RT-LIFE: Portable RISC-V Interface for Real-Time Lightweight Security Enforcement

Christoph Spang[0000−0003−1606−4474], Florian Meisel[0000−0002−4671−9526], and Andreas Koch[0000−0002−1164−3082]

Embedded Systems and Applications Group, TU Darmstadt, Germany
{spang✉, koch}@esa.tu-darmstadt.de
https://www.esa.informatik.tu-darmstadt.de

Abstract. With the ever-expanding attack surface of low-cost processors in IoT applications, the interest in lightweight hardware support for improving their security is growing. While industry has already adopted mostly static low-overhead mitigation approaches against code-*injection* attacks, the race against code-*reuse* attacks is not yet over. One commonly proposed measure against code-reuse attacks aims to enforce runtime-dynamic integrity. In contrast to runtime-dynamic remote attestation, which is limited by its periodic attestation interval (possibly hours or weeks), runtime-dynamic integrity enforcement performs runtime-integrity checking in parallel to the actual execution. This allows very short attack response times, ideally stopping all evil instructions in flight from actually taking effect. To guarantee a prevention-in-time, one requirement is a low latency trace of uncommitted instructions. This typically would require a deep and core-specific integration. As an abstraction layer, we present our highly portable Real-Time Lightweight Integrity enForcement intErface (RT-LIFE), which is optimized to provide the core's state (uncommitted instructions) to an arbitrary runtime-dynamic low-latency Security Enforcement Unit (SecEU) as early as possible, while minimizing the interface's area and clock frequency penalties. We demonstrate RT-LIFE for six very different RISC-V cores together with our initial control-flow-integrity-enforcing SecEU DExIE, discuss the hardware architecture and its timing in detail, and finally provide an open-source release of RT-LIFE.

Keywords: Hardware Security · Security Monitoring · Portable Uncommitted Instruction Tracing · Runtime-Dynamic Integrity · Real-Time · IoT · RISC-V · Attack Prevention · Code-Reuse Attacks · Open-Source

1 Introduction

General-purpose processors are vulnerable to different types of runtime attacks. One sub-class of these are sophisticated and at the same time practical code-reuse attacks, which cannot be mitigated by traditional techniques such as read-only memory, Write ⊕ Execute, or Address Space Layout Randomization [28]. Code-reuse attacks do not inject malicious code, but execute existing code gadgets in a

sequence not intended by the developer. This includes Return-into-libc, Return- and Jump-Oriented Programming (RoP, JoP) Control Flow attacks [1, 10, 30].

Without injecting new code instructions, Return-into-libc attacks exploit memory errors such as buffer overflows to replace the return address on a call stack to target on another subroutine [10]. RoP attacks extend this concept, and collect a potentially large number of code snippets, which are then concatenated and executed in an unintended order via manipulated return addresses [30]. Besides memory-safe programming languages, which trade performance for security [26], Return-into-libc and RoP attacks can be mitigated by storing a duplicate of the original return address on a shadow call stack, to be validated at return time [5, 8, 24, 31].

JoP attacks further extend the concept of RoP, but place their dispatcher gadget in heap memory [1]. By manipulating forward edges, JoP attacks bypass RoP countermeasures such as a shadow call stack. Beyond software solutions [34], one common mitigation approach is using a hardware monitor to safeguard inter- and intra-function Control Flow Integrity at runtime [8,22,25,29,33]. Depending on the attacker, this includes direct and indirect Control Flow (CF) [18, 21, 27].

Such a hardware monitor's [3] main functionality can be either deeply integrated into a core's pipeline [6, 7, 25], or in an on-chip module [8, 24, 33], or in a separate off-chip device, e.g. connected via a debug interface [4, 19].

In-pipeline monitors offer low-latency, but impose invasive changes to the pipeline, caches, memory, and executable binary [6, 7, 25]. On-chip and off-chip solutions are typically trace-based, and require only minimally-invasive changes (signal taps, stall, reset, interrupt). Off-chip monitors leave the entire SoC unchanged, but suffer from limited transmission data rates, data drops, and longer latency [4, 19]. Despite their tighter integration, many on-chip monitors cannot fully achieve short detection latencies. E.g. PHMon [8] evaluates only *fully-committed* instructions, which potentially cannot be reverted or aborted after detection. Additionally, PHMon relies on queues (2048 entries) and has multiple stages itself, thus no short detection is possible. This is potentially insecure, as an attack's impact may occur earlier than the monitor's delayed reaction, thereby circumventing the attack prevention capability. PHMon's design choice is a trade-off between low-invasiveness (tapping only the final stage) and performance (high F_{max}, no stalls) at the cost of latency-related security. However, PHMon gives an example (Heartbleed) where its detection latency is sufficient and network information leakage still *can* be prevented.

As an alternative, our prior work Dynamic Execution and Integrity Engine (DExIE) is an on-chip real-time SecEU for global and local control flow integrity enforcement [32] that is capable of stopping ongoing attacks early within short and guaranteed latency. To fulfill this guarantee, it must be supplied with a low-latency trace of early uncommitted instructions.

