eMPAC: Unlocking Full Potential of Pointer Authentication in Microcontrollers with Fat Pointers

Sungsoo Kim[0009−0007−0577−0740], Jihoon Kim[0009−0004−7197−0407], Kyuwon Cho[0000−0002−1082−3030], and Hojoon Lee*[0000−0001−5344−6266]

Sungkyunkwan University, Suwon, South Korea
{sskim71,rhgus862,kyuwon.cho,hojoon.lee}@skku.edu

Abstract. The Pointer Authentication (PA) extension introduced in ARMv8.3-A has significantly raised the bar for exploiting memory errors with hardware-accelerated pointer integrity. Unfortunately, its adaptation to the M-profile ARM architecture arrived with significantly reduced feature sets. As a result, existing works on PA-based system hardening do not apply to ARM-based MCU systems. In this paper, we present the eMPAC architecture, which extends the M-profile PA with an ABI extension to unlock its full potential. The key idea is to construct a compiler-defined extended architecture that embraces the fat pointer design. We show that the eMPAC architecture enables the existing PA-based security measures developed for A-profile processors to be applied to M-profile processors with moderate overhead.

Keywords: Pointer Authentication · ARMv8.1-M · Microcontrollers · Fat Pointers · Software Security

1 Introduction

Runtime software attack mitigation has played a pivotal role in raising the bar for adversaries, given that achieving complete memory safety is notoriously difficult. Thanks to modern operating systems and compilers, most programs today already run with layers of defense, including stack canaries, DEP, and Address Space Layout Randomization (ASLR). More recently, hardware extensions have enabled more powerful mitigations to become practical.

Pointer Authentication (PA), first introduced in Armv8.3-A [5], has become a standard hardware security feature that enforces pointer integrity by signing pointers with cryptographic keys. PA for the A-profile ARM processors— which we shall call *PA-A*—the ISA provides two instruction families, `PAC` and `AUT`, which sign and authenticate pointers, respectively. These instructions embed a *Pointer Authentication Code (PAC)* into the unused bits of a 64-bit pointer; the PAC is essentially a keyed hash of the pointer combined with a 64-bit *modifier* supplied to the instruction. By signing pointers before they are stored and authenticating

* Corresponding author

them when reloaded, PA-A prevents the use of corrupted pointers arising from memory-unsafe code.

PA-A has been embraced by the industry and research community alike. Both GCC [14] and LLVM [28] are already shipping with PA-based backward-edge control flow integrity (i.e., return address signing). Also, Apple's OS kernels and user-level programs have been further hardened with PA to protect critical pointers such as the C++ vtables [10]. Accumulated research also demonstrates that PA can be leveraged to design effective yet performant software security schemes encompassing PA-accelerated *Control-Flow Integrity (CFI)* [16,27,26], data pointer integrity [27,12], and software domain isolation [30,12]. Notably, PARTS [27] demonstrates the feasibility of full-coverage pointer authentication, ensuring the integrity of all pointers during program execution, including return addresses, function pointers, and data pointers. Previous works have also shown that software domain isolation can be achieved by dividing PA contexts with disjoint use of keys and modifiers [30,12].

Unfortunately, PA for the M-profile ARM processors [6] (*PA-M*) later arrived with significantly simplified feature sets compared to its A-profile counterpart. The changes made to PA-M, introduced along with the ARMv8.1-M architecture, render the accumulated research on PA incompatible with PA-capable MCU systems. Most notably, PA-M no longer attaches the PAC to a pointer, since the M-profile processors are 32-bit. Without the unused bits to leverage for a PAC, PA-M's PAC instruction family outputs a PAC in a designated general-purpose register operand. The detached PAC scheme breaks the underlying premise of previous works. That is, the PAC values no longer travel with the pointers and are left ephemeral without a systematic management scheme.

Moreover, PA-M provides only one key register as opposed to five key registers on A-profile that were used to isolate PA context for different types of pointers (e.g., code vs. data). Hence, the singular key register calls for a new signing scheme that can ensure sufficient entropy with diversified PA contexts.

In this paper, we present **eMPAC** (**e**xtensions for ARMv8.1-**M** **P**ointer **A**uthentication **C**ode) to retrofit the M-series architecture to render the existing PA-accelerated security schemes feasible on the PA-capable *Microcontroller units (MCUs)*. eMPAC proposes a compiler-defined extension to ARMv8.1-M that embraces the notion of fat pointers [43,17,32,41], allowing PACs to travel with pointers while the additional memory consumed scales precisely with the number of pointers stored in memory.

Our fat pointer design allows the PAC value to be conceptually integrated with the pointer, thereby providing basic interoperability with PA-A. Moreover, PA-signed fat pointers can be authenticated and used with a compact and efficient instruction sequence. eMPAC also compensates for the lack of multiple PA keys through a fine-grained compound modifier scheme that binds every pointer authentication to a specific per-type, per-task context. By taking full advantage of the modifier, eMPAC effectively achieves code/data pointer separation, type-based PA context isolation, and by-task isolation. The resulting *full-coverage fat pointer authentication scheme* protects return addresses, code pointers, and data

pointers with fine-grained modifier contexts, enforcing integrity and type safety before use. To realize these extensions, we design and implement a compiler based on LLVM that enforces the fat pointer ABI, as well as MCU OS support in FreeRTOS that partitions modifier contexts across tasks and establishes a cross-domain sharing model for compatibility and robustness.

In summary, this paper makes the following contributions:

- **Fat pointer-based architecture extension for PA-M.** We propose a compiler-defined architecture extension that introduces an *authenticated fat pointer* design and compensates for PA-M's single key with a fine-grained compound modifier that binds every signing and authenticating operation to a specific per-type, per-task context.
- **Enabling PA-A security schemes on MCUs.** We show that eMPAC closes the feasibility gap against PA-A, enabling mature security schemes developed for PA-A to be feasible on PA-M.
- **Open-source compiler and OS implementation.** We implement eMPAC as LLVM passes that enforce the fat pointer ABI across all pointer operations, along with minimal changes to the FreeRTOS scheduler for per-task modifier switching[1].
- **Evaluation on real hardware.** We evaluate eMPAC on a PA-M-capable Cortex-M85, demonstrating an average overhead of 10.81% on an HTTPS-capable Mongoose web server.

2 Background and related work

In this section, we provide additional details on the PA-M and PA-A implementations, including essential concepts of MCU OS such as tasks and queues, as well as on previous work that motivated eMPAC.

2.1 Real-time Operating Systems

A real-time operating system (RTOS) is an operating system designed to schedule and execute tasks under timing constraints. For eMPAC's implementation, we used FreeRTOS [15], although our design is not OS-specific. To aid understanding, we briefly discuss *tasks*, the units of work in MCU OSes such as FreeRTOS. FreeRTOS uses tasks for the main application, timers, networking, and other functions. Although tasks typically perform distinct functions, they often share global variables and a non-isolated heap. During a context switch, the scheduler saves the current task's register and stack state and restores those of the next task, enabling preemptive multitasking in a shared address space. Tasks also communicate through queues to handle asynchronous events such as network packets or user inputs.

