



Intellectual Property Exposure: Subverting and Securing Intellectual Property Encapsulation in Texas Instruments Microcontrollers

Marton Bognar, Cas Magnus, Frank Piessens, Jo Van Bulck

DistriNet, KU Leuven, 3001 Leuven, Belgium

Abstract

In contrast to high-end computing platforms, specialized memory protection features in low-end embedded devices remain relatively unexplored despite the ubiquity of these devices. Hence, we perform an in-depth security evaluation of the state-of-the-art Intellectual Property Encapsulation (IPE) technology found in widely used off-the-shelf, Texas Instruments MSP430 microcontrollers. While we find IPE to be promising, bearing remarkable similarities with trusted execution environments (TEEs) from research and industry, we reveal several fundamental protection shortcomings in current IPE hardware. We show that many software-level attack techniques from the academic TEE literature apply to this platform, and we discover a novel attack primitive, dubbed *controlled call corruption*, exploiting a vulnerability in the IPE access control mechanism. Our practical, end-to-end attack scenarios demonstrate a complete bypass of confidentiality and integrity guarantees of IPE-protected programs.

Informed by our systematic attack study on IPE and root-cause analysis, also considering related research prototypes, we propose lightweight hardware changes to secure IPE. Furthermore, we develop a prototype framework that transparently implements software responsibilities to reduce information leakage and repurposes the onboard memory protection unit to reinstate IPE security guarantees on currently vulnerable devices with low performance overheads.

1 Introduction

Memory isolation is a fundamental building block for security, safety, and privacy in computing systems; it protects secret keys, personal data, intellectual property, and other sensitive information. On high-end systems, the operating system provides isolation using established hardware features such as memory management units (MMUs) and CPU privilege levels. Recent years have also seen the rise of trusted execution environments (TEEs) shipped by major processor vendors, such as Intel SGX [13] and TDX [29], AMD SEV [5], and

ARM TrustZone [4], which provide hardware-backed isolation even in the presence of a compromised operating system. Still, low-end embedded devices, omnipresent in the Internet of things (IoT), often lack established memory isolation primitives [26] due to stringent demands on manufacturing costs and power consumption. In response to these unique challenges, a line of specialized, embedded TEE research prototypes [9, 15, 18, 32, 41, 42] has been developed. However, these TEE designs all require custom hardware changes and are, hence, incompatible with off-the-shelf IoT microcontrollers that often only offer a coarse-grained and error-prone memory protection unit (MPU) [24].

There is a clear demand for specialized IoT memory protection mechanisms, as also evidenced by the inclusion of diverse hardware security features with varying degrees of sophistication by embedded device manufacturers [4, 6, 10, 36–38, 48, 51, 52]. In this paper, we study an advanced example, the Intellectual Property Encapsulation (IPE) technology found in some popular, ultra-low-power MSP430 microcontrollers by Texas Instruments (TI). Interestingly, we find that IPE holds remarkable similarities to general-purpose TEEs in that it offers enclave-like, hardware-level isolation of arbitrary code and data, protecting from read or write access from *anywhere* outside of the encapsulated memory area, including the embedded operating system and even JTAG hardware debuggers [55, 61]. Unlike academic prototypes, IPE technology has been readily available in off-the-shelf devices since 2014 [55, 58], predating most academic works and several commercial technologies. IPE technology has been advertised to protect high-value proprietary code and sensitive data such as cryptographic keys [57, 60, 61]. Moreover, IPE has been used as a critical enabling technology for several recent academic security systems [17, 33, 34]. However, despite IPE’s potential and widespread availability, no critical analysis has yet been conducted to properly understand the guarantees and limitations of this security technology.

This paper performs such an analysis: we study TEE attack literature and comparable research prototypes [15, 18, 41, 42] to assess the security level offered by off-the-shelf IPE tech-

nology against software adversaries with arbitrary code execution on the device. Our findings paint a promising but currently disappointing picture. The lack of clear documentation makes it difficult to identify the exact security guarantees and attacks prevented by IPE. Furthermore, inadequate compiler toolchain support considerably increases the burden on developers and the likelihood of introducing unintentional interface sanitization vulnerabilities [69].

Most concerningly, we discover several hardware shortcomings that make it impossible to securely isolate application code and data with IPE on current devices. First, we discover *controlled call corruption*, a novel critical vulnerability in IPE’s memory access control logic, allowing an attacker-controlled `call` instruction to overwrite arbitrary protected memory locations. In our root-cause analysis, we attribute this behavior to the pipelined CPU architecture and show that at least two related open-source TEE implementations [41, 42] took measures to avoid similar flaws in their access control logic. Second, we show how the lack of enforcing a single entry point into the encapsulated IPE area, standard in comparable single-address-space TEE designs [13, 18, 32, 41, 42], opens the door to code-reuse attacks [35, 49]. Third, we demonstrate that IPE’s lack of enforcing atomic execution [18, 41, 42] or protecting the register state [13, 15, 30, 32, 71] upon interrupts allows to leak or modify CPU registers at arbitrary points during execution. In practical proof-of-concept demonstrations, we show how these attack primitives – individually and in potent combinations – allow us to bypass the IPE security objectives, i.e., confidentiality and integrity of encapsulated code and data.

Finally, we challenge assumptions [8, 16, 75] that deterministic MSP430 platforms would be less vulnerable to microarchitectural side-channel attacks. Specifically, inspired by attack research on popular high-end TEEs, including Intel SGX and AMD SEV, we experimentally show how IPE software adversaries can induce measurable cache timing differences [39] by manipulating CPU clock speed, reconstruct instruction timings via interrupt latency [71], and mount novel controlled-channel attacks [25, 77] through the MPU.

Building on our comprehensive attack analysis, we propose both hardware and software countermeasures. First, inspired by research prototypes [15, 18, 41, 42] on the open-source openMSP430 [20] processor, we formulate concrete refinements for IPE’s hardware-level memory access control logic. Next, we contribute a software framework, inspired by established TEE software development kits [28, 41, 69], to improve on TI’s rudimentary toolchain [61] by using a source-to-source translator and hardened entry and exit assembly stubs to automatically transform a developer-annotated C program and transparently safeguard IPE software transitions. Moreover, we extend this framework to secure currently vulnerable devices against our attacks by repurposing the onboard MPU to overlay an additional layer of fine-grained, execution-aware protection over the IPE memory area, reinstating the security

guarantees of IPE against software adversaries (without JTAG access) with minimal performance and code size impact.

Contributions. In summary, our main contributions are:

- Analyzing the security guarantees offered by IPE technology in off-the-shelf TI MSP430 microcontrollers, uncovering important vulnerabilities in current hardware, including the novel *controlled call corruption* attack.
- Showing that side channels from prior research can be adapted to leak information from vulnerable IPE code.
- Building practical, software-exploitable attack primitives and end-to-end attack demonstrations that completely bypass IPE confidentiality and integrity guarantees.
- Contributing hardware and software mitigations to improve security, including a software framework that transparently sanitizes IPE programs and mitigates our attacks on current hardware by utilizing the MPU.

Responsible disclosure. We disclosed our findings, including proof-of-concept exploits, to TI on April 20, 2023. After thorough analysis and successful reproduction of our attacks, TI issued a security advisory [65] for the *controlled call corruption* with a CVSS score of 7.1 (high). TI also fed back our findings to their internal design teams to ensure improved security practices for future devices and included our MPU approach as a suggested mitigation for current devices. TI currently considers the other issues out of scope, defining the goal of IPE as blocking the direct readout of encapsulated memory via the CPU or JTAG and not as preventing an attacker from indirectly inferring code or secrets. In this paper, we clearly show, however, that such indirect exposure is practical using our attack primitives.

Availability. To facilitate future research on IPE, our source code, including our attacks, the software framework, and the evaluation, is available at <https://github.com/martonbogvar/ipe-exposure/>.

2 Background and problem statement

2.1 MSP430 microcontrollers

The widely used [19] line of MSP430 microcontrollers, developed by Texas Instruments, is targeted at low-cost, ultra-low-power applications, such as vehicle ECUs and key fobs [56]. These microcontrollers feature a single address space without virtual memory or processor privilege levels and exhibit deterministic timing behavior, allowing their use in real-time systems. Over time, the 16-bit MSP430 instruction set and the devices received several extensions, such as an address space extension to 20 bits, direct memory access (DMA) capabilities, and security features such as an AES accelerator and an elementary MPU. More recently, TI has shipped millions of

MSP430 microcontrollers with non-volatile FRAM memory technology [58, 59], equipped with the IPE security feature described in the following subsection.

Memory protection unit (MPU). The most rudimentary mechanism to enforce isolation on the FRAM MSP430 devices is the MPU [55]. It allows partitioning the address space into three regions at 1 kB boundaries with separate access control permissions for each region. The MPU configuration registers are protected by fixed “passwords”, allowing to safeguard the memory partitions against accidental read, write, or execute accesses originating from code, DMA, and JTAG. The access control of the MPU is not program-counter-dependent, and while setting a configuration bit makes its configuration immutable, a device reset deactivates the protection entirely.

Research TEEs on openMSP430. Supported by the simplicity and popularity of the MSP430 instruction set, and the availability of the near-cycle-accurate openMSP430 [20] open-source software implementation, the MSP430 platform has been an attractive target for developing academic prototypes, among others TEEs [15, 18, 41, 42]. These systems employ a lightweight, program-counter-based access control mechanism [53] to seclude a contiguous memory region from external software accesses: additional hardware-level circuitry enforces that only when the program counter is inside the protected address range, access to protected memory is allowed.

2.2 Intellectual Property Encapsulation (IPE)

The security feature studied in this work is Intellectual Property Encapsulation, widely available since 2014 [58] in the MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx families of TI microcontrollers. In short, IPE isolates a region of the address space from all outside accesses. In the following, we elaborate on IPE’s design and precise security features based on several technical documents [54, 55] and whitepapers [57, 61, 64] released by TI, anticipating and contextualizing some of the attack primitives explored in depth in Section 3. Given the striking resemblance of IPE to (embedded) TEE designs, we structure the discussion based on requirements explicitly identified to realize enclave isolation [53].

Memory isolation. The most fundamental building block for trusted execution is that untrusted code should not be able to read or modify the data of the protected module. To this end, IPE includes hardware-level access control logic, schematized in Figure 1, that is strikingly similar to the program-counter-based access control mechanism [53] used in many embedded TEEs [18, 32, 41, 42]. The device user’s guide [55] explains unambiguously that “*only program code executed from the IPE-segment can access data stored in this segment*” and multiple TI documents [57, 60, 61] explicitly recommend using IPE to protect secret keys and configuration data. One

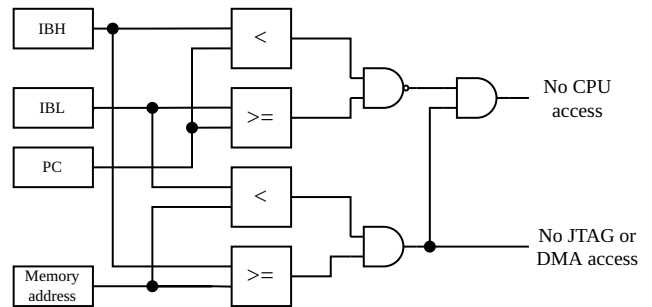


Figure 1: IPE access control logic [55]. IBH and IBL refer to the high and low boundaries of the IPE region.

important difference to many other TEEs is that, as hinted by the name, *Intellectual Property* Encapsulation, this technology is meant to protect code in addition to data. Hence, IPE protects a unified contiguous address range for both code and data, whereas research prototypes commonly separate executable code and (non-executable) data. Finally, IPE is limited to isolating a single “enclave”, similar to simpler designs [11, 15, 18, 42], whereas more advanced TEEs [32, 41] provide support for multiple, mutually distrusting enclaves.

