

# FireGuard: A Generalized Microarchitecture for Fine-Grained Security Analysis on OoO Superscalar Cores

Zhe Jiang<sup>†</sup> Sam Ainsworth<sup>||</sup>, Timothy Jones<sup>¶</sup>

<sup>†</sup>National Center of Technology Innovation for EDA, School of Integrated Circuits, South East University, People’s Republic of China,  
<sup>||</sup>University of Cambridge, United Kingdom, <sup>¶</sup>University of Edinburgh, United Kingdom

**Abstract**—High-performance security guarantees rely on hardware support. Generic programmable support for fine-grained instruction analysis has gained broad interest in the literature as a fundamental building block for the security of future processors. Yet, implementation in real out-of-order (OoO) superscalar processors presents tough challenges that cannot be explored in highly abstract simulators. We detail the challenges of implementing complex programmable pathways without critical paths or contention. We then introduce FireGuard, the first implementation of fine-grained instruction analysis on a real OoO superscalar processor. We establish an end-to-end system, including microarchitecture, SoC, ISA and programming model. Experiments show that our solution simultaneously ensures both security and performance of the system, with parallel scalability. We examine the feasibility of building FireGuard into modern SoCs: Apple’s M1-Pro, Huawei’s Kirin-960, and Intel’s i7-12700F, where less than 1% silicon area is introduced. The Repo. of FireGuard’s source code: <https://github.com/SEU-ACAL/reproduce-FireGuard-DAC-25>.

## I. INTRODUCTION

With ever-growing computation capacity, modern systems increasingly execute applications on shared platforms [1], [2]. The latest in-vehicle information systems from BYD, the world’s largest EV manufacturer [3], allow installation and execution of less-verified third-party workloads with life-critical workloads, threatening system-wide safety and trustworthiness [4], [5]. Similarly, Android phones allow untrustworthy applications to coexist with security-critical banking software [6]. Modern systems thus need the ability to analyze, detect and mitigate vulnerabilities in an always-on and comprehensive way. **Existing work.** Always-on, comprehensive security analysis relies on hardware support [7]–[16]. Current implementations, e.g., Arm’s MTE [7] and BTI [17], and Intel’s LAM [18] and CET [19], have very limited flexibility, allowing an attacker to bypass them by simply shifting their targets. Also, deployment against the latest threats requires lengthy development, leading to long vulnerability windows.

Hardware-assisted fine-grained instruction analysis [20]–[24] presents a new paradigm, adding observation channels into cores to filter and analyze execution (e.g., committed instructions, memory accesses and function calls), in programmable analysis engines, (e.g., microcontrollers, accelerators or FPGAs). Through reconfigurability and parallelism, they can adapt to cover a wide range of attacks.

**Challenges.** Existing efforts have been conducted through software simulation [21], [22], [24] or on simple in-order processors [23], [25]. They do not show how to build analysis into real OoO superscalar cores, or consider whether this is possible without a full overhaul of core design or at palatable overhead. When we tried to build such a mechanism into a real core, we hit bottlenecks and contention at every stage of the processing pipeline, from data collection and filtering to distribution and analysis. Typical programming models [21] require full generality: any and all instructions can be monitored simultaneously, with any and all data selected from them, and sent to multiple analyses at once. Inside the core, such data must be collected in a narrow time window, at commit-time rather than execute-time to reflect ordering, but before the data is overwritten. While data can be filtered down if it is irrelevant for currently running analyses, if every instruction could be monitored, the mechanism that decides which instructions are processed must be highly parallel and high throughput to avoid back-pressure slowing down the core. Even when data volume has been reduced using filtering, the generality required of distribution made it a challenge for us to send data to multiple analysis engines at once without unscalable broadcast.

**Contributions.** We introduce the *first microarchitecture* for hardware-assisted fine-grained instruction analysis, FireGuard, implemented in RISC-V BOOM [26]. To this end, we present:

- a buffer-free data-forwarding channel by inserting bypass circuits at key locations within the main core, giving fine-grained visibility of execution without significant microarchitectural invasion;
- a superscalar event filter, utilizing a fleet of SRAM-based *mini-filters* to handle commit of arbitrary instruction types at the same width as the core, shrinking the content volume for later analysis and preventing extra performance degradation.
- broadcast-free communication channels, partitioning a task mapper into a distributed fabric network and a scalable allocator. The former enables independent data paths per transaction, while the latter uses multiple *Scheduling Engines* to simultaneously route data to all interested engines;
- a microarchitecture-assisted programming model for analysis engines, with optimizations across the hardware-software stack via new queue-communication instructions, a novel tightly coupled ISA extension (ISAX) interface to minimize data hazards, and unrolling-aware custom instructions.

We implement FireGuard in Chisel, deploying on Virtex Ultra-Scale+ FPGAs using FireSim [27]. We boot full Linux and deploy different workloads with various security schemes: a Custom Performance Counter (PMC) [21], a shadow stack [28], AddressSanitizer [29], and Use-after-Free (UaF) detection [30]. The results show that employing four analysis engines is sufficient for running a PMC with 2.5% performance overhead, a shadow stack with 2.1% overhead, AddressSanitizer with 39% overhead (reducing to 6% with 12 engines), and UaF detection with 42% overhead (16% with 12 engines). Overhead can be completely removed by further integrating hardware accelerators. We evaluate the feasibility of building FireGuard into modern SoCs: M1-Pro, Kirin-960 and i7-12700F, where less than 1% silicon area is required.

## II. FINE-GRAINED INSTRUCTION ANALYSIS

Although studies on fine-grained instruction analysis converge on the same top-level idea, the architecture and APIs across them slightly differ. We adopt the Guardian Council [21] as a generalized example, build FireGuard upon it and discuss the feasibility of the others.

**The Guardian Council.** The architecture features a generic frontend that can analyze any instruction type. It inserts a data-forwarding channel into the compute core’s commit stage, collecting data (e.g., opcode, operands) from program execution. An event filter performs a pre-check on all observed data, selecting relevant information (dependent on the analyses being run) and sending it to the analysis engines through a mapper. The mapper chooses the target according to which analyses (*guardian kernels*) are running on each analysis engine. Using the passed information, the engines run guardian kernels in parallel, validating security and finding vulnerabilities.