[1] `https://github.com/sslab-skku/eMPAC`

2.2 Software Security and Isolation in MCUs

Control-flow integrity in MCUs. Achieving control-flow integrity (CFI) on MCUs remains challenging because of tight memory budgets, the lack of an MMU, and limited hardware support. Register-based designs such as μrai [1] protect return addresses with dedicated registers, while SHERLOC [38] and InsectACIDE [40] use hardware tracing support to detect violations. Compiler-based schemes such as Silhouette [44] and Kage [13] maintain protected shadow stacks, and attestation-based systems such as OAT [37] delegate verification to a trusted component. CFI via pointer integrity faces similar constraints. ParC-SPI [39] targets embedded systems but requires PA-A, so it does not apply to M-profile processors. EPI [46] embeds metadata in unused pointer bits for spatial and temporal safety, but requires custom hardware support. eMPAC shows that PA, widely used for CFI in desktop and mobile devices, can also be leveraged for the same purposes on MCUs by retrofitting PA-M to be compatible with PA-A.

Domain isolation in MCUs. Isolation has been explored as another line of defense. TrustZone-based systems like SeCloak [24] and TZ-DataShield [23] separate secure and non-secure worlds; micro-TEE designs such as TrustLite [22] and TyTAN [9] provide lightweight compartments without virtual memory. MPU-driven frameworks (Minion, ACES, EC) [20,11,19] partition firmware but are constrained by coarse-grained regions and limited MPU slots. Prior works [12,30] have shown that a form of domain isolation can be achieved with mutually disjoint assignment of PA keys or modifiers when a comprehensive pointer integrity scheme is in effect. Likewise with CFI, eMPAC demonstrates feasibility of domain isolation specialized for MCUs, namely the MCU task isolation through its fat-pointer-based architecture.

2.3 ARM Pointer authentication

We now detail the PA-A and PA-M mechanisms and emphasize their key differences.

Pointer Authentication in ARMv8.3-A (PA-A). PA-A introduces the `PACXX` and `AUTXX` instruction families, which sign and authenticate pointers using cryptographic message authentication codes bound to an execution context. A `PACXX` instruction computes a PAC over the pointer in Reg_{ptr} and the modifier in Reg_{mod} under a hardware-resident key, and embeds the resulting tag in the pointer's unused upper bits so that the PAC travels with the pointer through memory. A matching `AUTXX` instruction verifies and strips the PAC, raising an exception if the tag does not match. Notably, PA-A exposes five independent keys, enabling cryptographic separation of *PA key contexts* across code pointers, return addresses, and data pointers. As an example, `PACIA` (Reg_{ptr}, Reg_{mod}) signs the pointer in Reg_{ptr} by attaching a PAC using key register *IA* and a modifier in Reg_{mod}.

Pointer Authentication in ARMv8.1-M (PA-M). PA-M adapts PA-A to 32-bit MCU systems [4], but introduces two design simplifications that render existing PA-based security schemes infeasible. First, the PAC no longer

Table 1: Feasibility of PA-based security schemes on ARM architecture variants.

	PTR Width	PAC Pos.	Full-cov. PTR Integrity			III. Domain Isolation
			Ret. Addr	I. Code	II. Data	
ARMv8.3-A	64-bit	In-Ptr	✓	✓ [27,12]	✓ [27,12]	✓ [12,30]
ARMv8.1-M	32-bit	In-Reg	✓	✗	✗	✗
ARMv8.1-eMPAC	**64-bit**	**Fat Ptr**	✓	✓*	✓*	✓*

*Enabled through eMPAC

travels with the pointer it protects: PA-M computes a standalone 32-bit PAC over Reg_{ptr} and Reg_{mod} and deposits it in a separate register Reg_{pac}, making safe PAC storage and pointer-tag pairing a software responsibility. Second, PA-M exposes only one active key register per privilege mode, and because MCU systems typically lack user-kernel privilege separation [29], a single key is effectively shared system-wide. Accordingly, PA-M only provides two PA instruction types: `PAC/AUT` and `PACG/AUTG`. `PAC` specializes in backward-edge CFI and is hardwired to sign the `LR` register using `SP` as the modifier, writing the resulting PAC to `R12`. `PACG` (Reg_{pac}, Reg_{ptr}, Reg_{mod}) is the general-purpose form where the Reg_{pac} is decided by the first argument of the instruction, computing a 32-bit PAC over Reg_{ptr} and Reg_{mod} under the active key and storing the result in Reg_{pac}.

2.4 PA-based security schemes

The PA-A-based schemes shown below are well-established defenses built on PA-A primitives. Yet they are infeasible on MCU systems: detached PACs sever the pointer–authenticator coupling, and no prior design reconciles this shift in PA-M. Moreover, these schemes rely heavily on PA context separation realized through multiple PA keys. Under PA-M's single-key regime, their security arguments must be reconstructed from scratch. eMPAC therefore retrofits PA-M to enable these schemes on ARM-based MCU architectures.

Code pointer integrity. PA has been extensively adopted in CFI schemes to improve security while suppressing runtime overhead. In addition to the backward-edge CFI available in compilers, PA-based code pointer integrity (i.e., forward-edge CFI) has also been explored. The code pointers are signed with diversified modifiers constructed based on the pointer's object type and object address to enforce a fine-grained forward-edge CFI [3]. PARTS [27] proposed PA-based signing of all code pointers, thereby fully extending the forward-edge CFI coverage.

Data pointer integrity. Existing works have demonstrated *full-coverage pointer integrity* [27,12] that encompasses not only code pointers, but also data pointers. Notably, PARTS [27] sought to diversify PA *contexts* by fully utilizing the PA key registers and a meticulously designed modifier scheme. For instance, one of the PA key registers `IA` is used to sign code pointers, while `DA` and `IB` are used for data pointers and return addresses, respectively. PARTS also uses the hash values of the pointer types as modifiers. We use the term *full-coverage*

pointer integrity to refer to a scheme that enables both code pointer (forward-edge CFI) and data pointer integrity.

Domain isolation. Another branch of work has shown that a sensitive software component or modules (i.e., domains) can be isolated with PA. HAKC [30] protects both control-flow and data pointers across kernel modules, ensuring that even dynamically loaded code (e.g., device drivers) cannot forge or misuse authenticated pointers. It introduces a hardware-enforced authentication context so that PACs generated in one module cannot be reused in another. Capacity [12] introduced a PA-based domain isolation scheme. With full-coverage pointer authentication as its basis, Capacity devised a PA key switching mechanism that assigns each domain a unique key, with which the domain's sensitive pointers and file descriptors are signed.