Two explicitly documented [55] exceptions for the protection are that IPE cannot protect the fixed memory locations for the interrupt vector table (IVT) and that the first 8 bytes of IPE memory are reserved for a configuration data structure, called `ipe_init_struct`, and, hence, code executed from here cannot access the rest of the IPE region. This structure stores the boundaries of the IPE region along with other settings and a checksum, and is used by the boot process, explained next.

Minimal trusted computing base (TCB). The compelling feature of TEEs is that their security guarantees only rely on the correct functioning of the enclave code itself, the hardware, and possibly part of the boot process, not on the operating system or any other privileged software on the platform.

In line with this definition, IPE only requires trusting the hardware and a minimal part of the boot process. On startup, an immutable piece of trusted TI firmware (“bootcode”) initializes the IPE region boundary registers by reading the `ipe_init_struct`. This process allows IPE software to dynamically reconfigure its boundaries across resets. However, `ipe_init_struct` also includes an odd-bit-interleaved parity checksum, such that the bootcode can detect any accidental corruptions and erase all memory to avoid unintentional exposure of IPE secrets. Crucially, once the trusted bootcode finishes, the IPE boundary hardware registers are locked, and the processor’s program-counter-based access control logic protects the IPE region from *any* later direct accesses, including from the privileged MSP430 bootstrap loader (BSL), which was previously leveraged to extract firmware [21, 22].

While IPE does not explicitly support software attestation, commonly found in TEEs, the trusted bootcode can be considered an effective root of trust. That is, after IPE is first initialized in a secure environment, e.g., in the factory or before deploying the microcontroller to end users, stakeholders can assume that across system reboots the IPE code and data will remain persistent (via the non-volatile FRAM technology) and protected (via IPE).

Controlled entry points. If the protected module can be invoked at any location, its behavior will become unpredictable, posing a risk of inadvertent secret leakage through *code-reuse attacks* [49]. To avoid this issue, invocation should only happen at defined entry functions, as implemented in embedded [15, 18, 32, 41, 42] and high-end TEEs [13].

We found that TI’s documentation is unclear about restricting execution to defined entry points. The device user’s guide [55] writes: “*to execute code from the IPE-segment, branch into that segment or call functions stored in that segment*”. At least one document [54] describes IPE as protecting against “*any read, write, or execute access*”, implying that execute access can be disabled for part of the IPE region. Of course, completely disabling execute permissions would make the IPE section useless, as no code in the module could be invoked, which also means that the protected data could never be accessed. Another TI white paper [64] used to hint more explicitly towards defined entry points: “*execution of this portion of memory can be limited to specific callback functions that are defined at the time the IPE module is enabled*”.¹ However, we experimentally demonstrate in Section 3.2 that the current IPE hardware access control logic does *not* enforce entry points, giving rise to capable code-reuse attacks.

Interrupts and real-time guarantees. To prevent the corruption or leakage of CPU register values, some embedded TEE designs disable interrupts during enclaved execution [18, 41, 42, 53]. However, it is important to note that embedded devices are often used for real-time, safety-critical applications where timely handling of interrupts is crucial, and allowing enclaves to arbitrarily hold on to the CPU may be unacceptable. Thus, more advanced TEEs like Intel SGX and many embedded research prototypes [2, 11, 15, 32, 68] support interruptible enclaves via a hardware-level secure interrupt mechanism that transparently saves and restores enclave register contents when transitioning to and from the untrusted interrupt service routine (ISR). Interestingly, early versions of AMD SEV, before the introduction of SEV-ES [30], did not feature such a mechanism and were vulnerable as a result [73]. As we will demonstrate in Section 3.3, IPE also does not include a secure interrupt mechanism, nor forcibly disables interrupts in hardware.

¹Following our disclosure process, TI updated this document to reflect the offered security features better.

Table 1: Results of our IPE security analysis, listing software-driven architectural and side-channel attack primitives from prior TEE research or contributed by us (*new*). Symbols indicate whether the attack primitive can break IPE confidentiality (C~~X~~) and integrity (I~~X~~) guarantees directly (●) or indirectly (◐). Positive results are highlighted.

	Attack primitive	C X	I X	Section
Architectural	Controlled call corruption (<i>new</i>)	◐	●	§3.1
	Code gadget reuse [35]	◐	◐	§3.2
	Interrupt register state [73]	●	●	§3.3
	Interface sanitization [69]	◐	◐	§6.1
Side channels	Cache timing side channel [23, 39]	◐	◐	§3.4.1
	Interrupt latency side channel [71]	◐	◐	§3.4.2
	Controlled channel [25, 77]	◐	◐	§3.4.3
	Voltage fault injection [31, 40]	◐	◐	§A.1
	DMA contention side channel [7, 8]	◐	◐	§A.2

Attacker model and scope. The device user’s guide [55] describes IPE as a feature protecting an address range from accesses originating from any code outside this region. In addition to protection against software adversaries with arbitrary code execution, IPE also blocks DMA requests by peripherals and JTAG accesses by the debugger [55]. The FRAM technology used for IPE also has physical features specifically aimed at thwarting physical attackers with microscopy, voltage manipulation, and radiation capabilities [57].

To scope our work, we consider only software-exploitable attacks in our analysis. Specifically, in line with the standard threat model assumed by academic TEEs for embedded devices [9, 15, 18, 32, 41, 42], we consider a capable software adversary with arbitrary code execution on the device, but place physical attacks out of scope. Studying physical attacks against IPE and validating the relevant security measures is orthogonal to our analysis and left as future work.

3 Attack primitives

This section presents the attack primitives we discovered and reproduced, most of which build on existing TEE attack literature. Table 1 summarizes our results, where we distinguish between attack primitives that *directly* break confidentiality or integrity of IPE memory or register contents, and primitives that *indirectly* impact these properties through confused-deputy code gadgets or side channels. The table also includes two relevant negative results that we describe in Appendix A.

All attacks in this section have been implemented on proof-of-concept code based on TI’s official example IPE project [62] and experimentally validated on the following actively manufactured TI boards: MSP-EXP430FR5994, MSP-EXP430FR5969, EVM430-FR6047, MSP-EXP430FR6989.

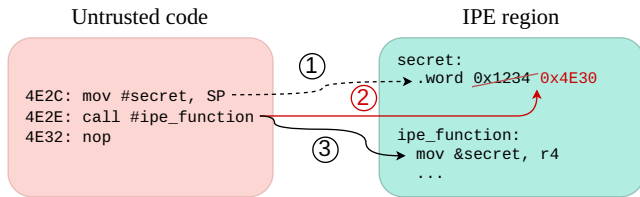


Figure 2: Principle of the controlled `call` corruption exploit.

3.1 Controlled `call` corruption

The processor’s access-control logic must protect IPE memory against both explicit and *implicit* accesses. One example of the latter is the processor’s interrupt logic, which pushes the interrupted instruction’s address and the value of the status register on the stack. In this scenario, we found that even if the attacker manipulates the stack pointer to point to IPE memory, the access rights are correctly enforced, and the interrupt logic will not write to the IPE region. Another example is the `call` instruction, which pushes the called function’s return address on the stack. Crucially, we found that the access control rules are not always correctly enforced for this instruction.

Figure 2 explains how to exploit this vulnerability. The attacker, executing from an untrusted region, first ① points the stack pointer inside the IPE region. Next, when executing a simple `call` instruction, ② the processor writes the return address of the function call (i.e., the address of the instruction below `call`) to the stack. As the stack pointer currently points inside IPE, data or code inside the protected region could be overwritten by the (known) return address if the access rules are not correctly enforced. Indeed, in our experiments, we found that this write access is incorrectly allowed in a specific scenario; ③ when the target of the `call` (i.e., the function being called) lies inside the IPE region.

This vulnerability gives the attacker a very strong primitive: arbitrary locations inside the IPE region, chosen by setting the stack pointer value, can be overwritten by a known value, i.e., the address of the instruction following the `call`.

Choosing corruption values. Attackers using the controlled `call` corruption primitive can overwrite code and data locations inside IPE. The exact written value is the return address of the `call` and, thus, depends on the location of the attacker code. For certain attacks, e.g., when overwriting secret keys, inserting a known arbitrary value is sufficient.

In other cases, however, the attacker may want to insert concrete values, such as the machine code of an assembly instruction. If the attacker had control over the entire address space, they could place the `call` instruction anywhere, inserting arbitrary values. In reality, however, the attacker is limited in the available locations. Table 2 lists the different memory segments, and Table 3 lists a subset of the instruction encodings on MSP430 (we refer to the user’s guide [55] for

Table 2: Address space of our target devices. Segments marked with a ✓ are writable and executable by the attacker unless protected by the MPU or IPE.

Segment	Address range	RWX
FRAM	0x4400 - 0x13FFF	✓
RAM	0x1C00 - 0x23FF	✓
Information memory	0x1800 - 0x1AFF	✗
Bootloader	0x1000 - 0x17FF	✗
Peripherals	0x0000 - 0x0FFF	✗

Table 3: A subset of instruction encodings on MSP430, falling in the RAM (*italic*) or FRAM ranges (cf. Table 2).

Encoding	Inst	Encoding	Inst
0xFxxx	and	0x8xxx	sub
0xExxx	xor	0x7xxx	subc
0xDxxx	bis	0x6xxx	addc
0xCxxx	bic	0x5xxx	add
0xBxxx	bit	0x4xxx	mov
0xAxxx	dadd	0x3xxx	<i>jmp, jl, jge, jn</i>
0x9xxx	cmp	0x2xxx	<i>jc, jnc, jz, jnz</i>

the complete list). The attacker can execute code from the entirety of the RAM and FRAM sections, except for regions protected by the MPU or IPE. Depending on the size and location of these restricted regions, the attacker can encode different binary instructions in the written value, as seen by comparing the two tables. A possible complication is that some instructions in MSP430 consist of multiple words, e.g., representing integer literals or memory addresses.