**The other architectures.** The key difference between the Guardian Council and similar mechanisms [22]–[24] is the analysis engines in the backend. It deploys a sea of microcontroller-sized cores ( $\mu$ cores) to run user-programmable validations (running on conventional *main cores*) in parallel. This allows for highly efficient, updateable and upgradeable analysis, by exploiting parallelism within validation tasks and between multiple independent tasks at once, and makes use of the fact that  $\mu$ cores consume significantly less hardware, by multiple

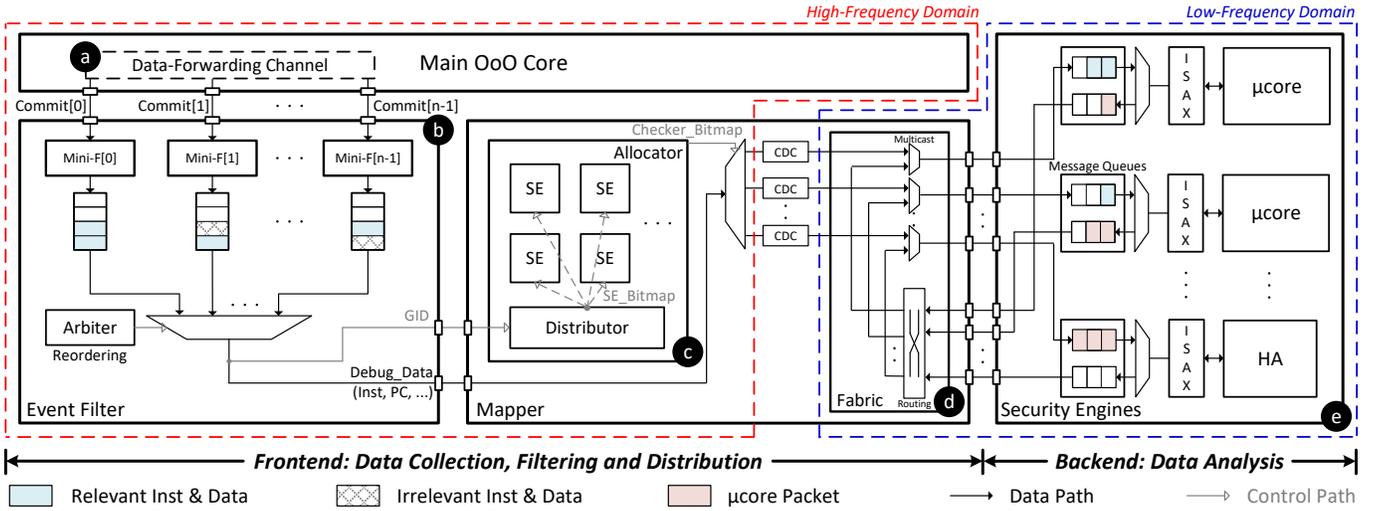


Fig. 1: Overview of FireGuard (*Mini-F*: *Mini-Filter*; *GID*: *Group Index*<sup>1</sup>; *HA*: *Hardware Accelerator*; *SE*: *Scheduling Engine*): **a** buffer-free data-forwarding channel extracts the main core’s execution events; **b** a superscalar filter pre-checks extracted events, identifying relevant instructions and selecting channels for analysis; **c** an allocator associates an SE to each kernel to distribute contents, and **d** a distributed fabric network transmits contents to the  $\mu$ cores or HAs; **e** kernels running on  $\mu$ cores or HAs fetch contents and validate their security.

orders of magnitude, compared to OoO superscalar main cores, which must achieve high single-threaded performance unnecessary for a  $\mu$ core. Deploying to an FPGA [23] or to multiple large analysis engines [22], [24] involves similar challenges of distribution.

### III. FIREGUARD

We show the concept of fine-grained instruction analysis is feasible by building a real system (FireGuard) upon the the Guardian Council architecture. Figure 1 fleshes out the data-forwarding channel, filter, mapper and analysis engines, with a careful redesign allowing practical implementation that can handle generality at each stage<sup>2</sup>.

To do so, we integrate simple read-only bypass circuits at various locations within the main core (figure 1 **a**) to extract data while avoiding new buffers between execute and commit. These forward on debug data associated with committed instructions for detailed analysis. Since the design only involves minor microarchitectural changes to the main core low and avoids adding significant hardware overhead. With that, we deploy a *superscalar* set of SRAM-based mini-filters to handle high commit widths in parallel (figure 1 **b**). It can be programmed to be sensitive to any arbitrary group of instructions, and guarantees that filtering can be completed while the data is still available in the core. To timely deliver the data to the analysis engines, we divide the mapper into a scalable allocator (figure 1 **c**) and a distributed fabric network (figure 1 **d**). The allocator associates a Scheduling Engine (SE) to each guardian kernel to independently transmit contents while avoiding broadcast, while the fabric network establishes dedicated paths for filtered contents and inter-checker communication to mitigate contention. In each analysis engine, we develop a data-hazard-aware ISAX interface (figure 1 **e**) and integrate it into the Memory Access (MA) stage of the  $\mu$ core’s pipeline, connecting its message queues. This allows us to develop drivers and programming models for guardian kernels, including instructions designed for efficient, hazard-minimizing design patterns.

#### A. Data-forwarding channel

The data-forwarding channel is deployed at the main core’s commit stage, extracting, selecting, and transporting debug data associated

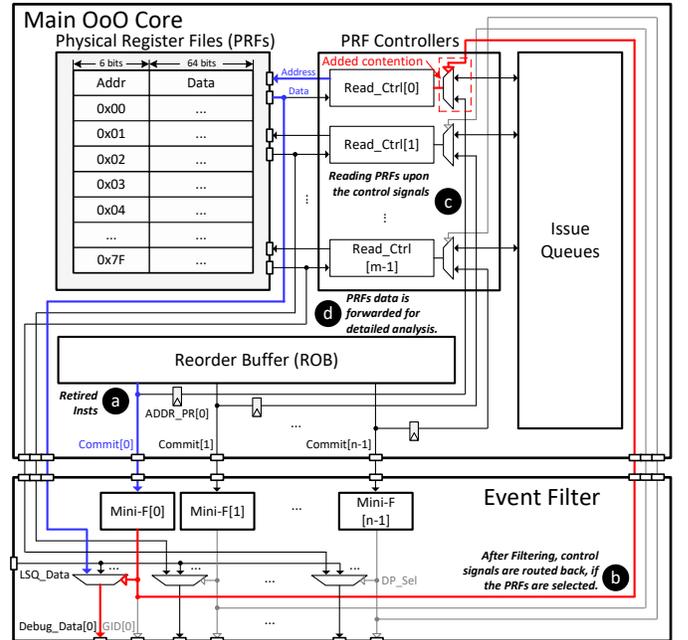


Fig. 2: Data-forwarding channel, using PRFs as an example (*blue lines*: *data-forwarding paths*; *red lines*: *filtering paths*; *gray lines*: *control paths*; *DP\_Sel*: *Data Path Selection*): **a** commit paths from the ROB are hooked, forwarding retired instructions to mini-filters and storing the PRF access addresses in temporary registers; **b** mini-filters pre-check the forwarded instructions, sending control signals to PRF controllers when PRF data is selected; **c** the data-forwarding channel preempts the controllers and feeds the addresses temporarily stored; **d** the read data is routed back for in-depth analysis.

with every retired instruction. Since the relevant data is already stored in different locations within the main core, we design a buffer-free implementation (i.e., adding no new intermediate storage between out-of-order execute and in-order commit), by inserting bypass circuits at the Reorder Buffer (ROB), Physical Register Files (PRFs), Load Store Queue (LSQ), and Fetch Target Queue (FTQ). This transports PC address, instruction data, operand data, and memory and jump addresses during commit for any instruction selected by the filter, avoiding reads of information not selected. These points cover all locations that store outputs of arithmetic-logic units, enabling fine-grained visibility with minimal contention.