2.5 Threat model

We assume a powerful remote software adversary targeting a networked MCU system, who may exploit memory-safety vulnerabilities in either the application or the MCU OS to obtain arbitrary read/write access within the MCU's physical address space. As pre-existing runtime mitigations, we assume W⊕X and *Branch Target Identification (BTI)*. Accordingly, the adversary cannot modify code regions or jump into the middle of functions, and must therefore rely on code-reuse attacks to carry out exploitation. W⊕X enforcement is widely supported on modern MCU systems [45], and BTI was introduced to ARM M-profile architectures alongside PA as *Pointer Authentication and Branch Target Identification (PACBTI)* [4]. We assume a trusted boot process and exclude hardware-level attacks such as Rowhammer [21] and microarchitectural side channels on PA [35], as well as attacks on the PA crypto algorithm (e.g., QARMA [8]) itself.

3 Design

eMPAC is a compiler-driven architectural extension that turns PA-M into a practical, system-wide pointer-integrity mechanism for MCU-class systems that is compatible with PA-A. In doing so, eMPAC enables on MCUs the PA-A-based defenses shown in Table 1—full-coverage pointer integrity, including both code-pointer and data-pointer, and PA-based domain isolation. To achieve this, eMPAC's design addresses the following challenges:

Authenticated fat pointer design exploration for MCUs. eMPAC adopts a *fat-pointer* representation that co-locates each pointer with its PAC, uniformly protecting pointers in globals, stack, heap, and shared objects. The pointer width is extended to 64 bits—a 32-bit pointer scalar paired with its PAC—so that the PAC persists across copies, spills, and context switches. Realizing this end-to-end for *full-coverage pointer authentication* requires coordinated changes across the compiler, linker, and RTOS scheduler, detailed in this section and implemented in §4. An alternative is to store each PAC in a separate metadata table [25], but the tight resource budgets of MCU targets make

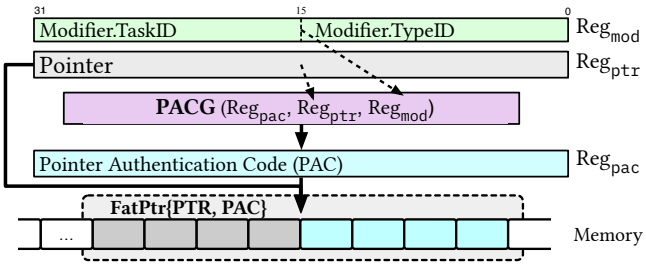

Fig. 1: eMPAC fat pointer generation

such disjoint schemes impractical. Our evaluation platform, for instance, runs at 480 MHz with only 1 MB of SRAM. Therefore, shadow memory or hash-table lookups add an indirection on every pointer access while consuming scarce SRAM for per-pointer bookkeeping.

Challenges of single-key PA on PA-M. PA-M exposes only a single PA key to the executing context, unlike PA-A's five keys, which prior work uses to separate code from data pointers or kernel from user authentication [27,12]. Because MCU OSes run in a single address space with a shared global segment (§2.1), per-task key switching is unavailable and all tasks authenticate under the same key. Consequently, the *modifier* becomes eMPAC's sole mechanism for fine-grained PA context diversification under a single key. To this end, eMPAC constructs a compound modifier whose two halves play distinct security roles: a secret, per-task taskID held strictly in a reserved register and switched by an eMPAC-modified scheduler. typeID, on the other hand, augments the diversity of the PA modifier using a fine-grained object type signature collected from the compiler frontend.

Enabling existing PA-Based Security on MCUs. Building on these primitives, eMPAC brings the security guarantees of PA-A-based defenses to PA-M through two coordinated mechanisms. First, a full-coverage authenticated fat pointer scheme (§3.1) protects both code (**I**) and data (**II**) pointers at every load and store, with a fine-grained compound modifier that provides cryptographic diversity under PA-M's single shared key. Second, task isolation (§3.3)—our MCU-specific adaptation of domain isolation—binds each PAC to its creating task, enabling safe sharing among mutually distrusting tasks (**III**).

3.1 Full-coverage authenticated fat pointers

Figure 1 captures the layout of the fat pointer and its generation procedure that occurs during pointer store into memory. The eMPAC compiler redefines the target architecture to treat the pointer types as 64-bit primitive data types. This transformation is applied consistently across globals, stack, and heap-allocated objects. This way, we can expect all pointer allocations and object definitions (e.g., pointers inside structures) to always allocate a 64-bit space for pointers. Then, eMPAC can enforce on-store/on-load pointer signing [12,27,30,16,25] and

```
1 ;Reg_mod = typeID("Socket_t*")
2 movw    r10, #38121
3 ;Reg_mod |= Reg_task
4 orr.w   r10, r10, r9
5 ;generate PAC into Reg_pac
6 pacg    r11, r0, r10
7 ;store [PTR, PAC]
8 strd    r0, r11, [r1, #164]
```

(a) Data pointer sign and store

```
1 ;load [PTR,PAC]
2 ldrd    r3, r11, [r0, #164]
3 ;Reg_mod = typeID("fn_ptr_t")
4 movw    r10, #48673
5 ;Reg_mod |= Reg_task
6 orr.w   r10, r10, r9
7 autg    r11, r3, r10
8 ;Authenticated call
9 blx     r3
```

(b) Code pointer load and authenticate

Listing 1.1: Fat pointer load and store procedure.

authentication by instrumenting all pointer load/store instructions. That is, a pointer is always packed into a fat pointer along with its PAC value before entering memory, and it is authenticated as it is fetched into a register.

Modifier scheme overview. eMPAC's fat pointer scheme uses two 16-bit fields as components of the modifier to compartmentalize memory accesses by pointer type and task. The typeID is a hashed representation of the pointer's language-level type (e.g., `struct hashtable*`), which is collected in the frontend of eMPAC compiler. The taskID is a *secret* per-task value kept strictly in the reserved register Reg_{task} under a register-only principle, and is switched by the eMPAC-modified scheduler on every task switch. Reg_{task} and Reg_{mod} are reserved registers intended to keep the taskID and the final compound modifier register-only throughout their lifetime. We discuss the security model of this modifier scheme separately in §3.2.

Fat pointer load and store procedures. This fat pointer scheme applies to both code and data pointers to achieve **I** and **II** from Table 1. Listing 1.1 illustrates the fat pointer load and store procedures that implement the scheme. Listing 1.1a captures the case of a store sequence involving a data pointer of type `Socket_t*`. Reg_{mod} is loaded with the typeID of the data type, which is materialized as `#38121` at compile time (Line 2). Then, Reg_{mod} is combined with the taskID of the current task to form the final modifier for the authentication (Line 4). The `pacg` instruction (Line 6) generates a PAC into Reg_{pac}. Finally, the paired memory store instruction `strd` efficiently stores the pointer (r0) and the PAC value (r11) into a fat pointer (Line 8).