Many of these challenges can be overcome depending on the goal of the attacker: alternative instructions may be inserted as gadgets to leak information (e.g., `add` instead of `mov`), instructions with indirect addressing modes (using addresses stored in registers) can be used to avoid having to insert additional literal values. Additionally, the 20-bit MSP430X instruction set used by our target microcontroller features a `calla` instruction that jumps to 20-bit target addresses and writes the 20-bit return address to the stack as a zero-extended 32-bit value, covering two 16-bit words. This way, values can be inserted as either the lower 16 bits of a 20-bit address or the zero-extended upper 4 bits, making it possible to insert a broader range of values. We, therefore, want to stress that merely overlapping the IPE region over a specific part of the address space (e.g., covering all the `mov` instructions) is decidedly *not* a sufficient defense against this attack.

Vulnerability root-cause hypothesis. Given that this exploit only works when the `call` target lies inside the IPE region, we suspect the vulnerability is caused by incorrectly performing the access control check on the updated program

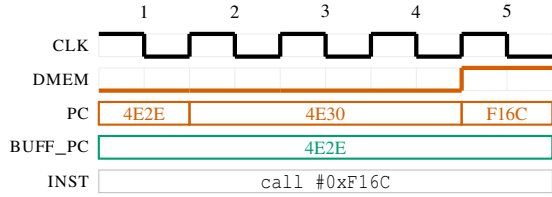


Figure 3: Changes to the program counter (PC) during a `call` on openMSP430. During the instruction’s execution, the program counter changes twice, in the 2nd and 5th clock cycles. Sancus [41] and VRASED [42] buffer the PC (BUFF_PC) for accurate program-counter-based access control.

counter. During the execution of the `call` instruction, the program counter changes to the target address, and the return address is written to the stack. This write will be subject to the access control of IPE. If, however, the program counter is updated before the access control check, this check will succeed when the target of the `call` is inside the IPE region.

This hypothesis is supported by an additional finding: as explained in Section 2.2, the first 4 words of the IPE region, storing `ipe_init_struct`, should not have access to the rest of the protected region. We experimentally confirmed that an instruction executing from the last word of the struct can access the IPE region. This incorrect behavior could again be explained by the program counter being updated too early, this time simply moving to the next instruction before the check. In the pipelined MSP430 design, the processor fetches the next instruction while executing the current one, explaining why the value of the program counter would change early.

The openMSP430 [20] two-stage pipeline behaves very similarly, and this open-source processor allows the inspection of its internal signals. Figure 3 shows the evolution of the program counter during the execution of a `call` instruction. The program counter updates multiple times during the instruction’s execution, first to fetch an additional word from memory (the target address of the `call`), then to change to the jump target while the instruction is still executing. The DMEM signal shows that the memory bus is active in the last cycle, where the return address is written to the stack, making it logical that the flawed access control check happens then.

Attack on modified Sancus. Building on the previous finding on openMSP430, we illustrate that the identified access-control flaw is not just a one-off bug in TI’s proprietary IPE implementation, but represents a more fundamental consideration for program-counter-based access control in pipelined CPU designs. To this end, we examined two mature and popular open-source TEE designs, Sancus [41] and VRASED [42], which feature similar program-counter-based access control logic and are built on openMSP430. We found that both San-

cus and VRASED include explicit patches² to avoid this issue in their hardware-level access control logic. Figure 3 shows how a new stable BUFF_PC signal is introduced that buffers the program counter of the currently executing instruction until its last cycle, such that access control is performed accurately for the full duration of the instruction. We theorize that these patches were introduced to prevent instructions on protection boundaries from being prematurely granted access to protected data, similar to the experiment with `ipe_init_struct`. To show the applicability of the controlled `call` corruption, we manually removed the buffered program counter from Sancus, effectively reverting the protection to that before the patch. With this modification, we experimentally validated in simulation that attackers from outside a Sancus enclave could indeed use the controlled `call` corruption primitive to corrupt data inside the enclave in the same way as on IPE.

3.2 Arbitrary jumps to IPE code

As explained in Section 2.2, arbitrary jumps to code in the protected region lead to a known class of code-reuse attacks. We experimentally validated that – in contrast to other TEEs, such as Intel SGX [13], Sancus [41], or VRASED [42] – IPE indeed does not enforce an entry point to its code, allowing untrusted code to jump to arbitrary locations in IPE. Interestingly, TI’s ARM-based MSP432 microcontrollers did enforce a mechanism similar to entry points in their IP protection [63] but have been discontinued [66], leaving the MSP430 line as the only active one supporting IPE. However, compared to traditional return-oriented programming (ROP) [49], the attacker on IPE is thwarted, as the code of the IPE region is also confidential. This makes finding useful gadgets more difficult, similar to blind ROP attacks on TEEs [35].

```

1  mov #source, r13
2  mov #dest, r14
3  mov 0(r13), 0(r14)

```

Listing 1: IPE code gadget copying secure data.

We first demonstrate an elementary proof-of-concept with this attack primitive assuming knowledge of the protected code but discuss the possibility of leaking IPE memory without this assumption in the following subsection. Consider the code in Listing 1, copying a word of data from one protected location to another with the `mov 0(r13), 0(r14)` instruction. This instruction takes the value from memory at the address contained in the `r13` register and stores it to the address taken from `r14`. While under normal operation these registers are set up with the correct, secure addresses, this instruction can be attacked with our primitive and used as a universal read or write gadget. By setting the registers `r13` and `r14` in untrusted code to point to a location inside and outside IPE respectively, then jumping directly to this instruction

²For Sancus, we could pinpoint the fix to a commit [bcc746a](#) dated Oct 23, 2012. For VRASED, this fix is similarly included in the initial release commit [d2d316c](#) dated May 15, 2019.

(line 3), it will unknowingly copy data or code from inside IPE to an unprotected outside location.

3.3 Arbitrary interrupts in IPE code

Section 2.2 explained that interrupting protected code without protections can lead to leaking or modifying the values of CPU registers. We experimentally validated that IPE does not offer such protection, and this class of attacks can be carried out by implementing a malicious ISR and overwriting an IVT entry (which remains writeable even if placed within IPE boundaries [55]). Our attack was again conducted on the code of Listing 1, this time interrupting the execution before the instruction on line 3 after the intended addresses are written to `r13` and `r14`. We overwrite the addresses in `r13` and `r14` inside the ISR, then resume execution to IPE, which will again unintentionally leak the secret values to unprotected memory.

TI discusses interrupt handling in multiple documents, none of which offer a fix for these attacks. While TI’s example code [62] does not disable interrupts, it clears the register state at the end of its execution. This is insufficient, as an attacker can interrupt the code before the register state is cleared and read out the registers. A different TI document [61] mentions interrupts as an attack vector and recommends disabling interrupts during IPE execution. There are different issues with this recommendation. First, it is not explained how to disable interrupts, and the `dint` (disable interrupts) instruction leaves non-maskable interrupts enabled, which could also be used for this attack. Second, using the previous arbitrary jump primitive, attackers could simply jump over the instructions disabling interrupts at the start of IPE code.

Blind attacks. The proofs-of-concept for the interrupt and the arbitrary jump primitive relied on the attacker knowing the location of a universal read gadget in the victim code. There are many such instructions, such as the `pop rx` instruction, which pops the last word from the stack into a register. This instruction is commonly emitted by compilers in function prologues and epilogues and is always present in code using stack operations, e.g., function arguments or local variables.

More generally, even if the code is protected, instructions may be reconstructed by observing the state changes they cause. In prior work, Schink and Obermaier [47] reconstructed code on MSP432 devices with IP protection enabled by interrupting individual instructions and observing their state changes. We, hence, anticipate that such an automated disassembler, augmented with the side-channel information described in the following section, could also be constructed for MSP430 IPE. Our attacks currently make such an effort unnecessary, as they allow the leakage of the entire IPE region.

Table 4: Evaluation of the side channels used in a covert-channel setting on MSP430FR5969. The extrapolated speeds are calculated for the maximum frequency of 16 MHz.

Side channel	Cycles to leak 16 bits	Extrapolated speed
FRAM cache patterns	1473	173 kbps
Instruction latencies	750	341 kbps
MPU access violations	475	538 kbps

3.4 IPE side-channel leakage

In addition to the attack primitives providing architectural leakage, we investigated the applicability of microarchitectural side-channel primitives from the TEE attack literature. We experimentally validated the presence of three different sources of side-channel leakage from IPE code, showing the relevance of these attacks on MSP430. For each side channel, we built and evaluated a proof-of-concept covert channel on the MSP430FR5969 to demonstrate how secrets can leak from the IPE region to untrusted code without shared memory, summarized in Table 4. While the control-channel setups rely on a cooperating victim, our examples show how deterministic and straightforward the channels are, making them also relevant in a classical attack scenario. In the following subsections, we describe these side channels in more detail and give some countermeasures that can limit the leakage from IPE. Importantly, the established defense against these types of side channels is writing data-oblivious (constant-time [3] or balanced [8, 75] code), avoiding any secret-dependent state changes in the microarchitecture. We only discuss different, specific defenses against the given side channels, possibly involving proposed hardware changes.

3.4.1 FRAM cache access patterns

Side-channel description. Cache attacks are one of the most well-known microarchitectural attacks in the literature. They have been extensively demonstrated on Intel SGX [23, 39], but, to the best of our knowledge, MSP430 microcontrollers were not considered vulnerable to fine-grained, cache-based access pattern leakage [8, 16, 75]. However, the FRAM MSP430 microcontrollers with IPE feature a small cache (2-way set-associative with two sets and 64-bit lines) to speed up accesses to FRAM [55]. This cache improves performance when the processor operates at frequencies above 8 MHz, in which case the CPU needs to insert wait cycles for FRAM reads unless they are served from the cache.

These differences in the timing of FRAM accesses can be observed accurately and deterministically on MSP430 thanks to the following properties. First, the attacker can configure the operating frequency of the core from untrusted software [55], forcing the insertion of wait cycles for cache misses. The attacker can also interrupt the execution of IPE after each instruction, similar to CacheZoom [39] or SGX-Step [70], and measure the cache state using a cycle-accurate timer. Fi-

```

1  configure_frequency();
2  ↵ prime_cache();
3      if (secret[0] == 1) touch(line1);
4      if (secret[1] == 1) touch(line2);
5  ↵ secret[1:0] = probe_cache();

```

Listing 2: Pseudocode of the cache covert channel.

nally, the instruction timings are deterministic, and, unlike in higher-end systems [23], there is no noise in the cache state due to parallel processes.

Proof-of-concept setup. Listing 2 shows the pseudocode of our covert channel setup, based on the Prime+Probe [44] technique. The *victim*, depending on the value of two secret bits, accesses two variables that fall into different cache sets, which we reverse-engineered to be determined by the 4th least significant bit of the address. The *attacker* sets up two interrupts (↵) that execute before and after these accesses. The first ISR, executing from RAM to avoid polluting the FRAM cache, primes the cache state by filling it with attacker-controlled values. Afterward, the *victim* performs the conditional accesses that can evict two lines from the cache. The *attacker*’s second interrupt hits after the conditional accesses, at which point the cache state can be probed and the two secret bits can be reconstructed based on which of the *attacker*’s values were evicted from the cache. Due to the IPE code executing from FRAM, its code accesses will also evict lines from the cache, limiting the number of cache lines that can be used for data transmission. In our proof-of-concept, we limit the executing code to two cache lines, leaving the two other lines for transmitting information. We successfully transmitted a secret word in a total of 1473 clock cycles (including both attacker and victim code), translating to an extrapolated bandwidth of 173 kbps at the maximum frequency of 16 MHz.

