openIPE: An Extensible Memory Isolation Framework for Microcontrollers

Marton Bognar
DistriNet, KU Leuven
Leuven, Belgium
marton.bognar@kuleuven.be

Jo Van Bulck
DistriNet, KU Leuven
Leuven, Belgium
jo.vanbulck@kuleuven.be

***Abstract*—Given the popularity of low-end microcontrollers, manufacturers and researchers have proposed memory isolation mechanisms for these devices. However, current proposals face two main shortcomings. First, due to a lack of extensible reference implementations of commercial specifications, academic systems commonly use custom memory isolation mechanisms, reducing compatibility and the chance of real-world adoption. Second, recent research has demonstrated crucial and overlapping vulnerabilities, including in commercial systems. Unfortunately, efforts to mitigate and validate these issues are hindered by the disconnect in codebases.**

This paper proposes openIPE, an open research platform for extensible, industry-compliant hardware-software co-designs. Our platform introduces minimal hardware extensions for memory isolation based on Texas Instruments' specification for Intellectual Property Encapsulation (IPE), alongside a versatile firmware layer enabling rapid prototyping of advanced security extensions. We establish a robust security testing infrastructure and demonstrate the capabilities of our framework through a comprehensive study on secure interrupt handling, an important research area for microcontrollers. Our evaluation shows that openIPE allows for the independent reproduction and comparison of existing proposals and enables a novel solution that achieves strong architectural and microarchitectural security with minimal hardware modifications and low overhead.

***Index Terms*—embedded security, memory isolation, secure interrupts**

1. Introduction

Effective memory isolation is a fundamental building block for security. In traditional computing systems, this is typically achieved through virtual memory mechanisms supported by advanced operating systems and memory management unit (MMU) hardware. However, as small embedded microcontrollers become increasingly prevalent in daily life, they introduce unique constraints related to production cost, power consumption, size, and extensibility. Consequently, specialized memory isolation features for embedded devices are generally less widespread and often limited in functionality [1]. In this respect, recent years have seen the emergence of uniform memory-isolation features on commodity 32-bit microcontrollers, including TrustZone-M for low-end Arm processors and the recently standardized physical memory protection (PMP) primitive for RISC-V. In contrast, the much more heterogeneous landscape of low-power 8-bit and 16-bit microcontrollers lacks effective standardization and remains largely unprotected in practice.

To address this gap, academia has developed a long line of specialized research prototypes [2]–[13], enabled in large part by open-source CPU designs [14]. At the same time, some vendors have begun to introduce limited hardware features for code protection and, to some extent, even data isolation in selected off-the-shelf microcontrollers [15]–[18]. Among the most advanced offerings in this area is Texas Instruments' Intellectual Property Encapsulation (IPE) [15] technology, included in its popular ultra-low-power MSP430 microcontrollers. IPE employs a capable, hardware-level isolation mechanism to create a secluded "enclave-like" memory region to protect software intellectual property (IP), such as confidential code and data. IPE is remarkably similar to independently developed academic prototypes based on program-counter-based access control [2]–[4], [13] or execution-aware memory isolation [5], [6]. However, recent analyses [19], [20] have revealed that current IPE-enabled microcontrollers suffer from critical design flaws and implementation oversights, which were known and successfully averted in their academic counterparts. Additionally, open-source academic prototypes often re-implement the same security functionalities, which has similarly resulted in design flaws and implementation errors, improper input sanitization vulnerabilities, and subtle microarchitectural side channels [21]–[24].

In conclusion, there is a notable disconnect between industry and academia: industry proposals, such as Texas Instruments' IPE, suffer from well-known vulnerabilities that have been addressed in academic architectures, while the latter face challenges related to incompatibility and deployment feasibility. This cycle of duplication and inconsistency ultimately undermines the overall security and effectiveness of memory isolation solutions in ultra-low-end microcontrollers.

openIPE. To bridge this gap, we propose openIPE, an extensible hardware-software co-design framework providing industry-compliant memory-isolation building blocks for rapid prototyping of innovative security features on low-end microcontrollers. We base openIPE on the mature openMSP430 [14] open-source softcore implementation, which lacks any built-in security features but has been a favored platform in academia. Over the past decade, more than 20 papers have focused on developing low-end memory isolation mechanisms on top of openMSP430, with new research still emerging (cf. Table 1 on page 5).

In contrast to prior academic prototypes, our design is compatible with TI's IPE specification, requiring only a recompilation step for existing IPE applications. As a result, openIPE enables the development and comparison of hardware-software security extensions within a unified open research infrastructure, akin to the prior examples of Keystone [25] and openSGX [26] for higher-end systems. As a case in point, we incorporate hardware modifications into IPE's access-control logic to protect against recently disclosed vulnerabilities [19]. Previously, these changes were only proposed in theory, but now they elevate the security guarantees to match those provided by state-of-the-art academic research prototypes. In addition to minimal hardware extensions, openIPE features a flexible privileged firmware layer, similar to the XuCode used in Intel SGX [27] or the Trusted Firmware in Arm CCA. Compared to existing academic designs, openIPE's programmable firmware layer ventures into a largely unexplored design space, enabling rapid prototyping and updateability of innovative security features encompassing hardware, software, and firmware modifications.

Security testing. Security validation can be streamlined through openIPE by consolidating efforts into a unified, reusable codebase. We conduct thorough testing for both hardware and software components. On the hardware side, we utilize openMSP430's thorough unit-testing framework to ensure backward compatibility, developing additional unit tests to ensure compliance with TI's base IPE specification and to validate our additional security features.

On the software side, inspired by recent validation efforts for Intel SGX enclaves [28]–[30], we employ Pandora [28], which uses the `angr` [31] binary symbolic execution tool. We symbolically validate both the firmware layer and IPE software components, specifically targeting confused-deputy pointer vulnerabilities and register leakage [22], [23], [32]. We contribute enhancements to `angr`'s MSP430 backend and identify a previously unknown vulnerability in the IPE Exposure framework [19] that permits secret leakage through register values.

Secure interrupts. To show how openIPE can be used as a unified framework for efficiently reproducing and comparing related work, as well as exploring new points in the design space, we conduct an extensive case study. Our case study concerns secure interrupt handling, a crucial aspect of providing isolation on embedded systems [5], [7], [8], [11], [33], [34]. Inadequate sanitization during interrupts can result in data leakage and corruption [19], while timing differences can leak side-channel information about executing instructions [21].

We implement and evaluate four approaches, ranging from hardware-only to software-only, using openIPE's flexible hardware-software co-design framework. This effort includes independent reproduction of a previously closed-source [7] proposal, as well as a novel solution using openIPE's unique firmware model. Notably, we demonstrate for the first time that the firmware layer, combined with trusted software in the IPE application, can effectively mitigate interrupt-related side-channel leakage, eliminating the need for inflexible and error-prone [23], [35] hardware solutions [33], while incurring much lower hardware overhead.

Evaluation. We synthesize the base openIPE design and the improved access-control logic and secure interrupt hardware extensions to an FPGA and report on their hardware cost. We, furthermore, evaluate and report on the runtime and code size overhead of our software development framework. To showcase a project that can directly use our isolation primitive, we use VRASED's [4] remote attestation codebase with minimal modifications. The results of the evaluation show that by focusing on the isolation primitive, we can provide strong security guarantees with low hardware cost and runtime overhead.