Listing 1.1b shows the load and authentication procedure for a code pointer. The pointer (r3) and its PAC value (r11) are loaded into separate registers using a paired load instruction (Line 2). A nearly identical modifier generation sequence follows, this time for a function pointer type (Line 4–5). The `autg` instruction authenticates the PAC value by internally regenerating the PAC for comparison and raises an exception upon mismatch (Line 6). Lastly, it should be noted that the indirect branch (Line 9) is executed only if the pointer has successfully cleared the authentication process. This means that forward-edge CFI, or pointer integrity and pointer type safety, has been enforced on the branch.

3.2 Modifier security model

PA-M exposes only a single PA key, so in eMPAC the modifier is the sole mechanism for diversifying PA contexts. eMPAC achieves both diversification and confidentiality of the modifier through its composite modifier scheme that incorporates two constituents with distinct security postures: a secret, per-task taskID that ensures modifier confidentiality, and a fine-grained typeID that diversifies the context.

Ensuring confidentiality of taskIDs. eMPAC generates and maintains the per-task taskID as the confidential component of the modifier and protects it end-to-end across its life cycle—from generation, through in-register use, to scheduler-mediated save and restore. By doing so, the modifier's confidentiality is maintained even when the typeIDs are compromised to the attacker who launches powerful code-reuse attacks to collect typeIDs from the immediates values encoded in eMPAC's pointer signing/authentication instruction sequences. This design allows eMPAC to bind *execution context* to the PA context that provides modifier security comparable to PA-key switching proposed by Capacity [12] under PA-M's single key constraint.

Each taskID is a 16-bit random value generated at task creation and confined to the reserved register Reg_{task}: Reg_{task} is removed from the compiler's general allocation pool to prevent spills during high register pressure. Hence, the value remains register-only throughout normal execution. eMPAC makes another change to the scheduler to complete the life-cycle protection of the taskIDs; eMPAC XOR-encrypts Reg_{task} with a boot-time random 32-bit key held in another reserved register Reg_{key} before writing the *task control block*, and decrypts on restore. Finally, eMPAC keeps the composed modifier (taskID | typeID) resident in the reserved Reg_{mod} register, closing the last gap in taskID's life-cycle protection. Together with the encrypted save/restore path, this guarantees that the secret taskID component never appears in adversary-readable memory at any point during its lifetime.

Fine-grained PA context diversification with typeIDs. In our modifier security model, typeIDs are combined with taskIDs for finer-grained diversification. eMPAC diversifies PA contexts *within* each task by assigning a distinct 16-bit salted hash to each source-level pointer type (§4.1), so that distinct pointer types receive distinct typeIDs. Each typeID is materialized as an immediate operand to a movw instruction preceding the corresponding `PACG/AUTG` sequence and moved into Reg_{mod} only for the duration of that operation.

3.3 Supporting task isolation

Figure 2 illustrates an overview of eMPAC's task isolation built on top of its modifier scheme. eMPAC isolates the PA contexts for MCU tasks with the aforementioned taskIDs, thereby achieving **III** from Table 1 with each task as a compartment. Task isolation also requires allowing *controlled* sharing among global variables and tasks. To this end, we systematically define three types of possible load and store operations and implement sharing as follows:

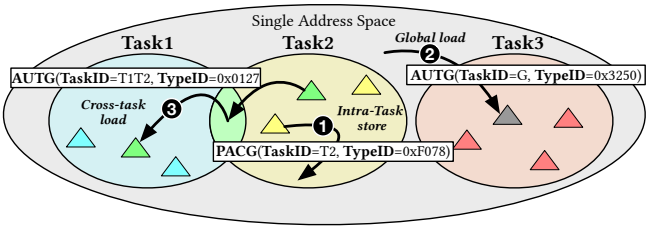

Fig. 2: Modifier diversification and task isolation model

Intra-task loads and stores. Intra-task load/stores (as illustrated by ❶ in Figure 2) are the dominant case of loads and stores that do not involve any sharing. These instructions do not require further consideration, and the pointer's typeID and the current taskID in Reg_{task} can be used.

Allowing global variable access. Use of system-wide global variables across tasks is a common practice in single-address-space MCU OSes (❷). To allow global variable access, eMPAC's modifier scheme assigns a dedicated taskID, called *G*. During compilation, the eMPAC compiler will identify and instrument loads and stores to `.data` or `.bss` sections to use the value *G* as the top bits value of the modifier instead of the current taskID in Reg_{task}.

Inter-task communication support. eMPAC also must support *cross-task* data sharing that occurs through *cross-task loads and stores* (❸). We address this challenge by creating a dedicated taskID for the data shared by two or more tasks via inter-task communications such as queues. For instance, we can derive a value, `T1T2`, for shared data between Task 1 and Task 2. Then, the load and store instructions to shared data space are instrumented similarly to the global data access; the instructions use `T1T2` encoded as an immediate operand and are loaded directly to Reg_{mod}, instead of using the current taskID in Reg_{task}. To this end, we provide developers with source-level annotations to mark data structures to be shared between two tasks. The eMPAC compiler will recognize the loads and stores to such data and instrument them accordingly. Specifically, when developers annotate pointer variables of a given type, the eMPAC compiler tracks the def–use chain of these pointers and instruments all loads and stores that dereference them. This annotation-based design is practical for embedded RTOS software, as mentioned in §2.1, where tasks are commonly structured as distinct functional components and inter-task sharing is typically limited to explicit communication objects. Pointer variables that share the same annotation ID (e.g., `__attribute__(annotation("SHR_T1T2"))`) are mapped to a common sharing domain, so their accesses are tagged with the same taskID. We found that such annotations were sufficient for supporting sharing, since cross-task interactions are infrequent and occur only in specific contexts.

Security Implications of Shared Domains. Pointers of ❷ and ❸ are deliberate design provisions that enable global data and inter-task communication; capabilities that strict per-task isolation alone cannot support. They extend the model with a controlled sharing mechanism: such pointers are bound to dedi-

cated sharing domains, providing well-defined cross-task access while preserving the full isolation of private memory. A pointer signed with G or `T1T2` will not authenticate under a task-local modifier, so cross-domain misuse is structurally prevented. eMPAC therefore treats ❷ and ❸ as explicit sharing boundaries, scoped narrowly by design to keep the attack surface minimal.