<sup>2</sup>We partition the microarchitecture into two clock domains. The main core, along with its associated modules (e.g., L1-cache), the data-forwarding channel, the filter, and the allocator, are within a high-frequency domain driven by a fast clock source. This avoids any resulting slowdown caused by data-forwarding, filtering, and allocating activities. The more parallel  $\mu$ cores and fabric network are within a low-frequency domain driven by a slower clock source, with handshake-based clock-domain crossing. This ensures energy efficiency and prevents the simple  $\mu$ cores from becoming the critical path.

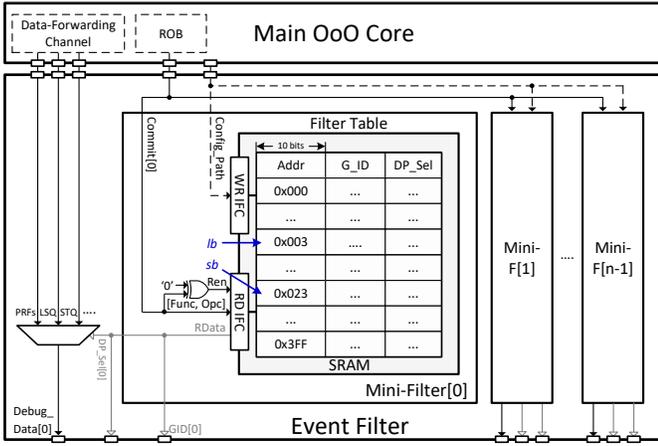


Fig. 3: Microarchitecture of a mini-filter (gray lines: control paths; WR/RD IFC: Write/Read Interface).

**PRF Example.** Figure 2 shows the microarchitecture using PRFs as an example, connecting the ROB, filter, and PRFs. On the ROB side, we add logic to hook onto each commit, transmitting the retired instructions to the mini-filters, as well as address registers storing the PRF indices accessed by each instruction (figure 2 a). In the cycle following retirement, the mini-filters identify Group Indexes (GIDs) of the transmitted instructions and select data based on programmed settings (section III-B). If the PRF data is selected, a control signal is routed back, preempting the PRF controller (figure 2 b).

The PRF’s read controllers are statically multiplexed (figure 2 c) between the issue queue (for executing instructions) and the mini-filters. Mini-Filter [x] has priority access to Read\_Ctrl [x] should it require it, to allow the data to be read and transported immediately (figure 2 d), avoiding any buffering or delays in freeing the physical register. This means that an instruction attempting to use the same port will be delayed until the next cycle, resulting in contention<sup>3</sup>.

### B. Event Filter

The event filter pre-checks all retired instructions. If instructions are selected for analysis, it returns their Group Indexes (GIDs) for the mapper and programs the data-forwarding channel to select data. We give a superscalar implementation that enables simultaneous filtering of all instructions, ensuring the filter keeps up with the main core.

Figure 1 b shows its design, including a set of mini-filters, FIFO queues, and an arbiter. A mini-filter is connected to each superscalar commit path of the ROB. Each is indexed by the instruction opcode from the data-forwarding channel, and returns programmed GIDs and selects debug data from the chosen channel for instructions analyzed. Filtered contents are buffered into paired FIFO queues, allowing a shared arbiter to arrange the output into sequence.

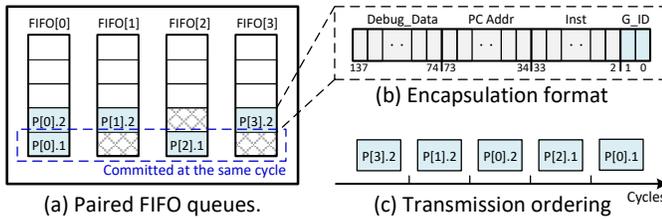


Fig. 4: Reordering with 4-width FIFOs. P[x].y: yth packet in FIFO[x].

**Mini-filters.** Mini-filters use a SRAM-based look-up table (figure 3). The address (10-bit) is an index formed of the concatenated RISC-V opcode (lower 7 bits) and function code (higher 3 bits) for all

<sup>3</sup>In LDQ, STQ, and FTQ, similar microarchitectures are deployed to obtain memory access or jump addresses. Unlike PRFs, where forwarded data can be stored at arbitrary addresses, the tops of these queues consistently hold the data associated with the most recently retired instructions. When a mini-filter decides to forward a load, store, or jump address, the bypass circuits directly transmit from the relevant queue’s top, avoiding contention.

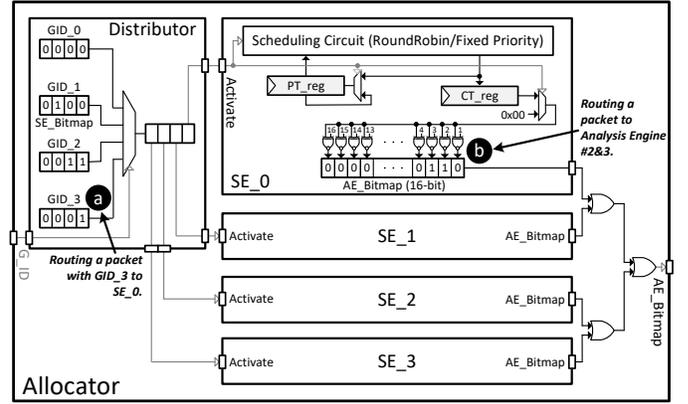


Fig. 5: Allocator microarchitecture. 3 GIDs and 4 SEs allocate contents to 16 engines (gray lines: control paths).

possible instructions, and stores the mapper’s GID and the desired data paths (PRF, LSQ and/or FTQ) for each instruction. For instance, 0x03 and 0x23 index RISC-V 1b and sb, respectively. We route the instructions’ opcodes and function codes to the address port of the SRAM’s read interface, and direct the read data to the data-forwarding channels and the mapper, assisting in data selection and allocation<sup>4</sup>.