Summary. In summary, our main contributions are:

- We design and implement openIPE, a research platform compliant with TI's IPE specification and extended with a trusted firmware layer, enabling research on hardware, software, and firmware extensions for memory isolation.
- We thoroughly validate openIPE's security and functionality through diverse unit tests and an openIPE-aware symbolic execution tool, which already found a bug in related work.
- We reproduce and compare secure interrupt handling mechanisms and devise a novel solution enabled by openIPE's flexible firmware layer, providing security guarantees against both architectural and microarchitectural attacks.

Ethical considerations and open science. We believe that openIPE and the surrounding ecosystem, including our research into secure interrupts, are important contributions to embedded security and open science. The openIPE implementation and evaluation are available at https://github.com/martonbognar/openipe, together with documentation and setup code aimed at researchers who want to build on our work. Additionally, we fixed the bug discovered in the IPE Exposure framework.

2. Background and related work

Memory isolation and secure interrupt handling are active research questions at all levels of the computing spectrum. On high-end systems, virtual memory and privilege rings provide isolation across processes, and trusted execution environments (TEEs) such as Intel SGX and Arm TrustZone can extend protection against a potentially compromised operating system. In research, multiple hardware and software extensions have been proposed that provide memory isolation with a minimal trusted computing base (TCB), often also ensuring secure interrupt handling and predating many commercial solutions [36]–[39]. However, most of these research proposals are aimed at desktops and mobile phones, only a minority of works target embedded systems [5], [6], [40], some of which will be discussed later. In this paper, we focus on low-end MSP430 microcontrollers, but our insights are generalizable to other embedded devices.

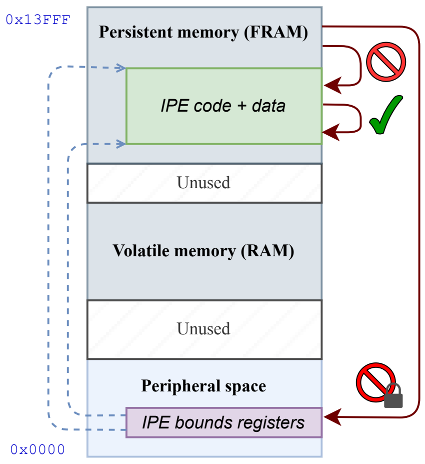

Figure 1. Memory layout of IPE (not drawn to scale). Arrows indicate accesses allowed and disallowed by the memory access control logic.

2.1. MSP430 microcontrollers

MSP430 is a family of microcontrollers developed by Texas Instruments (TI) with a focus on ultra-low-power operations. Originally a 16-bit architecture, TI has recently extended the MSP430 address space to 20 bits (referred to as MSP430X), adding new instructions to support this change. Recent MPS430 devices also feature non-volatile Ferroelectric Random Access Memory (FRAM) technology which enables the persistent storage of intellectual property on the device [41]. TI has sold millions of these FRAM MSP430 devices [42] and a recent embedded survey [43] ranks MSP430 as the second most popular 16-bit microcontroller. To address increased demands for security features, these recent devices also feature a memory protection unit (MPU) and the IPE technology, explained in Section 2.2.

The MSP430 instruction set has also been implemented in the open-source NEO430 [44] and openMSP430 [14] softcores, both of which have been used for research purposes. In particular, openMSP430 has been extended in many publications related to memory isolation, which we systematize later in this section.

2.2. Intellectual Property Encapsulation (IPE)

Intellectual Property Encapsulation (IPE) [45] is a code and data isolation feature on recent FRAM-based MSP430 devices. These devices feature a volatile RAM and a non-volatile FRAM partition (both writeable and executable), the latter of which can be configured to contain an isolated IPE region. Figure 1 summarizes the memory layout and the access control rules. IPE enforces a program-counter-based memory access control logic, which ensures that only code executing from within IPE can access data (or code) stored in the region. Accesses targeting the IPE region from any other source, including code executing outside IPE, the debugger unit, or direct memory access (DMA) requests, are disallowed. Illegal reads return the constant value `0x3FFF`, while illegal writes are ignored. Optionally, IPE can be configured to trigger a reset on these violations.

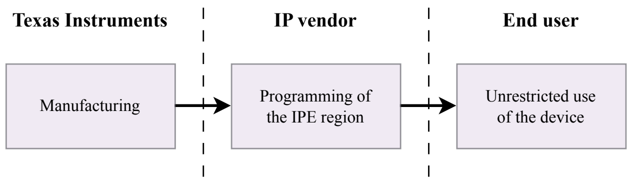

Figure 2. IPE deployment model: TI manufactures the device and loads the bootcode; IP vendor supplies proprietary code or keys; end user programs arbitrary code that can interact with the secluded IPE region.

The IPE region is defined by two hardware configuration registers, which store the boundaries at a granularity of 1 kB. These registers can be modified and locked from software directly, but in most cases, they are set up by the device's *bootcode*, detailed next.

Boot process. IPE-enabled devices execute a proprietary bootcode on every device reset before user code executes or the debugger can attach. While this bootcode is not modifiable and its source or binary code is not available, its behavior is described in detail by TI [45]. The role of the bootcode is to set up the IPE configuration registers based on an 8-byte configuration structure in user software. To prevent tampering, this structure should be stored in the protected IPE region itself.

On every reset, the bootcode checks two fixed FRAM locations which can contain the address of the configuration structure and an enable flag. If the enable flag is set or IPE was activated previously, the structure is evaluated, and either activates the IPE protection or, in case the structure is invalid (a potential sign of tampering or misconfiguration), triggers a mass erase loop that clears the entire persistent memory to prevent unintentional leakage. Since the structure is evaluated on every device reset, this also enables the IPE region to optionally reconfigure its boundaries across resets.

Deployment and threat model. IPE, similarly to other low-end commercial isolation schemes [16], assumes a deployment model involving multiple stakeholders. After TI manufactures the device and programs the bootcode, it is handed over to a second party, who programs the IPE region with their intellectual property, possibly proprietary code or secret keys [46]. Finally, the device is deployed in the field or delivered to an end user, who can arbitrarily reprogram the rest of the device and have physical access to it but should not be able to tamper with the protected code and data. This process is visualized in Figure 2.

More precisely, the attacker is considered to control all untrusted software on the device (outside the IPE region and the bootcode), and even external interfaces such as the DMA controller or JTAG debug units. To protect against physical threats, the FRAM technology used in IPE-enabled devices offers additional protections against attacks such as radiation and voltage manipulation [46].

2.3. Academic low-end memory isolation

Years before TI introduced its security features on modern MSP430 microcontrollers, several academic proposals for security features have been proposed on this architecture thanks to the extensible openMSP430 [14] softcore. The openMSP430 project is a mature, open-source re-implementation of the base 16-bit MSP430 instruction set with no built-in security features. This project has served as the basis of a long line of security research prototypes but has also enjoyed popularity in the industry, including applications in space [47].

As part of our work, we performed a literature study of MSP430-based security architectures that either provide memory isolation as their main feature or build on it to offer more advanced features, such as remote attestation or over-the-air updates. These systems are summarized in Table 1. We did not include openMSP430-based papers that only demonstrate the usage of security features in certain scenarios without changing the underlying hardware TCB [48]–[55].