4 Implementation

We implemented eMPAC mainly as two cooperating LLVM passes: a high-level *intermediate representation* (IR) pass, and a low-level *machine IR* (MIR) pass. The IR pass conceptually redefines pointers into eMPAC fat pointers. Also, it identifies pointer loads and stores, attaches metadata needed for later instrumentation, and then propagates it to the MIR pass. The MIR pass is in charge of performing actual instrumentation on each load and store instruction to enforce eMPAC's PA-based fat pointer authentication scheme. Our implementation is based on the toolchain for our target board (Renesas EK-RA8D1 [36]) that is a modified version of LLVM 18.1.0 [7].

4.1 Fine-grained typeID generation in Clang frontend

eMPAC's frontend component is responsible for computing fine-grained typeID from source-level type information. PARTS [27] relied on LLVM's typed-pointer model to recover pointee types. However, PARTS' approach is no longer feasible because LLVM has moved on to the opaque-pointer model [34] since LLVM 17 [7] and ARMv8.1-M toolchains require a minimum LLVM version of 17. For this reason, we made a design decision to perform typeID calculation in the frontend. eMPAC computes typeIDs by resolving source-level pointee object types to their canonical form and hashing them with per-compilation salt during IR generation. These typeID metadata entries are then attached directly onto the emitted IR instructions and global variable declarations, making them available to the downstream instrumentation pass.

4.2 LLVM IR Pass

The IR pass operates at a level where rich type, sharing, and pointer semantics information is still available, before instruction selection discards it. It exploits this visibility to classify every pointer operation and attach metadata that the subsequent MIR pass consumes to emit correct instrumentation.

IR-Level pointer redefinition. Pointer semantics redefinition is achieved by defining a new *named struct type*, `%FatPtr`, whose contents are described as `{ptr, i32}` at the IR level. The pass identifies all pointers residing in aggregate types (e.g., structs, arrays) in order to redefine the pointer types with eMPAC's fat pointer type. The redefinition is performed comprehensively to handle all sources of pointer allocation, including global variables, `alloca` sites (i.e., stack variables), and `malloc` and its variants. Notably, eMPAC adjusts the

size argument to `malloc` calls according to the identified number of pointers included in the aggregate type, to compensate for the increase in the type's size due to fat pointers. eMPAC's IR pass also handles nested aggregate types recursively. Pointers in global variables that are directly loaded from the `.data` segment lack PACs and therefore must be handled separately, as discussed in prior work [27,12]. eMPAC adopts a similar approach, in which PAC initializers for global variables are synthesized by the compiler and executed before `main()`.

Identifying and propagating instrumentation targets. The IR pass attaches metadata to all pointer load and store instructions such that the later MIR pass can instrument them accordingly. The pass traverses all functions in the module, collecting pointer-typed load and store instructions and annotating them with metadata keys such as `!Ptr.load`, `!Ptr.store`, and `!Ptr.shared`, where shared is determined through an interprocedural pointer analysis [2] that tracks whether a pointer value originates from or flows through globally-accessible memory. Additionally, pointer variables annotated in source code with inter-task communication type identifiers have their typeIDs propagated along the use-def chain to all reachable load instructions, marking each with the domain they belong to. The metadata, besides instructing the MIR pass to instrument them, describes if the instruction is 1) an inter-task pointer load/store or 2) a system-wide global variable access. In turn, the MIR pass can apply matching taskID as described in §3.3.

4.3 LLVM MIR Pass

The MIR level is where the materialization into concrete architecture-specific instructions occurs, and the MIR pass bridges the gap between semantic-rich IR instructions and benign MIR `load`/`store` instructions. The primary objective of the MIR pass is to employ the metadata emitted from the IR level pass to appropriately assemble sequences of modifier-constructing and sign/authenticating instructions.

Load/store instrumentation. First, the pass identifies `load`/`store` instructions and queries a helper that returns the corresponding IR instruction using the debug location. Then, the available metadata is used to determine whether the `load`/`store` instruction targets pointers that require instrumentation. Eligible instructions continue to be processed to construct a correct series of machine instructions, which can be classified into three types. If they target task-local pointers, they must be derived from the $\mathrm{Reg}_{\mathrm{task}}$, and globals must have the global ID G encoded in the instructions. Pointers shared between tasks are assigned a domain ID `T1T2` for each domain and must be encoded as well. G and `T1T2` are randomly generated at this stage. For all three cases, the typeID component of the modifier can be derived from the typeID metadata from the IR pass, and encoded in the machine instruction.

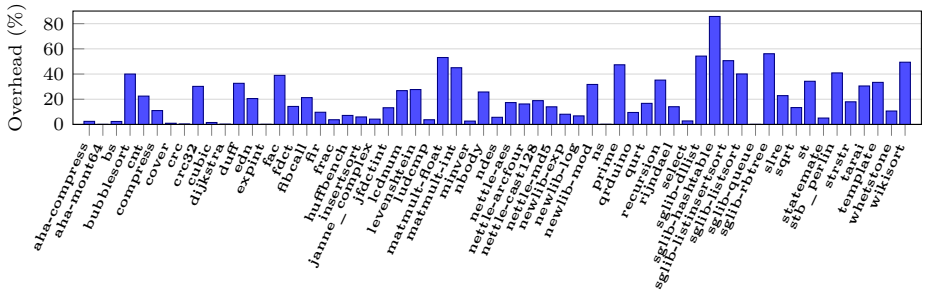

Fig. 3: BEEBS benchmark suite performance normalized to baseline.

5 Evaluation

We evaluated eMPAC with FreeRTOS V10.6.1 [15] running on a Renesas EK-RA8D1 [36] with a PA-M-capable 480 MHz Cortex-M85, 2 MB Code Flash, and 1 MB SRAM. The evaluation covers performance on a general benchmark suite, BEEBS [33] (§5.1), performance on a real-world web server application (§5.2), and code and memory footprint (§5.3).

5.1 Benchmark suite measurements

We used the BEEBS benchmark suite to evaluate the general performance profile of eMPAC, as BEEBS has been widely used to assess embedded system performance in prior works [38,18,44,39]. Figure 3 shows the measured performance results from 1,000 repetitions, normalized to the baseline system (without instrumentation). The average overhead across 61 benchmarks in the suite was 20.62%. The observed overhead characteristics are as expected; compute-heavy cases such as `cubic` (1.54%) and `frac` (3.74%) show almost negligible slowdowns. On the other hand, the highest overhead was observed in memory-access-intensive workloads, such as `sglib-hashtable` (85.70%) and `sglib-rbtree` (56.12%). Through manual analysis, we identified that the overhead arises from loops with only a small number of instructions and a high concentration of pointer load/stores. Overall, the average overhead was on a level comparable to prior work that leveraged PA-A [27,12,30].

