Noname manuscript No.
(will be inserted by the editor)

DExIE - An IoT-Class Hardware Monitor for Real-Time Fine-Grained Control-Flow Integrity

Christoph Spang · Yannick Lavan · Marco Hartmann · Florian Meisel · Andreas Koch

Received: 04/2021 / Accepted: 12/2021

Abstract The Dynamic Execution Integrity Engine (DExIE) is a lightweight hardware monitor that can be flexibly attached to many IoT-class processor pipelines. It is guaranteed to catch both inter- and intra-function illegal control flows in time to prevent any illegal instructions from touching memory. The performance impact of attaching DExIE to a core depends on the concrete pipeline structure. In some especially suitable cases, extending a processor with DExIE will have *no* performance penalty. DExIE is real-time capable, as it causes no or only up to 10.4 % additional and then predictable pipeline stalls. Depending on the monitored processor's size and structure, DExIE is faster than software-based monitoring and often smaller than a separate guard processor. We present not just the hardware architecture, but also the automated programming flow, and discuss compact adaptable storage formats to hold fine-grained control flow information.

Keywords IoT Security · Runtime-Dynamic Integrity · Fine-Grained Control Flow Integrity · RISC-V · Code Reuse Attacks · Real Time

1 Introduction

Internet of Things (IoT) devices have become omnipresent. Due to their resource constrained nature, they often provide insufficient security, making them vulnerable to different categories of code-reuse runtime attacks such as Return- and Jump Oriented Programming (RoP, JoP) [1, 32]. We propose the Dynamic Execution Integrity Engine (DExIE), which defends against *both* of these kinds of attacks. With minimally invasive changes, DExIE can be easily attached to existing processor pipelines, which we demonstrate on four very different RISC-V cores. In most cases, it requires less area than the processor itself, thus making it more attractive than using a second core acting as a guard processor for the first one (which would incur 100% area overhead). The DExIE architecture is guaranteed to always catch illegal control flows *before* illegal instructions are able to affect memory (which could be disastrous in case of memory-mapped I/O devices). To do so, DExIE causes only a very limited number of additional pipeline stall cycles (we observed at most 10%) that are statically perfectly predictable. The resulting execution behavior leads to very tight Worst-Case Execution Time (WCET) computations and makes DExIE suitable for monitoring real-time capable systems. A key contribution of our work is the development of highly compact adaptable storage layouts for fine-grained inter and intra-function control flow information. Thus, even for small-scale SoCs, practically useful configurations of DExIE require just 4.4-12.1% more on-chip memory capacity than an unmonitored system. In addition to the hardware architecture, we also introduce a toolchain that can extract the Enforcement Finite State Machines (EFSMs), which lie at the heart of DExIE's monitoring, from generic ELF binaries. In most of the cases we examined, DExIE monitoring incurs a wall-clock execution time slowdown of just 1.5-1.75x, which is better than pure software-based approaches that often exceed 2x [6]. Some base processors are especially suitable for DExIE, in that they induce *neither* a clock frequency penalty, *nor* a wall-clock execution-time slowdown.

After extensively covering related work (Sec. 2), the basic mechanism and security considerations are presented in Sec. 3. Next, the software toolchain and the

Christoph Spang, Yannick Lavan, Marco Hartmann, Florian Meisel, Andreas Koch
E-mail: {spang, koch}@esa.tu-darmstadt.de
TU Darmstadt, Embedded Systems & Applications Group

transformation of code into DExIE EFSMs is discussed (Sec. 4.1). This is followed by our hardware design (Sec. 4.2), optimization (Sec. 4.2.3), and microarchitecture (Sec. 4.2.4). Finally, we report evaluation results (Sec. 5) and conclude by looking to further work (Sec. 6).

2 Related Work

We will first introduce code-reuse attacks and countermeasures in Sec. 2.1. The next subsections (Sec. 2.2 - 2.6) discuss fundamental design decisions. Also, we categorize DExIE's main design choices.

2.1 Attacks, Countermeasures, and Granularity

In case of JoP [1] and RoP [32] code-reuse attacks, an attacker first analyses an application to collate a potentially large collection of abusable code snippets (gadgets). After exploiting a program bug as entry point, these gadgets are executed in an order unintended by the original developers. This creates a malicious and under some conditions even Turing-complete exploit without relying on the modification of existing, or the insertion of new code. As discussed by [32], a traditional Address Space Layout Randomization or $W \oplus E$ (Write Xor Execute) cannot fully mitigate such temporal anomalies. As RoP attack gadgets are concatenated via return address manipulation, a well-known and effective mitigation is a shadow stack holding a duplicate [6] [28] or hash [22] of the return address. Manipulation is detected by comparing the valid copy to the core's computed, possibly manipulated, return address.

In contrast to RoP attacks, a JoP attack's dispatcher gadget can be located in the heap memory, thereby bypassing the shadow stack. To stop JoP attacks, verification of inter- and intra-function Control Flow Instructions (CFI) is an effective method that can be realized in different ways and different levels of detail, with DExIE being just one possible solution. Pure-Call oriented Programming (PCoP) is a similar attack, chaining code gadgets via manipulated call statements [33].

Dover Microsystem's "Inherently Secure Processor" implements a stateful word-based *tag-map* [36] for return address CF policies but, depending on other policies, also supports dataflow integrity. We expect overheads to grow with the implemented policies and that their caches are likely being unsuitable for real-time applications. In contrast, DExIE stores its constraints in constant-latency BRAM with fully-predictable stalls.

The ARMv8.3 ISA [30], e.g., used by Apple and Qualcomm, implements *Pointer Authentication* Codes (PACs) by repurposing formerly unused address bits. Each PAC's value is computed using a combination of target address, context, and a chosen key. This approach requires extra instructions, which we avoid in DExIE, and an adapted compiler toolchain. Only limited documentation exists on the associated overheads. By running a simulation, [24] shows 12 to 16 extra clock cycles being required for safeguarding a return instruction, and 6 to 8 clock cycles for an indirect call. DExIE only requires up to two clock cycles for checking legality of a CF instruction, and it runs mostly *parallel* to the core's execution. Depending on the desired level of integrity, PAC's overhead was simulated to range between 0.5 % to 39.5 % (including data pointer integrity). For this aspect, a direct comparison with DExIE is difficult, since the performance of DExIE depends on the specifics of the monitored core. It may thus be both slower or faster than the overheads observed for PACs.

Intel Control Enforcement Technology [20] deploys a combination of a shadow stack and *indirect branch tracking*. The latter is realized in hardware, checking that each indirect CF is followed by an `ENDBRANCH` instruction. An attacker could still manipulate an indirect CF to target any other (at some point-in-time legal) indirect target address, though. DExIE can solve this by enforcing a finer CF granularity, ideally allowing only a single legal target address.

Li et al. [23] combine tight instruction and memory tagging, and deploy the *Bell-LaPadula* confidentiality and the *Biba* integrity model. Their highly-specialized solution's memory overhead is around 3.13 %, and the logic, register, and mux overheads range between 9.01 % and 12.06 %, which are all smaller than DExIE's overheads. However, they do not discuss stalls, real-time capability and performance overheads. Their model also cannot fulfill our portability and granularity requirements (Sec. 2.4).

Runtime-dynamic remote attestation often implements a *hash* that changes either for any executed CF instruction, or which changes only per executed indirect CF instruction. After execution finished the resulting final hash value allows to verify that a specific CF path has been taken. Unfortunately, computation of hashes for all legal CF paths (potentially thousands) can be difficult and attacks are detected only late (*after* already being successful) at the next attestation event. Depending on the implementation, hash computations may have a significant performance cost and hardware overhead [7, 8, 26, 27, 40].