This trace is realized by RT-LIFE (in this work and in [32]). As a security monitoring interface built for attack prevention, RT-LIFE is the first portable interface providing an attached Security Enforcement Unit (SecEU) such as DExIE sufficient time to reliably make its decision **(Decision Latency, DL)** to actu-

ally prevent illegal instructions from having any externally visible effects (Fig. 1). To this end, RT-LIFE retrieves the relevant signals of *uncommitted instructions* from the pipeline as early as possible and forwards them with low **Capture Latency (CL)** to the SecEU. The feedback loop (CL+DL) including the SecEU should be faster than the processor. No or only very few extra stall cycles are introduced into the regular pipeline, when the SecEU is operating (Sec. 8). If a SecEU introduces stall cycles, they can be fully-predictable at compile time, as this allows the tight Worst Case Execution Time (WCET) computations that are crucial for real-time applications.

After focusing on existing related interfaces (Sec. 2), Section 3 introduces RT-LIFE's security model and timings. In order to discuss requirements, Section 4 presents a case-study using our SecEU (DExIE). Section 5 explains RT-LIFE's behavior. The next section sets the design considerations (Sec. 6) which are followed by the concrete RT-LIFE implementations (Sec. 7). The final sections contain the evaluation (Sec. 8) and conclusion (Sec. 9).

Key contributions:

- Whereas existing portable tracing interfaces forward only fully-committed instructions, RT-LIFE reduces the latency by tracing *uncommitted* instructions in early pipeline states. Ideally, and also depending on the attached SecEU, this would allow to catch any malicious instruction in flight (and prevent it from being committed) without any extra stall cycles.
- We re-use our prior work DExIE to provide a practical use-case for RT-LIFE. As a trade-off between latency-related security, performance and portability, DExIE guarantees to stop any illegal CF in time and before any (directly) subsequent malicious and potentially irreversible Memory-Mapped I/O (MMIO) write access will be committed (take effect).
- We explain our hardware architecture and the given RISC-V cores in detail to facilitate reproducibility. We also publish our work in an open source repository [9]. This is an initial step to flexibly combine a variety of future attack-prevention SecEUs with different cores.

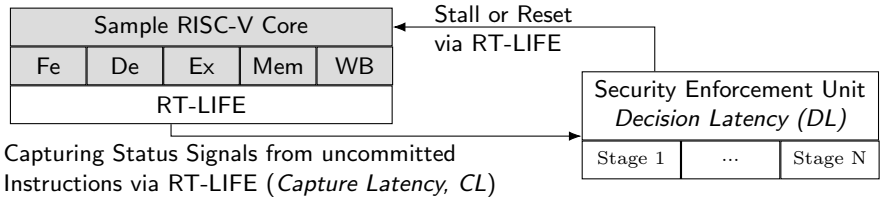


Fig. 1: Feedback loop for a generic RT-LIFE-enhanced *RISC-V* core and a generic SecEU. If the loop's accumulated latency (CL+DL) is *longer* than the core's latency to fully execute the first harmful instruction, stalls can be issued to halt the core, increasing the time interval available for detection and thus preventing an attack in time.

2 Related Work

We differentiate between open-loop *monitoring* for debugging and tracing purposes, and closed-loop security *enforcement* for attack prevention. Whereas the latter requires tight timing, monitoring does not. All interfaces discussed below are designed for monitoring only. As both use-cases require similar signals, we discuss which available standard RISC-V interfaces could be suitable for enforcement as well.

The **RISC-V Formal Interface (RVFI)** [14] contains a number of signals intended for formal verification. We used RVFI for an early draft monitoring unit. However, the RVFI focuses on *retired instructions* and only provides one global valid (*rvfi_valid*) signal for all captured data. Instead of directly forwarding captured data, earlier stages's signal values need to be delayed until the instruction reaches the pipeline's final stage. An earlier capture would require individual valid signals, which are not covered by the standard. Thus enforcement and attack prevention of an instruction is impossible. Stalling any pipeline stage would not solve the issue, as it equally delays an instruction and its CL, thus cannot increase the available DL. With RFVI, an attack would only be detected *after* ocurring, but could not be prevented in time. Additionally, RVFI does not integrate stall control signals, which are, under some conditions, necessary to fulfill our security guarantees.

The **RISC-V Debug Specification** [13] describes debugging interfaces for RISC-V processors. The execution of a Hardware Thread (HART) can be paused via explicit breakpoint instructions or debugger triggers. The debugger then has access to the state of the HART, including the Program Counter (PC) and registers. The interface would support our intended security guarantees for enforcement. However, continuous single-stepping would be necessary for capturing the instruction before execution, resulting in a massive performance drop. Therefore, we do not see this as a reasonable choice for SecEUs, except for very lightweight applications *without* real-time requirements.

In contrast to the RISC-V Debug Specification, the **RISC-V Trace Specification** [15] is designed for execution tracing without the externally induced stalls of single-stepping. However, compared to RT-LIFE, it focuses exclusively on CF. The specification differentiates between the *HART to Encoder Interface* and the Encoder's output, namely the *Branch Trace Interface*. For compatibility with off-chip debug units, the bandwidth must be reduced. Thus, the **Branch Trace Interface** [15] focuses on *retired* instruction blocks and uses compactly encoded packets. Both decisions lead to longer CL, making it incompatible for our attack prevention. The lower-level **HART To Encoder Interface** [15] also focuses on retired instruction blocks. Again, this interface can be used for CF *monitoring* without tight timing requirements, but is unsuitable for *enforcement*.