Mitigations. Hardware defenses include cache partitioning between trusted and untrusted code [14] or flushing the cache between context switches. Importantly, merely running the IPE code at a lower frequency is an insufficient defense, as we have observed that the cache state was kept updated at low clock speeds, making it possible to recover the cache state by increasing the frequency after the IPE’s execution.

3.4.2 Instruction timings through interrupt latency

Side-channel description. Interrupt latency attacks were first demonstrated on Intel SGX and Sancus platforms [71] and later extended to VRASED [7] and AMD SEV [74]. These attacks derive the execution time of individual instructions by measuring the time taken for interrupt handlers to fire, as the processor first waits for the current instruction to complete before handling a pending interrupt request.

```

1  setup_isr();
2      if (secret[1:0] == 0b00) inst1;
3      if (secret[1:0] == 0b01) inst2;
4      if (secret[1:0] == 0b10) inst3;
5      if (secret[1:0] == 0b11) inst4;
6  ↵ secret[1:0] = measure_time();

```

Listing 3: Pseudocode of the instruction-timing channel.

Our device is very similar to the openMSP430-based [20] Sancus and VRASED, having deterministic instruction timings and a cycle-accurate timer. Apart from having an extended instruction set [55], the most interesting difference on our device is the presence of the FRAM cache, which also influences the timing of instructions accessing memory. This means that interrupt-latency attackers can distinguish IPE load instructions that hit or miss the cache, similar to prior attacks on Intel SGX [71].

Proof-of-concept setup. Listing 3 shows our covert channel setup, similar to the cache setting. In this case, the *victim* executes instructions with different execution times depending on the secret bit values. The *attacker* times an interrupt (↵) to the start of the secret-dependent instruction’s execution and derives the leaked secret bits based on the delay before its ISR starts executing. In our setup, we encode two bits of information in instructions that take 1-4 cycles to execute. This setup leaks a secret word in a total of 750 cycles, resulting in an approximate bandwidth of 341 kbps at the maximum frequency of 16 MHz.

Mitigations. If interrupts could be securely disabled during the execution of IPE code, this attack would be mitigated. The design of Sancus_v [11] shows that with hardware support, it is possible to create provably secure interrupt handling that does not leak timing information. Alternatively, careful hardware-software co-design, such as AEX-Notify [12] on recent Intel SGX platforms, may thwart the attacker’s ability to precisely interrupt victim enclaves.

3.4.3 MPU-based controlled channel

Side-channel description. One of the first attacks on TEEs involved abusing the privileged operating system’s control over untrusted page tables, coining the term controlled-channel attacks [77]. Particularly, based on the page faults generated by the application running in the enclave, its memory access patterns could be reconstructed, which in turn might allow the reconstruction of its secret data.

While the MSP430 microcontrollers do not feature an advanced MMU or virtual memory paging, we show that a similar idea can be applied using the onboard MPU. On MSP430, the MPU can divide the address space into three regions at 1 kB boundaries with different access rights. The IPE access rules do not override MPU rules [55], so code within IPE can


```

1  setup_mpu();
2  if (secret[0] == 1) touch(segment1);
3  if (secret[1] == 1) touch(segment2);
4  if (secret[2] == 1) touch(segment3);
5  secret[2:0] = probe_mpu();

```

Listing 4: Pseudocode of the MPU-based covert channel.

still be denied from accessing data or executing code if the target address falls within an MPU segment with restricted permissions. By partially overlapping MPU segments with the IPE region and observing when an MPU violation occurs, an attacker can track when certain parts of the IPE memory are accessed. Similarly to the original controlled-channel attack, this can leak information about the location of the executing code or the data it is operating on.

Proof-of-concept setup. Our covert channel in Listing 4 follows a similar pattern to the previous setups. This time, the *attacker* sets up three MPU regions, all without write permissions, before launching the IPE code. The IPE code accesses the three regions depending on the secret value, encoding three bits of information. After the *victim*'s execution, the *attacker* checks the MPU violation flags, similar to monitoring page table metadata on Intel SGX [72], to see which of the three regions has a write violation, reconstructing the secret bits. Encoding three bits of information in each iteration leaks a secret word in 475 cycles, translating to a bandwidth of 538 kbps at the maximum frequency of 16 MHz.

Mitigations. The MPU configuration can be made immutable until a reset occurs [55] by setting a lock bit. Code in the IPE region could either configure the MPU to be in a known benign state at the start of its execution and lock it or validate that it is already locked and in a configuration that will not interfere with the IPE's execution or leak information. Alternatively, the hardware could conceivably be changed to ignore MPU access rights while the IPE code executes.

4 End-to-end attacks

This section demonstrates multiple end-to-end attacks to extract secret data and code from IPE using our architectural attack primitives. Our applications are based on TI's IPE example project [62], which developers are likely to build on when writing their code. We successfully reproduced the attacks on all four analyzed devices.

4.1 Overwriting sensitive data

Our first end-to-end attack scenario overwrites a secret key stored in IPE to demonstrate how an integrity violation can lead to loss of confidentiality. The TI example code loads a secret key into the AES accelerator of the device for demonstra-

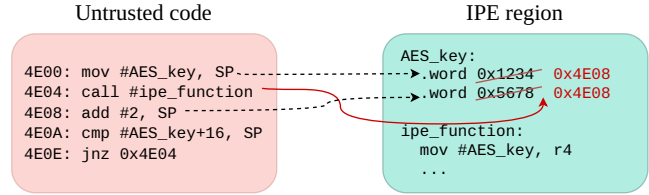


Figure 4: Using a controlled `call` corruption to overwrite a secret AES key in IPE memory.

tion purposes. We extended this code to perform encryption on a secret plaintext string stored in IPE memory, the result of which is then written to a public variable.

For this attack, we use a controlled `call` corruption to overwrite the AES key with a known value, gaining the ability to decrypt the resulting ciphertext. Overwriting the key assumes knowledge of its location, which, while part of the attacker model of many other TEEs, is not necessarily a capability on IPE, as the code is also protected. Nevertheless, the boundaries of the IPE region are visible to the attacker, and the default TI linker script places data at the end of the region, opening the door to a brute-force attack.

Figure 4 shows the basic idea of the attack. The attacker code first points the stack pointer to the (assumed) location of the AES key and calls an IPE function, which, through a controlled `call` corruption, will overwrite the first word of the key with the known return address. This process can be repeated for each word of the key (shown by the second dashed arrow), after which the full key will be overwritten with the same return value and, as we have shown experimentally, all output ciphertexts can be trivially decrypted.

4.2 Leaking the entire IPE region

While the previous attack can already pose a threat to real applications, it relies on two assumptions: the attacker knows (or can find) the location of some secret information, and that the corruption of this information leads to violating the security guarantees of IPE. In the following generic attacks, we go further; we leak the entire IPE region's memory without relying on *any* assumptions about the code or data.

4.2.1 Variant 1: Inserting a leaky gadget

For the arbitrary jump and interrupt examples in Section 3, we rely on a known instruction in the IPE region that can leak data to unprotected memory. With the controlled `call` corruption, this requirement is lifted, as the attacker can insert such an instruction inside the protected region. For example, to insert the `mov 0(r15), r12` instruction with machine code `0x4F2C`, we need to place the `call` instruction at the memory location preceding `0x4F2C`, as the value inserted by the attack is the return address, the address following the `call`.

After performing the corruption, we utilize the other attack primitives to jump directly to the inserted instruction with `r15` pointing to a protected location, then read out the leaked value from `r12` after the inserted instruction executes. The attack could also be orchestrated without relying on the other attack primitives, e.g., by encoding absolute addresses in the inserted instruction.

To mitigate any information loss from overwriting unknown IPE values, we can choose to insert the instruction at the end of the IPE region, as the region is aligned to 1 kB boundaries, leaving some protected memory unused. Overwriting the `ipe_init_struct` at the start of the region is an even better option, as this only contains public values.

4.2.2 Variant 2: Removing IPE protection

The `ipe_init_struct` data structure enables an even more interesting attack. The user’s guide mentions the possibility of reconfiguring the IPE boundaries dynamically [55], a feature that is not supported on comparable systems such as Sancus or VRASED. This reconfiguration happens on a reset if `ipe_init_struct` is updated with new boundary addresses and a valid checksum (invalid data leads to a mass erase).

Building on the previous exploit, we perform another end-to-end attack where we overwrite `ipe_init_struct` to move the boundaries of the protection. This is done by first inserting an arbitrary write gadget inside the IPE region (we chose `0x4FAE = mov 0(r15), 0(r14)`), and then using this instruction to overwrite `ipe_init_struct` one word at a time, taking care to also write a valid checksum for the modified configuration. Locating the currently active `ipe_init_struct` is trivial, as its address is stored at a fixed location [55]. After overwriting the boundaries to different values, we reset the device, which will update the IPE configuration and completely expose the original protected region, which can now be read out in full.

5 Hardware mitigations

This section proposes hardware mitigations to our architectural attack primitives inspired by related academic systems.

5.1 Single entry point and secure interrupts

Many TEEs [13, 18, 32, 41, 42] enforce a single entry point for protected regions to prevent code-reuse attacks that could bypass isolation. Such a modest change in the hardware would rule out most code gadget abuse attacks while only requiring limited extra resources. Specifically, compared to the existing IPE hardware circuitry in Figure 1, only a new internal hardware register `PREV_PC` is needed to buffer the previous value of the program counter (PC), plus additional comparators to ensure that when the current PC lies inside the protected IPE range, it is either the single entry point address or `PREV_PC`

already pointed inside IPE [41, 53]. Furthermore, existing applications could be transparently recompiled with our framework (cf. Section 6) to securely multiplex several software entry points through a single hardware-level entry point.

While it is possible to disable interrupts during the execution of enclaves from hardware as it is done in some embedded TEEs [18, 42], an enforced single entry point also makes it possible to disable interrupts in a non-bypassable way from enclave software. Alternatively, if interruptible and resumable IPE execution is desired, e.g., to ensure real-time guarantees, the hardware can be extended with support to securely save and clear registers upon interrupts, safeguarding the confidentiality and integrity of the register state from the untrusted ISR. In Sancus_v [11], the hardware cost of this mechanism when synthesized to an FPGA was 260 slice registers (+21%) and 142 slice LUTs (+5%), where most of the overhead comes from a shadow register file in the processor. This could be further reduced by storing the values securely in memory [2, 13, 15, 32, 68] (e.g., on the protected IPE stack).

5.2 Controlled `call` corruption mitigation

As discussed in Section 3.1, the likely cause of the controlled `call` corruption is performing the access control check on an updated program counter value, which we have shown may also affect the openMSP430 core [20]. While we do not have access to the design of the proprietary TI microcontrollers, we suspect that a defense that properly buffers the program counter register, similar to Sancus and VRASED, could be straightforwardly applied to mitigate our attacks.

