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

Christoph Spang
spang@esa.tu-darmstadt.de
Embedded Systems & Applications Group - TU Darmstadt
Darmstadt, Hessen, Germany

Yannick Lavan
Embedded Systems & Applications Group - TU Darmstadt
Darmstadt, Hessen, Germany

Marco Hartmann
Embedded Systems & Applications Group - TU Darmstadt
Darmstadt, Hessen, Germany

Florian Meisel
Embedded Systems & Applications Group - TU Darmstadt
Darmstadt, Hessen, Germany

Andreas Koch
koch@esa.tu-darmstadt.de
Embedded Systems & Applications Group - TU Darmstadt
Darmstadt, Hessen, Germany

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 only very few and then perfectly predictable pipeline stalls. It is often 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 for fine-grained control flow information.

CCS CONCEPTS

• **Security and privacy → Embedded systems security**; **Hardware security implementation**.

KEYWORDS

Fine-Grained Control Flow Integrity, RISC-V, Hardware Security, Real Time, Low Overhead, IoT

ACM Reference Format:
Christoph Spang, Yannick Lavan, Marco Hartmann, Florian Meisel, and Andreas Koch. 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 (14 th edition) (DASIP '21), January 18–20, 2021, Budapest (initially), Hungary.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3441110.3441146

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 run-time attacks such as Return- and Jump Oriented Programming (RoP, JoP) [1] [24]. 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 flow *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 10-12% more on-chip memories than an unmonitored system. In addition to the hardware architecture, we also introduce a toolchain that can extract the Enforcement FSMs (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 [4]. Some base processors are especially suitable for DExIE monitoring, in that they incur *neither* a clock frequency penalty, *nor* a wall-clock execution-time slowdown.

After covering related work (Section 2), the basic mechanism and security considerations are presented in Section 3. Next, the software toolchain and the transformation of code into DExIE EFSMs is discussed (Section 4.1). This is followed by our hardware design (Section 4.2) and microarchitecture (Section 4.2.4). In the final sections, we report evaluation results (Section 5), and conclude looking to further work (Section 6).

2 RELATED WORK

In case of JoP [1] and RoP [24] code-reuse attacks, an attacker first analyses an application to collect a potentially large collection of abusable code snippets (gadgets). After exploiting a program bug as entry point, these gadgets are executed in an unintended order, thus creating a malicious and under some conditions even Turing-complete exploit without relying on the modification of existing or insertion of new code. Discussed by [24], a traditional Address Space Layout Randomization or $W \oplus E$ 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 [4] [20] or hash of the return address [18]. By comparing the valid copy to the core's computed return address, manipulated return instructions are detected. 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, with DExIE being one alternative:

CFIGuard [27] is a software solution that uses Control Flow Graphs (CFG) and x86-64 ISA hardware capabilities (Last Branch Recording and Performance Monitoring Unit), for a fully-transparent control flow integrity enforcement (CFIE) in Linux. By monitoring only indirect branches (16.7 % of all branch instructions executed) of four different server workloads, CFIGuard reports between 23.9 % and 32.3 % memory overhead, and a maximum of 5.6 % performance overhead. The limited performance overhead likely results from the exclusion of monitoring direct control flow (CF) and the inclusion of IO-time in the server benchmarks. Their constraints are stored in *sparse* bitmaps. We started our own research using *sparse* bitmaps, as well, to achieve higher speeds and to reduce logic overhead, but gave up that approach due to an unreasonably high and poorly-scaling memory overhead, which would render DExIE too expensive for low-cost IoT devices lacking DDR memory. Instead, one major contribution of DExIE is a fast, and at the same time, compact encoding of CF constraints.

Dover Microsystem's hardware uses a word-based tag-map with instruction-level granularity [25]. These stateful tags are used for CFIE with lower than CFG-granularity, but depending on the implemented policies, their concept also covers memory safety and dataflow integrity. The RISC-V Rocket core, used as an example, executes C-based programs, which are compiled using an adapted GCC toolchain. The "Inherently Secure Processor's" (ISP) configuration is also stored apart from the application, similar to DExIE's. Neither memory nor performance overheads are given by Dover in the literature - we expect them to correlate with individual policies. The use of caches in their design is a sign of both higher clock frequencies, but also for unpredictable stalls, making the design most likely real-time incapable. In contrast, DExIE's tight encoding is stored in BRAM and requires no caching, resulting in very few and fully-predictable stalls.