### C. Mapper

The mapper routes filtered packets to engines based on configured parallelization policies, and also enables data exchange between the analysis engines, fostering complex parallelism schemes for guardian kernels. It is fully programmable, allowing any instruction to flow to all interested guardian kernels and be scheduled to an analysis engine core for each. Figure 1 c and d illustrate the microarchitecture of the mapper, which we partition into a scalable allocator followed by a distributed fabric network. The mapper transitions FireGuard’s processing from superscalar to scalar: unlike the filter, it only handles one packet per cycle. This rarely impedes a 4-wide BOOM’s performance (we saw less than 0.5% slowdown)<sup>5</sup>.

**Allocator.** The allocator uses a 2-level indirection bitmap to allocate packets across the analysis engines. Figure 5 shows the allocator, including a distributor and a set of Scheduling Engines (SEs). The distributor manages a bitmap between GIDs and SEs, deciding which SE(s) should be activated during packet transmission. SEs are one-to-one associated to a guardian kernel, and each maintains another bitmap between itself and analysis engines, allocating filtered packets to groups of analysis engines executing one guardian kernel.

In the distributor, an SE\_Bitmap register is assigned to each GID, and individual register bits are used to index the SEs that are interested in that specific GID. For instance, if packets with GID 3 should be sent to SE 0, bit 0 in SE\_Bitmap[3] is set (figure 5 a). SEs use a scheduling circuit, two scheduling registers (PT\_reg and CT\_reg), and an AE\_Bitmap (Analysis Engine Bitmap) register. The scheduling circuit implements several policies, e.g., fixed, round-robin, and block mode [21], where the latter is used to send all messages to one  $\mu$ core until it is full before moving to the next, for when message locality is important, e.g., shadow stack. When the scheduling circuit is activated, the PT\_reg (previous target) is

<sup>4</sup>We place a pair of FIFO queues to connect the mini-filters, buffering the filtered contents (figure 4(a)). Although these filtered contents are produced by the mini-filters in parallel, they must be sent sequentially, aligning with their commit order, since analyses can be sensitive to program order (e.g., shadow stack [28]). We encapsulate filtered contents to achieve this (figure 4(b)). If an instruction is discarded, an invalid packet is generated and also pushed into the FIFO queue in order to preserve ordering at the end of the FIFO. The arbiter uses a finite state machine to transmit packets (figure 4(c)) in-order, consuming one clock cycle for a valid packet while skipping invalid packets.

<sup>5</sup>Even so, a superscalar mapper could be considered for a more powerful core. To achieve this, modifications are necessary for both the fabric and allocator, including duplicating communication channels and SEs. Extra arbiters must be deployed to manage contention, e.g., when multiple packets are sent to the same security engine.

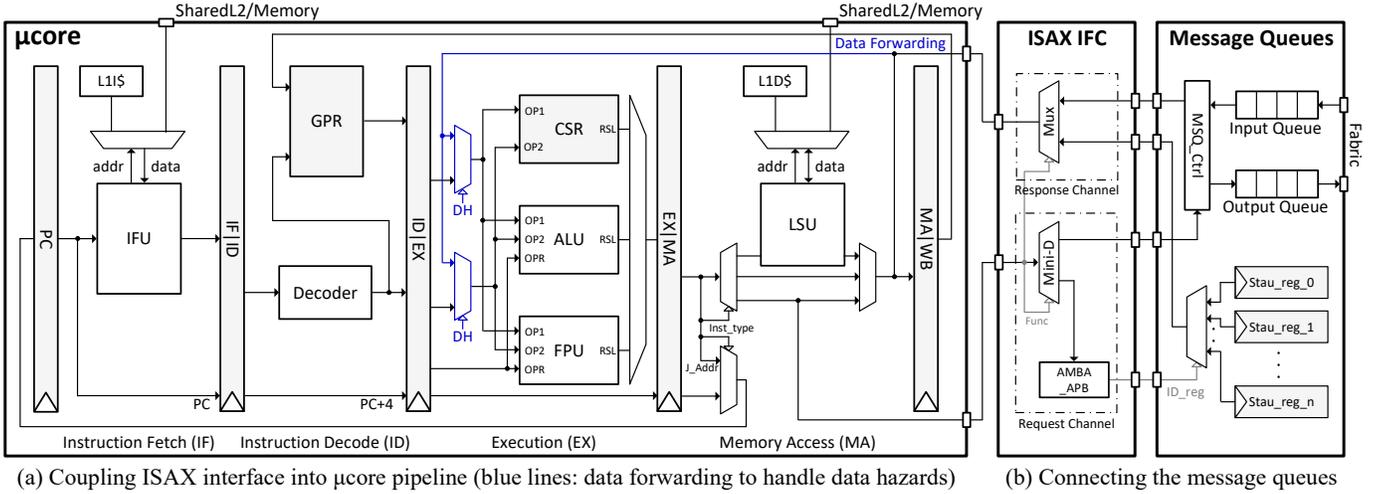


Fig. 6: Custom instructions are integrated into the  $\mu$ core’s pipeline at the MA stage. (gray lines: control paths; MSQ\_Ctrl: Message Queue Controller; IFU: Instruction Fetch Unit; Mini-D: Mini-Decoder). For the ISAX interface (IFC), a full-duplex microarchitecture handles requests and responses independently, with a mini-decoder and multiplexer. In the request channel, a mini-decoder directs an ISAX request to the message queues’ MSQ\_Ctrl or status registers (via an APB bridge). In the response channel, a multiplexer selects returned data, routing it to the commit stage and the EX stage using forwarding logic for handling data hazards.

Instruction	Target Queue	Descriptions
count rd, rs1	Input/Output	Count number of packets buffered in message queue rs1, returning it to rd.
top rd, rs1	Input	Return bitfields [rs1+63:rs1] of the first element to rd.
pop rd, rs1	Input	Remove the first element and return its bitfields [rs1+63:rs1] to rd.
recent rd, rs1	Input	Return bitfields [rs1+63:rs1] of the most recently removed element to rd.
push rs1	Output	Push packet rs1 for transmission.

TABLE I: Main control instructions for the message queues.

used to generate the *current target*, buffered in CT\_reg until it is moved to the PT\_reg after packet transmission. The CT\_reg value sets the relevant target bit in the SE’s AE\_Bitmap (figure 5 (b)). The AE\_Bitmap returned by all SEs are combined using OR gates to form the decision, selectively broadcasting to the analysis engines.

**Fabric network.** The fabric network features a half-duplex multicast (1-to-N) channel and a full-duplex routing (N-to-N) channel. The multicast channel selectively broadcasts packets, while the routing channel allows the checkers to transmit packets among themselves.