Although one could use other interfaces for monitoring, there is no other option, which forwards *uncommitted* instructions. Hence, none of the existing interfaces is suitable for analyzing and eventually stopping the currently ongoing (possibly malicious) instruction before it will be committed.

3 Fundamentals

3.1 RT-LIFE's Security Model

Attacker model: Whereas the specific *Threat Model* defended against depends on the attached SecEU unit, RT-LIFE is designed to thwart an attacker who has access to the core's state and can arbitrarily alter CF, register write instructions, and memory store instructions [1, 10, 21, 30].
Guarantees: With constant and short CL, RT-LIFE provides the core's current state early and thus allows a long DL to the SecEU attached to the core. RT-LIFE's signals provide support for trace-based SecEUs [3] that require the core's current status. This includes SecEUs that support a broad range of security policies, including Control Flow (CF), Data Flow (DF), Memory Security, and Value Invariant Enforcement [3, 8]. With the information captured by RT-LIFE, attached SecEUs can guarantee to *prevent* illegal instructions from execution, often without incurring additional pipeline stalls.
Assumptions: RT-LIFE is intended to support mitigating code-reuse attacks, it thus monitors the dynamic execution of instructions in the core. As we assume read-only memory (enforced via a Memory Protection Unit - MPU, or static partitioning), RT-LIFE does not perform static (memory) integrity attestation (against code injection attacks).

3.2 Decision Latency for CF, DF and Memory Attack Prevention

To our current knowledge, all existing interfaces (Sec. 2) have CLs that are too long, leading to the remaining DL between capture and an attack with real-world impact to be too short to make a decision (Fig. 1). Stalling each instruction to meet DL requirements is possible, but this would slow down code execution. Instead, we optimize the interface as well as the SecEU for a *reduced* latency, where additional stalls are avoided, and will only be introduced for handling edge cases.

We define a **successful attack** by its immediate real-world impact, which can be caused by **(A)** MMIO write instructions, or **(B)** tampering with Control and Status Registers (CSR). These attacks should be stopped before they take effect. Other scenarios without immediate real-world impact, e.g., combinatorial General Purpose Register (GPR) writes, are categorized as less harmful, and can be safely stopped just *after* the manipulation occurred. In closer detail, four latency guarantees, grouped by **MMIO (A)** and **CSR (B)** tampering, have been implemented:

(A1) If a manipulated CF Instruction (CFI) is followed by a memory instruction potentially writing to an attached MMIO device, RT-LIFE guarantees to provide a DL of at least one clock cycle to the SecEU to make its decision. **(B1)** If a manipulated CFI is followed by a malicious CSR register write, RT-LIFE guarantees to provide a DL of one cycle to the SecEU. **(A2)** For a malicious memory write access, RT-LIFE can guarantee to capture its value and address, such that a SecEU can combinatorially (DL=0) decide and prevent the write

from taking effect. **(B2)** In the RISC-V ISA, a CSR cannot be directly written. Instead, its new value is *moved* from a GPR. Therefore, we are focusing on GPR integrity. At the latest safe moment, RT-LIFE allows stopping code execution directly *after* a GPR is maliciously written, with a guaranteed combinatorial DL. But this is still sufficiently early to *prevent* a subsequent CSR write from actually taking effect.

4 DExIE - A sample Security Enforcement Unit

Before further elaborating on RT-LIFE's details, this section introduces our sample implementation of a SecEU, which itself is called DExIE - Dynamic Execution Integrity Engine [32]. It requires RT-LIFE's low latency, and already uses the CF-focused subset of RT-LIFE's functionality for a low-overhead fine-grained Control Flow Enforcement. A forward edge in a Control Flow Graph's (CFG) corresponds either to a jump, branch or call instruction. Backward edges always correspond to return instructions. DExIE enforces forward edges via auto-generated CFG- or profiling-based (for increased granularity) Enforcement FSMs (EFSM). In contrast, backward edges are safeguarded by an EFSM-state-agnostic Shadow Stack. Each subroutine corresponds to one EFSM at a time. Branches and jumps correspond to the current function's EFSM-internal transitions. For calls and returns, that function's EFSM becomes active. Per call, the Shadow Stack holds return address, return EFSM, and return EFSM state.

After discussing DExIE's security model, which employs a subset of RT-LIFE's features, the DExIE tool-chain and architecture are explained, and the realization of security guarantees as well as DExIE's behavior is presented.