Intel Tiger Lake cores introduce Intel Control Enforcement Technology (CET) [16], which is a combination of a shadow stack and Indirect Branch Tracking. Additional CET instructions mark valid branch target addresses of indirect branches. A hardware state-machine verifies that each indirect CF is followed by an ENDBRANCH and stops the program otherwise. No overheads for performance or memory are given in the specification. We assume that their performance overhead correlates with the number of inserted instructions, which itself depends on the number of indirect CFIs.

The ARMv8.3 ISA [22], used by i.e., Apple and Qualcomm, implements Pointer Authentication Codes (PACs) by repurposing formerly unused address bits of 64 bit pointers. Each PAC's value is computed using a combination of target address, context and a chosen key. Again, no overheads are given in the literature.

Alternatively, tags can be used to enforce more sophisticated security models besides FSM logic. Li et al. [19] 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 below DExIE's overheads. Unfortunately stalls, real-time capability and performance overheads are not discussed, and their model cannot fulfill our portability and granularity requirements.

Security tags are often stored as additional instruction bits or a new instruction type within instruction memory. Both, the resulting increase of word width and new instructions, require deep changes of the individual core (Fig. 5 in [19]). This is sub-optimal for a portable solution optimized for compatibility to many IoT cores. Instead, DExIE only requires the tapping of few status signals, and a stall, and a reset input. Its FSMs are stored in separated memories. Thereby, all of the cores presented in Section 5 can be fitted with a common DExIE implementation.

When loading tags together with the corresponding CF instruction or its target instruction, the constraints required for checking only become available late during the instruction execution cycle [19] [22]. In contrast, for each CFI, DExIE directly transitions to the next CFI's state and then waits there for the core to catch-up executing intermediate straight-line code. As DExIE loads the new constraints with the activation of the *previous* transition with single or double-cycle latency, even without cache, the number of stall cycles per CFI is reduced ideally to zero, see Section 4.2.4.

Tagging of only indirect CF is reasonable for reducing overheads, but possibly insufficient, as [13] and [15] demonstrate attacks running on legal CFGs. Such attacks can also bypass DExIE's standard checking granularity with CFG-based EFSMs executing one EFSM transition for any CFI (Section 4.1.1). Therefore, and depending on the supplied EFSMs, DExIE supports going beyond CFG-grade granularity, and implements an optional decoupling of function identifiers (Global Address ID - GAID) and EFSM identifiers (EFSM ID). For any function call (depending on the callers EFSM and state), this allows to *individually* constrain a called function to an alternative even tighter EFSM. DExIE's hardware also provides an option for decoupling individual CF instructions (Local Address ID - LAID) and their corresponding EFSM's state (StateID). This enables alternative constraints on any individual CFI depending on its state. With functions corresponding to alternative EFSMs, and CFIs corresponding to alternative states, this variable granularity allows to tighten the amount of legal CF paths, ideally down to one, thereby mitigating *any* CF-attack based on even a single CF-derivation.

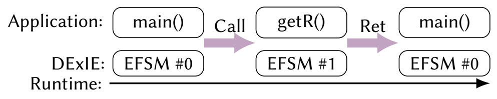


Figure 1: Active pairs of (function, EFSM) over time

This paper discusses DExIE in conjunction with auto-generated CFG-based EFSMs that can optionally be hand-edited for finer granularity (Section 4.1).

3 MECHANISM & SECURITY CONSIDERATIONS

DExIE by default uses one CFG-based EFSM per function to statefully constrain an application function's legal CF. 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. 1). 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] [24].

Guarantees: DExIE will react to any CFI violating the currently active EFSM. It will stop an attacker from calling (jumping, 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 write instruction.

Assumptions: DExIE is designed with a focus on code-reuse attacks. A non-write-protected 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 lack 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 read-only program memory for defending against code injection attacks.

Real-time: DExIE is real-time capable. Specifically, this means that a given code sequence will always take the same execution time. The monitoring also introduces only very limited (if any) stall cycles, that are also statically predictable, compared to an unmonitored execution (Section 4.2.4 and 5).