Safeguarding only indirect CF is reasonable for reducing overheads, but possibly insufficient, as [16] and [19] demonstrate attacks running on legal CFGs. On the other hand, legalizing only a single path may cause incompatibility especially for more complex or interactive applications. Therefore, and depending on the supplied EFSMs, DExIE supports *flexibly* going *beyond CFG-grade granularity.* This functionality is implemented via a One-to-N relationship between a function and its possibly many context-specific EFSMs. For any function call, and according to the caller's EFSM's state, this allows to *individually* constrain a called function to an alternative, possibly even tighter, EFSM. DExIE's hardware also provides an option for decoupling individual branch and jump CF instructions from their corresponding EFSM's state. With functions corresponding to alternative EFSMs, and CFIs corresponding to alternative states, this variable granularity allows to flexibly tighten the number of legal CF paths, ideally down to one, thereby mitigating *any* CF-attack, even one based on a single CF-deviation. In this work, we discusses DExIE in conjunction with auto-generated EFSMs at the CFG-level, that can optionally be annotated to reach finer granularity (Sec. 4.1).

2.2 Software-Hardware Partitioning

Runtime-dynamic integrity, implemented as a software-only solution, requires additional integrity-checking instructions, which can dramatically increase binary size and wall clock execution time. The actual overhead depends on the granularity of control flow integrity and can be higher than a factor of 2.5 x even for RoP-only mitigation [6]. Besides safeguarding fewer indirect CF instructions [38], extra hardware and power can be traded to limit wall clock-penalty. Running integrity-checking on a *second* dedicated core, called a “Guard Processor”, doubles the hardware area [39]. As an alternative, a specialized hardware unit can implement the desired functionality, thereby optimizing attack response latency, runtime-penalty, and hardware overhead.

2.3 Attack Response Latency: Monitoring vs. Low-Latency Enforcement

Fully-committed instructions can be monitored via a number of existing RISC-V specific interfaces [12–14] (Fig. 1). An attack's execution of evil instructions can be either just logged, or actually be prevented in time. For the latter and more secure approach, which is implemented by DExIE, monitoring already *fully-committed* instructions is not sufficient, as this would be too late to prevent in-flight instructions from being committed and taking effect [35]. Depending on the individual implementation, attack response latency varies between a few [34] or many thousand clock cycles [23], and can even be hours or weeks for remote attestation approaches (depending on the attestation interval) [7, 8, 26, 27, 40].

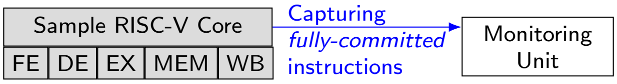


Fig. 1 One-way monitoring of fully-committed instructions. Monitoring can be used for debugging and tracing, but is not capable of preventing the most recent (evil) instructions from being (potentially irreversibly) executed.

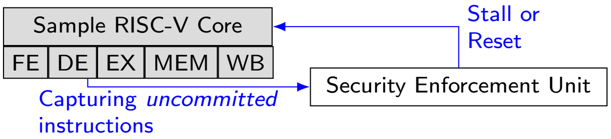


Fig. 2 A Security Enforcement Unit taps uncommitted instructions early from a RISC-V core's pipeline, and has only very limited time to react to prevent an evil instruction from being (potentially irreversibly) committed to memory (MEM) or the write-back (WB) stage.

To reduce attack response latency and ideally achieve *attack prevention* at the very first evil instruction in flight, capturing *uncommitted* instructions in an earlier pipeline stage is a *key requirement* (Fig. 2). With only a very limited time budget, a Security Enforcement Unit can then check the legality and *prevent* the illegal instruction from being fully-committed in a later pipeline stage. Statically or dynamically increasing the available time budget is not desirable, as it either affects attack response latency, or impacts performance, either by slowing the maximum clock frequency or by causing additional stall cycles [34, 35]. The alternative idea of tracing read accesses on the instruction memory bus could provide shorter latency, but is prone to extra traffic due to speculative accesses and pipeline flushes.

2.4 Location of an Integrity Enforcement Unit

Beyond software-only solutions [6], a hardware integrity unit can be either located within the core's pipeline [23], on-chip (without deeply affecting the pipeline) [34] or on a separate chip (Fig. 3).

Deep pipeline-specific integration yields best attack response latency but limits portability. Both the resulting increase of word width and new instructions necessitate deep changes of the existing design (memory word width, ALU, executable binary image, compiler toolchain [23]). This is sub-optimal for portable solutions such as DExIE that are aiming for compatibility with many different IoT cores.

Deeply integrated designs often store their constraints within a modified instruction memory. When loading constraints (e.g., memory tags), either together

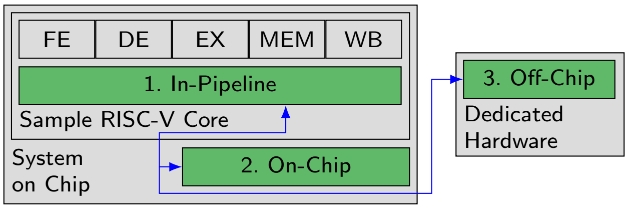


Fig. 3 Possible Architectural Locations: 1. In-Pipeline, 2. On-Chip, 3. Off-Chip. Deep integration limits inter-core portability. Transmission latency lengthens minimum attack response latency.

with the corresponding CF instruction or with its target instruction, the constraints required for checking only become available late during the instruction execution cycle [23] [30]. In contrast for each CFI the corresponding DExIE EFSM directly transitions to the next state and then waits there for the core to catch-up by executing straight-line code. As DExIE loads the new constraints at the activation time of the *previous* transition with only a single or double-cycle latency, even without cache, the number of stall cycles per CFI is ideally reduced to zero, see Sec. 4.2.4.

Compared to deeply integrated designs, on-chip out-of-pipeline monitors require only limited changes to the pipeline and instruction memory [34]. Ideally, existing debug/trace interfaces could be re-purposed for connecting them [12–14]. Since enforcement requires a low-latency trace of *uncommitted instructions*, which is not provided by any existing interface, we implemented such an interface for four different RISC-V cores [35]. Our interface taps the relevant status signals and provides a stall and a reset input. As DExIE's constraints are stored in separated BRAM memories, all of the cores presented in Sec. 5 can be safeguarded with a single common DExIE implementation.

Finally, off-chip monitors reach highest portability, as no changes to the System-On-Chip are made at all (Sec. 7 in [4]). However, the inter-chip communication comes with bandwidth and latency limitations. Therefore, these devices cannot be used to *prevent* instructions in flight from being executed, without imposing significant runtime overheads (stalls). Yet, when security timing requirements are relaxed, and without guaranteed timing, some longer-running network-based attacks might still be stopped in time before information secrets are leaked through the network [23].

2.5 Performance: Parallelism of Instruction Checking

Enforcement requires continuous integrity checking, which can be either realized serial or in parallel to the actual program execution (Fig. 4). The former requires program execution to be paused and the evaluation to run exclusively, eventually causing a high performance impact. Alternatively, a fully-parallel implementation will never cause the program execution to be stalled. However, we have shown that this may come at high hardware overhead or clock frequency reduction [34]. Finally, the hybrid approach runs in parallel most of the time, but *can* induce stalls. Depending on the implementation (e.g., avoidance of caches), the hybrid approach can still allow to exactly compute the Worst Case Execution Time (WCET), as required for real-time applications.

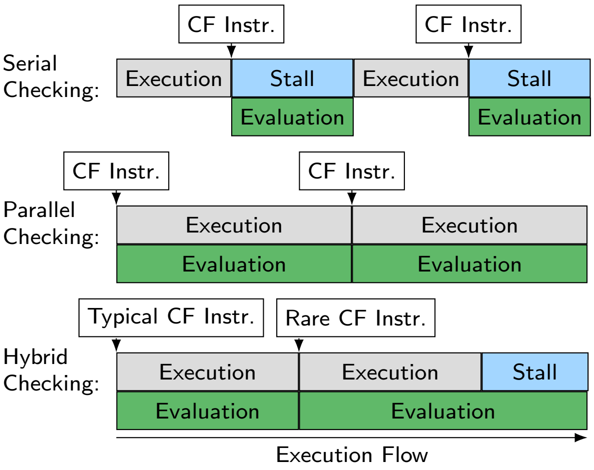


Fig. 4 Serial vs. parallel vs. hybrid integrity checking: Program execution is either stalled on a regular basis, not stalled at all, or only stalled for handling edge cases. Pipelining the processing of CF events would improve performance, but delay the attack responses, and thus is avoided.