This table provides several interesting insights. First, it is clear that research into the security features of low-end microcontrollers is an active area, and (open)MSP430 is a popular platform to build on. Second, systems such as Sancus [2] and VRASED [4] show that research can build on a useful security primitive to provide additional guarantees in follow-up work. In the case of Sancus, this primitive is an embedded TEE, whereas, for VRASED, it is a hardware-software co-design for remote attestation. Third, only a few systems comply with a commercial specification for isolation, such as IPE, most likely because most of the proposed security architectures require changes in the hardware. However, such custom, non-standard-compliant hardware extensions, reduce the chances of being adopted in the industry or run on off-the-shelf devices. Finally, multiple systems have known vulnerabilities (cf. Section 2.4), including IPE itself. This is especially concerning for systems that serve as a base for derived architectures, which may inherit such vulnerabilities. Moreover, systems building on vulnerable off-the-shelf IPE hardware can currently not be patched.

With openIPE, we aim to address some of these challenges, inspired by the observation that extensible security platforms enable a wide range of research. In our work, we replicate the behavior of TI's proprietary devices to an almost cycle-accurate level, making research findings on openIPE relevant to TI's devices. Importantly, openIPE offers open-source, extensible components for the entire system stack: hardware, firmware, compiler, software development framework, and application software.

2.4. Attacks and mitigations on MSP430

Commercial isolation primitives on microcontrollers and academic proposals show demand for secure isolation features in this space. Unfortunately, recent research has uncovered numerous vulnerabilities in these systems. Influential embedded research prototypes based on openMSP430, like Sancus [2] and VRASED [4] that have served as the basis for numerous derived systems, were shown vulnerable to microarchitectural side-channel leakage [21], [23], [24], subtle software sanitization oversights [22], [23], [32], and even design flaws that were not discovered by formal verification [23], [34], [35]. Furthermore, even the production-quality IPE has recently been shown [19], [20] to suffer from critical design oversights and implementation vulnerabilities opening up the possibility for remote exploits and eliminating its security guarantees and those of all systems making use of it.

Importantly, we observe a large overlap in the impacted security features and, therefore, hypothesize that many of these attacks could have been avoided with more coordination. In the following, we overview attacks and mitigations on MSP430-based systems and how they motivate openIPE's design goals.

Bootstrap loader attacks. Early work from Goodspeed et al. exploited timing side channels [76] and ROP-style attacks [77] against the TI serial bootstrap loader software to extract code from a locked MSP430 device. This highlights the need for principled validation and updateability of critical firmware components, as supported by openIPE.

Attacks on attestation. Castellucia et al. [78] demonstrated attacks on early software-based attestation schemes, one of which was implemented on an MSP430 microcontroller [79]. These attacks show the difficulty of providing security guarantees on microcontrollers without any isolation primitives like the one offered by openIPE.

Interrupt-latency attacks. Van Bulck et al. [21] first showed that subtle timing differences in interrupt handlers may reveal instruction lengths of protected Sancus enclaves. This attack has since been generalized to VRASED [23] and even TI IPE [19]. Sancus$_V$ provides a principled, provably secure hardware mitigation, which was, however, shown to be vulnerable to several implementation oversights [23], [34], [35]. This highlights the need for a single, unified and vetted prototyping framework such as openIPE.

Single entry point. Multiple works [19], [20] have independently exploited that TI's IPE does not properly enforce a single entry point and is, hence, vulnerable to advanced code-reuse attacks. In contrast, enforcing a single entry point is a well-understood design requirement for embedded TEEs [2]–[4], [13], and openIPE concretely implements such a single entry point as suggested in previous IPE security analyses [19].

Software sanitization oversights. Both the Sancus [22], [32] and VRASED [23] implementations were shown to suffer from subtle software interface sanitization bugs allowing secret leakage. Similarly, it has also been shown that the example IPE project provided by TI [80] also suffers from similar oversights [19]. We address these challenges by providing a unified software development framework for transparent interface sanitization, which can be further vetted using symbolic execution.

Interrupt register leaks. IPE has been shown to straightforwardly leak register contents on interrupts [19], [20], which was properly avoided in Sancus and VRASED via guaranteed atomic execution or custom secure interrupt extensions [4], [7], [8], [11], [33], [64]. We provide a unified research platform to implement and compare these proposals. As a concrete contribution in Section 6, we re-implement a previously closed-source secure interrupt proposal [7], as well as explore new points in the hardware-software co-design space.

TABLE 1. Overview of security architectures based on MSP430 with memory isolation. Systems with demonstrated vulnerabilities are highlighted (🐞). The columns indicate support for confidential (🔒) code; confidential data; dynamic configuration of isolation boundaries; whether the extension is implemented in hardware, software, or both; whether untrusted code can interrupt isolated code; whether the code is open source; and whether the system complies with a commercial specification for isolation.

	name	year	venue	🔒 code	🔒 data	dyn. 🔒	extension	untr. ISR	open src.	ind. spec.	attacks
openMSP430	SMART [3] 🐞	2012	NDSS	○	●	○	Hybrid	○	○	○	[4], [56], [57]
	↪ ERASMUS [58]	2018	DATE	○	●	○	Hybrid	○	○	○	–
	Sancus 1.0 [59]	2013	USENIX	○	●	●	Hardware	○	●	○	–
	↪ Soteria [60]	2015	ACSAC	●	●	●	Hardware	○	●	○	–
	↪ Towards Availability [11]	2016	MASS	○	●	●	Hardware	●	○	○	–
	↪ Sancus 2.0 [2] 🐞	2017	TOPS	●	●	●	Hardware	○	●	○	[21], [22]
	↪ Sancus$_V$ [33] 🐞	2020	CSF	●	●	●	Hardware	●	●	○	[23], [34], [35]
	↪ Aion [8]	2021	CCS	●	●	●	Hybrid	●	●	○	–
	↪ Authentic Execution [61]	2023	TOPS	●	●	●	Hybrid	○	●	○	–
	de Clercq et al. [7]	2014	ASAP	○	●	●	Hybrid	●	○	○	–
	VRASED [4] 🐞	2019	USENIX	○	●	○	Hybrid	○	●	○	[23]
	↪ APEX [57] 🐞	2020	USENIX	○	●	○	Hybrid	○	●	○	[23]
	↪ ASAP [62]	2022	DAC	○	●	○	Hybrid	●	●	○	–
	↪ RARES [63]	2023	arXiv	○	●	○	Hybrid	○	○	○	–
	↪ RATA [64]	2021	CCS	○	●	○	Hybrid	○	●	○	–
	↪ CASU [65]	2022	ICCAD	○	●	○	Hybrid	●	●	○	–
	↪ VERSA [66]	2022	S&P	○	●	○	Hybrid	○	●	○	–
	↪ ACFA [67]	2023	USENIX	○	●	○	Hybrid	○	●	○	–
	GAROTA [68]	2022	USENIX	◐	◐	○	Hybrid	○	●	○	–
	IDA [10]	2024	NDSS	○	●	○	Hybrid	●	○	○	–
	UCCA [69]	2024	TCAD	●	●	○	Hardware	●	●	○	–
	openIPE *(this work)*	2025	EuroS&P	●	●	●	Hybrid	●	●	●	–
TI MSP430	IPE [46] 🐞	2014	–	●	●	●	Hardware	○	○	●	[19], [20]
	↪ SIA [70]	2019	HOST	●	●	●	Software	○	○	●	–
	↪ SICP [71]	2020	JHSS	○	●	●	Software	○	○	●	–
	↪ Optimized SICP [72]	2022	TECS	○	●	●	Software	○	●	●	–
	↪ IPE Exposure [19] 🐞	2024	USENIX	●	●	●	Software	○	●	●	§4.2
	Hardin et al. [73]	2018	ATC	●	●	○	Software	○	○	●	–
	PISTIS [74]	2022	USENIX	●	●	●	Software	●	●	●	–
	↪ FLAShadow [75]	2024	TIOT	●	●	●	Software	●	●	●	–

Controlled `call` corruption. The most crucial vulnerability on IPE, which was assigned high severity by TI [81], is *controlled `call` corruption*, which can completely bypass the protection from software [19]. This vulnerability was found to be already mitigated via minimal hardware changes in both Sancus and VRASED. This highlights the advantage of a unified shared code base. Concretely, openIPE for the first time allows implementing and evaluating the previously hypothesized hardware changes [19] to mitigate controlled `call` corruption.