4.1 DExIE's Security Model

Threat Model: DExIE is fitted for (industrial) real-time IoT devices with MMIO peripherals. The device's firmware includes memory unsafe languages such as C with possible vulnerabilities that are (remotely) exploitable.
Attacker model: The attacker (in)directly and arbitrarily tampers with control flow instructions [1, 10, 30].
Guarantees: For any illegal CFI, DExIE immediately resets the core, thus prevents it from executing any subsequent memory write instruction, which might have a potentially irreversible real-world impact. As EFSMs are stored and protected in on-chip SRAM and no caching is used, DExIE guarantees to react in *constant* time. DExIE and RT-LIFE operate faster than the attached core can fully execute a memory write instruction following an illegal CF.
Assumptions: By exploiting a software weakness (e.g., a huge overflow), an attacker could potentially overwrite a function's code with new instructions which do not include *any* CF, or have *identical* CF, and thus would not violate any EFSM imposed by DExIE. Therefore, we assume read-only program memory (e.g., enforced via a MPU).

4.2 DExIE's Fundamentals

Figure 2, shows DExIE's key idea. First, the sample application C-Code (a) is compiled into RISC-V assembly code (b). The DExIE [32] compiler reconstructs the program structure, to build and interconnect the CFG-based function-individual EFSMs (c), that are actually being used for enforcement [2].

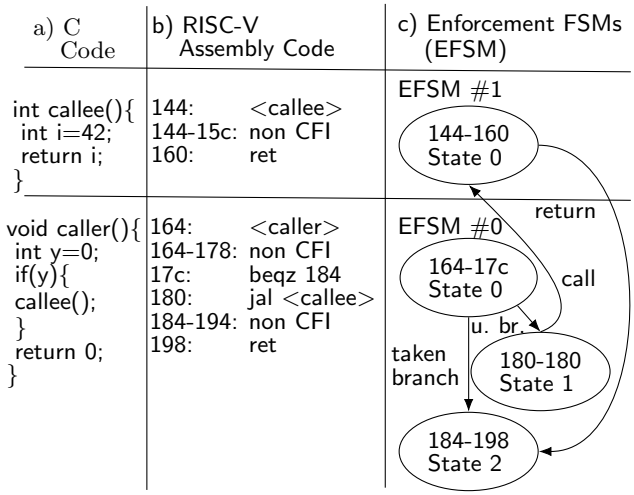
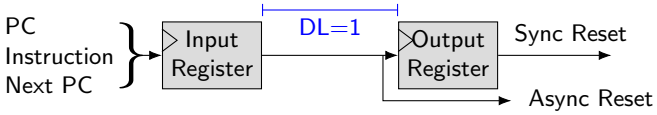


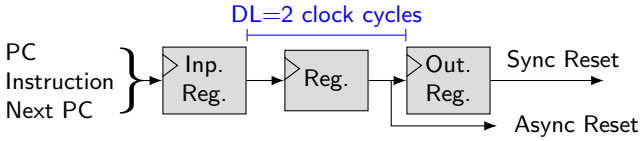
Fig. 2: A standard compiler compiles C-code (a) into Assembly code (b), which gets automatically converted into interconnected enforcement FSMs (c)

4.3 DExIE's Behavior and Interface Requirements

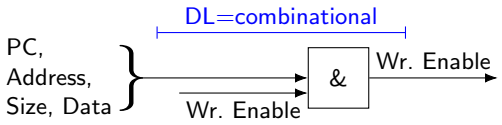
CF monitoring is limited by the frequency of CFI in the executable. We call the number of CFI per clock cycle the *CFRate*. DExIE's microarchitecture can cope with a CFRate of one CFI per cycle for calls and returns, and still achieves a single clock cycle of DL (Fig. 3a). For branches and jumps, the required DL is two cycles, thus dropping the maximum CFRate that can be handled without stalling to 1/2 (Fig. 3b). Stalls are only needed for chained branches and jumps in combination with a successful branch prediction. For DExIE, the ideal interface to the core would collect all required CF-related data (PC, Instruction, Next PC), write it into a register, and ideally leave two (or more) cycles of DL headroom. For memory writes, combinational comparators will validate values and addresses against DExIE's statefully loaded constraints (Fig. 3c). For GPR writes, DExIE will stop execution directly *after* the write occured (Fig.3d).



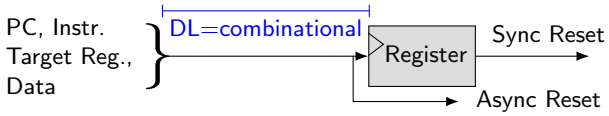
(a) DExIE's timing behaviour for *calls* and *returns*: After RT-LIFE's signal capture, which happens before the input register, DExIE needs one clock cycle of DL for its decision to reach the output register.



(b) DExIE's timing behaviour for *jumps* and *branches*: Two cycles of Decision Latency (DL) are required.



(c) DExIE's timing behaviour for memory writes: Combinational comparators for memory writes react within the same clock cycle.



(d) DExIE's timing behaviour for general purpose register (GPR) writes: Combinational logic stops a core directly after an illegal GPR write.

Fig. 3: DExIE's timing behaviour under different conditions

5 RT-LIFE: Signals and Behavior

After the discussion of DExIE as a sample SecEU to motivate the design of RT-LIFE, this section focuses on RT-LIFE's actual implementation.

Table 1 gives an overview of the signals used in RT-LIFE, grouped by their corresponding type of enforcement function. Columns (a) to (c) contain CF and DF signals, which are captured from the processor. Column (d) provides the control signals from SecEU to the core, closing the feedback loop.