Remote Attestation often performs *queuing* of control flow events. In this case, execution can temporarily overtake the monitoring unit for improved performance, but at the cost of a longer attack response latency. In their literature, remote attestation often does not state stalls during the process of monitoring (hash computation). However, program execution is typically paused at attestation time [7, 8]. In contrast, and depending on the supervised core, DExIE induces *no* or only few stalls, which are also statically-predictable, for handling edge cases.

2.6 Location of the Trust Anchor

A trust anchor is an authoritative entity (or a device component) for which trust is assumed and not derived [18]. Related work differentiates between device-local or remote trust anchors. The basis of a local trust anchor can be a trusted local memory, which cannot be manipulated during runtime by any (untrusted) application [34]. Alternatively, with *reduced local hardware* *overhead*, runtime-dynamic *remote* attestation allows to verify device integrity without local trust anchors [7, 8, 26, 27, 40]. Beyond continuous logging, runtime-dynamic remote attestation only allows sporadic integrity checking, thus increasing attack response times, and lacks code-reuse attack *prevention* guarantees.

3 Mechanism and Security Considerations

DExIE by default uses one CFG-based EFSM per function to statefully constrain an application function's legal CF. To show a practical code-to-EFSM-mapping example, Columns (a) and (f) in the later Figure 7 introduce the idea. Whereas inter-function control flows such as calls and returns switch the currently active EFSM, function-local control flows such as branches and jumps cause an EFSM-internal transition, thereby replicating the application's CFG. As we focus on single-core RISC-V implementations, exactly one pair of EFSM and software-function is active at any point in time (Fig. 5) with exactly one state being active. In the hardware, narrow bit-width numerical IDs are used to represent wider CF target addresses, different EFSMs, and their states. DExIE has been designed with the following in mind:

Attacker model: The attacker can (in)directly and arbitrarily tamper with control flow instructions [1] [32].

Guarantees: DExIE will react to any CFI violating the currently active EFSM. It will stop an attacker from targeting (calling, jumping to, branching to) a violating code gadget (address). Using a small number of predictable stalls, DExIE *guarantees* to react *faster* than the core can execute any subsequent illegal (Memory Mapped I/O - MMIO) memory write instruction, thus preventing an attack's potentially irreversible malicious outside world effects. As faster attack response times would require further stall cycles, the implemented and guaranteed attack response latency is a trade-off between security and performance, making DExIE especially suitable for safeguarding realtime-capable IoT devices that come attached to potentially irreversibly harmful MMIO periphery. For example, this could be a smart doorlock or a medical radiation device, which would quickly and automatically reboot and recover after a DExIE-induced device reset.

Assumptions: DExIE is designed with a focus on code-reuse attacks. An unprotected program memory would potentially allow an attacker to exploit a software weakness (e.g. a buffer overflow) for replacing a function with malicious code. If the malicious code had similar CF structure to the original, or lacked any CF at all, it potentially would not violate the active EFSM, and thus would be undetectable. Therefore, we assume the existence of a mechanism, e.g., a separate address space or a Memory Protection Unit, to enforce program memory being read-only for defending against code injections.

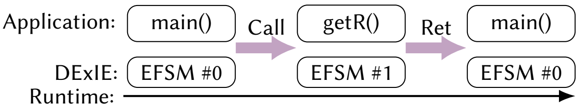


Fig. 5 Active pairs of (function, EFSM) over time

Real-time: DExIE is real-time capable. Specifically, this means that DExIE's runtime enforcement process always takes identical time for a given supervised instruction type. DExIE also introduces only very limited (if any) stall cycles, that are also statically predictable, compared to an unmonitored execution (Sec. 4.2.4, 5).

4 Implementation

We will first focus on DExIE's static analysis based software toolchain (Sec. 4.1) and later-on explain DExIE's hardware architecture for enforcing hierarchical EFSMs (Sec. 4.2).

4.1 Software Toolchain

Specifying DExIE's configurations by hand is possible but time-consuming. Thus, the goal of our software toolchain is the automatic construction of DExIE configurations for arbitrary software applications. A configuration contains a set of EFSMs and corresponding mappings from wide addresses to narrow IDs.

As an advantage, our toolflow (Fig. 6) uses an *unchanged conventional* RISC-V ISA compliant compiler to compile the program code into an Executable and Linking Format (ELF) object file. The object file is used twice: First, it is converted into a binary image to be executed by the RISC-V core using *objcopy*. Second, the object file is also fed into the DExIE-EFSM-Generator, which generates DExIE's EFSM and address-to-ID mapping configuration. We make use of the freely available Capstone disassembly library [9], which we utilize to reconstruct each function's DExIE Code Graph (DCG) from the CFG via *static analysis* (further details in Sec. 4.1.1). We use this somewhat indirect approach to ensure that the generated EFSMs match the actual machine instructions in the binary executable file. The easier approach of constructing the EFSMs, e.g., from the assembly-level instructions during compiler code generation, might be inaccurate, as later tools, such as the assembler or linker, could change the binary code again.

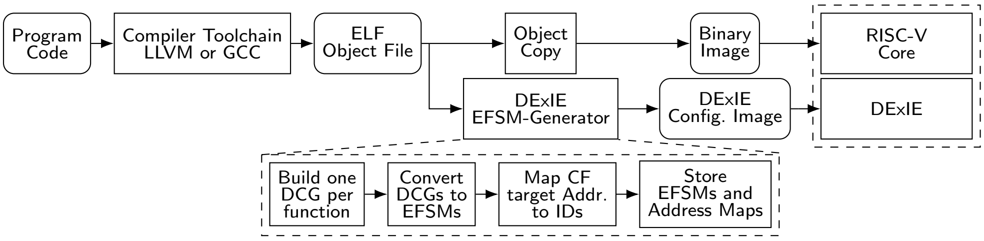


Fig. 6 Compilation into RISC-V and DExIE binaries: The DExIE-EFSM-Generator reads the ELF file, constructs the DExIE Code Graphs (DCGs) and EFSMs, performs the address-to-ID mapping, and writes the DExIE configuration image.

Using a later-explained set of transformation steps, the Capstone-generated DCGs get converted into EFSMs. Next, 32 bit addresses are mapped to narrow IDs, reducing DExIE's memory overhead. Lastly, EFSMs and mapping IDs are converted into a dense encoding and stored as a DExIE configuration image.

Statically disambiguating legal target addresses of indirect control flows (e.g., jumps via runtime-dynamic register) can be difficult. Manual code annotations or hand annotated assembly can resolve this uncertainty (Listing 1). Additionally, DExIE's hardware is not only compatible with EFSMs from static analysis similar to [3], but already supports runtime-profiling-based EFSMs generated using the Spike simulator [15], thereby also covering the profiled indirect CFs. As a benefit, profiling EFSMs ideally specify a single CF path, thereby mitigating any CF-deviating attacks. Symbolic execution is another possible source for EFSMs [2] [29].

```
int a(){[...]} //Not to be called
int b(){[...]} //To be called
int main() {
    typedef int (*fp_t)();
    fp_t arr[2] = {&a, &b};
    int a = 1;
calling_b: //DExIE annotation as C label
    return (*arr[a])(); //The indirect call
}
```

Listing 1 Code example for an indirect control flow: Using compiler optimization and the Capstone framework [9], our toolchain will identify the function (b), which is to be called in this example. Additionally, the optional comma-separable label also specifies that only function (b) will be legally callable.

4.1.1 Toolchain Details - Creation of EFSMs

Figure 7 shows an example code-to-EFSM-transformation containing two functions `main()` and `getR()`. Each column contains an individual toolflow stage's result.