4 IMPLEMENTATION

4.1 Software Toolchain

Our toolflow (Fig. 2) uses a conventional compiler to compile the program code into an Executable and Linking Format (ELF) object file. The object file is used twice: Using *objdump* it is converted into a binary image, which is then executed by the RISC-V core. The object file is also fed into the DExIE-Compiler, which generates DExIE's EFSM and address-to-ID mapping configuration. We make use of the freely available Capstone disassembly library [6], which we utilize to reconstruct each function's DExIE Code Graph (DCG) from the CFG via *static analysis* (further details in Section 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. 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. This approach does not require the original source code. However, in case of indirect control flows (e.g., jump via register) that are not statically disambiguable, extra code annotations or hand-annotated assembly are needed to define the valid target(s).

DExIE's hardware is not only compatible with EFSMs from static analysis similar to [12], but already supports runtime-profiling-based EFSMs generated using the Spike simulator [11]. Symbolic execution is another possible source for EFSMs [2] [21].

4.1.1 Toolchain Details - Creation of EFSMs.

Figure 3 shows an example code-to-EFSM-transformation containing two functions `main()` and `getR()`. Each column contains the result of different stages in the DExIE toolflow.

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 (e.g. deactivate edges, explicit states per loop run, or an alternative FSM per call) for increased granularity.

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 3.

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 capable 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. It is an alternative to prior work, which deploys EFSMs for inter-function CFI [23] or system calls [26] only, aiming for lower overheads, but also coarsening the EFSM CFIE granularity.

This simplified example does not use compiler optimization, and no optimization is performed on the EFSMs. Currently, for inter-function CF, only Call and Return instructions are supported.

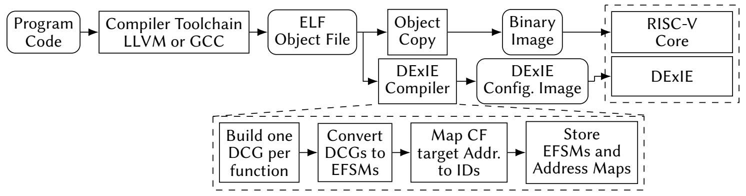


Figure 2: Compilation into RISC-V and DExIE binaries: The DExIE Compiler reads the ELF file, constructs the DExIE Code Graphs (DCGs) and EFSMs, performs the address-to-ID mapping, and writes the DExIE configuration image.