DMA access control. VRASED's DMA access control check was shown to suffer from a subtle implementation bug that was properly handled in both Sancus and TI IPE [23]. Furthermore, DMA has been abused as a capable side-channel attack primitive on Sancus, Sancus$_V$, and VRASED [23], [24]. We believe that openIPE facilitates specialized compile-time mitigations [24] that rely on a principled, open-source understanding of the microcontroller's internals and protection mechanisms.

Other systems. Given all these remarkable overlaps and further evidence from research [21], [22], [82], [83], we can only assume that other, closed-source prototype implementations that are more difficult to analyze likely suffer from similar recurring vulnerabilities.

3. openIPE design and implementation

To tackle the challenges outlined in the previous sections, we propose a hardware-software co-design for memory isolation based on the specification of TI's IPE. The advantage of such an open platform is that it concentrates the efforts on implementing, improving, and validating the security primitives that more complex systems can build on. By basing our system on openMSP430, we provide a platform that is familiar to many researchers in the area and has a proven track record.

Compared to many open-source security architectures in Table 1, openIPE has a number of advantages. First, openIPE offers compliance with an industry specification, potentially leading to a more direct industry impact and enabling the extension and comparison of security guarantees in a unified manner. Second, our design features a flexible firmware layer inspired by IPE's bootcode, allowing rapid prototyping of security features without requiring extensive hardware changes, which are often more complex to carry out and more challenging to validate. In Section 6, we perform a case study in secure interrupts to showcase this rapid prototyping and evaluation capability.

Based on the IPE Exposure work [19], we also implemented an open-source toolchain and software development framework, largely providing source-level compatibility with current IPE projects that can be recompiled to run on openIPE. Relying on a fully open-source toolchain also enables research on compiler modifications [24], [84], which is currently hindered on commercial IPE platforms due to reliance on the proprietary TI compiler.

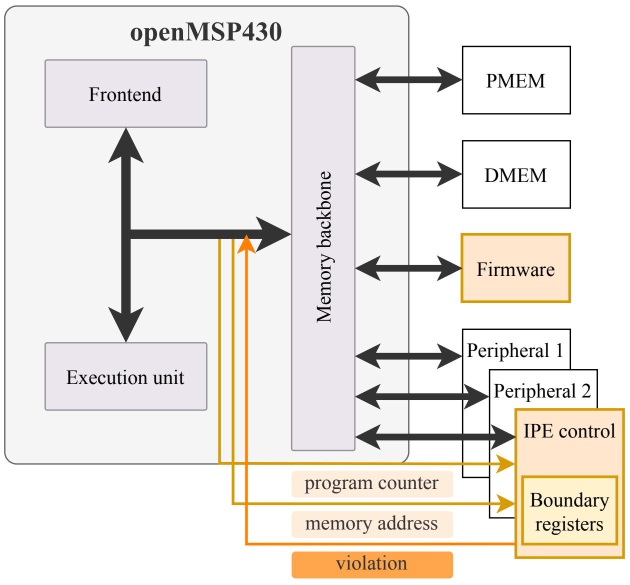

Figure 3. Our design extends the openMSP430 core with a *Firmware* memory and an *IPE control* peripheral, implementing program-counter-based access control based on the highlighted added connections.

3.1. Deployment and threat model

As our goal is to enable research and extensions on the isolation provided by IPE, we aim to stay as close as possible to its deployment and threat model involving multiple stakeholders, with some deviations necessary for practical reasons. When considering software adversaries, we offer the same protection as intended by IPE: we consider *all* untrusted software on the device, the DMA controller, and the openMSP430 debug unit to be compromised.

As our prototype implementation is not a physical device, protecting against physical attacks on the memory as offered by TI is outside our scope. In a real deployment scenario, it could be possible to implement our design on an FPGA or as an ASIC and connect an external memory chip with similar security properties as TI's FRAM. Similarly, our small trusted firmware layer, which can be implemented as an external memory unit, could also benefit from similar physical protections. Investigating and implementing these protections is left as future work, in our contribution we focus on software-driven attacks.

3.2. Processor architecture

As numerous academic prototypes [2]–[4], [7], [10] discussed earlier, we implement our design on the open-source openMSP430 [14] softcore. Figure 3 overviews the high-level design, highlighting the main components and connections added to the openMSP430 architecture. In broad terms, the main changes are a new *IPE control* peripheral that implements the access-control logic based on the configuration registers (Section 3.3) and the *firmware* that is securely executed on device reset (Section 3.4).

To stay as close as possible to TI's specification and preserve compatibility with existing IPE software, we performed a number of modifications in the openMSP430 core. Notably, our design does not necessitate any new instructions and remains fully backward compatible with

TABLE 2. ACCESS CONTROL RIGHTS (READ, WRITE, EXECUTE) FOR THE NEWLY INTRODUCED FIRMWARE AND IPE MEMORY REGIONS.

From \ To	**Untrusted**	**Firmware**	**IPE**	**IPE entry**
Untrusted	`rwx`	`r--`	`---`	`--x`
Firmware	`rxw`	`rwx`	`rwx`	`rwx`
IPE + entry	`rwx`	`r--`	`rwx`	`rwx`
DMA	`rw-`	`r--`	`---`	`---`
Debug unit	`rw-`	`r--`	`---`	`---`

existing openMSP430 software, being implemented solely through new memory-mapped I/O registers. To support IPE code and data being placed in a continuous memory region as on TI's FRAM chip, we modified the openMSP430 memory backbone and two-stage pipeline to support write operations to the originally read-only program memory, enabling dynamic data updates for the secluded IPE region. In addition, we extended the size of the peripheral space to allow placing the IPE configuration registers at the same addresses as on TI's microcontrollers.

3.3. Memory access control

An important aspect of our system is the implementation of memory access control based on the IPE specification and extended to the trusted firmware region. Access control is largely handled by the IPE control peripheral, shown in the bottom right of Figure 3. This peripheral stores the configuration registers and is connected to the CPU by several monitoring and control signals, signaling access violations to the memory backbone. Table 2 summarizes the access-control rules enforced by openIPE. Untrusted code or peripherals are prohibited from writing to or executing the newly introduced firmware memory partition, whereas reading is permitted, as maintaining the secrecy of the firmware is not a design objective in our current implementation. The privileged firmware has unrestricted access to the entire memory including the IPE region, but untrusted code cannot read IPE memory and can only jump to a newly introduced single entry point at the start of the IPE region (cf. Section 3.5). Similar to TI's design, openIPE currently does not enforce read-write-execute separation within the single IPE region, leaving this as a potential area for future work.

The backbone immediately returns the value `0x3FFF` for illegal reads without forwarding the operation to the memory, preventing possible microarchitectural leakage by the resulting value's propagation. Illegal stores are also suppressed. These access control checks are performed for accesses by the debug unit and DMA requests as well. Moreover, during the execution of the IPE region, the debug unit is disabled, preventing it from leaking or corrupting register values.