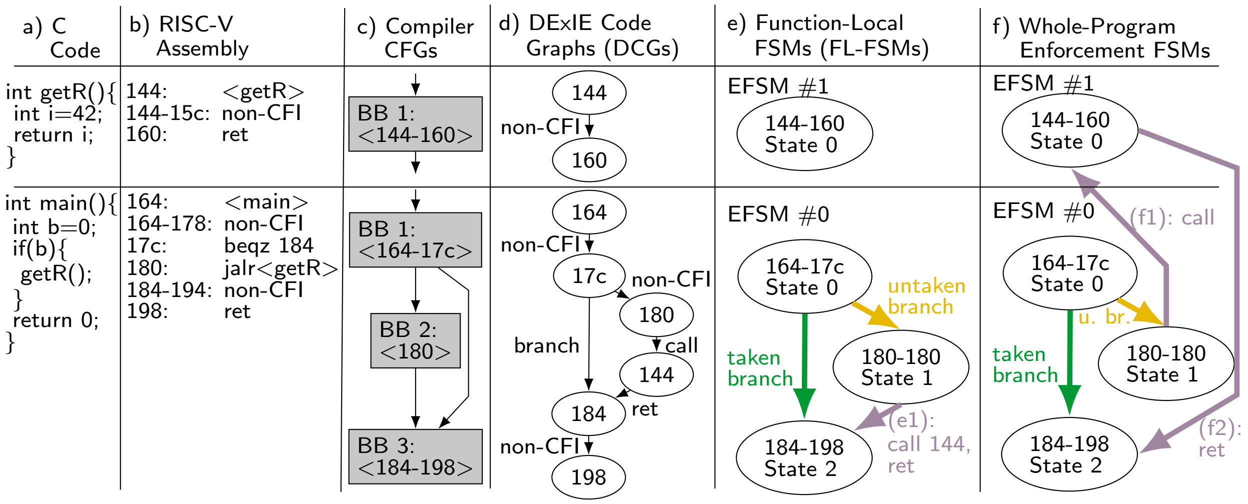
Column (a) contains each function's source code. Using a toolchain like LLVM or GCC, the code gets compiled into an ELF file, containing the assembly code shown in Column (b). For reference, Column (c) holds each function's traditional compiler CFG. Its nodes contain control flow instructions (CFI) and non-CFI (nCFI) and its edges are intra-function CFI (jumps and branches). Column (d) shows our refined DExIE Code Graph (DCG). Its nodes are code addresses. Each edge represents a single CFI, or sequences of nCFIs. Based on DCGs, Algorithm 1 constructs the Function-Local FSMs (FL-FSM) (Column (e)). After getting interconnected, they become the Whole-Program EFSMs actually being used for enforcement (Column (f)). The automatically created result can optionally be hand-tightened. One can deactivate edges, define explicit states per loop iteration, or specify a function's alternative EFSMs per call.

Applying Algorithm 1 to `getR()` function's DCG results in a single-state EFSM. Its entry state is also the return state. Because the function does not contain any CFI, the EFSM lacks any transitions. For `main()`, these rules lead to the removal of nodes 164 (no CFI), 144 (another function) and 184 (no CFI). The result of the algorithm are two FL-FSMs, which are shown in Column (e) of Figure 7.

The final transformation step performs the interconnection between FL-FSMs. The result can be seen in the figure's last Column (f). The purple intra-FSM arrow (e1) is split up into two arrows, namely one call (f1) to the first state of the called function's EFSM and one return (f2) from its accepting state. As result, we create a model consisting of two interconnected Whole-Program Enforcement FSMs. The `main` function's EFSM is ca-

Algorithm 1: EFSM generation algorithm

```
Input: One DExIE Code Graph (DCG)
Output: One Function-Local FSM
for each DCG-node do
    rename node to state;
    if state is exit state then
        mark state accepting, allow return to caller;
    else if a state's single out edge has no CFI then
        delete state & out-edge, transfer in-edges &
          address to next state;
    else if state is located in other function then
        delete state & out-edge, forward in-edge to
          next state, assign state's address to edge;
```



```
int getR(){
 int i=42;
 return i;
}
```

```
144:      <getR>
144-15c:  non-CFI
160:      ret
```

```
int main(){
 int b=0;
 if(b){
  getR();
 }
 return 0;
}
```

```
164:      <main>
164-178:  non-CFI
17c:      beqz 184
180:      jalr<getR>
184-194:  non-CFI
198:      ret
```

Fig. 7 From left to right: Two functions are transformed into interconnected whole-program Enforcement FSMs (EFSM). DExIE's attack mitigation capability depends on the ESFM's granularity. In the given example, manipulated calls and returns will be detected via ESFM (PCoP) or shadow stack violations (RoP). Fine-granular (e.g. profiling-generated) EFSMs can also selectively enforce branches (JoP, DoP).

pable of calling `getR()`'s EFSM, which in turn allows to return back to `main()`'s EFSM. The example demonstrates our concepts for intra- and inter-function CF. This is an alternative to prior work, which employs EFSMs only for inter-function CF [31], or system calls [37], thereby limiting overheads and CF granularity.

This simplified example does not use compiler optimizations, and no optimization is performed on the EFSMs. Currently, *inter-function CF* is limited to Call and Return instructions. Thus, DExIE does not yet allow Branches and Jumps *between* functions and EFSMs. Typically, these result from compiler optimizations for *inter-function CF* without stack interaction (e.g., tail-calls), and have to be avoided for now as DExIE would misinterpret them as CF violations and reset the core. We use the `-fno-optimize-sibling-calls` GCC flag for deactivating the optimization of *sibling and tail recursive calls*, and thus making DExIE compatible with all other optimizations at the `-O3` level.

4.1.2 DExIE Enforcement FSM (EFSM) Rules

Non-optimized non-edited DExIE EFSMs (Column (f), Fig. 7) obey a set of basic rules: Only one FSM and state is active at a time. Each function has one EFSM. Each CF target address corresponds to one EFSM-state. EFSM-states begin with a CFI, or alternatively the function's first instruction. Execution of a CF instruction always triggers an EFSM state transition. EFSM-edges specify the legal transitions. In case of a function call, an EFSM-state can call the first state of another EFSM. States containing a return instruction are designated as "accepting" states. A return instruction also reactivates the caller's EFSM at the correct state.

4.2 Hardware Architecture

4.2.1 System Architecture and DExIE Interface

Figure 8 shows a sample RISC-V core with a common 5-stage RISC pipeline, the DExIE component, corresponding memories, and a Secure Platform Controller for initiating the startup sequence.

As part of the startup sequence a Secure Platform Controller first loads and activates DExIE's configuration (1.). Next, the application is loaded (2.). Finally, DExIE releases the processor core's reset (3.), thereby starting the supervised program execution, which is guaranteed to not miss any control flow events.

For the DExIE pipeline taps, we initially considered using standard interfaces [12][13][14] to attach DExIE to the core. But as these interfaces only report *retired* instructions or instruction blocks, respectively, they come too late and would lead to DExIE missing its goal of detecting a violation *earlier* than the next instruction's commit, which might be a write instruction to a dangerous memory-mapped device, having irreversible real-world impact. As part of prior work, we thus implemented the *RT-LIFE* interface, which taps early pipeline stages [35]. DExIE uses this RT-LIFE interface to obtain the current Program Counter (PC), the current instruction, and the next PC.

At runtime, DExIE first identifies the obtained control flow instructions. Next, the current EFSM's state's

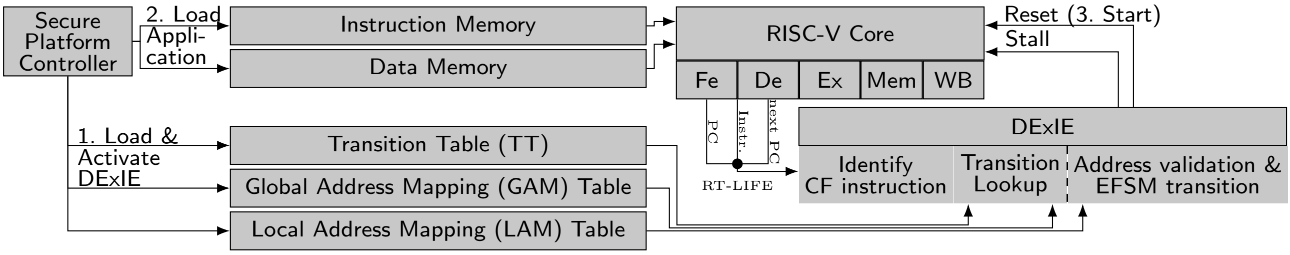


Fig. 8 RISC-V core with an attached DExIE monitor, memories and a Secure Platform Controller. The core provides the current PC, instruction, and the next PC. For any CF anomaly, DExIE resets the core in time, and thus prevents any subsequent malicious instruction from being committed to memory. Depending on the individual core's signal taps, its pipeline structure, and its latency for memory writes, stalling the core mitigates latency-related security risks. With its close coupling to the monitored processor's pipeline, DExIE will also have far tighter, low-latency control than would be possible with a more loosely-coupled Guard Processor. DExIE's precise microarchitecture and latency (1-2 cycles) are discussed in Sec. 4.2.4.

legal transitions are retrieved from the Transition Table. Each transition contains a narrow Address ID, which indexes one out of two address mapping memory tables in order to determine the corresponding full-width legal CF target address. Finally, the state's transition addresses are compared against the core's next PC (CF target address). If a match is found, the CF is valid and the corresponding transition into the next EFSM and the next EFSM state fires. For an unknown address, or non-matching transitions, the CF is deemed invalid, and DExIE immediately resets the core.