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;
```

Thus, DExIE does not yet allow Branches and Jumps *between* functions and EFSMs. Typically, these result from compiler optimization for inter-function CF without stack interaction (e.g., tail-calls), and have to be avoided for now as DExIE would misinterpret them as CFIE violations and reset the core. We use the GCC flag `-fno-optimize-sibling-calls` 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. 3) 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 4 shows a sample RISC-V core with a common 5-stage RISC pipeline and instruction and data memories. Early pipeline stages are fitted with custom DExIE wire taps, to forward control flow information to DExIE as soon as possible. DExIE needs the current Program Counter (PC), the current instruction, and the next PC. As a first step in monitoring, DExIE identifies control flow instructions. Next, the current EFSM's state's legal transitions are retrieved from the Transition Memory. 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

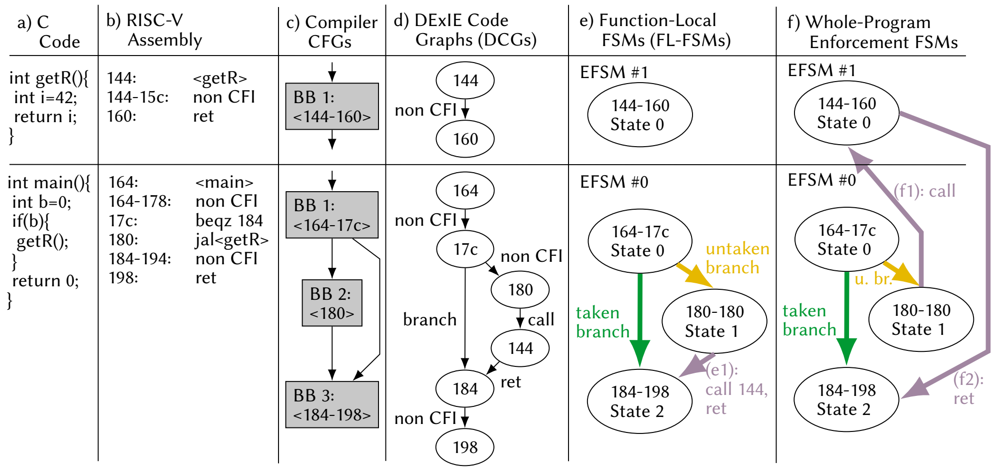


Figure 3: From left to right: Two functions are transformed into interconnected whole-program Enforcement FSMs (EFSM).

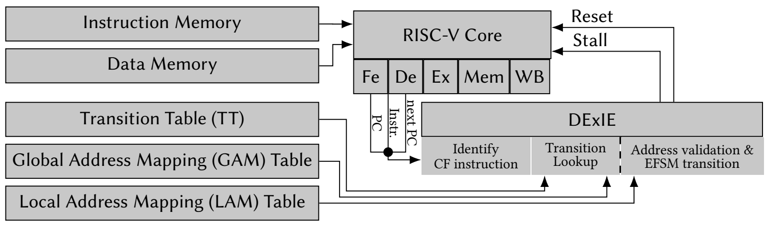


Figure 4: RISC-V core with an attached DExIE monitor. The core provides the current PC, the current instruction, and the next PC. For any CF anomaly, DExIE resets the core in time, thus prevents any subsequent malicious instruction from being committed to memory. Depending on the individual core's signal 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 later in Section 4.2.4.

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.

For the DExIE pipeline taps, we initially considered using standard interfaces [8][9][10] 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.

4.2.2 Data Structures and Lookup Sequence. This section describes DExIE's configuration memories (Fig. 4) and refers to our previous code example (Fig. 3) to discuss the corresponding memory contents. In addition, DExIE's Shadow Stack (DSS) is introduced.

In Figure 5(a), 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* GAID and NextFsmID (enabling independence of functions and EFSMs), as well as LAID and NextStateID (enabling independence of CFIs and states), for an optional refinement of CF granularity.

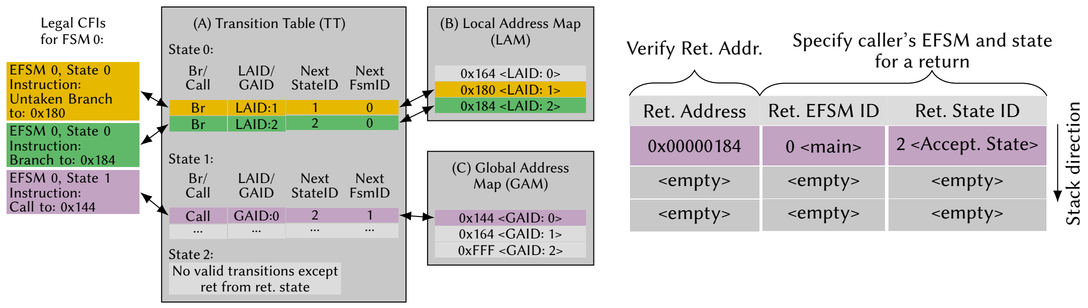
Next, we focus on the colors shown in Figure 5(a) to demonstrate sample lookups. The yellow (untaken branch), green (taken branch), and purple (call) colours correspond to the same-colored transitions in Figure 3.

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 nextPC 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, in case of calls, DExIE repurposes the entry to hold the Return State Identifier. This Return State ID is temporarily stored on DExIE's Shadow Stack (DSS) (Fig. 5(b)), and indicates the caller's EFSM's state when the callee's EFSM returns.

Return instructions are enabled via the DSS (Fig. 5(b)) - a second independent stack, which is not accessible by the core, similar to [20]. As in a traditional stack, entries are pushed and popped for function calls and returns. As shown in the first column of Figure 5(b), 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. 5(a)), 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.

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. For hardware/software systems that do not fully exploit 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.



(a) DExIE's memory contents: (A) Transition Table, (B) Local Address Mapping (LAM) and (C) Global Address Mapping (GAM) tables.

(b) DExIE Shadow Stack

Figure 5: DExIE configuration memory contents (a) and Shadow Stack (b). Colors and contents correspond to Figure 3.