On TI devices featuring a 20-bit address space, the configuration registers store the most significant 10 address bits, aligning the region to 1 kB boundaries. In our implementation, we adhere to this design for compatibility reasons, but as the openMSP430 only has a 16-bit address space, storing the full boundary address and implementing byte-granular protection boundaries is an interesting alternative in future work.

3.4. Secure firmware

When TI devices are reset, the immutable *bootcode* is responsible for setting up the IPE registers before user code can execute or the debugger can attach. With openIPE, we took a more general approach: we provide a flexible protected firmware layer that can implement this task, but can also be extended to offer broader functionality and security features. This firmware layer is stored in a separate memory partition with a single entry point and can contain arbitrary machine code that sets up the CPU state, or, as seen in Section 6, can even be vectored to during runtime. In the base configuration, openIPE contains our open-source implementation of TI's proprietary bootcode, reconstructed based on the available documentation [45]. Implementing the firmware in MSP430 assembly code (or compiling it from C) enables rapid prototyping of extensions to it, and the use of software testing and verification tools ensures its correctness and enforced security guarantees.

Importantly, as shown in Table 2, the firmware memory must have protections similar to those of the IPE region to safeguard against tampering by attackers, which could compromise system security by, for instance, manipulating the setup of the IPE configuration registers.

Figure 3 shows the firmware memory as a separate memory partition connected to the core. In practice, this memory could be implemented in different ways. One option is connecting an external writeable non-volatile memory, such as EEPROM or FRAM, possibly including the physical protections offered by TI [46], which could also enable firmware updates to the device. The firmware could also be implemented on the same persistent memory chip as the one used for user code and data.

Security considerations. As the firmware has complete access to the protected IPE section, it inevitably opens up a new attack vector for compromising the IPE protection. This is similar to how the MSP430 bootloader was exploited to read out protected Flash memory on earlier MSP430 devices [76], [77]. The base IPE firmware is relatively simple, and symbolic execution (cf. Section 4.2) can be used to validate that it does not perform any dangerous memory accesses.

For firmware extensions and real hardware deployments, a number of additional steps can be taken to reduce the security risks. First, reducing access rights to both the IPE region and the firmware memory itself might be possible depending on the desired functionality (e.g., making parts of the firmware non-writeable or non-executable). Second, extensions to the firmware should be tested with the provided security testing mechanisms (Section 4.2), which can be extended further if new functionality is provided by the new firmware code. Third, additional security measures need to be taken if physically switching out the firmware memory is a potential threat. This could be achieved either with physical protections or the hardware could be extended with a secure boot-type mechanism that verifies the authenticity of the firmware before execution. We note that our current threat model (cf. Section 3.1) does not provide protection against physical attackers that are capable of reading out the IPE memory directly.

3.5. Security improvements over TI IPE

In addition to implementing the base IPE specification, we integrated hardware fixes suggested by IPE Exposure [19] in response to vulnerabilities uncovered on TI devices. Our framework, for the first time, enables the analysis and comparison of these hardware fixes with the software mitigation framework previously proposed as a stopgap solution. Moreover, we implemented an automated stack pointer switching mechanism between the IPE region and untrusted code, similar to other solutions in the literature [7].

Single entry point. As opposed to research TEEs [2]–[4], [13], off-the-shelf TI devices have been shown to allow jumps to *any* address inside the IPE region, leading to practical code re-use attacks [19], [20]. As a mitigation, we modify the IPE access control logic to restrict access to the IPE section through a single predefined entry point, referred to as *IPE entry* in Table 2. Any illegal jumps trigger a non-maskable interrupt (NMI), allowing the software to recover without causing a device reset. Care needs to be taken to enforce the single entry point for all control-flow transfers, not just direct jumps. This includes return from interrupt (`reti`) instructions and interrupt vector table (IVT) entries, including the NMI handler itself.

We chose the location of the entry point to be placed 8 bytes after the start of the IPE region, automatically calculated from the configuration registers. In practice, this is similar to TI devices, where the convention is to place the IPE configuration structure in the first 8 bytes of the protected region. Using the software framework of IPE Exposure [19], software can still vector to multiple logical entry points through the single physical entry point.

Mitigating controlled `call` corruption. Controlled `call` corruption is an attack that uses the `call` MSP430 instruction to corrupt memory inside the IPE region, leading to a complete loss of security guarantees. When performing a function call, the CPU writes the return address on the stack. On microcontrollers with IPE, the access control rules for this write were found to not always be enforced, enabling untrusted code to corrupt protected code and data using a poisoned stack pointer register pointing to IPE, making it possible to leak the contents of the entire IPE region [19].

The cause of the vulnerability was identified to be incorrect handling of the program counter register during the execution of a `call` instruction [19]. As a mitigation, buffering the program counter register was proposed, keeping its value stable until the execution of the `call` instruction finishes. Interestingly, it was shown that both Sancus [2] and VRASED [4] contain similar changes in their implementation, avoiding this vulnerability.

Since no open-source implementation of IPE previously existed, the cost and effectiveness of this mitigation could only be approximated [19]. We implement this suggested mitigation and demonstrate that it effectively mitigates the attack with minimal hardware overhead.

Automated stack switching. Controlled `call` corruption is not the only attack enabled by attacker-poisoned stack pointer values. Previous research has identified vulnerabilities in Sancus and VRASED that involve invoking the enclave with poisoned stack pointer values, possibly resulting in the leakage of part of a secret key [22], [23].

While the IPE Exposure mitigation framework implemented secure stack switching in software, a more efficient and less error-prone solution is to perform it from hardware on context switches [7], [85]. In our work, we implemented this automatic stack pointer switching mechanism in hardware, reducing the amount of sanitization code required in the mitigation framework and enabling better security guarantees, for example in the context of secure interrupt handling (cf. Section 6).

3.6. Software toolchain

While we cannot offer full binary compatibility with TI IPE due to practical limitations of the openMSP430 core, such as lack of support for the 20-bit address space and the extended MSP430X instructions [45], we strive to provide full software compatibility for C projects, only requiring recompilation of code written for TI. Our software development framework is based on TI's example IPE project [15] and the software mitigation framework introduced in IPE Exposure [19]. Our framework transparently inserts hand-crafted assembly stubs on IPE entry and exit transitions, taking care of low-level concerns such as vectoring multiple logical entry functions through the single physical entry point and sanitizing and cleansing CPU registers [22]. Importantly, openIPE development is fully compatible with open-source compilers for MSP430, opening up the space to investigating and applying compiler-based mitigations and extensions that require changes in the compiler [24], [84].

As part of our work, we discovered and fixed various bugs in the open-source MSP430 ecosystem, including the `angr` MSP430 backend and the original IPE Exposure framework. Additionally, we discovered and reported a null pointer dereference bug in the proprietary disassembler tool shipped by TI, which has been confirmed by the developers, but to the best of our knowledge, has not yet been fixed. These contributions underline the importance of having an open-source toolchain for development, where the community can contribute to fixing issues and improving the development tools.

4. Functional and security testing

As evidenced by prior work [23], inductive testing and deductive methods are complementary approaches for early detection of implementation bugs and mismatches with the specification in security architectures. Following this advice, we took a multi-faceted approach to validating the correctness and security of our implementation, using a combination of diverse unit testing and symbolic execution with `angr`. By using multiple tools, we can test changes across all layers of our implementation: hardware, firmware, and software.

4.1. Unit test framework

The original openMSP430 core includes a regression test suite of 94 test cases, 62 of which are applicable to our Verilog simulator setup. These tests can be run using the cycle-accurate `iverilog` simulator. First, we validated that all of these tests finish successfully on openIPE, even with the IPE firmware implementation included, validating that our changes do not interfere with the normal execution of the device when IPE is not enabled.