5.2 Real-world application benchmark

To understand the real-world performance of eMPAC, we conducted an experiment using Mongoose [31], a full-fledged HTTPS web server, as shown in Figure 4. The board manufacturer provides the Ethernet peripheral drivers, the TCP/IP stack is provided by FreeRTOS, and our entire firmware and OS run on 6 tasks created by the OS, which are switched frequently during execution. The application tests various aspects of eMPAC's performance impact on the system: establishing a TCP connection, performing a TLS handshake, parsing

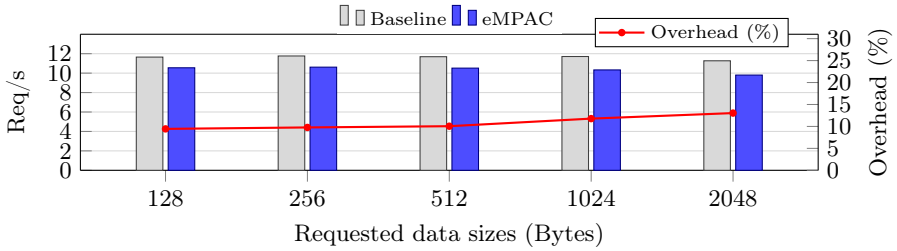

Fig. 4: Web server benchmark results

HTTP requests, and performing kernel-side operations, including packet transmission. Hence, the application should experience very frequent fat pointer authentication across the application and the kernel.

Performance overhead. We measured the *requests processed per second* for baseline (unmodified software) and the eMPAC version of the web server and FreeRTOS. Each request was made individually; each request establishes a connection, negotiates TLS, receives the data, and then disconnects. This way, the overhead from eMPAC is more clearly shown in the results. We measured requests per second for both baseline and eMPAC, averaging over 100 accesses. The average overhead across all data sizes was approximately 10.81%. The overhead increases by up to 13.04% as the serving data size grows. The observed overhead is as expected and consistent with a similar web server experiment conducted on the A-series ARM processors. Although a direct comparison is infeasible, Capacity [12] reported an average of approximately 17% across the {128, 256, 512, 1024, 2048} data size interval, suggesting that this level of overhead is expected even with the more powerful A-series processors.

5.3 Code and Memory Footprint

We measured code and global footprints from binary section sizes (`.text` and `.data+.bss`) and from stack/heap usage measured at runtime.

BEEBS benchmark suite. Table 2 shows that average code size (`.text`) increases by 102.1% (9.04 KB → 18.27 KB) due to load/store instrumentation. The widened pointer size from 32-bit to 64-bit affects data sections as well as runtime memory usage; `.data+.bss` showed a 40.89% increase (2.03 KB → 2.86 KB). Also, runtime memory usage increased by 13.46% on average (0.52 KB → 0.59 KB). Note that we measured the *highest* observed stack memory usage for BEEBS because it does not use dynamic memory.

Mongoose web server. For the Mongoose web server as shown in Table 2, code size increases by 75.1% (869 → 1,522 KB), while global memory and heap remain nearly unchanged (518 → 521 KB, 0.06%; 107 → 108 KB, 0.9%). Compared to BEEBS, this indicates stronger amortization in a larger real-world software stack, where instrumentation costs contribute less outside `.text`. Overall, despite noticeable code growth, the full firmware still fits within the platform's limits, staying within the 2 MB flash and 1 MB SRAM budgets.

Table 2: Section-wise memory footprint. All sizes are in KB.

Binary	Code (`.text`)			Global (`.data + .bss`)			Runtime Mem.		
	Base.	Inst.	Incr (%)	Base.	Inst.	Incr (%)	Base.	Inst.	Incr (%)
BEEBS (Avg.)	9.04	18.27	102.1	2.03	2.86	40.89	0.52	0.59	13.46
Web server	869	1522	75.1	518	521	0.06	107	108	0.9

6 Discussion

In this section, we discuss the security of eMPAC's design, its current limitations, and future work.

6.1 Security analysis

Here, we provide a point-by-point security analysis of potential attacks that the adversary may launch. More specifically, eMPAC's security model must uphold the same level of security achieved by previous works implemented using PA-A.

PA key leak or modification. The adversary may attempt to leak or modify the singular PA key register, the source of all entropy in PA-M. In fact, the lack of user-kernel privilege separation [29] in MCUs makes this attack vector highly plausible and dangerous. Specifically, the adversary may launch code-reuse attacks to abuse the PA key management instructions `msr` and `mrs`. However, eMPAC's implementation specifically mitigates such attacks by erasing these instructions from memory after key register initialization during the boot sequence.

Pointer reuse. An adversary with arbitrary memory read capability may capture a valid signed fat pointer from memory and attempt to reuse it elsewhere. The possibility of such a replay attack is an inherent attack vector in cryptographic pointer protection in PA, and is to be mitigated through context diversification as discussed in previous works [27,12]. eMPAC heavily constrains such replay through its modifier security model despite the limitations of PA-M. For instance, in our webserver case study, the program spans 6 RTOS tasks and 454 distinct pointer types, yielding 2,724 possible task–type contexts. A stolen pointer is therefore valid in only one of these contexts, leaving the adversary a narrow, context-specific window for reuse.

Pointer forgery. An adversary may attempt to forge a fat pointer to an arbitrary address by invoking a `pacg` gadget with attacker-chosen modifiers harvested from instruction encodings and spilled register contents. eMPAC blocks this path on two fronts. First, reserving $\mathrm{Reg}_{\mathrm{mod}}$ and $\mathrm{Reg}_{\mathrm{task}}$ sharply contracts the usable gadget space: instructions that write these registers appear only within signing/authentication sequences ($\mathrm{Reg}_{\mathrm{mod}}$) and scheduler code ($\mathrm{Reg}_{\mathrm{task}}$), and the surrounding code offers little expressiveness for an attacker to repurpose. Second, taskID isolation prevents forgery from crossing task boundaries in either direction. Producing a pointer that authenticates in a victim task requires that task's $\mathrm{Reg}_{\mathrm{task}}$, which is installed exclusively by the scheduler on context switch and is never exposed to attacker-reachable code; conversely, a pointer

forged under one task's Reg_{task} fails authentication the moment it is dereferenced from another task. A forgery is therefore confined to the task in which it was mounted, and eMPAC realizes the *domain isolation* property for PA first articulated by Capacity [12]. Moreover, BTI further narrows the code-reuse surface by forbidding entry into any instruction other than a function's designated entry point.

Nevertheless, prior work discussed the possibility of function-granular code reuse to achieve pointer forgery [42]. That is, there may exist a function in the program that naturally produces a pointer signed with the adversary-desired modifier into an adversary-controlled region. However, as discussed in that work, such functions are rare. Additionally, the authors provide a static validator that can identify such gadgets. We leave incorporating the tool as a validator for eMPAC-produced binaries to future work.

