unit to disable. To find which functional unit has the permanent fault, the functional units both perform a calculation on the same, altered inputs to see if they get an expected value. For example, recomputing an ADD operation would use inverted operands and carry-in supplied to the module. Therefore, the output should be the inverse of the original addition. Whichever module does not have this is the faulty module, and the processor can decide to trap or disable this functional unit and continue execution. This approach allows the processor to respond to transient and permanent faults in the execution stage of the processor.

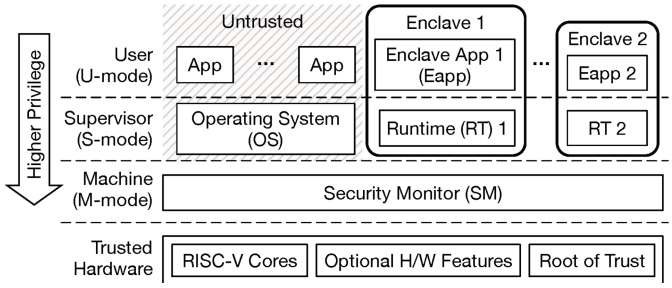

Figure 1: Diagram of Keystone System (via the Keystone Docs)

2.3 RISC-V Keystone

Once Linux capability is achieved, we will proceed to include hardware support for Keystone features in parallel with the previous hardware features mentioned, which require PMP (Protected Memory Protection) CSRs and a hardware RoT, including a device specific key and a secure boot ROM.

2.3.1 Keystone Overview. We provide a brief summary of Keystone, mostly copied from the original Keystone paper [4], website, and GitHub [3].

Keystone is an open framework for architecting trusted execution environments (TEEs) based on hardware enclaves. As hardware designers, we will be developing the Trusted Hardware component (see 1). During design and bringup, we will compile the **Security Monitor** (SM). The SM provides an interface for managing the lifecycle of enclaves as well as for utilizing platform-specific features. **Enclaves** are environments isolated from the untrusted OS and other enclaves. Each enclave is given a private physical memory region which is only accessible by the itself and the SM. Each enclave consists of a user-level enclave application (eapp) and a supervisor-level runtime. The **Enclave Application** (eapp) is the user-level application that executes in the enclave. One can build a custom eapp from scratch or modify an existing RISC-V executable in Keystone. **Runtime (Eyrie)** is S-mode software which implements functionality such as system calls, trap handling, virtual memory management and so on.

2.3.2 Keystone Hardware Requirements. In addition to the standard privileged spec, FORTE will implement 16 PMP CSRs as specified in the Section 3.7 of the privileged spec [11]. These registers will define separate regions of memory and will only be writable by the SM. These CSRs contain an address and flags indicating what permissions are given to that memory region. These include read/write access, instruction execution, address matching mode, and the ability to lock the PMP region to be permanent until reset. PMP checks apply to all memory transactions, except M-mode transactions on unlocked regions.

To create keys and attestation reports, the SM needs access to a True Random Number Generator (TRNG). To achieve this we will use several ring oscillators of prime number lengths combined with an XOR tree and sampled with a flip-flop, with optional post-processing. Software will have access to generated random numbers through the RISC-V Zkr extension.

A device-specific private key is needed for the hardware RoT to attest the SM and create a secure key for it. Due to the scope of this project and hardware limitations, the private key will be hard-encoded with a known value.

We propose some additional features to enable secure execution, as area permits: Architectural State Scrubbing [12], which clears all structures and caches on enclave entry/exit, and Debug locking [10], which mitigates DFT side-channels by securing debug ports.

2.3.3 Boot "ROM" for Keystone. The Secure Boot "ROM" is part of the RoT in our system and contains the minimal functionality to authenticate the next software stage. After a CPU reset, the Boot "ROM" starts executing, initializes the CPU state, and loads the next-stage firmware into memory. The Boot "ROM" also contains a software cryptographic hash function (such as SHA-256) to measure the loaded image and a signature verification method (such as RSA-2048) that uses a public key embedded in the "ROM" to validate the image. On any verification failure, the "ROM" has logic to halt execution. To facilitate software encryption, we add support for the Zbkb bit manipulation extension. The "ROM" also contains a boot measurement that is passed on to the next stage to support the attestation required for Keystone.