TABLE 3. Overview of the unit tests.

# tests	Tested functionality
62	Original openMSP430 regression tests
6	IPE boundary setup and modification
3	IPE protection from untrusted code
3	IPE protection from the debugger
2	IPE protection from DMA
1	Allowed internal IPE access
4	IPE protection from known attacks
4	Protection of the firmware region
3	Secure interrupt handling case study (Section 6)

To validate the correct behavior and security guarantees of our implementation, we extended this test suite with 26 new tests, summarized in Table 3. These tests examine the behavior of the bootcode implementation and the memory access control logic, including DMA and debugger accesses to both the IPE and the firmware region. Additionally, some of the tests confirm the presence of known vulnerabilities on the base IPE platform, such as controlled `call` corruption or unrestricted entry points [19], [20]. These tests are subsequently used to validate the effectiveness of the security improvements added in Section 3.5.

These unit tests are run in the continuous integration (CI) environment of the open-source openIPE release and can serve as the basis for evaluating the compatibility and correctness of future openIPE extensions. In this spirit, three of the unit tests relate to the extensions developed as part of our case study (cf. Section 6).

4.2. Symbolic execution

Framework. Recent findings [28]–[30] on Intel SGX have shown that symbolic execution tools are particularly effective at detecting software sanitization vulnerabilities in TEEs. Even for MSP430 applications, symbolic execution has been used in the past to discover memory safety issues [86]. For our work, we extended Pandora [28], a recent open-source symbolic execution tool that was specifically designed for the principled validation of Intel SGX enclaves. We selected Pandora because it includes a capable plugin system to detect a wide range of vulnerabilities, including confused-deputy pointer dereferences, control-flow hijacking, and register sanitization bugs. Additionally, Pandora features a mature command-line interface and can generate actionable, human-readable analysis reports. While initially limited to validating Intel SGX enclaves, recent work [32] has extended Pandora with a hardware abstraction layer, which we utilize for our openIPE extension. Specifically, we implemented an openIPE enclave loader extension to automatically recognize openIPE binaries, expose protection boundaries to Pandora's enclave-aware symbolic memory model, and accurately model execution semantics according to our implementation with security improvements. Our loader recognizes both firmware and application binaries, allowing the security validation of the critical bootcode as well as IPE applications, including the entry and exit assembly stubs inserted by the software framework.

Under the hood, Pandora utilizes `angr` [31] as its symbolic execution engine. Unlike alternative tools like KLEE [87], `angr` performs symbolic execution at the binary level, enabling it to reason about low-level concerns such as compiler optimizations and CPU register cleansing. However, during our research, we found `angr`'s MSP430 backend to be unstable. To support future symbolic execution research on MSP430 platforms, we contributed essential patches to upstream `angr`, fixing several bugs that led to the incorrect lifting of MSP430 assembly code into the VEX intermediate representation.

Evaluation. We validate both the firmware code developed in assembly and a sample IPE application developed in C and compiled with our framework, which includes the critical entry and exit stubs. We focus our symbolic exploration on validating the basic firmware and assembly stubs, i.e., excluding the extensions for secure interrupts (Section 6), which could potentially be validated similarly. We utilize Pandora's built-in `ptrsan`, `cfsan`, and `abisan` plugins to rule out *(i)* confused-deputy attacks via attacker-tainted pointers resolving within the currently executing region; *(ii)* arbitrary control flow hijacking; and *(iii)* leakage through unscrubbed registers during context switches, respectively. These issues have been shown to cause severe vulnerabilities in both openMSP430-based systems [22], [23] and TI IPE [19].

Notably, `abisan` autonomously discovered a subtle bug[1] in the previously published IPE Exposure [19] framework where the secret registers were not properly cleared during one of the exit flows from IPE, clearly showing the power of this approach. This specific bug, which we reproduced on a TI microcontroller and patched in our framework, left five registers uncleared after the invocation of an IPE function, leaving their contents exposed to the calling untrusted code. Depending on the implementation of the called IPE function, this might leak secret data to the calling code.

When validating the firmware, `ptrsan` correctly identified the potential for confused-deputy reads through the attacker-provided IPE configuration structure (cf. Section 2). However, since we explicitly excluded the secrecy of the firmware from our objectives, we determined that these instances do not require mitigation.

Discussion. Symbolic execution is notorious for suffering from state explosion. However, we found this to be less of a concern for the small embedded programs we aim to validate, specifically the firmware and IPE assembly entry/exit stubs. When run on a minimal IPE "hello world" C program, our Pandora port finishes in seconds, making it suitable to run in a CI environment. Similarly, validating the main firmware execution path completes in seconds, but the failure path, which includes a mass-erase loop to clear the entire program memory (cf. Section 2), would require much longer.

Additionally, symbolic execution is known to produce false positives. Indeed, our modified `abisan` reports false-positive warnings for non-zeroed registers that are used as return values or untrusted bridge parameters. Furthermore, the `ptrsan` plugin flags the mass-erase loop as a false positive when validating the firmware and warns for a benign attacker-provided index in the dispatch table with logical entry points when validating the IPE stubs. The `cfsan` plugin similarly reports a false-positive warning for benign callbacks to untrusted code. Thanks to Pandora's detailed and interactive analysis reports, we were able to efficiently sift through these false positives.

Similarly to the unit test suite, we integrated our symbolic execution framework into openIPE's CI environment. We intend this to be extended in the future to provide additional security guarantees, possibly relating to future openIPE extensions.

1. https://github.com/martonbognar/ipe-exposure/commit/92c01ef2

TABLE 4. HARDWARE COST OF OUR IMPLEMENTATIONS IN LOOK-UP TABLES (LUTS) AND FLIP-FLOPS (FFS). OPENIPE REFERS TO OUR DESIGN WITH ALL HARDWARE-BASED SECURITY IMPROVEMENTS OF SECTION 3.5.

Design	LUTs	Δ LUTs	FFs	Δ FFs
openMSP430 (baseline)	2,311	-	1,110	-
IPE specification	2,510	+8.6%	1,162	+4.7%
openIPE	2,582	+11.7%	1,191	+7.3%

5. Performance evaluation

This section evaluates the changes of openIPE to the openMSP430 hardware and the IPE Exposure software mitigation framework, including the security improvements over TI's implementation (Section 3.5). The evaluation of our case study in secure interrupts follows later in Section 6.

5.1. Hardware cost

When proposing extensions to the hardware, the incurred overhead on the physical realization of the design is an important metric for evaluating the feasibility of the proposal. To evaluate this cost for our design, we follow the strategy of related work in the literature [4], [59] and report on the number of look-up tables (LUTs) and flip-flops (FFs) used when synthesizing the hardware design for an FPGA. Concretely, we synthesize variants of our design for the Basys 3 FPGA using Xilinx Vivado 2024.2 configured with the default optimization strategy.

Table 4 summarizes the results of this evaluation, showing synthesis results for designs with a connected 1024-byte external firmware memory, such as an EEPROM chip. Compared to the unmodified openMSP430 core, shown in the first line, implementing the (insecure) IPE specification incurs an 8.6% overhead in the number of look-up tables and a 4.7% overhead in the number of flip-flops. Adding the security improvements to enforce a single entry point to the IPE region, mitigate the controlled `call` corruption attack, and perform automated stack switching increases the overhead only slightly, mostly due to storing the extra stack pointer register.