Brute-forcing PACs. Finally, PA is fundamentally an entropy-based defense, and hence an adversary may be able to brute-force a pointer authentication. In rare circumstances, the adversary may be able to brute-force without crashing the entire system. For example, a crash resulting from an unsuccessful pointer authentication may affect only a single thread without causing side effects. While these attacks are a fundamental limitation of PA in general, eMPAC uses 32-bit PACs, which provide greater entropy than PA-A's PAC sizes of 11, 15, or 24 bits.

Exceptional cases: hardware-defined pointers. A limited number of hardware-defined structures, notably the interrupt vector table and DMA peripheral instance descriptors, cannot be migrated to the fat-pointer representation because they are consumed by hardware and therefore require a fixed binary layout. The vector table resides in read-only flash by default, so exempting it from instrumentation poses no risk. For DMA descriptors, we manually modified the kernel code to store their PAC values in a separate PAC table.

6.2 Handling of Type-casting and Arithmetic of Pointers

It is imperative to maintain correctness of eMPAC even in cases of type or value alteration of pointers. Type-casting authenticates the pointer with previous typeID and resigns with the new typeID, similar to how PARTS [27] handles pointer conversion. Pointer arithmetic such as `ptr++` translates to storing a new value to the pointer in LLVM IR, which eMPAC's LLVM IR pass instruments to resign the old pointer with the new value after authenticating with the old value.

6.3 Limitation and future work

The following are the current limitations and future work that we plan to explore. First, eMPAC does not currently provide real-time guarantees. Although our implementation targets FreeRTOS, a widely used real-time operating system, we have not validated whether the instrumentation overhead satisfies hard

real-time deadline requirements. Our choice of RTOS was driven by the limited OS support available for the recently released Cortex-M85 board, rather than a specific intention to target real-time systems. We consider further optimizing eMPAC for real-time workloads as a worthwhile direction for future work. Second, eMPAC requires whole-program instrumentation, so uninstrumented binary-only libraries or drivers will not correctly handle fat pointers. In practice, MCU firmware is typically compiled from source with a board-specific toolchain, making it reasonable to compile all components with the eMPAC compiler.

7 Conclusion

We presented eMPAC, a compiler and ABI extension to ARMv8.1-M's pointer authentication that enables full-coverage pointer authentication and task isolation in MCU systems. eMPAC retrofits the limited PA-M primitives through a fat pointer design and modifier-based diversification, achieving fine-grained pointer integrity and task-level domain isolation. Our evaluation showed that eMPAC enables such security schemes with moderate overhead.

Acknowledgments. This work was supported by grants funded by the Korean government: Institute of Information & Communications Technology Planning & Evaluation (IITP) grants (RS-2022-II220688, RS-2026-25525457, RS-2024-00439819, RS-2024-00437306, RS-2023-00259497, RS-2024-00437849)

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Almakhdhub, N.S., Clements, A.A., Bagchi, S., Payer, M.: μRAI: Securing Embedded Systems with Return Address Integrity. In: Network and Distributed Systems Security (NDSS) Symposium (2020)
2. Andersen, L.O.: Program analysis and specialization for the C programming language. PhD Thesis, University of Copenhagen (1994)
3. Apple: Apple Platform Security: Operating System Security. https://support.apple.com/guide/security/operating-system-integrity-sec8b776536b/web (2024)
4. Arm: Armv8.1-M PACBTI Extensions, https://developer.arm.com/documentation/107976/22-1-0/Clang-reference/Other-compiler-specific-features/Product-architecture-features/Armv8-1-M-PACBTI-extension, official Arm documentation
5. Arm: Arm architecture reference manual armv8, for armv8-a architecture profile. https://developer.arm.com/documentation/ddi0487/ga (2021)
6. Arm: Armv8.1-M Pointer Authentication and Branch Target Identification Extension. https://developer.arm.com/community/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension (2021)

7. ARM-software: LLVM-embedded-toolchain-for-Arm. https://github.com/ARM-software/LLVM-embedded-toolchain-for-Arm/ (2024)
8. Avanzi, R.: The QARMA block cipher family – almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. Cryptology ePrint Archive, Paper 2016/444 (2016), https://eprint.iacr.org/2016/444
9. Brasser, F., El Mahjoub, B., Sadeghi, A.R., Wachsmann, C., Koeberl, P.: TyTAN: tiny trust anchor for tiny devices. In: Proceedings of the 52nd Annual Design Automation Conference. DAC '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2744769.2744922, https://doi.org/10.1145/2744769.2744922
10. Cai, Z., Zhu, J., Shen, W., Yang, Y., Chang, R., Wang, Y., Li, J., Ren, K.: Demystifying Pointer Authentication on Apple M1. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 2833–2848. USENIX Association, Anaheim, CA (Aug 2023), https://www.usenix.org/conference/usenixsecurity23/presentation/cai-zechao
11. Clements, A.A., Almakhdhub, N.S., Bagchi, S., Payer, M.: ACES: Automatic Compartments for Embedded Systems. In: USENIX Security Symposium (2018), https://api.semanticscholar.org/CorpusID:46939467
12. Dinh Duy, K., Cho, K., Noh, T., Lee, H.: Capacity: Cryptographically-Enforced In-Process Capabilities for Modern ARM Architectures. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. p. 874–888. CCS '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3576915.3623079, https://doi.org/10.1145/3576915.3623079
13. Du, Y., Shen, Z., Dharsee, K., Zhou, J., Walls, R.J., Criswell, J.: Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 2281–2298. USENIX Association, Boston, MA (Aug 2022), https://www.usenix.org/conference/usenixsecurity22/presentation/du
14. Free Software Foundation: Using the gnu compiler collection (gcc): Arm options. https://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html
15. FreeRTOS™: FreeRTOS. https://freertos.org, https://freertos.org
16. Ismail, M., Quach, A., Jelesnianski, C., Jang, Y., Min, C.: Tightly seal your sensitive pointers with PACTight. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 3717–3734 (2022)
17. Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., Wang, Y.: Cyclone: A Safe Dialect of C. In: 2002 USENIX Annual Technical Conference (USENIX ATC 02). USENIX Association, Monterey, CA (Jun 2002), https://www.usenix.org/conference/2002-usenix-annual-technical-conference/cyclone-safe-dialect-c
18. Kang, J., Park, J., Seo, J., Kwon, D.: Look before you access: Efficient heap memory safety for embedded systems on armv8-m. In: Proceedings of the 61st ACM/IEEE Design Automation Conference. DAC '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3649329.3655949, https://doi.org/10.1145/3649329.3655949
19. Khan, A., Xu, D., Tian, D.J.: EC: Embedded Systems Compartmentalization via Intra-Kernel Isolation. In: 2023 IEEE Symposium on Security and Privacy (SP). p. 2990–3007 (May 2023). https://doi.org/10.1109/SP46215.2023.10179285, https://ieeexplore.ieee.org/document/10179285, 3 citations (Semantic Scholar/DOI) [2024-05-24]