4.2.2 Data Structures, Lookup Sequence and Scaling

This section describes DExIE's configuration memories (Fig. 8) and refers to our previous code example (Fig. 7) to discuss the corresponding memory contents. In addition, DExIE's Shadow Stack (DSS) is introduced.

In Figure 9, DExIE's on-chip memories are shown as grey boxes (A), (B) and (C). For larger designs, these could be extended by cached DRAM memory, introducing new stalls. The transition memory (A) contains a legal set of transitions for all states and EFSMs (Transition Table, TT). A transition consists of activation information (Boolean: branch or call, and an Address ID), as well as the transition's next EFSM and next EFSM state. Each inter- or intra-EFSM CF-target address must be known in advance. In particular, these addresses are stored in one of the two DExIE address mapping tables, namely the (B) intra-function Local Address Mapping (LAM) table and the (C) inter-function Global Address Mapping (GAM) table. Both tables are indexed by narrow Address IDs, and contain one full-width address per index. Depending on its purpose, an Address ID can either be a Local Address ID (LAID) or a Global Address ID (GAID). Each EFSM has its own LAM table, but only a single GAM table is used for the entire program. Again focusing on the TT, notice the possibility for *decoupling* GAIDs and next-EFSM IDs (enabling independence of functions and EFSMs), as well as LAIDs and next-state IDs (enabling independence of CFIs and states), for an optional refinement of CF granularity.

Next, we focus on the colors shown in Figure 9 to demonstrate sample lookups. The yellow (untaken branch), green (taken branch), and purple (call) colours correspond to the same-colored transitions in Figure 7.

Yellow and **green** refers to the taken and un-taken **branches** from State 0 of EFSM 0. First, the instruction is identified as function-internal CF. Next, from the Transition Table (A) the LAIDs of the current state's (State = 0) transitions are both speculatively accessed in parallel (LAID = 1 & 2), and used to index the LAM Table (B) to read the addresses 0x180 (untaken branch target) and 0x184 (taken branch target). Finally, both addresses are compared against the next PC computed by the core. In case of a match, DExIE performs the corresponding EFSM-internal transition into EFSM 0 and State 1 or 2, as set by the jump decision.

The **purple** marker refers to the **call** in State 1 of EFSM 0. Analogously, the instruction is identified as inter-EFSM call. GAIDs are read from the Transition Table (A). Each legal transition's GAID (here: GAID=0) speculatively indexes the GAM Table (C) to obtain the corresponding legal target function entry point address (0x144), which is then validated against the actual next PC value to finally transition into the entry state of EFSM 1. Called EFSMs are always entered in their State 0. Thus, the call transition's next State ID entry is not used. Instead, and in case of calls, DExIE repurposes the entry to hold the Return State Identifier. It is temporarily stored on DExIE's Shadow Stack (DSS) (Fig. 10), and indicates the caller's EFSM's state when the callee's EFSM returns.

Return instructions are enabled via the DSS (Fig. 10) - a second independent stack, which is not accessible by the core, similar to [28]. As in a traditional stack, entries are pushed and popped for function calls and re-

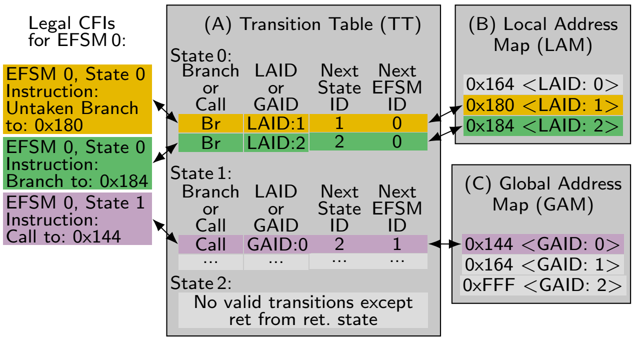
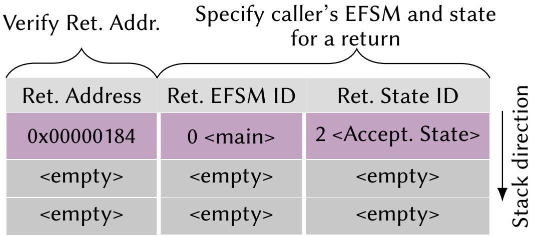


Fig. 9 DExIE's memory contents: (A) Transition Table, (B) Local Address Mapping (LAM) for (branches and jumps) and (C) Global Address Mapping (GAM) table table (for calls). Colors and contents correspond to Figure 7.


Verify Ret. Addr.
Specify caller's EFSM and state for a return
Stack direction

Ret. Address	Ret. EFSM ID	Ret. State ID
0x00000184	0 <main>	2 <Accept. State>
<empty>	<empty>	<empty>
<empty>	<empty>	<empty>

Fig. 10 When calling a function, DExIE's Shadow Stack stores the return address, the ESFM ID, and its state, which are restored (verified) at return time.

turns. As shown in the first column of Figure 10, each entry holds a copy of the RISC-V core's return address. The Columns 2 and 3 show that each entry also contains DExIE's return EFSM ID and Return State ID. For any call transition, like the one described in the previous example (Fig. 9), DExIE pushes this information onto the stack and enters the called EFSM's entry state. For a return from a previously called function, DExIE pops the top-most entry, verifies the return address, and activates the return EFSM in the given return state.

A naive implementation would *scale* approximately according to the simplified Fomulae 1 – 4. These assume one legal target address per CFI, no manual state duplication, e.g. via FSM state loop unrolling, a constant size for all functions, and CFIs targeting only Basic Block (BB) entrypoints.

$$\text{Dupl. Stack}_{\text{size}} = \text{RISC-V-Addr.-Width} \cdot \text{Call-Depth} \quad (1)$$

$$\text{GAM}_{\text{size}} = \text{RISC-V-Addr.-Width} \cdot \#\text{Funcs} \quad (2)$$

$$\text{LAM}_{\text{size}} = (\text{RISC-V-Addr.-Width} \cdot \#\text{BBs}) \cdot \#\text{Funcs} \quad (3)$$

$$\begin{aligned}\text{TT}_{\text{size}} = {} & \#\text{BBs} \cdot (1 + max(log_2(\#\text{BBs}), log_2(\#\text{Funcs})) \\ & + log_2(\#\text{BBs}) + log_2(\#\text{Funcs}))\end{aligned} \quad (4)$$

4.2.3 Optimization of Data Structures

In practice, our design implements optimizations, which were not described in the simplified example, but which significantly reduce memory requirements by more than 50 %. For hardware/software systems that do not fully leverage a 32-bit address space, DExIE address entries within the GAM table can be narrowed to match the extent of the address space actually used. Next, our LAM table does not implement wide absolute, but narrow function-local addresses, which can be sized to fit the largest function expected to be executed on this processor. Finally, un-taken branches that transition into the current EFSM's next state can be encoded using just a single additional bit per state. This lazy-next-state encoding (LNSE) requires sequential state IDs for subsequent untaken branches, which is realized by a prior automatic reordering of states. When looking again at Figure 9, LNSE significantly reduces overheads for the TT and LAM tables, as the yellow untaken branch transition to `0x180` is expressed by a single bit. Notice that the transition connects State 0 and State 1 (+1), thus no reordering is needed in this example (Fig. 7).