Including the firmware memory using RAM blocks on the FPGA naturally increases the overhead. Adding a 128-byte firmware block to openIPE (which is enough to store our implementation of IPE bootcode) uses 2640 LUTs and 1197 FFs, while a 1024-byte firmware memory uses 2900 LUTs and 1200 FFs (in addition to 32 and 256 memory blocks, respectively).

When comparing the hardware cost of openIPE to other openMSP430-based systems, we see that it is similar to the reported overhead of VRASED (+7% LUTs, +5% FFs [4]) while offering a more flexible isolation primitive, and is much smaller than Sancus (+62% LUTs, +80% FFs [59]). It is important to note that hardware cost is not a straightforward comparison, as the different systems serve different purposes and extend different versions of the openMSP430 core.

5.2. Binary size and runtime overhead

Besides hardware cost, another important evaluation aspect is the size and execution time of the additional software introduced by openIPE. First, we evaluate our implementation of the IPE bootcode. Second, we measure the impact of our changes on the assembly stubs used by the IPE Exposure framework [19]. All execution time measurements in this section were obtained through the cycle-accurate openMSP430 simulator.

Bootcode. The performance impact of the IPE bootcode is negligible. Our implementation consists of 78 bytes of code, which takes at most 58 cycles to execute unless a mass reset needs to be performed as a result of misconfiguration or tampering.

Case study: code attestation. To evaluate the impact of our changes on the mitigation framework, we ported VRASED's remote attestation primitive to openIPE as a case study. VRASED [4] provides a hardware-software co-design for attestation to validate the integrity of untrusted code running on the device. For this purpose, they apply an HMAC algorithm to the untrusted memory, provided by the HACL* library [88]. To protect the software running the attestation, they modify openMSP430 to provide a read-only code memory region for the attestation code, together with a private data memory section.

In IPE Exposure [19], this application has been modified to make use of the isolation primitive of IPE, co-locating the code and the stack in the isolated IPE region, performing attestation on the code and data outside the IPE region. We used this IPE-based version to show that IPE applications written in C can be recompiled to run on openIPE as long as they do not use special peripherals unavailable on openMSP430. Interestingly, openIPE also enables the possibility of moving the attestation code to the firmware, which in turn would enable the attestation of IPE-protected applications or a secure boot mechanism.

Case study results. Compared to the IPE Exposure mitigation framework [19], our security improvements in hardware allow us to relax the responsibilities of the software, reducing the overhead of the framework. The most significant improvement comes from the fact that IPE Exposure had to rely on the MPU to provide protection against the controlled `call` corruption vulnerability and to enforce the single entry point. These features necessitated performing a brown-out reset on every context switch from untrusted to IPE code to remove the MPU protection. Eliminating these brown-out resets improves performance, as they took between 0.3-0.6 ms to complete on TI devices [19]. Moreover, openIPE is better suited for implementing systems with hard real-time guarantees that could not be ensured with nondeterministic reset times.

Table 5 shows the assembly stubs of the mitigation framework and microbenchmarks obtained from IPE Exposure [19], together with our evaluation after adapting the framework to openIPE. These results show that even beyond eliminating the brown-out resets, the hardware mitigations such as the automated stack switching enable further optimizations in the code.

As an end-to-end benchmark, we also measured the execution time of the VRASED attestation routine on openIPE, which was used as the macrobenchmark of IPE Exposure. We measured an execution time of 3,651,333 cycles for the attestation of a 2 kB region, corresponding to a wall clock time of 456.42 ms at 8 MHz, a figure in line with the numbers reported in IPE Exposure and the original VRASED benchmark [4]. As the execution time of some instructions differs between TI MSP430 and openMSP430 [14], the 43 ms speedup compared to running the code on TI devices cannot be solely attributed to the improvements in the framework and the lack of brown-out resets, but they are certainly a factor.

6. Case study: Secure interrupt handling

Interrupt handling is a crucial feature to enable real-time functionality in microcontrollers, which is often required in safety-critical applications [7]. At the same time, numerous studies have demonstrated that enabling interrupts can compromise architectural [19], [20] and microarchitectural [21], [23] security. As a result, many security architectures choose to disable interrupts.

In the following, we survey different approaches to secure interrupt handling from the literature and implement three of them on openIPE. Additionally, we design and implement a secure interrupt handling scheme that utilizes openIPE's flexible firmware layer to act as a trusted security monitor and handle interrupts. As part of our case study, we compare the security guarantees and the overhead of these different approaches.

Security concerns. Improper handling of interrupts can cause security violations in different scenarios. First, the attacker may have control over an untrusted interrupt service routine (ISR) that gets triggered during the execution of IPE code. From the ISR, the attacker can then read out the current register values or corrupt them, leading to secret leakage or other unintended behavior if the register values are not cleared and restored before and after executing the untrusted ISR [19], [20], [89].

Second, the interrupt handling mechanism itself can leak information. Nemesis [19], [21] is a microarchitectural attack that exposes timing differences in the interrupt handling mechanism based on the currently executing instruction in the IPE region at the time of the interrupt. The attack reveals the execution time of this individual instruction, which, if it depends on a secret condition, can leak this information.

TABLE 5. OVERVIEW OF ASSEMBLY STUBS IN OUR FRAMEWORK, COMPARED TO THOSE REPORTED FOR THE IPE EXPOSURE FRAMEWORK. STUBS MARKED WITH A (*) COULD BE COMPLETELY ELIMINATED THANKS TO THE HARDWARE IMPROVEMENTS (SECTION 3.5).

Stub	# instances	Old size	Old execution time (cycles)	New size	New execution time
`ipe_entry`	global	76 B	63	**84 B**	**62**
`ipe_ocall`	global	60 B	51	**56 B**	**50**
`ocall_stub`	per ocall fn	20 B	25	**20 B**	**23**
`ecall_table`	per ecall fn	6 B	-	**4 B**	**-**
`ipe_ocall_cont`	global	72 B	54	**6 B**	**6**
`ecall_ret`	global	68 B	51	**2 B**	**3**
Untrusted `ecall_stub`	per ecall fn	12 B	16	**12 B**	**13**
`_system_pre_init` (*)	global	72 B	28/42	-	-
`_system_post_cinit` (*)	global	36 B	28	-	-
`reset_into_ipe` (*)	global	66 B	49	-	-
`new_reset_isr` (*)	global	120 B	45/62	-	-

Third, an improperly set up stack pointer in combination with interrupt handling can also lead to security issues. On MSP430, when handling an interrupt, the interrupted instruction's address and the status register are backed up on the stack, pointed to by the stack pointer register. If the attacker can control the value of the stack pointer, e.g., because there is no automated stack switching mechanism during context switches, this behavior can be exploited. By pointing the stack pointer inside the IPE region and triggering an interrupt after calling protected code, the attacker can force the interrupt handler mechanism to corrupt code or data inside the region. On TI IPE, this attack reportedly does not succeed [19] (which also means that interrupt handling will always fail if the stack points inside the IPE region as the ISR cannot return), but it does on our base implementation. On the other hand, if the stack pointer points to unprotected memory when the IPE code is interrupted, the backed-up values of the program counter and the status register will be visible to the attacker, which can again leak information. These attack vectors show the importance of correctly managing the stack pointer across context switches.

6.1. Disabling interrupts in software

The most straightforward solution, also employed by the IPE Exposure [19] framework, is to disable interrupts from software. This is also the recommended approach by TI to "ensure that the registers or RAM are cleared before servicing any ISR outside of the IPE region" [15]. The IPE Exposure framework includes instructions to disable both maskable and non-maskable interrupts on context switches to the IPE region, either when calling a protected function or when returning from an outside call. This mitigation can be easily adapted to openIPE by changing the instructions managing non-maskable interrupts.