To estimate the cost of this mitigation, we compare two versions of Sancus: the current upstream version (commit `d83a5207`) and the modified version on which we demonstrated our attack in Section 3.1 by reversing the buffered program counter defense. We synthesized both versions for the Xilinx XC6SLX25-2FTG256 FPGA at 20 MHz using the Xilinx ISE 14.7 tool. Based on this analysis, applying the mitigation incurs an overhead of 20 slice registers, from 3078 to 3098 (less than 1% increase), while slightly decreasing the number of necessary slice LUTs from 5628 to 5613.

6 Software mitigations

This section presents our software framework, which helps secure IPE applications on both future devices with mitigations for our attacks and currently deployed hardware. While TEEs like Sancus [41] and Intel SGX [28] provide mature toolchains to help enforce software requirements to achieve the security guarantees of the system, TI only offers an automated setup of the IPE boundaries based on developer annotations via their CCS IDE [61] and an example IPE project [62] demonstrating best practices. This lack of support motivated our framework, which, in addition to using the MPU to reinstate most security guarantees of IPE on current devices with vulnerable

hardware, offers compiler toolchain support to automatically prevent information leakage from sensitive programs, also providing utility on future devices with hardware fixes for our attack primitives. Our mitigation design, like IPE itself, focuses only on preserving the confidentiality and integrity of IPE memory and register contents, placing availability and real-time guarantees out of scope.

Our framework consists of a collection of assembly and C code files and a source-to-source translation script. The developer is responsible for adding intuitive `IPE_ENTRY`, `IPE_FUNC`, and `IPE_DATA` C preprocessor annotations in the source code to define the (entry) functions and data objects that are to be protected. This source file can then be applied to our translator script, which will generate additional assembly stubs and a slightly modified C source code, all of which compiled together will produce a protected binary. As the source files and the changes generated by the translator script are minimal, they can be subjected to manual security analysis or symbolic validation efforts [1] in future work.

6.1 Secure entry and exit stubs

Similar to the Sancus toolchain [41] and the Intel SGX SDK [28], we support a (non-nested) secure function call abstraction, where *ecalls* represent context switches from untrusted code to IPE and *ocalls* refer to calls from secure IPE code to untrusted code. Our framework automatically inserts dedicated assembly stubs whenever the control flow enters or leaves the protected code to sanitize low-level CPU registers and switch between call stacks, known sources of unintentional information leakage [7, 69]. These stubs, shown in orange in the first section of Table 5, are protected by IPE.

In addition to low-level assembly stubs, our framework includes a small C library with support functions for securely validating that a given pointer lies outside the IPE region and for constant-time comparison. These functions can help prevent known issues demonstrated earlier on MSP430 microcontrollers, such as inadvertently leaking secrets through unchecked pointers [69] or timing side channels [7, 21].

IPE ecalls. For each `IPE_ENTRY` function, the translator assigns a logical `eID` and generates an `ecall_stub` placed in untrusted memory with the name of the original function (to preserve compatibility with other code). This untrusted stub transparently invokes the single protected `ipe_entry` entry point, passing the logical `eID` in a defined register, ensuring that third-party code calling the entry functions will be vectored correctly through the single entry point. The translator, furthermore, automatically generates an `ecall_table` in IPE memory. This table, indexed by the `eID`, stores the addresses of the `ecall` functions and a bitmap indicating which registers hold return values and should thus not be cleared on exit.

The single entry point `ipe_entry` performs multiple security-critical tasks. First, it disables all (maskable and non-

Table 5: Assembly stubs provided or generated by our framework. Orange entries in the first section are protected by IPE (and also by the MPU when untrusted code is executing). Blue entries in the second section are only protected by the MPU. Black entries in the last section are not protected.

	Stub	# instances	Size	Cycles
IPE + MPU	<code>ipe_entry</code>	global	76 B	63
	<code>ipe_ocall</code>	global	60 B	51
	<code>ocall_stub</code>	per ocall fn	20 B	25
	<code>ecall_table entry</code>	per ecall fn	6 B	-
MPU only	<code>_system_pre_init</code>	global	72 B	28/42
	<code>_system_post_cinit</code>	global	36 B	28
	<code>reset_into_ipe</code>	global	66 B	49
	<code>ipe_ocall_cont</code>	global	72 B	54
	<code>ecall_ret</code>	global	68 B	51
	<code>new_reset_isr</code>	global	120 B	45/62
	<code>Interrupt vector table (IVT)</code>	global	-	-
	Untrusted <code>ecall_stub</code>	per ecall fn	12 B	16

maskable) interrupts and switches to a known secure stack in IPE memory. The assembly logic then decides if the IPE code previously performed an `ocall`, in which case it restores the saved register state and resumes IPE execution. Otherwise, the stub vectors to the `ecall` function address retrieved from `ecall_table[eID]`. Upon return from the protected function, the remainder of the assembly code clears all registers except those containing a return value, switches back to the untrusted stack, and returns to untrusted code.

IPE ocalls. For each untrusted function called from IPE code, the translator generates an `ocall_stub` in IPE memory and redirects the call site there. This stub vectors to a single `ipe_ocall` exit point, passing the address of the untrusted function to be called and a bitmap indicating argument register usage. The `ipe_ocall` stub will securely exit IPE by clearing unused registers and switching to the untrusted stack before vectoring to the called function.

TI example code and toolchain limitations. As a comparison, the TI example project [62] only includes inline assembly to demonstrate register clearing before returning from an `ecall` but does not provide any guidance for `ocalls`. Furthermore, we found that the inline assembly code does not clear the value of the MSP430 status register, which contains sensitive information on whether the previous branch was taken, potentially leaking secret information for non-constant-time programs. More worryingly, the TI example code offers no guidance on sanitizing the stack pointer, which could lead to severe confused-deputy memory corruption or leakage [7, 69].

Finally, we found that the compiler – both GCC and TI’s proprietary MSP430 compiler – may silently include calls to built-in runtime helper functions, such as `memset` or software-emulated numerical operations. By default, these functions re-

Table 6: Timeline (top to bottom) of MPU-based protection showing the initialization of the device and a subsequent execution consisting of an ecall followed by an ocall. Device resets disabling the MPU are highlighted.

Executing function	MPU
* Untrusted initial startup code	✗
↳ <code>_system_pre_init</code>	✗
↳ <code>_system_post_cinit</code>	✓
↳ Untrusted main → <code>ecall_stub(eID)</code>	✓
↳ <code>reset_into_ipe</code>	✓
* Software brownout reset	✗
↳ <code>new_reset_isr</code>	✗
↳ <code>ipe_entry(eID) → ecall_table[eID](args)</code>	✗
↳ <code>ocall_stub → ipe_ocall</code>	✗
↳ <code>ipe_ocall_cont</code>	✓
↳ Untrusted ocall function	✓
↳ <code>reset_into_ipe</code>	✓
* Software brownout reset	✗
↳ <code>new_reset_isr</code>	✗
↳ <code>ipe_entry → resume ecall function</code>	✗
↳ <code>ecall_ret</code>	✓
Resume untrusted main	✓

side in untrusted memory, even when called from IPE. Hence, any such compiler-generated calls would inadvertently transfer control to the attacker and allow to expose arguments and inject arbitrary return values through CPU registers. It is well-known that compilers that are not aware of security objectives of code can introduce serious vulnerabilities [50], and this is a particularly dangerous illustration of that problem. Therefore, we developed a script in our framework that runs before the linker and intercepts any relocations to such compiler-inserted functions, transparently redirecting them to trusted, intra-IPE counterparts offering the required functionality.

6.2 MPU-based isolation

To protect currently deployed devices with no mitigations against our architectural attack primitives, we leverage the MPU in our framework. This allows reinstating the security guarantees of IPE at the cost of increasing the TCB with the hardware MPU and security-sensitive code protected by the MPU: a few stubs and the IVT (cf. blue entries in Table 5).

Preventing controlled `ca.1.1` corruption. Our framework dynamically disables write access in the MPU over the IPE region when untrusted code is executing, preventing the corruption of protected values by untrusted `call` instructions.

Table 6 shows the evolution of the MPU protection for an execution consisting of an ecall with a single ocall. During the first boot after flashing, `_system_pre_init` sets up our reset vector handler `new_reset_isr` in the IVT. Following the initialization of global variables, this one-time setup finishes with `_system_post_cinit` enabling and locking the MPU

configuration, then calling the untrusted `main` function.

When untrusted code calls (via `ecall_stub`) or returns to (via `ipe_ocall_cont`) the IPE region, the protected `reset_into_ipe` stub is called, which saves all register values and triggers a software brown-out reset. Upon such a reset, the MPU is disabled, and the CPU executes the reset vector, `new_reset_isr`. This reset vector re-enables the secure MPU configuration unless a persistent flag is set, indicating that the untrusted code requested an ecall. In this case, the MPU is not enabled and `ipe_entry` is directly invoked from the reset vector after restoring the previously saved register values to support passing arguments or return values from untrusted code to IPE ecall functions. Two additional stubs, `ipe_ocall_cont` and `ecall_ret`, enable and lock the MPU configuration before calling or returning to untrusted code.

Single entry point and interrupts. In addition to restricting reads and writes to the sensitive MPU region, we also remove execute privileges from this region. By restricting the execute permissions, we prevent attackers from being able to jump to arbitrary points inside the protected region. To launch the protected code, untrusted code needs to call `reset_into_ipe` as explained before, which our framework automatically handles in the generated entry stubs.

Once the previous fixes are in place, and code injection or reuse attacks are not possible, interrupts can be disabled from software in `ipe_entry` before the protected code executes. Even if not all non-maskable interrupt sources can be disabled, the MPU-protected IVT entries can be set to trusted interrupt handlers that prevent register values from being exposed.

6.3 Limitations

Our framework has a few prototype limitations that we believe could be addressed with further engineering effort in future work. For instance, we currently only support automatically generating stubs for functions that only pass basic primitive data types as arguments and return values in registers.

A more fundamental limitation of using the MPU is that we inherit a weaker attacker model than IPE. That is, while JTAG can be, and often is deactivated on devices sold to end users [22], if it is enabled, JTAG can easily bypass our defense if execution is halted during `new_reset_isr` before the MPU protection is enabled and locked, which disables JTAG accesses. This is in contrast to IPE, which is enabled earlier in the boot process before the debugger can attach. Furthermore, our design requires using two of the three MPU regions to protect the original IPE code and our stubs. Due to the IVT residing in the middle of the address space (near `0xFFFF`), the MPU protection also reduces the available storage for untrusted code and data, as only one MPU region can have no protection, and this region cannot contain the IVT.

We use software-induced brown-out resets to remove the MPU protection, which also comes with limitations. These

resets may add non-deterministic delays, making them inappropriate in certain hard real-time settings. They also clear peripherals and configuration settings (e.g., the processor frequency), making it necessary to manually move the required setup code to a section (e.g., `ipe_entry`) that executes before the protected code on each reset. However, both the FRAM and the RAM contents are preserved on resets, and our framework handles the saving of register contents.