The multicast channel uses multiplexers to direct packets from the event filter to the message queues in the analysis engines. All multiplexers are controlled by the allocator, regulating transmission and masking of each packet. The routing channel uses a Manhattan grid [31]–[34] Network-on-Chip (NoC) mesh. Each router has five bi-directional ports, connected to other routers located to its north, south, east, and west, as well as to an analysis engine.

#### D. ISA and Programming Model

The analysis engines run guardian kernels concurrently on  $\mu$ cores, finding vulnerabilities. To handle the frequent hardware-software interactions, we use a FIFO-based programming model implemented into RISC-V Rocket cores via custom instructions. As these instructions take up a large fraction of total  $\mu$ core cycles, we integrate FIFO-management instructions using a new tight-coupling arrangement; Rocket’s existing ISAX interface, which executes custom instructions post-commit, caused too many data hazards. We also redesign our instruction set to better support high-throughput programming design patterns that minimize hazards further.

**ISA.** We connect the message queues to the fabric network in the mapper, allowing the  $\mu$ core to receive packets via an input queue and send packets through an output queue, in order to support pipelined parallelism strategies such as used in the shadow stack [21]. Our

Main core	
Core	4-Width, out-of-order SonicBOOM [26], @3.2GHz 128-Entry ROB, 96-Entry IQ, 32-entry LDQ/STQ, 128 Int/FP Phy Registers, 2 Int ALUs, 1 FP/Multi/Div ALU, 2 MEM, 1 Jump Unit, 1 CSR Unit
Pipeline	TAGE algorithm, 256-entry BTB, 32-entry RAS, 6 TAGE table with 2 - 64 bits history
Branch Predictor	
Memory	
L1 ICache	32KB, 8-way, 8 MSHRs
L1 DCache	32KB, 8-way, 8 MSHRs
L2 Cache	512KB, 8-way, 12 MSHRs
LLC	4MB, 8-way, 8 MSHRs
Memory	16 GB DDR3 @1066MHz, max 32 requests
FireGuard and Interconnects	
Event Filter	4-width, 16-entry FIFO
Mapper	4 SEs, 8-entry CDC, fabric @1.6GHz
Analysis Engine	In-order Rocket $\mu$ core [35], 5-stage pipeline, @1.6GHz, 32-entry message queues, no FPU
L1 Cache	4KB, 2-way for both I- and D-Cache
Interconnect	Memory bus @ 1GHz, others @ 3.2GHz

TABLE II: Hardware configurations evaluated.

queue controller, with status registers and software drivers, allows guardian kernels to manage the message queues using ISAX [35] custom instructions (see table I). Top, push and pop were inherited from the original design [21]; count was added to aid in loop unrolling, and recent to allow accessing extra information about an element already processed: for example, the PC is only needed on a detected error in AddressSanitizer, and is discarded otherwise.

**Microarchitecture Support.** RISC-V Rocket runs custom instructions post-commit [35]. The routing to the ISAX peripheral blocks the core for at least 3 cycles for each instruction and can extend up to 13 cycles in the presence of data hazards and contention [36]. This causes data hazards and large slowdowns when such instructions are as commonly used as our queue operations. We redesigned Rocket’s interface to move it into the MA stage<sup>6</sup> of the pipeline (figure 6), multiplexing between the ISAX unit and the load-store unit.

## IV. EVALUATION

**Experimental Setup** To comprehensively evaluate the feasibility of FireGuard, we built FireGuard into RISC-V cores. BOOM main cores were augmented with data-forwarding channels, filters, and mappers; Rocket  $\mu$ cores were configured as security engines, running different safeguards, i.e., kernels. We implemented the microarchitecture with

<sup>6</sup>The MA stage was chosen as it is the first stage in the pipeline that is non-speculative, simplifying the implementation of the state-destructive pop but otherwise minimizing hazards, such that only one bubble is required for an instruction immediately following the custom instruction that uses its output.

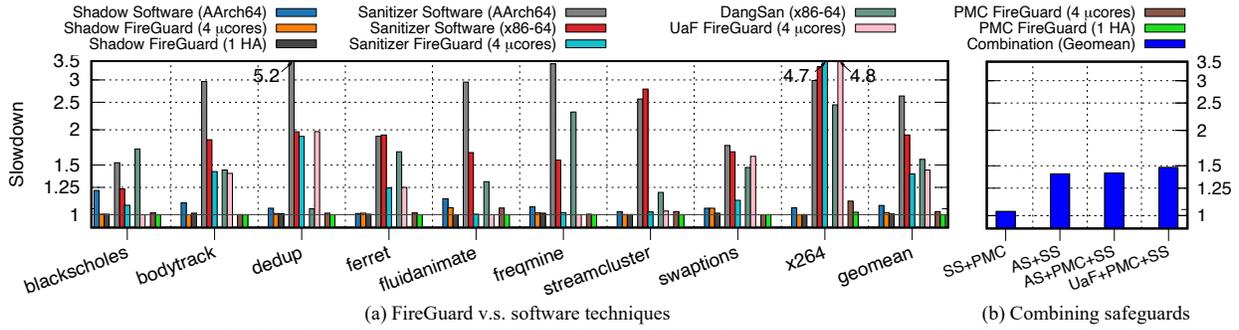


Fig. 7: Performance results for FireGuard (4  $\mu$ cores or 1 HA for each kernel) running Parsec with different/combined safeguards (AS: AddressSanitizer; SS: Shadow Stack; in (b), SS is implemented as a HA when three guardian kernels are deployed).

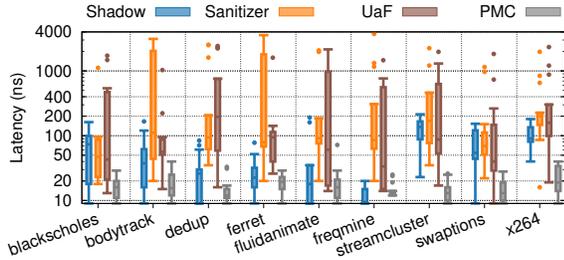


Fig. 8: Detection latency while using 4  $\mu$ cores (unit: ns).

Chisel (v3.4) and synthesized the RTL using Vivado toolchains (v2021.2). The generated netlist was deployed on Virtex UltraScale+ FPGAs using FireSim [27], [37], emulating the setup in table II.

We booted Linux (with kernel v5.7.0) and executed Parsec [38], running the simmedium dataset with kernels: Custom Performance Counter with bounds check (PMC) [21], AddressSanitizer [29], Use-after-Free detector (UaF)<sup>7</sup> and shadow stack [28], [39].