In the discussion so far, all tables were assumed to have the same fixed sizes. By analyzing typical IoT baremetal applications from the Embench benchmark suite, as well as a sample program using Contiki-NG (an embedded OS) [10], we verified that common applications contain a broad range of function sizes (Fig. 11). This would lead to wasted memory space in the "one size fits all" approach, since all memory blocks for TT and LAM tables would have to be configured to fit the *largest* function's EFSM number of the states. Therefore, DExIE allows to *dynamically* re-partition its internal memory at configuration load-time, right before the system boots. Multiples of 2^n are used to define the number of EFSM-instances and the number of states per EFSM-type, for up to four EFSM-types. As completely flexible EFSM sizes would require a more complex additional layer of indirection (for flexible addressing of constraints), we consider our decision to yield a viable trade-off. For the experiments in Sec. 5, four different EFSM table *sizes* (2, 16, 64 and 512 states), as well as 8,16,4, and 1 table *instances* of these sizes are configured. As smaller EFSMs can be placed in larger EFSM tables, this configuration fits all benchmarks (Fig. 11).

In our first design, we marked accepting states in a dedicated accepting table with one bit per EFSM per state. A more recent attempt avoided the additional accepting table, and defined an EFSM's highest state as implicitly accepting. However, this requirement fails with more complex EFSMs having more than one accepting state, as we could only define one accepting state. It also conflicts with EFSMs that have a number of states other than 2^n and contain sequences of LNSE-encoded untaken branches, which in turn must

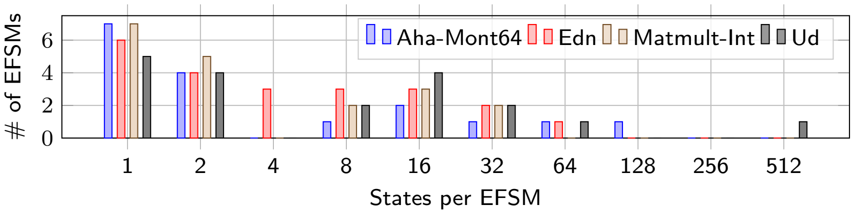


Fig. 11 Benchmarks and their corresponding EFSM sizes in grouped by 2^n states per EFSM. Function size and complexity largely varies, thus should be reflected by flexible hardware EFSM sizes (flexible configuration memory partitioning required).

have ascending State IDs from lowest to highest state; in this case, no legal sequence can be represented.

We first solved this by implementing untaken branch transitions in the traditional encoding (Fig. 9), which required additional memory. For sequences of untaken branches, our next approach solved the problem of ascending-only State IDs by auto-inserting single special states, that implicitly automatically transitioned to freely configurable State IDs. This solved the incompatibility, but required two instead of one hardware-transitions, and therefore affected either performance or latency-related security guarantees (Sec. 3). Finally, we decided to not mark accepting *states*, but instead mark *transitions* to the accepting states (with all transition bits set to one). As a result, accepting states themselves are not encoded within the Transition Table, reducing memory overhead. Returning into the previous EFSM is still constrained to take place at accepting EFSM States, and is also safeguarded by our EFSM-aware DExIE Shadow Stack (Fig. 10).

4.2.4 Microarchitecture and Parallel Table Lookups

In contrast to our efforts, most related work (Sec. 2) focuses on solutions that are either not real-time capable, or do not explicitly guarantee stopping execution earlier than any subsequent malicious access to a MMIO device can happen [5].

The performance overhead of a CF integrity enforcement depends on the dynamic frequency of CFIs. We express this as the *CFRate*, defined as the number of CFIs per clock cycle. A *CFRate* = 1 indicates a CFI *every* clock cycle, = 1/2 one every *second* clock cycle etc. Without pipelining EFSM transitions, DExIE has a maximum CFRate it can process without requiring stalls. This depends on the core-specific latency between getting the data from the taps and when DExIE has to make the valid/invalid decision. In case the core's CFRate is temporarily higher than DExIE's, automatic stalls are used, preventing DExIE from being overtaken.

Our actual microarchitecture targets at *CFRates* between 1 and 1/2 (Fig. 12). In order to achieve such high throughput/ low-latency monitoring, speculative queries to our TT held in FPGA BRAM are implemented, with a maximum number of legal CF targets per EFSM state configured to 2. These accesses can be performed in parallel using Dual-Ported BRAM. Therefore, this implementation supports all directly addressed CFI, but limits indirectly addressed CFIs to a maximum of 2 targets. Note that for more complex codes using a larger number of indirect targets, DExIE can be configured to either employ slower sequential lookups, or use multiple memory blocks to perform multiple lookups in parallel. Another compiler-based solution would be to *split* valid targets by constructing a binary tree of branches via a compiler plugin.

Fig. 12 shows DExIE's operation at the microarchitecture level. First, the CFI is identified as a branch/ jump (a), call (b) or return (c). In case of **branches and jumps (a)** at a *CFRate* up to 1/2, valid LAIDs and their corresponding next states are read from the state's TT entry (queried in advance). The LAIDs index the LAM Table, which provides both valid target

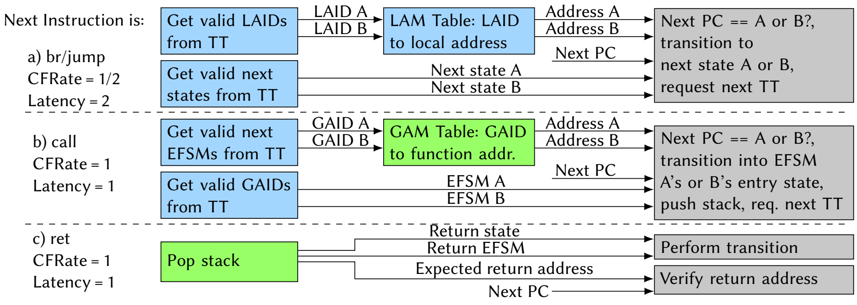


Fig. 12 DExIE's microarchitecture: TT lookup before address mapping, TT BRAM is queried at transition time, two alternative CF targets are loaded in parallel to hide memory latency. BRAM reads are marked blue, LUTRAM reads green.

addresses at the beginning of the next cycle. Next, the valid addresses are compared against the next PC address. If a match is found, the CF is valid, and DExIE requests the next state's TT entry and transitions to the current EFSM's next state. The **call (b)** mechanism is similar. The GAIDs of the valid targets are read from the TT entry (queried in advance), which are then used to combinationally index the LUTRAM-based GAM Table to retrieve the corresponding function addresses. In parallel, both target EFSM IDs are read from the TT entry. Finally, the RISC-V core's next PC address is compared to both legal addresses, and if a match is found, the corresponding transition into the EFSM's entry state, as well as a stack push and the next state's TT query, are performed. Because the GAM Table holds far fewer entries than all of the LAM Tables combined, the GAM Table is implemented in LUTRAM, which is faster than BRAM, thus supporting a *CFRate* of 1. **Returns (c)** are also supported at a *CFRate* of 1: First, the DExIE stack in LUTRAM is popped. DExIE transitions into that return state in the return EFSM. In parallel, DExIE verifies the next PC by comparison with the popped valid return address.

5 Evaluation

Using its default configuration with up to 2 clock cycles attack response latency, DExIE is attached and evaluated in combination with different RISC-V cores. As stated in the title, our project focuses on small IoT-class RISC-V cores, for which we selected Piccolo, PicoRV32, Taiga, and VexRiscv [25]. We selected these cores, which are detailed in [35], to demonstrate DExIE's potability. Each FPGA design's clock frequency, LUTs, Register and BRAM usage is compared to the corresponding core-only implementation. We evaluate our design using four benchmarks from Embench-IoT [11], which covers real-world IoT tasks. In the prior Figure 11, the benchmark's corresponding EFSMs are grouped by their number of states in multiples of 2^n.

The size of DExIE's maximum total configuration memory should be chosen to fit all applications that are expected to run on the processor (here: the four benchmarks). At boot-time, the configuration memory can then be re-partitioned to fit a specific application's EFSMs. As all benchmarks need only one legal call and branch-taken target per CFI, we configure DExIE to use only single- instead of dual-ported memories.