In case the core processes a CF instruction (a), RT-LIFE provides the PC, instruction and the Next PC together with a valid signal. For memory store instructions (b), the instruction's PC, the target address, the access size, the data to be written, and a valid signal are captured. For a register write instruction (c), the interface provides the PC, the target register ID, and the corresponding data.

Table 1: RT-LIFE's signals

(a) CF	(b) Mem. Store	(c) Reg. Write	(d) SecEU control
To SecEU			From SecEU
Valid, PC, Instruction, Next PC	Valid, PC, Address, Size, Data	PC Target Register (0: invalid), Data	CF-Stall (CFS), Stall-on-Store (SoS), Continue-Store (CS) Reset

If a SecEU's decision latency is too long to prevent real-world impact of an instruction, it can request additional time by asserting the stall signals (d) CF-Stall (CFS), Stall-on-Store (SoS), and Continue-Store (CS). The CFS signal is used if a CF instruction decision takes too long, and the following instruction with potential real-world impact could not be stopped otherwise. In case the signal is set, the following instruction is to be stalled *before* it reaches the WB and MEM stages, gaining additional DL clock cycles for the SecEU. For memory writes, the SoS signal allows a SecEU to combinatorially validate the data to be written, and combinatorially stall a memory write operation before the validation is complete. To prevent combinatorial loops, which can be caused by RT-LIFE's constantly captured signals, the SecEU then asserts a *separate* combinatorial CS signal, if the write operation is deemed valid.

6 RT-LIFE Design and Behavior Considerations

For reduced logic overhead, RT-LIFE by default does not compute the next PC itself, but utilizes the core's computation. We decided against branch prediction awareness, as it would increase RT-LIFE's complexity and potentially degrade portability. Per group of signals (each column in Table 1), DExIE captures *all* signals in the same cycle, and thus would not benefit if only a subset of a group's signals were valid. Thus, we capture the signals as soon as *all* of them are valid.

7 RT-LIFE Implementation

To achieve portability for SecEUs and maintain compliance with the specified behaviour (Sec. 5), the microarchitecture of RT-LIFE is adapted individually to each core. We show six examples here for different cores.

With the exception of their pipeline depth, which is 3 and 5 stages respectively, **Piccolo** (Fig. 4) and **Flute** are closely related RISC-V cores [11]. Figure 4 shows Piccolo and draws vertical separation lines between its pipeline stages. As Flute adds additional separators between FE, DE and EX, and RT-LIFE only interacts with EX and later stages, the *same* RT-LIFE microarchitecture can be used for both cores.

VectorBlox' **Orca** (Fig. 5) is a 5-stage core. One interface difference to Piccolo and Flute is the possibility of an *unknown* Next PC in the EX stage, which

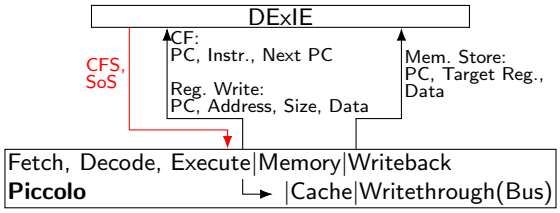


Fig. 4: RT-LIFE microarchitecture for the Piccolo RISC-V core. Vertical lines separate the three pipeline stages and the dedicated memory write stages (Cache, Writethrough(Bus)). Flute is similar, with FE, DE, EX being separated.

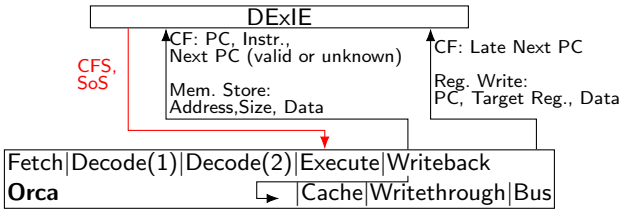


Fig. 5: RT-LIFE microarchitecture for Orca RISC-V core. Vertical lines separate pipeline stages. Note the increased memory write latency.

becomes only known later in the WB stage. Another difference is the additional clock cycle for memory write accesses after the Execute stage (Cache, Writethrough, Bus), increasing DL headrom for memory writes.

PicoRV32 (Fig. 6) [12] is a fast-clocked *non-pipelined* multicycle core, which already implements the RVFI. PicoRV32 uses an FSM to control the current instruction's execution. Figure 6 shows the control FSM extended with RT-LIFE. A CF-stall blocks *all* FSM transitions towards the `fetch` FSM state, which are also marked as red crosses in Figure 6. The SoS signal only stalls the `stmem` FSM state.

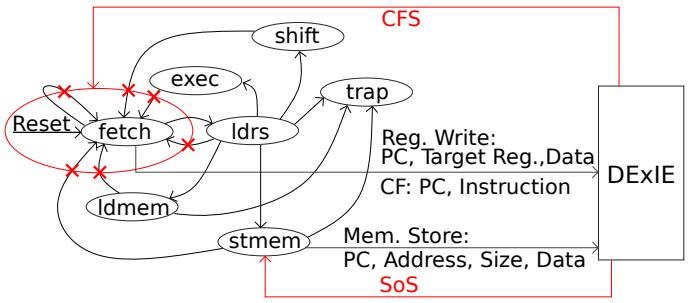


Fig. 6: RT-LIFE microarchitecture for PicoRV32 RISC-V core.