2.4 Bring-up and High-Level Board Configuration

2.4.1 Bring-up Configuration and I/O Pins. Given the current configuration, we expect our I/O pins to be split across several major categories. FORTE will require an interface to an FPGA that can interact with external devices and peripherals such as memory or UART. The primary off-chip communication path will be a strobe-based AXI-Lite interface, using a bidirectional 16-bit data bus and an 8-bit address bus for memory-mapped transactions between the FORTE and an external FPGA or board-level system that communicates with peripheral devices. We also reserve 6 pins for QSPI boot/configuration access and 4 pins for DFT and manufacturing test. Finally, we allocate relevant power pairs, as well as clock and reset pins. We also plan to have an interrupt request pin to communicate with peripherals. This gives us a rough estimate of 45-51 pins, depending on the number of peripherals we support. This pinout remains a high-level estimate, and we expect to refine the exact allocation as the AXI-Lite strobe protocol, pad-ring constraints, test requirements, and package-level power integrity requirements are finalized.

2.4.2 Kernel Configuration. We plan to boot the mainline Linux kernel, specifically v6.6, since this is the earliest and most lightweight version of the kernel that contains all of the patches for Zbkb and Zkr support (see 2.3.3). For bringup we plan to use an FPGA to interface over a modified AXI4-lite bus with the chip. The FPGA will provide controllers for all peripherals including UART and DRAM. The chip PCB will include status LEDs and possibly power regulators.

3 Timeline

- **Milestone 1 (by July)**: Complete high-level architecture review, and finish RTL/Verification of baseline RV64IMAZicsr _Zifenci processor. Verification would consist of fuzzing and co-sim using Spike. This sets up our baseline for future feature implementation
- **Milestone 2 (by August)**: Complete the Keystone Hardware Features along with the Fault Injection Features.
- **Milestone 3 (by September)**: Implementation and module-level verification of the Hardware Root of Trust and Shadow Pipeline. In parallel, start FPGA emulation of FORTE with Linux, and setting up scaffolding for testing Linux boot.
- **Milestone 4 (by October/RTL Freeze)**: Formal verification of all security features, as well as processor-level testing and I/O and DFT interfaces on an FPGA.
- **Milestone 5 (by GDSII Handoff)**: Physical design and chip-level testing.

4 Team Division and Strengths

4.1 Team Strengths

We will separate the areas of this project into these distinct categories: Architecture/RTL, Verification, Physical Design (PD), Security, and Software/Modeling. All members will be available for bringup in the spring.

Pranav Bellur – Rising Senior in Computer Engineering

- *Relevant Coursework:* 411, 425, 483, 391, CS 461
- *Experience:* 120 CA, Hardware Research w/ Prof. Deming Chen
- *Strengths:* Architecture/RTL, Circuit Design, PD, Security

Rohan Das – Rising Senior in Computer Engineering

- *Relevant Coursework:* ECE 411, 425, 482, 483, 391
- *Experience:* 425 CA, RF Systems Intern at SpaceX
- *Strengths:* Architecture/RTL, Circuit Design, PD

Swetha Karthikeyan – Rising Senior in Computer Engineering

- *Relevant Coursework:* ECE 411, ECE 425, ECE 391
- *Experience:* 385 CA (First to boot Microblaze Linux on FPGA), ECE 397 KL Course Staff (SIGPwny Embedded CTF Lead), Hardware Engineering Intern at Microsoft
- *Strengths:* Architecture/RTL, Software/Modeling, Security

Siddharth Rau – Rising Senior in Computer Engineering

- *Relevant Coursework:* ECE 411, ECE 391, ECE 511, ECE 425
- *Experience:* 4th Place in ECE 411 competition, DV Intern at Ambarella
- *Strengths:* Architecture/RTL, Verification, Physical Design