All CPU cores are implemented as Processing Elements (PE) in the Task Parallel System Composer (TaPaSCo) FPGA SoC framework [17] [21] targeting the VC709 Virtex 7 device prototyping board using Xilinx Vivado 2018.3, which, in our case, yields better results than more recent versions. On the software side, we use GCC 9.2.0 and Embench 0.5 Draft compiled at Embench default `-O2` with RV32IM, but disallowing inter-function branches and jumps (as described in Sec. 4.1.1) for DExIE. To find each design's highest frequency, synthesis was run iteratively using TaPaSCo's Design Space Exploration feature. Note that performance baselines for the cores can be found in [17].

While not being part of our project, we expect DExIE to scale even when combined with larger processor cores. However, and to prevent evil attacks in flight, low-latency tracing of uncommitted instructions from an *out-of-order* pipeline is expected to be more complex, likely causing extra overhead on the tracing interface [35].

Figure 13(a) shows the achieved maximum clock frequencies for the core-only and DExIE-extended implementations. As expected, achieving DExIE's strong security guarantee of preventing any outside-world impact via MMIO-Devices, and at the same time staying real-time capable at CFRates between 1 and 1/2, often comes at the price of a slower clock frequency. Using an asynchronous reset, all cores but VexRiscv give DExIE two cycles of latency between sending their combined <PC, instruction, next PC>message to DExIE and the commit of the next instruction to the memory interface, which is the point where the valid/invalid decision has to have been made by DExIE. For all cores but VexRiscv, single-cycle stalls only occur for back-to-back CFIs (which rarely occurs in typical applications). VexRiscv is stalled an additional cycle, if a CFI is followed by a memory write instruction.

Depending on the core's size, which in turn varies with the scope of the instruction set being supported (Table 1), LUT requirements increase by 54 % to 124 %, as shown in Figure 13(b). The absolute overhead depends on the core-specific interface and Vivado's optimization algorithm, which duplicates logic for better timings. Figure 13(c) shows an increased register usage between 2.24 and 7.04 kilobit (kb). This is mainly caused by the GAM table being implemented in LUTRAM. When comparing the BRAM cost of using DExIE (Fig. 13(d)), we use the *minimal* Embench target system as a baseline, which has 64 kB data + 64 kB of instruction memories in BRAM.

The slight performance *improvement* for Piccolo is due to variations in the Vivado toolchain, and not related to DExIE. Depending on the core, the performance overhead ranges from 0 % to more than 100 % (Table 1). PicoRV32 has been optimized for very high $f_{\max}$ and small area. It thus is a "worst-case" for DExIE monitoring, which carries a comparatively high area and performance overhead. At the other end of the spectrum lies the Piccolo core, which carries a far lower

Benchmark / Core	Core's ISA	Aha-Mont64 w/o	Aha-Mont64 w	Edn w/o	Edn w	Matmult-Int w/o	Matmult-Int w	Ud w/o	Ud w
Piccolo	RV32ACIMU	5.17 s	4.88 s	31.56 s	29.82 s	38.86 s	36.73 s	15.83 s	14.96 s
PicoRV32	RV32IM	17.43 s	40.88 s	23.75 s	55.70 s	24.40 s	57.20 s	16.32 s	38.28 s
Taiga	RV32IMA	1.79 s	2.65 s	1.86 s	2.75 s	2.03 s	3.01 s	1.57 s	2.33 s
VexRiscv	RV32IM	8.79 s	15.34 s	6.68 s	11.61 s	6.93 s	12.06 s	6.99 s	12.00 s

Table 1 Each core's ISA as well as wall-clock execution time per core and benchmark, without and with the DExIE unit attached. As described, Piccolo's improvement is not caused by DExIE, but relates to the proprietary FPGA toolchain.

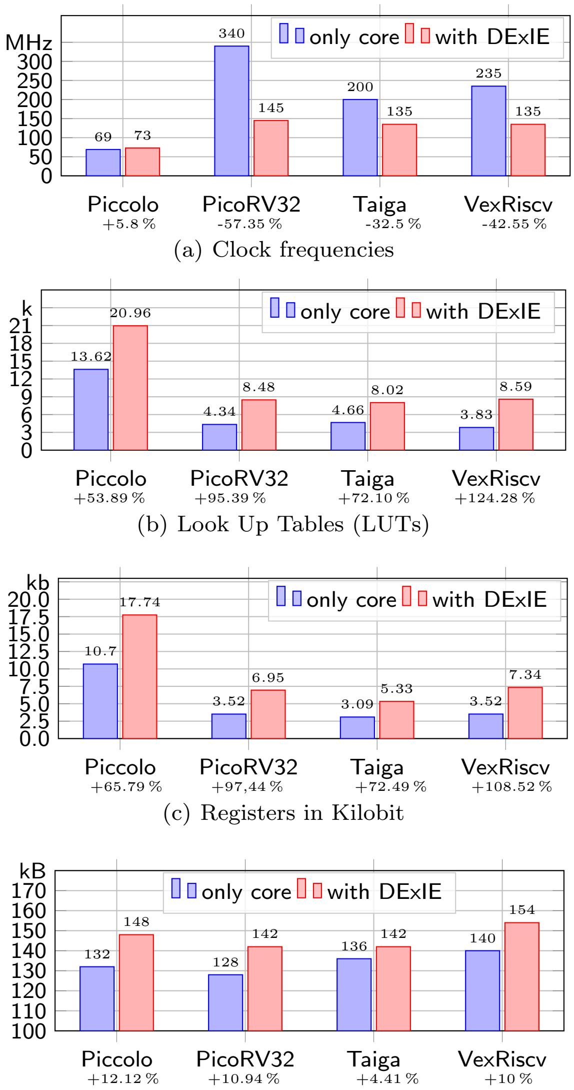
overhead and *no* performance slowdown. The percentage of DExIE's extra clock cycles for stalls ranges from 0 % for the fast-clocking and higher-latency PicoRV32, to 10.4 % for Taiga with its partially independent execution units.

When referring back to Sec. 2 and comparing DExIE with related work other approaches, such as [23], imply less memory overhead (no performance overhead is mentioned, deep pipeline-specific integration, low portability to other cores, not real-time capable), but do *not* guarantee tight timing for attack prevention. For improved performance, they implement queuing for up to 2000 control flows.

DExIE instead does not use any queuing, but immediately *stalls* program execution to prevent the core from overtaking DExIE (to guarantee tight attack response latency). To achieve smaller overheads, other approaches limit CF granularity, and thus are not capable of mitigating DoP or JoP attacks [6, 20, 22, 28, 36].

In contrast, runtime dynamic remote attestation *is* capable of detecting very fine granular CF deviations. But densely storing the logged taken-path information can lead to unacceptable (>100 %) memory overheads [26]. This can be reduced by *hashing* the taken-path information, and by offloading the attestor to a server class machine [8]. DExIE guarantees very short attack response times, which are required to guarantee attack prevention, and still reaches a relatively high maximum clock frequency of 145 MHz in our experiments, with no or only very few statically predictable stalls. Even Remote Attestation approaches, which do not implement tight latency guarantees, and thus should easily achieve higher clock frequencies, are often limited to lower frequencies (70 - 80 MHz) [7, 40], or at best achieve comparable clock frequencies to DExIE (150 MHz) [8]. For a more detailed comparison, please see [4].

Overall, we have not found *any* prior work, which is compatible with a number of different processor cores and is capable of actually preventing runtime-dynamic code-reuse attacks with tight and guaranteed latency, with real-time support, and a flexible (EFSM-defined) CF checking granularity.



(a) Clock frequencies

(b) Look Up Tables (LUTs)

(c) Registers in Kilobit

(d) BRAM in Kilobyte

Fig. 13 Results for Piccolo, PicoRV32, Taiga and VexRiscv.