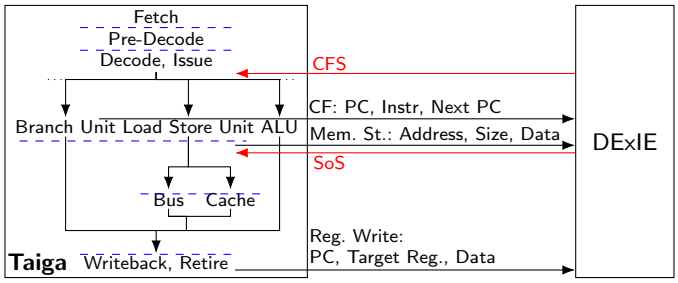


Fig. 7: RT-LIFE microarchitecture for Taiga RISC-V core. Dashed blue horizontal lines separate pipeline stages.

Taiga's (Fig. 7) [16] execution units work partly independently and in parallel. The elastic pipeline also causes instructions to reach the WB stage possibly out-of-order, and bypassing of values can happen even earlier. However, the final WB always happens in-order at retirement time. Our interface's SoS affects the Load Store Unit (LSU), which is only stalled if a subsequent *write* instruction enters the LSU. Compared to other cores with intermediary AXI busses, Taiga employs *directly* attached BRAMs, resulting in a shorter latency for memory accesses.

VexRiscv (Fig. 8) [17] is a modular core of adaptable pipeline depth with a plugin-based implementation. It supports RVFI via its FormalPlugin. We implemented a new plugin to externally stall the execute stage. The DBusSimplePlugin is extended to autonomously stall one cycle, if a CF instruction is *directly* followed by a memory write operation ($\mathrm{DL} = \mathrm{DL} + 1$).

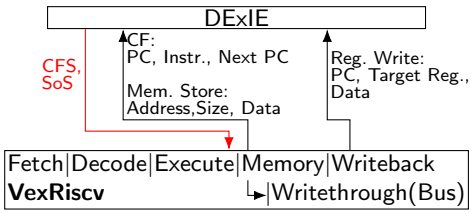


Fig. 8: RT-LIFE microarchitecture for VexRiscv RISC-V core. Vertical lines separate pipeline stages.

8 Evaluation

We implemented all designs as Processing Elements (PE) in the FPGA SoC framework Task Parallel System Composer (TaPaSCo) [20, 23] on the VC709 Xilinx Virtex 7 device prototyping board using Vivado 2018.3 (in this particular use case 2018.3 reached higher clock frequencies than more recent versions). This includes the original cores, the RT-LIFE-enabled cores, and the DExIE-monitored cores. Table 2 shows each core's RISC-V ISA type, the Hardware Description Language (HDL) used, and its number of pipeline stages. Regarding the interface, the table lists two decision latencies. First, it gives the number of clock cycles between a captured CFI and a subsequent *memory* store instruction taking effect. Second, it gives the latency between captured CFI signals and a subsequent *register* write instruction taking effect. The number of clock cycles can be seen directly in the core diagrams, except for PicoRV32 (Sec. 7). For all cores behavior is constant and identical (therefore not shown in the table) for plain memory stores (combinatorial DL, can be blocked in time) and GPR writes (comb. DL, safe to stop directly *after* malicious GPR write).