Disabling interrupts for the duration of IPE execution and handling them afterward, as is done currently in our framework, might also be unacceptable in real-time settings. If real-time handling of interrupts is desired, our design could conceivably be extended to handle interrupts securely instead of disabling them. This would require adding a trusted interrupt handler to all IVT entries, which first enables the MPU protection and cleans the execution state, then invokes the untrusted ISR. The downside of this approach is that after the untrusted ISR’s execution, IPE needs to be resumed through `reset_into_ipe`, causing a reset.

6.4 Evaluation

To evaluate the security and performance of our framework, we ported a representative embedded enclaved application, VRASED’s memory attestation [42] using the HACLS* cryptographic library [78], to the MSP-EXP430FR5969 board.

Security. To validate the security guarantees of our framework, we first implemented the attestation in IPE, following TI’s guidelines and example project [62]. We then used an attack chain with the three architectural primitives to leak the secret attestation key, very similar to Section 4.2.1. After applying our framework to the application, we observed that the key could no longer be leaked. We also validated that each of the three primitives is individually mitigated by running more self-contained examples with our framework. The MPU protection for overwriting, reading, or executing the protected region is active during the execution of untrusted code and causes a device reset when the controlled `call` corruption or the arbitrary jump primitive is attempted. Interrupts are also disabled during execution of the secure attestation code, in line with VRASED’s original requirements [42], and interrupting the code at the start (before interrupts are disabled) does not leak information. More rigorous or formal testing of the security of our framework is left as future work.

Microbenchmarks. To assess the individual overhead contributions of the different components introduced by our framework, we performed a series of microbenchmarks. First, Table 5 provides the binary size and execution times³ of all stubs provided or generated by our framework, which are deterministic and minimal. A more important measurement is

³For stubs containing a branch, both possible execution times are listed.

Table 7: Mean duration of software-induced brown-out reset on tested devices ($n = 100$), calculated from cycle counts of the measuring device. Note that due to the frequency of the measuring device, the granularity of our timer is $0.0625 \mu\text{s}$.

Device	Reset duration	Standard deviation
MSP-EXP430FR5969	347.3 μs	0.109 μs
MSP-EXP430FR5994	428.5 μs	0.095 μs
EVM430-FR6047	581.4 μs	0.099 μs
MSP-EXP430FR6989	383.9 μs	0.107 μs

the time taken for the brown-out resets that need to be induced before executing trusted code. Table 7 shows these measurements for each of our devices. In our experimental setup, the target device was performing brown-out resets in a loop, sending a GPIO signal on each iteration to a dedicated MSP-EXP430FR5969 performing the timing measurement. While these timings are not entirely deterministic and might depend on physical properties, we see that, in all our experiments, the reset times were fast and very consistent.

Macrobenchmark. As an end-to-end benchmark, we also measured (using the above method) the average execution time of attesting the entire RAM (2 kB). Running at 8 MHz, the attestation takes 497.0777 ms ($n = 100, \sigma = 0.0149$) with only IPE protection, similar to the figure reported in the original paper (450.15 ms for 4 kB) [42]. Using our framework to protect this vulnerable implementation, the execution time increases to 497.7123 ms ($n = 100, \sigma = 0.0182$), an overhead of 634.6 μs (0.13%), which corresponds to the two resets required (the initial `ecall`, then returning from an `ocall`) as shown in Table 6. This result shows that, as is common with TEEs, the overhead of our framework seems to closely correspond with the number of context switches between the enclave and the untrusted world.

7 Related work

7.1 Code protection on microcontrollers

While this paper focused on IPE, in the following, we summarize other code protection or isolation features on currently manufactured microcontrollers.

Arm TrustZone. Arm TrustZone [4, 45], introduced in 2004 and predating most other code protection and TEE technologies, isolates secure and non-secure software on 32-bit devices using support from the hardware, which maintains a security state and propagates it for bus operations, making peripherals also aware of the current security level.

Microchip CodeGuard. Microchip CodeGuard [36, 37], introduced in 2006, offers more comprehensive protection

than IPE on a similar class of devices. CodeGuard allows the flash code memory to be partitioned into three segments, called Boot, Secure, and General. These segments can also claim exclusive access over parts of the device’s data memory, enabling the isolation of dynamic data. At the highest security setting, CodeGuard also enforces entry points, making only the first 32 instructions of the Boot and Secure regions executable from outside. Finally, interrupts are also secured. If a Boot segment is defined, only this segment has access to the IVT. Furthermore, if an interrupt is triggered during the execution of the Boot or Secure regions, the device first vectors to a special ISR in these regions that can clean the register and memory contents before invoking untrusted code.

Flash protection. Multiple vendors offer read and write protection of program code on embedded devices. While useful in certain scenarios, these solutions cannot offer the isolation guarantees outlined in this paper, as they do not protect application data. Microchip Selective Code Protection [38] separates program memory into blocks and can restrict read and write accesses to these, allowing read access to constant values to code from the same block. Armv7-M and v8-M processors are equipped with eExecute-Only-Memory (XOM) [6], which disallows read and write access to specific code regions. Certain Arm-based ST microcontrollers also feature global read protection (RDP) and sector-granular write protection, but importantly, RDP only disallows reads from debug interfaces, not user code [52]. Some ST devices also feature proprietary code read-out protection (PCROP), which is a sector-granular protection of sensitive code from read and write accesses, also from untrusted user code [51].

7.2 Attacks on code protection technologies

While not as actively researched as high-end TEEs, some code protection technologies have already been the subject of security analysis, exposing their vulnerabilities to different classes of attacks. Half-blind attacks [22] exploit the boot-loader through a memory safety vulnerability in application code to extract protected code from MSP430 microcontrollers. Obermaier and Tatschner [43] completely remove the protection on ST’s RDP [52] by downgrading the security level of the device and bypass the access control check on the debugger through a race condition. They also discovered that SRAM contents are readable for a debugger while stepping through code protected by RDP, potentially revealing the firmware when a CRC or hash value is calculated over it, storing intermediate values in SRAM.

This last exploit joins a line of work reconstructing protected firmware based on visible execution state changes. The first such attack [10] bypasses code protection on Cortex-M-based Nordic Semiconductor devices. This technique relies on finding indirect load instructions by single-stepping the protected code with the debugger, setting each register to a

memory address and examining the register bank after the instruction. Once a load instruction is found, this is used to leak the entire firmware. Schink and Obermaier [47] performed similar attacks on other Cortex-M-based microcontrollers implementing XOM [6]. Their approach uses values in both memory and registers (and proposes using side-channel information) to recover all instructions without relying on load instructions to leak the firmware. They performed the attack on ST PCROP [51] and NXP FAC [48] (now deprecated) by using a debugger to single-step, and on MSP432’s IP protection by using interrupts to execute instructions in isolation. In addition, they found implementation issues and load gadgets that can straightforwardly leak the firmware, similar to the attack on Nordic devices [10]. Finally, in concurrent work, RIPencapsulation [46] develops and evaluates a framework using a similar approach on MSP432 and MSP430 IPE.

8 Conclusion

This paper presented the first in-depth security analysis of the IPE security technology found in popular, off-the-shelf TI MSP430 microcontrollers. While this technology is promising and bears a striking resemblance to embedded TEE research prototypes that have been widely studied in recent years, we discovered multiple crucial issues in current IPE hardware and software. Among other attacks inspired by the attack literature, we showcased the novel controlled `call` corruption, which may critically undermine the security guarantees of systems relying on IPE as a secure key storage or intellectual property protection mechanism. To demonstrate the practicality of our attack primitives, we presented proof-of-concept exploits and end-to-end attacks.

Based on our thorough root-cause analysis and the design of similar TEE research prototypes, we formulated actionable hardware-level mitigations for next-generation hardened IPE microcontrollers. Furthermore, we contributed a practical, end-to-end framework that automates software security responsibilities on existing and future IPE devices at low overheads and provides security guarantees similar to IPE using the onboard MPU on existing devices.

Acknowledgments

We thank Jolan Hofmans for the IPE experiments in his master’s thesis [27] and the TI PSIRT for their cooperation during the disclosure process. This research was partially funded by the ORSHIN project (Horizon Europe grant agreement #101070008), the Research Foundation – Flanders (FWO) via grants #G081322N and #1261222N, the Research Fund KU Leuven, and the Flemish Research Programme Cybersecurity.

References

- [1] Fritz Alder, Lesly-Ann Daniel, David Oswald, Frank Piessens, and Jo Van Bulck. Pandora: Principled symbolic validation of Intel SGX enclave runtimes. In *45th IEEE Symposium on Security and Privacy (S&P)*, 2024.
- [2] Fritz Alder, Jo Van Bulck, Frank Piessens, and Jan Tobias Mühlberg. Aion: Enabling open systems through strong availability guarantees for enclaves. In *28th ACM Conference on Computer and Communications Security (CCS)*, 2021.
- [3] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.
- [4] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. *Information Quarterly*, 3, 2004.
- [5] AMD. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. White Paper, 2020.
- [6] Arm Community blogs. What is execute-only-memory (xom)? <https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/what-is-execute-only-memory-xom>, 2017.
- [7] Marton Bogнар, Jo Van Bulck, and Frank Piessens. Mind the gap: Studying the insecurity of provably secure embedded trusted execution architectures. In *43rd IEEE Symposium on Security and Privacy (S&P)*, 2022.
- [8] Marton Bogнар, Hans Winderix, Jo Van Bulck, and Frank Piessens. Microprofiler: Principled side-channel mitigation through microarchitectural profiling. In *8th IEEE European Symposium on Security and Privacy (EuroS&P)*, 2023.
- [9] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. TyTAN: Tiny trust anchor for tiny devices. In *52nd ACM/IEEE Design Automation Conference (DAC)*, 2015.
- [10] Kris Brosch. Firmware dumping technique for an ARM Cortex-M0 SoC. <https://blog.includesecurity.com/2015/11/firmware-dumping-technique-for-an-arm-cortex-m0-soc/>, 2015.
- [11] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. Provably secure isolation for interruptible enclaved execution on small microprocessors. In *33rd IEEE Computer Security Foundations Symposium (CSF)*, 2020.
- [12] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. AEX-Notify: thwarting precise single-stepping attacks through interrupt awareness for Intel SGX enclaves. In *32nd USENIX Security Symposium*, August 2023.
- [13] V. Costan and S. Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016(086), 2016.
- [14] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium*, 2016.
- [15] Ruan De Clercq, Frank Piessens, Dries Schellekens, and Ingrid Verbauwhede. Secure interrupts on low-end microcontrollers. In *25th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2014.
- [16] Florian Dewald, Heiko Mantel, and Alexandra Weber. AVR processors as a platform for language-based security. In *European Symposium on Research in Computer Security (ESORICS)*, 2017.
- [17] Daniel Dinu, Archanaa S Khrishnan, and Patrick Schaumont. SIA: Secure intermittent architecture for off-the-shelf resource-constrained microcontrollers. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2019.
- [18] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: Secure and minimal architecture for (establishing a dynamic) root of trust. In *19th Annual Network and Distributed System Security Symposium (NDSS)*, 2012.
- [19] Stephen Evanczuk. Slideshow: The most-popular MCUs ever. <https://www.edn.com/slideshow-the-most-popular-mcus-ever/>, 2013.
- [20] Olivier Girard. openmsp430 rev 1.17. <https://github.com/olgirard/openmsp430/blob/master/doc/openMSP430.pdf>, 2017.
- [21] Travis Goodspeed. Practical attacks against the MSP430 BSL. In *25th Chaos Communications Congress*, 2008.
- [22] Travis Goodspeed and Aurélien Francillon. Half-blind attacks: Mask ROM bootloaders are dangerous. In Dan Boneh and Alexander Sotirov, editors, *3rd USENIX Workshop on Offensive Technologies, WOOT*. USENIX Association, 2009.
- [23] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *10th European Workshop on Systems Security (EuroSec)*, 2017.