#### A. Performance Overhead

Figure 7(a) shows the slowdown experienced by the main core while executing kernels in FireGuard, in comparison to software-based counterparts (implemented via LLVM). In all instances, FireGuard is configured with four  $\mu$ cores. Hardware Accelerators (HAs) within FireGuard are also presented for PMC and shadow stack.

Using 4  $\mu$ cores is sufficient to run a PMC with overheads of 2.5% geomean, shadow stack at 2.1%, AddressSanitizer at 39%, or UaF at 42%. By replacing the  $\mu$ cores with a single HA, the overhead of PMC and shadow stack can be reduced to 0%. Software techniques give 7.9% overhead for shadow stack, and 163.5% for AArch64 and 91.5% for x86-64 architectures on AddressSanitizer<sup>8</sup>. FireGuard consistently outperforms software techniques, with the exception of x264 with AddressSanitizer, and Dedup on UaF. The former is due to the exceptionally high volume of load and store instructions executed in x264, where four  $\mu$ cores fail to keep up to such a degree that sharing compute resources with the compute core, as in software schemes, is favorable (though FireGuard can do better with more  $\mu$ cores, see section IV-D). The latter is caused by DangSan [40] being more suited to dedup’s allocation patterns than our UaF detector’s underlying Minesweeper [30], where DangSan itself on FireGuard would give false negatives, as it relies on zeroing pointers, which may occur only after the UaF access if offloaded to a  $\mu$ core.

**Combining kernels.** Figure 7(b) reports the slowdown experienced by the main core while executing multiple kernels simultaneously. The kernel that incurs the most slowdown dominates the performance, but slowdowns are not multiplied. This is benefited from the parallel execution of the kernels. When one kernel halts the execution of the

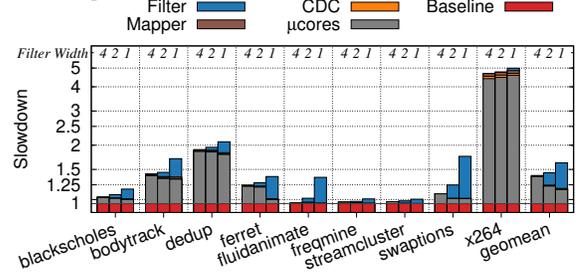


Fig. 9: Cumulative bottlenecks v.s. filter widths (on 4- $\mu$ core Sanitizer), measured by the proportion of time queues are full.

main core, decreasing the number of instructions, other kernels can persist in their function with reduced computational overhead.

#### B. Detection Latency

To examine the detection latency, we inject erroneous input at various locations in the core, e.g., the jump unit, the LDQ and the STQ, etc, simulating e.g., jump to a hijacked PC and access a freed memory address. For each workload, 50–100 attacks are generated, and the detection latencies are shown in figure 8. PMC experiences the lowest detection latency, consistently detecting attacks under 50 ns. Due to its complex block parallelism model [21], the shadow stack sees marginally higher latencies than PMC. In the worst-case scenario (x264), a latency of 220ns is observed. For AddressSanitizer, though the median is always <200ns, the tail exceeds 2,000ns. This occurs when TLB and cache misses co-occur on many accesses in the same queue: our accurate modeling of TLB misses in FireSim causes worst-case delays to be higher than in the original work [21].

#### C. Analysis of Microarchitecture Bottlenecks

Figure 9 decomposes overheads in a 4- $\mu$ core FireGuard when running AddressSanitizer using varying event-filter widths (1, 2, or 4), highlighting the accumulated bottlenecks stemming from different elements. A 4-width event filter is as wide as the core’s commit, so can always keep up. As the width decreases, overhead increases significantly, from 16% at 2-width to 34% at 1-width. The geomean overhead from the filter jumps from 16% (2-width) to 34% (1-width).

#### D. Analysis of Scalability

Figure 10 shows the slowdown with varying  $\mu$ core count. **PMC.** Figure 10(a) shows that using two  $\mu$ cores to perform PMC leads to a 20% geomean slowdown. Four  $\mu$ cores reduce this to 2%; yet, x264 still has a 17% slowdown. When scaling the  $\mu$ cores number to six, all workloads can be run with an overhead of less than 5%.

**Shadow stack.** Figure 10(b) shows that using two  $\mu$ cores for executing shadow stack leads to 7.3% slowdown (geomean), which is comparable to the performance of LLVM’s software mechanism. With four  $\mu$ cores, the geomean overhead is reduced to 2.1%, decreasing to 0.4% when using six  $\mu$ cores. With six  $\mu$ cores, all workloads can be executed with <1% slowdown, except fluidanimate (1.8%).

**AddressSanitizer.** Figure 10(c) is heavier than (a) or (b). Using two  $\mu$ cores results in 86% overhead, still outperforming software (section IV-A). However, three workloads (bodytrack, dedup, x264)

<sup>7</sup>Our design takes MineSweeper [30] and uses analysis engines to find loads and stores to quarantined regions, to find as well as prevent bugs.

<sup>8</sup>AddressSanitizer is not supported for RISC-V in either GCC or LLVM, so the comparison focuses on AArch64 and x86-64. For shadow stack, support is absent for both RISC-V and x86-64, so we compare against AArch64.

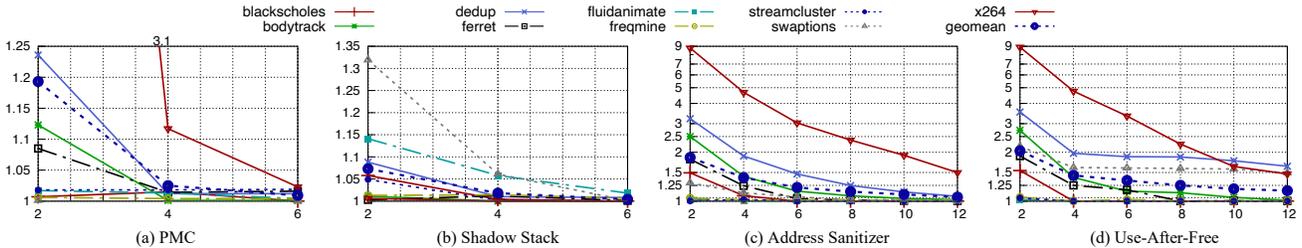


Fig. 10: Slowdown when using varying numbers of  $\mu$ cores for guardian kernels ( $x$ -axis: number of  $\mu$ cores;  $y$ -axis: slowdown).