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 reordering of states. When looking again at Figure 5(a), LNSE significantly reduces overheads for the TT and LAM tables, as the yellow transition to `0x180` is expressed by a single bit.

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) [7], we verified that common applications contain a broad range of function sizes. This would lead to wasted memory space in the "one size for 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 FSM-instances and the number of states per FSM-type for up to four FSM-types. For the experiments in Section 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.

4.2.4 Microarchitecture and Parallel Table Lookups. In contrast to our efforts, most related work (Section 2) focuses on solutions, which are either not real-time capable, or do not explicitly guarantee to stop execution earlier than any subsequent malicious access to a MMIO device can happen [3].

The performance overhead of a CFIE monitor 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, which 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. 6). 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

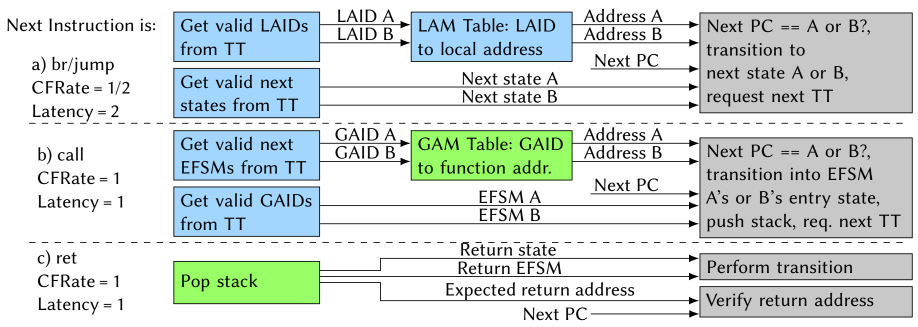


Figure 6: DExIE's microarchitecture: TT lookup before address mapping, TT BRAM is queried at transition time, two alternative CF targets are loaded in parallel to hide memory latency. BRAM reads are marked blue, LUTRAM reads green.

blocks to perform multiple lookups in parallel. Another compiler-based solution would be an additional splitting of valid targets by constructing a binary tree of branches.

Fig. 6 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 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 address by comparison with the popped valid return address.

5 EVALUATION

DExIE is evaluated in combination with different RISC-V cores (Piccolo, PicoRV32, Taiga and VexRiscv). 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 [5], which covers real-world IoT tasks. In Figure 7, 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, the evaluation configures 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 [14] [17] 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 Section 4.1.1) for DExIE. To find each design's highest frequency, synthesis was run iteratively. Note that performance baselines for the cores can be found in [14].

Figure 8(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, 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 8(b). The absolute overhead depends on the core-specific interface and Vivado's optimization algorithm, which duplicates logic for better timings. Figure 8(c) shows an increased register usage between 2.24 and 7.04 kilobit. This is mainly caused by the GAM table being implemented in LUTRAM. When comparing the BRAM cost of using DExIE (Fig. 8(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. 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 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.

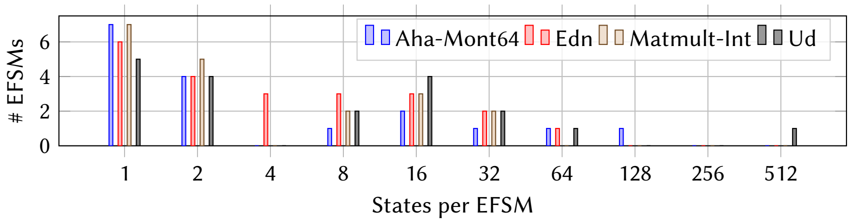
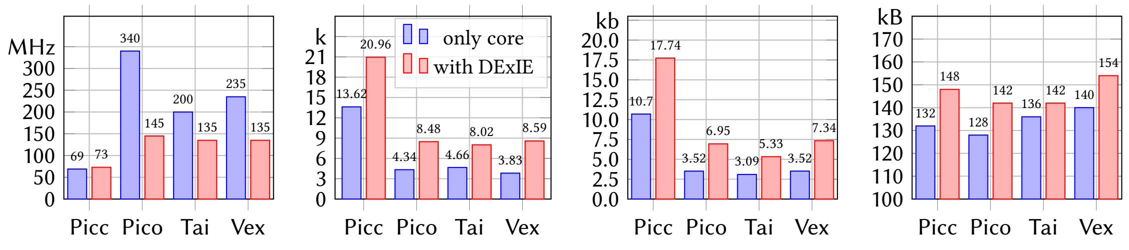


Figure 7: Benchmarks and their corresponding EFSMs

Benchmark / Core	Core's ISA	Aha-Mont64		Edn		Matmult-Int		Ud	
		w/o	w	w/o	w	w/o	w	w/o	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



(a) Clock frequencies (b) Look Up Tables (LUTs) (c) Registers in Kilobit (d) BRAM in Kilobyte

Figure 8: DExIE evaluation results: Clock frequencies, Look Up Tables (LUTs), Registers in Kilobit, BRAM in Kilobyte

6 CONCLUSION

DExIE is an on-chip low-overhead fine-grained CFIE monitor 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. In our next steps, we will scale DExIE and its toolflow up to better support multi-target indirect branches, interrupts, contexts, further reduce overheads, and continue exploring another toolchain for finer than CFG granularity, fully-utilizing DExIE's capabilities.

ACKNOWLEDGMENTS