- [24] Michele Grisafi, Mahmoud Ammar, and Bruno Crispo. On the (in) security of memory protection units: A cautionary. In *2022 IEEE International Conference on Cyber Security and Resilience (CSR)*. IEEE, 2022.
- [25] Jago Gyselink, Jo Van Bulck, Frank Piessens, and Raoul Strackx. Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*, 2018.
- [26] Vikas Hassija, Vinay Chamola, Vikas Saxena, Divyansh Jain, Pranav Goyal, and Biplab Sikdar. A survey on IoT security: application areas, security threats, and solution architectures. *IEEE Access*, 7, 2019.
- [27] Jolan Hofmans. A comparative analysis of security features between Sancus and TI MSP430 IPE. Master’s thesis, KU Leuven, 2022.
- [28] Intel. Intel Software Guard Extensions – Get started with the SDK. <https://software.intel.com/en-us/sgx/sdk>, 2019.
- [29] Intel. *Intel 64 and IA-32 Architectures Software Developer’s Manual – Combined volumes*, 2023. Reference no. 325462-062US.
- [30] David Kaplan. Protecting VM register state with SEVES. White Paper, 2017.
- [31] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. VOLTpwn: Attacking x86 processor integrity from software. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.
- [32] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. TrustLite: A security architecture for tiny embedded devices. In *9th European Conference on Computer Systems (EuroSys)*. ACM, 2014.
- [33] Archanaa S Krishnan and Patrick Schaumont. Benchmarking and configuring security levels in intermittent computing. *ACM Transactions on Embedded Computing Systems (TECS)*, 21(4), 2022.
- [34] Archanaa S Krishnan, Charles Suslowicz, and Patrick Schaumont. Secure and stateful power transitions in embedded systems. *Journal of Hardware and Systems Security*, 4, 2020.
- [35] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. Byunghoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium*, 2017.
- [36] Microchip. Codeguard security: Protecting intellectual property in collaborative system designs. <http://ww1.microchip.com/downloads/en/DeviceDoc/70179a.pdf>, 2006.
- [37] Microchip. dspic30f family reference manual - section 26. codeguard security. <http://ww1.microchip.com/downloads/en/DeviceDoc/70275A.pdf>, 2007.
- [38] Microchip. Selective code protection - microchip developer help. <https://microchipdeveloper.com/xc8:selective-code-protection>, 2021.
- [39] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *19th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2017.
- [40] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *41st IEEE Symposium on Security and Privacy (S&P)*, 2020.
- [41] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix C. Freiling. Sancus 2.0: A low-cost security architecture for IoT devices. *ACM Transactions on Privacy and Security*, 20(3), 2017.
- [42] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED: A verified hardware/software co-design for remote attestation. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.
- [43] Johannes Obermaier and Stefan Tatschner. Shedding too much light on a microcontroller’s firmware protection. In *11th USENIX Workshop on Offensive Technologies, WOOT 2017*. USENIX Association, 2017.
- [44] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology – CT-RSA 2006*, 2006.
- [45] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Comput. Surv.*, 51(6), 2019.
- [46] Prakhar Sah and Matthew Hicks. RIPencapsulation: Defeating IP encapsulation on TI MSP devices. *arXiv preprint arXiv:2310.16433*, 2023.
- [47] Marc Schink and Johannes Obermaier. Taking a look into execute-only memory. In *13th USENIX Workshop on Offensive Technologies, WOOT*. USENIX Association, 2019.

- [48] NXP Semiconductors. Using the kinetis flash execute-only access control feature. <https://www.nxp.com/docs/en/application-note/AN5112.pdf>, 2015.
- [49] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *14th ACM Conference on Computer and Communications Security (CCS)*, 2007.
- [50] Laurent Simon, David Chisnall, and Ross Anderson. What you get is what you c: Controlling side effects in mainstream c compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018.
- [51] STMicroelectronics. AN4968 application note: Proprietary code read out protection (PCROP) on STM32F72xxx and STM32F73xxx microcontrollers. https://www.st.com/resource/en/application_note/dm00346619-proprietary-code-read-out-protection-pcrop-on-stm32f72xxx-and-stm32f73xxx-microcontrollers-stmicroelectronics.pdf, 2017.
- [52] STMicroelectronics. RM0091 reference manual: STM32F0x1/STM32F0x2/STM32F0x8 advanced arm-based 32-bit MCUs. https://www.st.com/resource/en/reference_manual/rm0091-stm32f0x1stm32f0x2stm32f0x8-advanced-armbased-32bit-mcus-stmicroelectronics.pdf, 2022.
- [53] Raoul Strackx, Frank Piessens, and Bart Preneel. Efficient isolation of trusted subsystems in embedded systems. In *Security and Privacy in Communication Networks*, 2010.
- [54] Texas Instruments. MSP430 programming with the JTAG interface. <https://www.ti.com/lit/ug/slau320aj/slau320aj.pdf>, 2010.
- [55] Texas Instruments. MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx family user’s guide. <https://www.ti.com/lit/ug/slau367p/slau367p.pdf>, 2012.
- [56] Texas Instruments. Car access product family. <https://www.ti.com/lit/ml/slyt455a/slyt455a.pdf>, 2013.
- [57] Texas Instruments. Closing the security gap with TI’s MSP430 FRAM-based microcontrollers. <https://www.ti.com/lit/wp/slay035/slay035.pdf>, 2014.
- [58] Texas Instruments. FRAM FAQs. <https://www.ti.com/lit/wp/slat151/slat151.pdf>, 2014.
- [59] Texas Instruments. Introduction to MSP430FR5969. <https://www.youtube.com/watch?v=QRJ0r-Zx2Hk>, 2014.
- [60] Texas Instruments. MSP430 FRAM technology – how to and best practices. <https://www.ti.com/lit/an/slaa628b/slaa628b.pdf>, 2014.
- [61] Texas Instruments. MSP code protection features. <https://www.ti.com/lit/an/slaa685/slaa685.pdf>, 2015.
- [62] Texas Instruments. MSP code protection features: Source code. <http://www.ti.com/lit/zip/slaa685>, 2015.
- [63] Texas Instruments. Software IP protection on MSP432P4xx microcontrollers. <https://web.archive.org/web/20191213154223/http://www.ti.com/lit/an/slaa660b/slaa660b.pdf>, 2015.
- [64] Texas Instruments. Understanding security features for MSP430 microcontrollers. <https://www.ti.com/lit/ml/swpb018/swpb018.pdf>, 2017.
- [65] Texas Instruments. PSIRT notification: MSP430FR5xxx and MSP430FR6xxx IP encapsulation write vulnerability. <https://web.archive.org/web/20231030234254/https://www.ti.com/lit/an/swra792/swra792.pdf>, 2023.
- [66] TI Support Forum. MSP432P401R: Is the MSP432 line discontinued? <https://e2e.ti.com/support/microcontrollers/arm-based-microcontrollers-group/arm-based-microcontrollers-f/arm-based-microcontrollers-forum/1007640/msp432p401r-is-the-msp432-line-discontinued>, 2021.
- [67] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium*, 2018.
- [68] Jo Van Bulck, Job Noorman, Jan Tobias Mühlberg, and Frank Piessens. Towards availability and real-time guarantees for protected module architectures. In *Companion Proceedings of the 15th International Conference on Modularity (MASS)*, 2016.
- [69] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *26th ACM Conference on Computer and Communications Security (CCS)*, 2019.

- [70] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *2nd Workshop on System Software for Trusted Execution (SysTEX)*. ACM, 2017.
- [71] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *25th ACM Conference on Computer and Communications Security (CCS)*, 2018.
- [72] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium*, 2017.
- [73] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The severest of them all: Inference attacks against secure virtual enclaves. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019.
- [74] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. SEV-Step: A single-stepping framework for AMD-SEV. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(1), 2024.
- [75] Hans Winderix, Jan Tobias Mühlberg, and Frank Piessens. Compiler-assisted hardening of embedded software against interrupt latency side-channel attacks. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2021.
- [76] Lennert Wouters, Benedikt Gierlichs, and Bart Preneel. On the susceptibility of Texas Instruments SimpleLink platform microcontrollers to non-invasive physical attacks. In *Constructive Side-Channel Analysis and Secure Design*, 2022.
- [77] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *36th IEEE Symposium on Security and Privacy (S&P)*, 2015.
- [78] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACl*: A verified modern cryptographic library. In *24th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017.

A Negative results: Inapplicable attacks

Some attacks from high-end or embedded TEEs have turned out not to apply to the microcontrollers in our analysis. Most notably, embedded MSP430 microcontrollers are not optimized for high performance and do not come equipped with

advanced microarchitectural features, such as out-of-order execution and branch prediction that give rise to the potent line of transient execution attacks, which has critically affected high-end TEEs like Intel SGX [67]. Below, we summarize two non-trivial negative results that we found unexploitable from software in our analysis.

A.1 Voltage fault injection

While voltage fault injection attacks, also performed on MSP430 [21] and other TI microcontrollers [76], have traditionally required physical access and are thus out of scope for our analysis, on certain systems it is also possible to control the frequency and voltage of the system through software interfaces. On Intel SGX, changing the voltage through these interfaces could compromise the confidentiality and integrity of enclaves through fault injection, even leading to leaking cryptographic keys from enclaves [31, 40].

The only related configuration setting our microcontroller exposes to software is the frequency of the device. Fault injection could occur by running the device at a high frequency, requiring wait states for the FRAM, but omitting this configuration step. However, the device implements a protection mechanism against this attack and resets the device upon such misconfiguration, avoiding fault injection [55]. In addition, the device contains additional measures against physical attacks [57], including voltage attacks, but these are out of scope for our software-based analysis.

A.2 Direct memory access (DMA) contention

A recent side-channel attack demonstrated on openMSP430-based TEEs uses contention between the CPU and DMA devices to measure the memory activity of enclaves [7, 8]. While on our device, it is possible to schedule DMA transfers from software (putting it in scope for our analysis), due to the DMA implementation on TI's devices, this attack is not possible in its current form. The side-channel attack exploits subtle timing differences caused by contention when DMA transfers and CPU activity happen in parallel, whereas on TI devices, DMA transfers happen while the CPU is disabled in low-power mode and not executing code [55].