20. Kim, C.H., Kim, T., Choi, H., Gu, Z., Zhang, X., Xu, D.: Securing Real-Time Microcontroller Systems through Customized Memory View Switching. In: Proceedings of the 25th Network and Distributed System Security Symposium (NDSS 2018). San Diego, CA (Feb 2018). https://doi.org/10.14722/ndss.2018.23107, http://doi.org/10.14722/ndss.2018.23107
21. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. ACM SIGARCH Computer Architecture News **42**(3), 361–372 (2014)
22. Koeberl, P., Schulz, S., Sadeghi, A.R., Varadharajan, V.: TrustLite: a security architecture for tiny embedded devices. In: Proceedings of the Ninth European Conference on Computer Systems. EuroSys '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2592798.2592824, https://doi.org/10.1145/2592798.2592824
23. Kong, Z., Park, M., Guan, L., Zhang, N., Kim, C.H.: TZ-DATASHIELD: Automated Data Protection for Embedded Systems via Data-Flow-Based Compartmentalization. Network **15**, 16 (2025)
24. Lentz, M., Sen, R., Druschel, P., Bhattacharjee, B.: SeCloak: ARM Trustzone-based Mobile Peripheral Control. In: Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services. pp. 1–13 (2018)
25. Li, Y., Tan, W., Lv, Z., Yang, S., Payer, M., Liu, Y., Zhang, C.: PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 1901–1915 (2022)
26. Liljestrand, H., Nyman, T., Ekberg, J.E., Asokan, N.: PACStack: an Authenticated Call Stack. In: Proceedings of the 56th Annual Design Automation Conference 2019. pp. 1–2 (2019)
27. Liljestrand, H., Nyman, T., Wang, K., Perez, C.C., Ekberg, J.E., Asokan, N.: PAC it up: Towards pointer integrity using ARM pointer authentication. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 177–194. USENIX Association, Santa Clara, CA (Aug 2019), https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand
28. LLVM Project: Pointer authentication in llvm. https://llvm.org/docs/PointerAuth.html
29. Ma, Z., Liu, G., Eastman, A., Kaufman, K., Armanuzzaman, M., Tan, X., Jesse, K., Walls, R., Zhao, Z.: " we just did not have that on the embedded system": Insights and challenges for securing microcontroller systems from the embedded ctf competitions. arXiv preprint arXiv:2503.08053 (2025)
30. McKee, D.P., Giannaris, Y., Ortega, C., Shrobe, H.E., Payer, M., Okhravi, H., Burow, N.: Preventing Kernel Hacks with HAKCs. In: NDSS. pp. 1–17 (2022)
31. Mongoose: Mongoose web server. https://mongoose.ws/
32. Necula, G.C., McPeak, S., Weimer, W.: CCured: type-safe retrofitting of legacy code. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 128–139. POPL '02, Association for Computing Machinery, New York, NY, USA (2002). https://doi.org/10.1145/503272.503286, https://doi.org/10.1145/503272.503286
33. Pallister, J., Hollis, S., Bennett, J.: BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms (2013), https://arxiv.org/abs/1308.5174
34. Project, L.: Opaque pointers. https://llvm.org/docs/OpaquePointers.html

35. Ravichandran, J., Na, W.T., Lang, J., Yan, M.: PACMAN: attacking ARM pointer authentication with speculative execution. In: Proceedings of the 49th Annual International Symposium on Computer Architecture. p. 685–698. ISCA '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3470496.3527429, https://doi.org/10.1145/3470496.3527429
36. Renesas Electronics Corporation: Renesas EK-RA8D1 Evaluation Kit. https://www.renesas.com/en/design-resources/boards-kits/ek-ra8d1 (2024)
37. Sun, Z., Feng, B., Lu, L., Jha, S.: OAT: Attesting Operation Integrity of Embedded Devices. In: 2020 IEEE Symposium on Security and Privacy (SP). p. 1433–1449 (May 2020). https://doi.org/10.1109/SP40000.2020.00042, https://ieeexplore.ieee.org/abstract/document/9152803
38. Tan, X., Zhao, Z.: SHERLOC: Secure and Holistic Control-Flow Violation Detection on Embedded Systems. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. p. 1332–1346. CCS '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3576915.3623077, https://doi.org/10.1145/3576915.3623077
39. Wang, Y., Lemieux-Mack, C., Chantem, T., Baruah, S., Zhang, N., Ward, B.C.: Partial Context-Sensitive Pointer Integrity for Real-time Embedded Systems. In: 2024 IEEE Real-Time Systems Symposium (RTSS). pp. 415–426. IEEE (2024)
40. Wang, Y., Mack, C.L., Tan, X., Zhang, N., Zhao, Z., Baruah, S., Ward, B.C.: InsectACIDE: Debugger-Based Holistic Asynchronous CFI for Embedded System. In: 2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 360–372 (2024). https://doi.org/10.1109/RTAS61025.2024.00036
41. Watson, R.N., Woodruff, J., Neumann, P.G., Moore, S.W., Anderson, J., Chisnall, D., Dave, N., Davis, B., Gudka, K., Laurie, B., Murdoch, S.J., Norton, R., Roe, M., Son, S., Vadera, M.: Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In: 2015 IEEE Symposium on Security and Privacy. pp. 20–37 (2015). https://doi.org/10.1109/SP.2015.9
42. Yoo, S., Park, J., Kim, S., Kim, Y., Kim, T.: In-Kernel Control-Flow integrity on commodity OSes using ARM pointer authentication. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 89–106. USENIX Association, Boston, MA (Aug 2022), https://www.usenix.org/conference/usenixsecurity22/presentation/yoo
43. Zhou, J., Criswell, J., Hicks, M.: Fat Pointers for Temporal Memory Safety of C. Proc. ACM Program. Lang. **7**(OOPSLA1) (Apr 2023). https://doi.org/10.1145/3586038, https://doi.org/10.1145/3586038
44. Zhou, J., Du, Y., Shen, Z., Ma, L., Criswell, J., Walls, R.J.: Silhouette: Efficient Protected Shadow Stacks for Embedded Systems. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 1219–1236. USENIX Association (Aug 2020), https://www.usenix.org/conference/usenixsecurity20/presentation/zhou-jie
45. Zhou, W., Jiang, Z., Guan, L.: Understanding mpu usage in microcontroller-based systems in the wild. In: Proceedings 2023 Workshop on Binary Analysis Research. San Diego, CA, USA: Internet Society (2023)
46. Ziad, M.T.I., Arroyo, M.A., Manzhosov, E., Kemerlis, V.P., Sethumadhavan, S.: Epi: Efficient pointer integrity for securing embedded systems. In: 2021 International Symposium on Secure and Private Execution Environment Design (SEED). pp. 163–175. IEEE (2021)