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] Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. 30–40. https://doi.org/10.1145/1966913.1966919

[2] Frank Busse, Martin Nowack, and Cristian Cadar. 2020. Running Symbolic Execution Forever. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/3395363.3397360

[3] S. Das, W. Zhang, and Y. Liu. 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 (2016), 3193–3207.

[4] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 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*. 40–51.

[5] Div. 2019. Embench-Iot Github Repository. https://github.com/embench/embench-iot

[6] Div. 2020. Capstone The Ultimate Disassembly Framework. http://www.capstone-engine.org/

[7] Div. 2020. Contiki-NG: The OS for Next Generation IoT Devices. https://github.com/contiki-ng/contiki-ngf

[8] Div. 2020. RISC-V Debug Specification. https://github.com/riscv/riscv-debug-spec

[9] Div. 2020. RISC-V Formal Verification Framework. https://github.com/SymbioticEDA/riscv-formal

[10] Div. 2020. RISC-V Trace Specification. https://github.com/riscv/riscv-trace-spec

[11] Div. 2020. Spike RISC-V ISA Simulator. https://github.com/riscv/riscv-isa-sim

[12] Chen et al. 2019. Automated Finite State Machine Extraction. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation* (London, United Kingdom) *(FEAST'19)*. Association for Computing Machinery. https://doi.org/10.1145/3338502.3359760

[13] 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* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 901–913. https://doi.org/10.1145/2810103.2813646

[14] Carsten Heinz, Yannick Lavan, Jaco Hofmann, and Andreas Koch. 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.

[15] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*. 969–986.

[16] Intel. 2020. Control-flow Enforcement Technology Specification, Rev. 3.0. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

[17] Jens Korinth, Jaco Hofmann, Carsten Heinz, and Andreas Koch. 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, 214–229.

[18] Jinfeng Li, Liwei Chen, Qizhen Xu, et al. 2019. Zipper Stack: Shadow Stacks Without Shadow. *ArXiv* (2019).

[19] Yang LI and Jun-wei LI. 2018. 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

[20] H. Ozdoganoglu, T. N. Vijaykumar, C. E. Brodley, B. A. Kuperman, and A. Jalote. 2006. SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. *IEEE Trans. Comput.* 55, 10 (2006), 1271–1285. https://doi.org/10.1109/TC.2006.166

[21] Tran Nghi Phu, L. Hoang, N. Toan, Nguyen Dai Tho, and N. N. Binh. 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)* (2019), 568–573.

[22] Qualcomm. 2017. Pointer Authentication. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf

[23] M. Rahmatian, H. Kooti, I. G. Harris, and E. Bozorgzadeh. 2012. Hardware-Assisted Detection of Malicious Software in Embedded Systems. *IEEE Embedded Systems Letters* 4, 4 (2012), 94–97. https://doi.org/10.1109/LES.2012.2218630

[24] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.* 15, 1, Article 2 (March 2012), 34 pages. https://doi.org/10.1145/2133375.2133377

[25] G. T. Sullivan, A. DeHon, S. Milburn, E. Boling, M. Ciaffi, J. Rosenberg, and A. Sutherland. 2017. The Dover inherently secure processor. In *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*. 1–5.

[26] Nai Xia, Bing Mao, Qingkai Zeng, and Li Xie. 2007. Efficient and Practical Control Flow Monitoring for Program Security. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*, Mitsu Okada and Ichiro Satoh (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 90–104.

[27] Pinghai Yuan, Qingkai Zeng, and Xuhua Ding. 2015. Hardware-Assisted Fine-Grained Code-Reuse Attack Detection. In *Research in Attacks, Intrusions, and Defenses*, Herbert Bos, Fabian Monrose, and Gregory Blanc (Eds.). Springer International Publishing, Cham, 66–85.