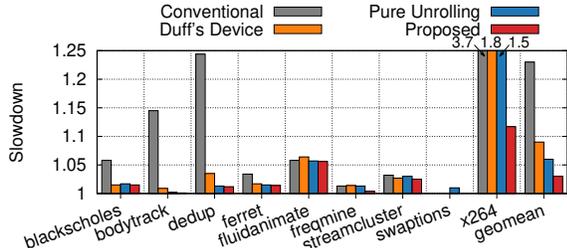


Fig. 11: Slowdown of programming models (4- $\mu$ core PMC).

exhibit over 100% overhead. This is due to the extremely high volume of load and store instructions that necessitate analysis. Even with 12  $\mu$ cores, x264 continues to endure a 58.9% slowdown, while all other workloads see negligible overhead (less than 1%).

**UaF.** Figure 10(d) is the heaviest workload. As well as monitoring all loads and stores, the extra work to quarantine and release memory allocations only when safe [30] causes some overheads that do not parallelize away; for example, dedup sees flat overheads due to extra memory-allocation work. The geomean reaches  $1.16\times$  by 12  $\mu$ cores, lower than the  $1.6\times$  that is achieved by software mechanisms [40].

#### E. Analysis of Programming Models

Figure 11 shows the performance overhead when executing PMC using different programming models (section III-D) with 4  $\mu$ cores. By being aware of the remaining hazards in the  $\mu$ cores, we can improve performance significantly. A conventional single-iteration loop suffers from frequent data hazards due to both queue `count` checks and `pop` instructions. Duff’s device [41] reduces hazards on size checks, made possible by the queue `count` instruction, and pure unrolling helps further, by removing hazards for `pop` instructions if the queue is relatively full. A hybrid strategy, using unrolling when possible and Duff’s device otherwise, is uniformly best.

#### F. Hardware Overhead

We study hardware overhead of the microarchitecture by performing a physical implementation of a 4- $\mu$ core FireGuard at the post-layout stage using Synopsys 14nm Generic PDKs. The RTL is synthesized with Design Compiler (v2022.12), and the netlist is placed and routed via IC Compiler 2 (v2022.12).

The area of the SoC is  $2.91mm^2$ , where the BOOM is  $1.107mm^2$  and each Rocket is  $0.061mm^2$ . The total area of FireGuard’s transport mechanisms is  $0.043mm^2$  (3.88% and 1.48% of the BOOM and the SoC): the filter occupies  $0.032mm^2$  and the mapper  $0.011mm^2$ . Building a 4- $\mu$ core FireGuard upon a BOOM thus needs  $0.287mm^2$ , i.e., 25.9% and 9.86% of the BOOM and the SoC.

Intuitively, this observation diverges from the goal of low-overhead design. But, this is only because the prototype BOOM cores are small compared with commercial designs, yet the Rocket cores are very large, with closed-source in-order 64-bit cores available using half the resources [17]. To provide more comprehensive insights, we present a detailed analysis below on real-world SoCs.

#### G. Feasibility Analysis

We compare *performance-area trade-offs* between BOOM and the OoO processors in commercial SoCs [42], [43], [45]–[47], [47], [47], [48], and examine the overhead associated while constructing

Processors (Performance Core only)				
Core (SoC)	BOOM [26]	FireStorm [42] (M1-Pro)	Cortex-A76 [43] (Kirin-960)	AlderLake-S [44] (i7-12700F)
Peak Freq.	3.2GHz	3.2GHz	2.8GHz	4.9GHz
Tech.	14nm	5nm	7nm	10nm
Area	$1.11mm^2$	$2.53mm^2$	$1.23mm^2$	$7.30mm^2$
@ 14nm	$1.11mm^2$	$22.55mm^2$	$3.61mm^2$	$22.63mm^2$
IPC	1.3	3.79	2.07	2.83
<b>Normalized Throughput</b>	<b>1</b>	<b>2.92</b>	<b>1.27</b>	<b>3.35</b>
FireGuard Elements				
Filter Width	4-way	8-way	4-way	6-way
# $\mu$ cores	4	12	5	13
<b>Overhead (% / Core)</b>	<b>0.29mm<sup>2</sup> (25.9%)</b>	<b>0.81mm<sup>2</sup> (3.6%)</b>	<b>0.35mm<sup>2</sup> (9.6%)</b>	<b>0.85mm<sup>2</sup> (3.8%)</b>
An Independent Kernel for All Cores				
<b>Overhead (% / SoC)</b>	<b>0.29mm<sup>2</sup> (9.86%)</b>	<b>6.10mm<sup>2</sup> (0.47%)</b>	<b>1.23mm<sup>2</sup> (0.57%)</b>	<b>6.67mm<sup>2</sup> (0.99%)</b>

TABLE III: Feasibility of FireGuard in commercial SoCs.

FireGuard upon these processors and SoCs. To do so, we first estimate the core sizes from die shots of each system [44]–[46], then normalize the area using 14 nm technology based on the density difference [49]. We then scale up the number of  $\mu$ cores to match the increased IPC and clock relative to BOOM (calculated using  $IPC \times peak\ frequency$ ), where the IPC is measured through single-thread Parsec execution. **CPU-level overhead.** Table III’s upper portion examines throughput and area between BOOM and performance cores in the SoCs; the middle portion estimates the overhead to support a kernel, while attaining the performance in section IV-A.

Modern OoO cores are significantly larger than BOOM. To achieve a linear increase in processor throughput, a superlinear increase in hardware overhead is required. FireStorm is  $2.92\times$  faster than BOOM, but each core consumes  $20.3\times$  the area. While building FireGuard with a bigger core would require extra  $\mu$ cores to keep up with the execution, the increase is only linear. Building FireGuard upon FireStorm, Cortex-A76, and AlderLake-S requires 12, 5, and 13  $\mu$ cores respectively, giving 3.6%, 9.6%, and 3.8% overhead/core. **SoC-level overhead.** To ensure global security, FireGuard elements must be equipped for all processors in the SoC. Hence, we apply the same analysis on all core types in the SoCs and report the overhead.

Table III’s bottom portion indicates that integrating a kernel in M1-Pro, Kirin-980, and i7-12700F leads to an overhead of less than 1%. This negligible impact makes implementing several kernels per-core practical. Moreover, the energy overhead would be even lower, since the majority of FireGuard operates within a low-frequency domain.

## V. CONCLUSION

We have presented FireGuard, a microarchitecture for fine-grained instruction analysis. To make the design practical for deployment, we presented a buffer-free data forwarding channel, a superscalar event filter and a broadcast-free mapper. Feasibility analysis shows FireGuard can be integrated into modern SoCs with less than 1% increase in area. In summary, comprehensive in-core analysis is practical and efficient to build into real cores and SoCs.