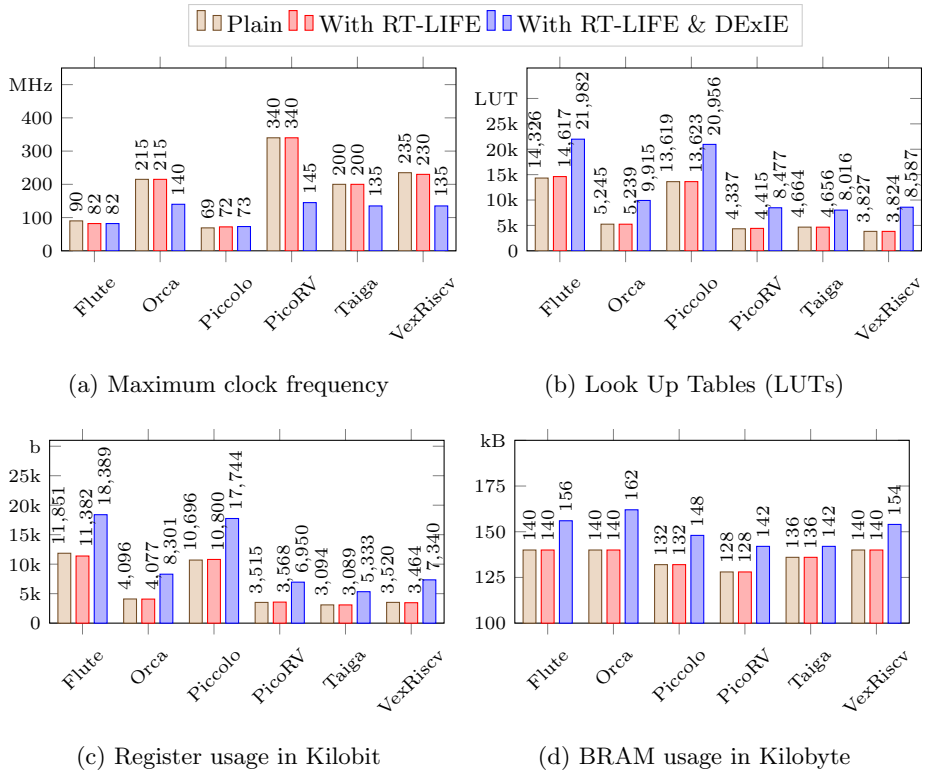
The following diagrams (Fig. 9a to 9d) show the f_{max}, LUTs, register and BRAM usage for all cores. By comparing each core's implementation against the corresponding RT-LIFE-augmented implementation, we show that RT-LIFE itself has *no* or only *minimal* overheads. In some cases, RT-LIFE seems to even improve the performance. This is an artifact and *unrelated* to RT-LIFE. It is caused by the Xilinx Vivado proprietary logic synthesis flow, which also includes heuristic algorithms, which may produce slightly better or worse results in different runs, even on the same design. Compared to the other cores, RT-LIFE shows somewhat higher overheads for PicoRV32 (due to our FSM modifications, see Fig. 6). Only when combined with a full-blown SecEU like DExIE do the overheads increase. This is expected, as enforcing fine-grained Control Flow Integrity within only 1-2 clock cycles is quite challenging. The critical path lies within DExIE for four out of the six cores. With DExIE being attached, the number of additional stalls introduced ranges from 0 % for the higher clocking, but longer latency PicoRV32, to 10.4 % for Taiga, which employs partially parallelized execution units. The wall-clock performance penalty with DExIE

Table 2: Characteristics and timing headroom for different RISC-V cores and CF scenarios

Core	ISA RV32	HDL	Pipeline Stages	Cycles betw. CF & subseq. memory store	Cycles betw. CF & subseq. reg. WB
Flute	ACIMU	BlueSpec	5	2	2
Orca	IM	VHDL	5	3	1
Piccolo	ACIMU	BlueSpec	3	2	2
PicoRV32	IM	Verilog	Multicycle Core	4	0
Taiga	IMA	SystemVerilog	3 (var.)	3	2
VexRiscv	IM	SpinalHDL	5	2	2

ranges from 0 % for Piccolo, to 134 % for PicoRV32. The latter is the worst-case scenario, as its f_{max} suffers most. For all of these tests, DExIE was configured identical to monitor the execution of Embench-IoT 0.5 draft benchmarks, namely Aha-Mont64, Edn, Matmult-Int, and Ud.

As we have described in our related work (Sec. 2), we are not aware of any other *portable* interface for tracing *uncommitted* instructions. Thus, we cannot directly compare RT-LIFE to any similar implementation. Also, attaching DExIE [32] to one of the many interfaces tracing *committed* instructions would be insecure, as the SecEU would no longer be able to stop evil instructions *before* taking effect.



(a) Maximum clock frequency

(b) Look Up Tables (LUTs)

(c) Register usage in Kilobit

(d) BRAM usage in Kilobyte

Fig. 9: Maximum frequency in MHz, number of Look Up Tables and Registers, BRAM in kB

9 Conclusion

To the best of our knowledge, RT-LIFE is the first approach for building a *portable* security monitoring interface, aiming for reduced *latency*, *guaranteed timing*, and *low overhead* that captures *uncommitted* instructions. We identified and demonstrated these attributes as key requirements for SecEUs with *guaranteed* attack prevention, with *no* or only limited performance overhead.

With its inter-core portability, RT-LIFE can ease future research in the area of real-time low-overhead SecEUs, with our SecEU DExIE serving as an initial use-case. Future work will further reduce DExIE's overhead, and add Data Flow and Invariant Enforcement to DExIE. The RT-LIFE specifications, the RT-LIFE-extended RISC-V cores, and a simple demonstration SecEU have been released as open-source [9].

Acknowledgement

This research work has been funded by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity *ATHENE.*

References

1. *Bletsch, T., Jiang, X., Freeh, V., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. pp. 30–40 (01 2011). https://doi.org/10.1145/1966913.1966919*
2. *Chen, et al.: Automated finite state machine extraction. In: Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation. FEAST'19, Association for Computing Machinery (2019). https://doi.org/10.1145/3338502.3359760*
3. *Clercq, R., Verbauwhede, I.: A survey of hardware-based control flow integrity (cfi) (06 2017)*
4. *Das, S., Zhang, W., Liu, Y.: A fine-grained control flow integrity approach against runtime memory attacks for embedded systems. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **24**(11), 3193–3207 (2016)*
5. *Davi, L., Koeberl, P., Sadeghi, A.: Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In: 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC). pp. 1–6 (2014)*
6. *Davi, L., Koeberl, P., Sadeghi, A.: Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In: 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC). pp. 1–6 (2014). https://doi.org/10.1109/DAC.2014.6881460*
7. *de Clercq, R., Götzfried, J., Übler, D., Maene, P., Verbauwhede, I.: Sofia: Software and control flow integrity architecture. Computers & Security **68**, 16 – 35 (2017), http://www.sciencedirect.com/science/article/pii/S0167404817300664*