6 Conclusion

DExIE is an on-chip low-overhead fine-grained CF integrity enforcement device that guarantees to react faster than a subsequent illegal instruction may perform a memory write, blocking an attack's potentially irreversible malicious real-world impact. Its limited area and performance costs often make DExIE a better solution than alternative approaches, such as software instrumentation, or the use of a full-scale guard processor. DExIE is especially attractive when it can be attached to a suitable base pipeline. For such pipelines, which are not primarily optimized for f_{max}, DExIE can operate with *no* clock frequency penalty or wallclock slowdown. Because it is designed with reduced latency in mind, DExIE causes no stalls for PicoRV32 and only few and fully-predictable stalls for other cores. As future work, we will extend DExIE to support context-switching by alternating per-context EFSM groups, thereby safeguarding IoT using more complex operating systems.

Acknowledgements 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 (2011) Jump-oriented programming: a new class of code-reuse attack. pp 30–40, DOI 10.1145/1966913.1966919
2. Busse F, Nowack M, Cadar C (2020) Running symbolic execution forever. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2020, p 63–74, DOI 10.1145/3395363.3397360, URL `https://doi.org/10.1145/3395363.3397360`
3. Chen, et al. (2019) Automated finite state machine extraction. In: Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation, Association for Computing Machinery, FEAST'19, DOI 10.1145/3338502.3359760, URL `https://doi.org/10.1145/3338502.3359760`
4. de Clercq R, Verbauwhede I (2017) A survey of hardware-based control flow integrity (CFI). `1706.07257`
5. Das S, Zhang W, Liu Y (2016) 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
6. Davi L, Sadeghi AR, Winandy M (2011) ROPdefender: A detection tool to defend against return-oriented programming attacks. In: Proceedings of the 6th International Symposium on Information, Computer and Communications Security, ASIACCS 2011, pp 40–51
7. Dessouky G, Zeitouni S, Nyman T, Paverd A, Davi L, Koeberl P, Asokan N, Sadeghi AR (2017) LO-FAT: Low-overhead control flow attestation in hardware. pp 1–6, DOI 10.1145/3061639.3062276
8. Dessouky G, Abera T, Ibrahim A, Sadeghi AR (2018) LiteHAX: Lightweight hardware-assisted attestation of program execution. In: Proceedings of the International Conference on Computer-Aided Design, Association for Computing Machinery, New York, NY, USA, ICCAD '18, DOI 10.1145/3240765.3240821, URL `https://doi.org/10.1145/3240765.3240821`
9. Div (2021) Capstone the ultimate disassembly framework. URL `http://www.capstone-engine.org/`
10. Div (2021) Contiki-NG: The os for next generation iot devices. URL `https://github.com/contiki-ng/contiki-ng`
11. Div (2021) Embench-iot github repository. URL `https://github.com/embench/embench-iot`
12. Div (2021) RISC-V debug specification. URL `https://github.com/riscv/riscv-debug-spec`
13. Div (2021) RISC-V Formal verification framework. URL `https://github.com/SymbioticEDA/riscv-formal`
14. Div (2021) RISC-V trace specification. URL `https://github.com/riscv/riscv-trace-spec`
15. Div (2021) Spike RISC-V ISA simulator. URL `https://github.com/riscv/riscv-isa-sim`
16. Evans, et al. (2015) Control jujutsu: On the weaknesses of fine-grained control flow integrity. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA, CCS '15, p 901–913, DOI 10.1145/2810103.2813646, URL `https://doi.org/10.1145/2810103.2813646`
17. Heinz C, Lavan Y, Hofmann J, Koch A (2019) 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
18. Housley R, Ashmore S, Wallace C (2010) Trust Anchor Format. RFC 5914, RFC Editor, URL `https://www.rfc-editor.org/rfc/rfc5914.txt`
19. Hu H, Shinde S, Adrian S, Chua ZL, Saxena P, Liang Z (2016) Data-oriented programming: On the expressiveness of non-control data attacks. In: 2016 IEEE Symposium on Security and Privacy (SP), pp 969–986
20. Intel (2020) Control-flow Enforcement Technology specification, rev. 3.0. URL `https:`

//software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

21. Korinth J, Hofmann J, Heinz C, Koch A (2019) The TaPaSCo open-source toolflow for the automated composition of task-based parallel reconfigurable computing systems. In: Applied Reconfigurable Computing, Springer International Publishing, Cham, pp 214–229
22. Li J, Chen L, Xu Q, et al. (2019) Zipper stack: Shadow stacks without shadow. ArXiv
23. LI Y, LI Jw (2018) A technique preventing code reuse attacks based on RISC processor. DEStech Transactions on Computer Science and Engineering DOI 10.12783/dtcse/CCNT2018/24682
24. Liljestrand H, Nyman T, Wang K, Perez CC, Ekberg JE, Asokan N (2019) PAC it up: Towards pointer integrity using ARM pointer authentication. In: 28th USENIX Security Symposium (USENIX Security 19), USENIX Association, Santa Clara, CA, pp 177–194, URL https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand
25. MultiMedia LLC (2021) RISC-V Cores list. URL https://github.com/riscv/riscv-cores-list
26. Nunes IDO, Jakkamsetti S, Tsudik G (2020) Tiny-CFA: A minimalistic approach for control-flow attestation using verified proofs of execution. 2011.07400
27. Nyman T, Dessouky G, Zeitouni S, Lehikoinen A, Paverd A, Asokan N, Sadeghi AR (2018) HardScope: Thwarting DOP with hardware-assisted run-time scope enforcement. 1705.10295
28. Ozdoganoglu H, Vijaykumar TN, Brodley CE, Kuperman BA, Jalote A (2006) SmashGuard: A hardware solution to prevent security attacks on the function return address. IEEE Transactions on Computers 55(10):1271–1285, DOI 10.1109/TC.2006.166
29. Phu TN, Hoang L, Toan N, Tho ND, Binh NN (2019) C500-CFG: A novel algorithm to extract control flow-based features for iot malware detection. 2019 19th International Symposium on Communications and Information Technologies (ISCIT)
30. Qualcomm (2017) Pointer Authentication on ARMv8.3. URL https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf
31. Rahmatian M, Kooti H, Harris IG, Bozorgzadeh E (2012) Hardware-assisted detection of malicious software in embedded systems. IEEE Embedded Systems Letters 4(4):94–97, DOI 10.1109/LES.2012.2218630
32. Roemer R, Buchanan E, Shacham H, Savage S (2012) Return-oriented programming: Systems, languages, and applications. ACM Trans Inf Syst Secur 15(1), DOI 10.1145/2133375.2133377, URL https://doi.org/10.1145/2133375.2133377
33. Sadeghi A, Niksefat S, Rostamipour M (2018) Pure-call oriented programming (PCOP): chaining the gadgets using call instructions. Journal of Computer Virology and Hacking Techniques 14:1–18, DOI 10.1007/s11416-017-0299-1
34. Spang C, Lavan Y, Hartmann M, Meisel F, Koch A (2021) 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), Association for Computing Machinery, New York, NY, USA, DASIP '21, p 26–34, DOI 10.1145/3441110.3441146, URL https://doi.org/10.1145/3441110.3441146
35. Spang C, Meisel F, Koch A (2021) RT-LIFE: Portable RISC-V interface for real-time lightweight security enforcement. In: Intl. Conf. on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS), Springer International Publishing
36. Sullivan GT, DeHon A, Milburn S, Boling E, Ciaffi M, Rosenberg J, Sutherland A (2017) The dover inherently secure processor. In: 2017 IEEE International Symposium on Technologies for Homeland Security (HST), pp 1–5
37. Xia N, Mao B, Zeng Q, Xie L (2007) Efficient and practical control flow monitoring for program security. In: Okada M, Satoh I (eds) Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 90–104
38. Yuan P, Zeng Q, Ding X (2015) Hardware-assisted fine-grained code-reuse attack detection. In: Bos H, Monrose F, Blanc G (eds) Research in Attacks, Intrusions, and Defenses, Springer International Publishing, Cham, pp 66–85
39. Yubin Xia, Yutao Liu, Chen H, Zang B (2012) CFIMon: Detecting violation of control flow integrity using performance counters. In: IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012), pp 1–12, DOI 10.1109/DSN.2012.6263958
40. Zeitouni S, Dessouky G, Arias O, Sullivan D, Ibrahim A, Jin Y, Sadeghi A (2017) ATRIUM: Runtime attestation resilient under memory attacks. In: 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp 384–391, DOI 10.1109/ICCAD.2017.8203803