## VI. ACKNOWLEDGEMENT

We appreciate the anonymous reviewers for their helpful feedback. This work is supported by the National Key Research and Development Program (Grant No. 2024YFB4405600), the National Natural Science Foundation of China (Grant No. 62472086, 62204036) and the Basic Research Program of Jiangsu (Grants No. BK20243042).

## REFERENCES

- [1] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," vol. 50, no. 6. ACM New York, NY, USA, 2017, pp. 1–37.
- [2] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 457–468, 2014.
- [3] Zecar, in <https://zecar.com/reviews/>, 2023.
- [4] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive experimental analyses of automotive attack surfaces," in *20th USENIX security symposium (USENIX Security 11)*, 2011.
- [5] T. Wehbe, V. Mooney, and D. Keezer, "Hardware-based run-time code integrity in embedded devices," *Cryptography*, vol. 2, no. 3, p. 20, 2018.
- [6] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 627–638.
- [7] S. Bannister, "Memory tagging extension," 2019.
- [8] X. Vera, "Inside tiger lake: Intel's next generation mobile client cpu," in *Hot Chips Symposium*, 2020.
- [9] K. N. Khasawneh, M. Ozsoy, C. Donovanick, N. Abu-Ghazaleh, and D. Ponomarev, "Ensemble learning for low-level hardware-supported malware detection," in *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings*. Springer, 2015, pp. 3–25.
- [10] M. Milenković, A. Milenković, and E. Jovanov, "Hardware support for code integrity in embedded processors," in *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, 2005, pp. 55–65.
- [11] T. Nyman, G. Dessouky, S. Zeitouni, A. Lehtikoinen, A. Paverd, N. Asokan, and A.-R. Sadeghi, "Hardscope: Thwarting dop with hardware-assisted run-time scope enforcement," 2017.
- [12] D. Kuzhiyelil, P. Zieris, M. Kadar, S. Tverdyshev, and G. Fohler, "Towards transparent control-flow integrity in safety-critical systems," in *International Conference on Information Security*. Springer, 2020, pp. 290–311.
- [13] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic, "Hardbound: Architectural support for spatial safety of the c programming language," vol. 42, no. 2. ACM New York, NY, USA, 2008, pp. 103–114.
- [14] S. M. A. Zeinolabedin, J. Partzsch, and C. Mayr, "Real-time hardware implementation of arm coresight trace decoder," *IEEE Design & Test*, vol. 38, no. 1, pp. 69–77, 2020.
- [15] J. Zhang, B. Qi, Z. Qin, and G. Qu, "Hcic: Hardware-assisted control-flow integrity checking." IEEE, 2018.
- [16] M. Ozsoy, C. Donovanick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 651–661.
- [17] ARM BTI, in <https://developer.arm.com/documentation/100748/latest/Security-features-supported-in-Arm-Compiler-for-Embedded/>, 2023.
- [18] Intel LAM, <https://lpc.events/event/11/contributions/LAM-LPC-2021.pdf>, 2021.
- [19] Intel CET, 2023.
- [20] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "Hcfi: Hardware-enforced control-flow integrity," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, 2016, pp. 38–49.
- [21] S. Ainsworth and T. M. Jones, "The guardian council: Parallel programmable hardware security," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1277–1293.
- [22] S. Fytraki, E. Vlachos, O. Kocerberber, B. Falsafi, and B. Grot, "Fade: A programmable filtering accelerator for instruction-grain monitoring," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 108–119.
- [23] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh, "Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2010, pp. 137–148.
- [24] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, "Flexible hardware acceleration for instruction-grain program monitoring," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 585–598, 2017.
- [25] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 585–598, 2017.
- [26] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*, vol. 5, 2020, pp. 1–7.
- [27] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra *et al.*, "Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 29–42.
- [28] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.
- [29] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A fast address sanity checker," in *2012 USENIX annual technical conference (USENIX ATC 12)*, 2012, pp. 309–318.
- [30] M. Erdős, S. Ainsworth, and T. M. Jones, "Minesweeper: a "clean sweep" for drop-in use-after-free prevention," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 212–225.
- [31] A. Ankur, A. Pandya, A. Abu, and L. Young-Uhg, "Noc architecture design methodology," *Journal of the Korea Institute of Information and Communication Engineering*, vol. 10, no. 1, pp. 57–64, 2006.
- [32] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann *et al.*, "T-crest: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.
- [33] Z. Jiang, N. C. Audsley, and P. Dong, "Bluevisor: A scalable real-time hardware hypervisor for many-core embedded systems," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2018, pp. 75–84.
- [34] Z. Jiang, K. Yang, N. Audsley, N. Fisher, W. Shi, and Z. Dong, "Bluescale: a scalable memory architecture for predictable real-time computing on highly integrated socs," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 1261–1266.
- [35] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, pp. 6–2, 2016.
- [36] P. Davide, "Design and programming of a coprocessor for a risc-v architecture," Master's thesis, Politecnico di Torino, 2017.
- [37] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.
- [38] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.
- [39] LLVM, 2023.
- [40] E. Van Der Kouwe, V. Nigade, and C. Giuffrida, "Dangsan: Scalable use-after-free detection," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 405–419.
- [41] T. Duff, "Duff's device," 1988.
- [42] D. Johnson, "Firestorm pipeline overview," in <https://dougallj.github.io/applecpu/firestorm.html>, 2022.
- [43] ARM, "Cortex-a76," in <https://developer.arm.com/Processors/Cortex-A76>, 2018.
- [44] Intel, "Alder lake: Microarchitecture," in [https://en.wikichip.org/wiki/intel/microarchitectures/alder\\_lake](https://en.wikichip.org/wiki/intel/microarchitectures/alder_lake), 2022.
- [45] "Apple m1 pro die shot," in <https://www.anandtech.com/show/17019/apple-announced-m1-pro-m1-max-giant-new-socs-with-allout-performance>, 2021.
- [46] "Kirin 980 die shot," in <https://www.anandtech.com/show/13564/chiprebel-releases-kirin-980-die-shot-cortex-a76s-mali-g76-in-view>, 2018.
- [47] E. Rotem, A. Yoaz, L. Rappoport, S. J. Robinson, J. Y. Mandelblat, A. Gihon, E. Weissmann, R. Chabukswar *et al.*, "Intel alder lake cpu architectures," *IEEE Micro*, vol. 42, no. 3, pp. 13–19, 2022.
- [48] Intel, "Alder lake: Specification," in <https://www.intel.co.uk/content/www/uk/en/products/platforms/details/alder-lake-s.html>, 2022.
- [49] K. Kumawat, "7nm vs 10nm vs 14nm: Fabrication process," in <https://www.techcenturion.com/7nm-10nm-14nm-fabrication>, 2019.