8. *Delshadtehrani, L., Canakci, S., Zhou, B., Eldridge, S., Joshi, A., Egele, M.: Phmon: A programmable hardware monitor and its security use cases. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 807–824. USENIX Association (Aug 2020), http://usenix.org/conference/usenixsecurity20/presentation/delshadtehrani*
9. *Div.: Rt-life: An interface to easily attach integrity monitors to iot-class processors, https://github.com/esa-tu-darmstadt/RT-LIFE*
10. *Div.: Getting around non-executable stack (and fix) (1997), https://seclists.org/bugtraq/1997/Aug/63*
11. *Div.: Open-source risc-v cpus from bluespec, inc. (2020), https://github.com/bluespec/Flute*
12. *Div.: PicoRV32 - a size-optimized risc-v cpu (2020), https://github.com/cliffordwolf/picorv32*
13. *Div.: RISC-V debug specification (2020), https://github.com/riscv/riscv-debug-spec*
14. *Div.: RISC-V Formal verification framework (2020), https://github.com/SymbioticEDA/riscv-formal*
15. *Div.: RISC-V trace specification (2020), https://github.com/riscv/riscv-trace-spec*
16. *Div.: Taiga gitlab repository (2020), https://gitlab.com/sfu-rcl/Taiga*
17. *Div.: Vexriscv github repository (2020), https://github.com/SpinalHDL/VexRiscv*
18. *Evans, et al.: Control jujutsu: On the weaknesses of fine-grained control flow integrity. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. p. 901–913. CCS '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2810103.2813646*
19. *Guo, Z., Bhakta, R., Harris, I.G.: Control-flow checking for intrusion detection via a real-time debug interface. In: 2014 SMARTCOMP Workshops. IEEE Computer Society (2014). https://doi.org/10.1109/SMARTCOMP-W.2014.7046672, https://doi.ieeecomputersociety.org/10.1109/SMARTCOMP-W.2014.7046672*
20. *Heinz, C., Lavan, Y., Hofmann, J., Koch, A.: A Catalog and In-Hardware Evaluation of Open-Source Drop-In Compatible RISC-V Softcore Processors. In: IEEE Proc. International Conference on ReConFigurable Computing and FPGAs (ReConFig). IEEE (2019)*
21. *Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z.: Data-oriented programming: On the expressiveness of non-control data attacks. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 969–986 (2016)*
22. *Intel: Control-flow Enforcement Technology specification, rev. 3.0 (2020), https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf*
23. *Korinth, J., Hofmann, J., Heinz, C., Koch, A.: The tapasco open-source toolflow for the automated composition of task-based parallel reconfigurable computing systems. In: et al., H. (ed.) Applied Reconfigurable Computing. Springer International Publishing, Cham (2019)*
24. *Li, J., Chen, L., Xu, Q., et al.: Zipper stack: Shadow stacks without shadow. ArXiv (2019)*
25. *LI, Y., LI, J.w.: A technique preventing code reuse attacks based on risc processor. DEStech Transactions on Computer Science and Engineering (08 2018). https://doi.org/10.12783/dtcse/CCNT2018/24682*
26. *Nagarakatte, S.: Practical low-overhead enforcement of memory safety for c programs (01 2012)*

27. *Palmiero, C., Di Guglielmo, G., Lavagno, L., Carloni, L.P.: Design and implementation of a dynamic information flow tracking architecture to secure a risc-v core for iot applications. In: 2018 IEEE High Performance extreme Computing Conference (HPEC). pp. 1–7 (2018)*
28. *Prandini, M., Ramilli, M.: Return-Oriented Programming. IEEE Security & Privacy **10**(6), 84–87 (2012)*
29. *Qualcomm: Pointer Authentication on armv8.3 (2017), https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf*
30. *Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: Systems, languages, and applications. ACM Trans. Inf. Syst. Secur. **15**(1) (Mar 2012). https://doi.org/10.1145/2133375.2133377*
31. *Sinnadurai, S., Zhao, Q., Wong, W.F.: Transparent runtime shadow stack: Protection against malicious return address modifications (2008)*
32. *Spang, C., Lavan, Y., Hartmann, M., Meisel, F., Koch, A.: Dexie - an iot-class hardware monitor for real-time fine-grained control-flow integrity. In: Workshop on Design and Architectures for Signal and Image Processing (14th Edition). p. 26–34. DASIP '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3441110.3441146*
33. *Sullivan, G.T., DeHon, A., Milburn, S., Boling, E., Ciaffi, M., Rosenberg, J., Sutherland, A.: The dover inherently secure processor. In: 2017 IEEE International Symposium on Technologies for Homeland Security (HST). pp. 1–5 (2017)*
34. *Yuan, P., Zeng, Q., Ding, X.: Hardware-assisted fine-grained code-reuse attack detection. In: Bos, H., Monrose, F., Blanc, G. (eds.) Research in Attacks, Intrusions, and Defenses. pp. 66–85. Springer International Publishing, Cham (2015)*