B Artifact Appendix

B.1 Abstract

This artifact provides source code for the individual attack primitives and end-to-end attack scenarios that can be run on off-the-shelf TI MSP430 microcontrollers with Intellectual Property Encapsulation (IPE) support. We also provide source code to reproduce evaluation results for the software mitigation framework, as well as the openMSP430/Sancus-based hardware mitigation against controlled `call` corruption.

B.2 Description & Requirements

B.2.1 Security, privacy, and ethical concerns

This artifact demonstrates attacks on real-world TI MSP430 microcontrollers. There are no security risks for evaluators, as the only code executed on the host machine is compilation of the example projects using standard tools. All attack code runs locally on the specific device under test.

The attack code provided in this artifact is solely intended for reproduction of our results. Any uses of these results on real-world microcontrollers should be conducted responsibly.

B.2.2 How to access

The artifact files are accessible in the following repository: <https://github.com/martonbognar/ipe-exposure/tree/usenix24-artifact>.

B.2.3 Hardware dependencies

Our repository contains proof-of-concept code for four TI development boards: MSP-EXP430FR5994, MSP-EXP430FR5969, EVM430-FR6047, MSP-EXP430FR6989.

B.2.4 Software dependencies

Compiling and running our artifact requires the following software, available on all major operating systems:

- Code Composer Studio (CCS) integrated development environment (IDE). Can be downloaded from the TI website (we used regular CCSTUDIO version 12.6.0): <https://www.ti.com/tool/CCSTUDIO#downloads>.
- Python 3 with `pycparser` (v2.21), `pycparserext` (v2021.1), and `pyelftools` (v0.29) libraries.
- For the Sancus experiment: `gcc-msp430`, `cmake`, `iverilog` (e.g., via standard Ubuntu packages).

B.2.5 Benchmarks

No external benchmarks were used for our evaluation.

B.3 Set-up

B.3.1 Installation

First, clone the artifact repository:

```
$ git clone --recurse-submodules \  
--branch usenix24-artifact \  
https://github.com/martonbognar/ipe-exposure
```

Then, install the software dependencies from above.

1. Download CCS via the link above and proceed with the installation, then install the necessary drivers:

```
$ wget https://dr-download.ti.com/software-development  
  ↳ /ide-configuration-compiler-or-debugger/MD-  
  ↳ J1VdearkvK/12.6.0/CCS12.6.0.00008_linux-x64.  
  ↳ tar.gz  
$ tar -xvzf CCS12.6.0.00008_linux-x64.tar.gz  
$ cd CCS12.6.0.00008_linux-x64/  
$ ./ccs_setup_12.6.0.00008.run # choose ~/ti as  
  ↳ installation directory  
$ cd ~/ti/ccs1260/ccs/install_scripts/  
$ sudo ./install_drivers.sh
```

2. Install the required Python 3 dependencies:




```
$ cd ipe-exposure/05_framework/framework  
$ pip install -r requirements.txt --no-deps
```

3. Install Sancus dependencies:





```
$ sudo apt install gcc-msp430 cmake iverilog
```

B.3.2 Basic Test


To test out the setup, we provide a simple “hello world” CCS project that can be run on the target MSP430 board with IPE as per the steps below (please refer to the repository’s top-level `README.md` for detailed screenshots and troubleshooting):

1. Launch the CCS IDE and create a new workspace in an empty directory when prompted on startup.
2. In CCS, choose *File* > *Open Projects from File System*. Now select the directory `00_helloworld` in the cloned `ipe-exposure` repository.
3. With the microcontroller connected to the system, start the debug session (F11,  .
4. After successfully launching the debug session, resume the program (F8, .
5. Expected output should now appear in the Console pane:

```
-----  
Reading secret from main: 1234 (IPE disabled)  
Reading secret from IPE : 1234
```

6. In order to activate IPE, the device needs a hard reset. For this, first pause the running debug session (Alt+F8, ), then select “Hard Reset” from the dropdown next to the Reset button ( .
7. The microcontroller will now reboot with IPE enabled. After resuming the program (F8, ) , you should see the following output in the Console pane:

```
-----  
Reading secret from main: 3fff (IPE enabled)  
Reading secret from IPE : 1234
```

8. The CCS debug session can now be terminated via the stop button (Ctrl+F2, .

B.4 Evaluation workflow

B.4.1 Major Claims

- C1** The attack primitives from [Section 3](#) directly or indirectly break confidentiality and integrity of IPE-protected memory, as summarized in [Table 1](#) (cf. E1).
- C2** The covert channels from [Section 3.4](#) enable deterministic leakage with performance as reported in [Table 4](#) (cf. E2).
- C3** The three end-to-end attack scenarios from [Section 4](#) can be reproduced, showing successful corruption or leakage of secrets from complete programs (cf. E3).
- C4** Buffering the program counter register as a hardware mitigation prevents similar attacks on openMSP430/Sancus, as explained in [Section 3.1](#) (cf. E4).
- C5** Our software mitigation framework blocks all architectural attacks demonstrated in this paper (cf. E5).
- C6** The micro- and macrobenchmarks in [Section 6.4](#) ([Table 5](#) and [Table 7](#)) describe the software framework’s overhead (cf. E6).

B.4.2 Experiments

- E1:** [*Attack primitives*] [40 human-minutes]: Reproduction of three architectural and three side-channel attack primitives by running minimal proof-of-concept programs, one per primitive, in standalone CCS projects.
Preparation: Launch CCS and open all relevant projects under the `01_attack_primitives` directory.
Execution: For every project individually, analogous to [§B.3.2](#): launch the debug session, trigger a hard reset (to activate IPE), then run the code.
Results: Refer to the README of each project for the expected output, which should match the console. These projects demonstrate the effectiveness of the attack primitives and, thus, validate claim C1.
- E2:** [*Covert channels*] [20 human-minutes]: Reproduction of the three covert channel setups.
Preparation: Launch CCS and open all projects under the `02_covert_channel` directory.
Execution: For every project individually, analogous to [§B.3.2](#): launch the debug session, trigger a hard reset (to activate IPE), then run the code.
Results: Refer to the README of each project for the expected output, which should match the console and the numbers in the second column of [Table 4](#). These projects demonstrate the presence of the covert channels and their measured performance, validating claim C2.
- E3:** [*End-to-end attacks*] [20 human-minutes]: Reproduction of end-to-end attacks.
Preparation: Launch CCS and open all relevant projects under the `03_end_to_end_attacks` directory.
Execution: For every project individually, analogous to [§B.3.2](#): launch the debug session, trigger a hard re-

set (to activate IPE), then run the code. Note: for the `init_struct_overwrite` exploit, two successive hard resets are required (see the corresponding README).

Results: Refer to the README of each project for the expected output, which should match the console. These projects demonstrate the effectiveness of the attacks and, thus, validate claim C3.

- E4:** [*Sancus defense*] [10 human-minutes]: Reproduction of the existing hardware mitigation preventing controlled `call` corruption on Sancus. The cycle-accurate openMSP430 Verilog simulation shows that the attack fails on the original Sancus, but succeeds after deliberately omitting the program counter buffering.

Preparation: Make sure the `sancus-core` git submodule is initialized (execute `git submodule init; git submodule update` if needed).

Execution: Run the `run-sancus-eval.sh` script in the `04_sancus_exploit/sancus-exploit/` directory. This will perform all necessary steps for this experiment.

Results: The script will first run a controlled `call` corruption attack against the upstream version of Sancus. This attack should result in a memory violation error, without overwriting the secret value. Next, the script will apply a minimal patch that removes the buffered program counter. After running the same attack again, no memory violation will occur, and the secret value will be overwritten, validating claim C4.

- E5:** [*Framework security*] [40 human-minutes]: Demonstration of the mitigation framework’s security by recompiling and running a vulnerable example project.

Preparation: Launch CCS and open the `demo_all` project under the `05_framework/security_eval/` directory. Now execute `run.sh` in that same directory to apply the framework (cf. [Figure 5](#)) on the vulnerable application and generate a new `demo_all_mitigated` project. Open this new project in CCS as well.

Execution: Run both the vulnerable and the mitigated projects and examine the `fail_code` and `public` variables using the CCS debugger, as shown in the screenshots of the README file. Repeat this three times, for all values of the `attack` global variable in `main.c`.

Results: The values will show that while in the unprotected version all attacks successfully change and leak values from the IPE region, applying our framework disables these attacks, validating claim C5.

- E6:** [*Benchmarks*] [40 human-minutes]: Reproduction of the micro- and macrobenchmarks by measuring the timing of projects secured by the framework.

Preparation: Follow the steps in the README in the `06_benchmarks` directory to simultaneously debug a timer and a benchmark project on two connected boards.

Execution: Always start a new debug session of the timer project and resume its execution first before launching and resuming the benchmarked program in

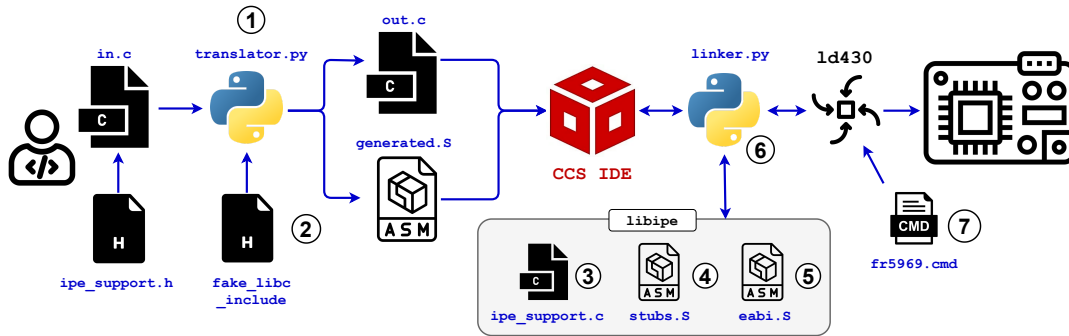


Figure 5: Overview of the general workflow for the software mitigation framework.

a separate CCS instance. After successfully collecting 100 measurements, the `timer` project prints the collected numbers in a comma-separated value format to the console, which can be copied verbatim into a `.csv` file. Collect microbenchmark measurements for a `software-bor/bor_timing_*/` project, depending on the evaluation target. Next, collect macrobenchmark measurements for both the `hmac/base_attestation` and `hmac/translated_attestation` projects.

Results: Use the `measurements/calculator.py` script to compute the mean and standard deviation for the collected `.csv` files. These values should be similar to those reported in the paper (small deviations are expected), showing the limited overhead of our defense and validating claim C6.

B.5 Notes on Reusability

Based on the `05_framework/security-eval` and `06_benchmarks/hmac` examples, our software mitigation framework (cf. Figure 5) could be applied to other programs, collecting additional evidence for its overhead and effectiveness. In future work, the framework could also be improved further to support a wider range of programs.

B.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at <https://secartifacts.github.io/usenixsec2024/>.