Sai Venkatakrishnan – Rising 1st yr MS in Computer Engineering

- *Relevant Coursework:* ECE 411, ECE 391, ECE 425, ECE 385
- *Experience:* ECE 411 CA, ASIC Design Intern at NVIDIA
- *Strengths:* Architecture/RTL, Verification, Software/Modeling

Ved Vyas – Rising Senior in Computer Engineering

- *Relevant Coursework:* ECE 411, ECE 425, ECE 391, ECE 479
- *Experience:* HW Engineering Intern at IMC Trading, Digital Design Intern at Plexus, Research - FAST Lab
- *Strengths:* Architecture/RTL, Verification, PD, Security

4.2 Division of Work

All team members are strong in multiple technical areas and will be able to actively participate in most or all stages of development and bringup. To ensure timely completion of milestone and associated subtasks, we designate 2-3 people per category to oversee progress:

- **Architecture and RTL**: Siddharth, Sai, and Ved
- **Verification**: Siddharth and Pranav
- **Physical Design**: Rohan and Pranav
- **Bringup and Software**: Swetha and Sai
- **Validation**: Rohan, Ved, and Swetha

We all have a good understanding of architecture, low-level software, and physical design principles, so we believe we can tackle the complexity of the project laid out in this document.

References

[1] S. Gupta, N. Gala, G. S. Madhusudan, and V. Kamakoti. 2015. SHAKTI-F: A Fault Tolerant Microprocessor Architecture. doi:10.1109/ATS.2015.35

[2] Zhangqing He, Tianyong Ao, Meilin Wan, Kui Dai, and Xuecheng Zou. [n. d.]. https://www.iaeng.org/IJCS/issues_v43/issue_1/IJCS_43_1_08.pdf

[3] Dayeol Lee, Gregor Haas, Gui Andrade, alexthomasv, David Kohlbrenner, Stephan K., zchn, cathylu10, red-robby, Mohit Singla, Akash, Andreas Kuster, philippgie, Ahmad Syarif, Ben Laurie, Edward Im, Eric Schneider, Evgeny P, Gubaer, Jarkko Sakkinen, Moritz Sanft, Pascal Cotret, Tsukasa OI, Akihiro Saiki, Akira Tsukamoto, Alejandro Cabrera Aldaya, Collin Reilly Clark, Deepak Sirone, Kazumoto Kojima, and Kevin Laeufer. 2025. keystone-enclave/keystone. https://github.com/keystone-enclave/keystone

[4] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[5] lowRISC. 2015. Bootload Procedure. https://www.cl.cam.ac.uk/~jrrk2/docs/untether-v0.2/bootload/.

[6] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2009. CACTI: An Integrated Cache and Memory Access Time, Cycle Time, Area, Leakage, and Dynamic Power Model. Hewlett-Packard Laboratories, https://github.com/HewlettPackard/cacti.

[7] Nangate Inc. 2011. NangateOpenCellLibrary: An Open Standard Cell Library. http://www.nangate.com.

[8] OpenHW Group. 2026. CORE-V Wally: A Configurable RISC-V Processor. https://github.com/openhwgroup/cvw. Accessed: 2026-06-11.

[9] RISC-V Software Sources. 2024. OpenSBI: RISC-V Open Source Supervisor Binary Interface. https://github.com/riscv-software-src/opensbi.

[10] Paul Donahue Tim Newsome. [n. d.]. https://docs.riscv.org/reference/debug-trace-ras/debug/_attachments/riscv-debug-specification.pdf

[11] Andrew Waterman and Krste Asanović. 2021. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture.* Technical Report. RISC-V Foundation.

[12] Nils Wistoff, Moritz Schneider, Frank K. Gürkaynak, Gernot Heiser, and Luca Benini. 2025. Systematic prevention of on-core timing channels by full temporal partitioning. https://arxiv.org/abs/2202.12029