Of course, applying this mitigation comes with the obvious downside of not being able to handle interrupts during IPE execution, which might be prohibitive for certain applications. Disabling interrupts limits the functionality of the protected code, as it cannot communicate with peripherals that rely on interrupts to communicate with the CPU, or reduces performance by relying on polling instead. Moreover, it makes time-sharing the CPU very difficult, as an operating system cannot use interrupts to pause the execution of the enclave to run another process. A malicious enclave can also easily hijack the CPU and never return control to untrusted code, although this is already a concern on MSP430 as untrusted code can always disable interrupts.

There is an additional risk that an interrupt might be triggered and handled between the start of IPE execution and the interrupt-disabling instructions finishing. Especially if automatic stack switching is not enabled, this can already open up the device to some of the previously described vulnerabilities.

6.2. Disabling interrupts in hardware

Instead of relying on the previous error-prone approach of disabling interrupts from software, the hardware can also be modified to disable interrupt handling during the execution of protected code. This is the approach taken by VRASED [4] and the original Sancus [2] architecture to avoid secret leakage with minimal hardware modifications. Of course, this comes with the same limitations in functionality as disabling interrupts from software.

6.3. Secure interrupts

In prior work, de Clercq et al. investigated secure interrupts on low-end embedded devices with enclaved execution [7]. Their goal was to share the responsibilities of interrupt handling between trusted and untrusted code, enabling interrupt requests during the execution of both while allowing ISRs to reside in either section. They proposed two approaches, called "software-based" (which we will call `SW-IRQ`) and "hardware-based" implementations. The two designs are very similar and have the same goals, with `SW-IRQ` aiming to minimize the amount of required hardware changes. In our work, we focus on this implementation as it offers more extensibility. Unfortunately, the authors did not release their code. As part of our case study, we implemented `SW-IRQ` on openIPE while staying as close as possible to the original design, making it the first available implementation of this system which could inspire further research in this domain.

Figure 4 shows the main components of the `SW-IRQ` design. Both security domains have their own IVT, and, depending on the currently executing domain at the time of the interrupt, the hardware selects the handler from the corresponding IVT. As a convention, we store the secure IVT at the highest addresses of the IPE region, determined by the configuration registers. We modify the software

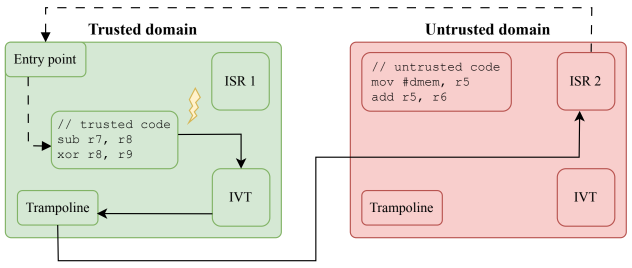

Figure 4. Control flow transfers on interrupts in SW-IRQ, showing how an untrusted ISR is invoked during enclaved execution. Solid lines indicate jumps and dotted lines indicate returns.

framework to add trampoline functions in both domains that handle the case in which a context switch is required to handle the interrupt. The trampoline inside IPE needs to back up and clear registers before invoking the untrusted ISR. The entry stub of the framework is also modified to handle the case when the IPE is invoked to handle an interrupt from untrusted code, or when control is returned after handling an interrupt in untrusted code.

While this solution offers strong isolation guarantees and a high degree of flexibility, it lacks certain availability guarantees. The two domains rely on each other's cooperation to set up the IVT and trampoline functions correctly, meaning that malicious untrusted code can prevent interrupts from being handled in the IPE region by replacing the entries in the untrusted IVT.

The "hardware-based" approach by de Clercq et al. uses a single IVT and additional hardware modifications to perform the vectoring based on whether a context switch is required. Their paper does not mention where this IVT should be located, but if untrusted software can access it, it suffers from the same limitation as SW-IRQ.

6.4. Novel firmware-based solution

Inspired by these prior proposals, we design a new secure interrupt scheme, FW-IRQ, outlined in Figure 5. The main insight we offer is that openIPE's firmware layer can be used as a trusted secure monitor that is responsible for handling the interrupt vectoring. Our design splits the IVT into two parts, both the IPE enclave and the untrusted code can register handlers that are located in their own domain. The firmware maintains for each interrupt source whether the corresponding handler is stored in IPE or untrusted code. Upon an interrupt, the control flow is always redirected to the firmware, which executes an interrupt dispatcher function that checks whether a context switch is required for handling the interrupt, saving and clearing registers accordingly. Similarly to SW-IRQ, small modifications in the IPE stubs and untrusted code are required to correctly handle all control transfers, and a second entry point is added to the firmware for interrupts.

By moving the interrupt handling responsibilities to the firmware, we strengthen the availability guarantees compared to the SW-IRQ approach, making designs with enforceable (and secure) resource sharing possible [6], [8], [11], [12]. Moreover, by using the firmware for this functionality, we could quickly prototype our design and enable further modifications and improvements. As an

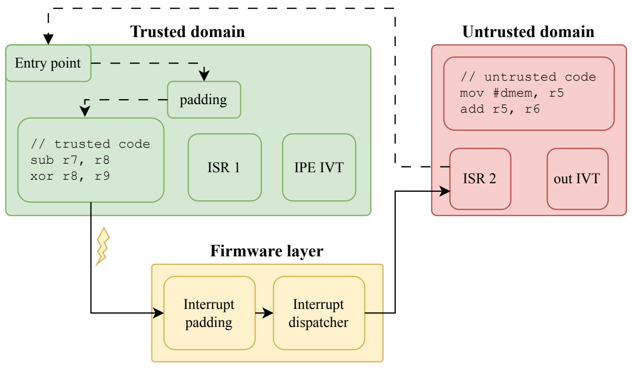

Figure 5. Control flow transfers on interrupts in FW-IRQ when vectoring from trusted code to an untrusted handler. Solid lines indicate the flow to the handler, and dotted lines indicate the return flow.

example, if support for multiple distrusting enclaves was added to openIPE, the firmware could be modified to contain a mapping of interrupt sources to different enclaves.

Eliminating side-channel leakage. To showcase the strength of extensibility in the firmware-based approach, we integrate a defense against the Nemesis interrupt latency attack [21], which has been extensively demonstrated on a range of low- and high-end CPUs, including Sancus [21], VRASED [23], and TI IPE [19]. Our defense is inspired by the design of Sancus$_V$ [33], a hardware-based extension that adds a variable amount of padding to the interrupt handler logic to mask the execution time of the interrupted instruction. This padding needs to be applied before handing control over to an untrusted ISR that could deduce the execution time of the interrupted instruction by measuring the start of its own execution. In addition, the padding needs to be complemented by additional padding cycles before returning from the interrupt handler to ensure that the execution time of the rest of the enclaved code also remains constant.

We divide the responsibility of padding between the firmware and the IPE software itself. When enclaved execution is interrupted, the firmware layer calculates the interrupted instruction's execution time based on the current timer value and performs the appropriate padding with no-op instructions. To handle the return padding, the firmware writes the number of required padding cycles into IPE memory, which is used by the IPE entry point logic after the return from interrupt for a similar padding routine before continuing the execution.

Our prototype implementation suffers from a number of limitations that could be addressed in future work. The padding is currently only performed for interrupts caused by the primary timer peripheral of openMSP430 and requires that the timer's period is set higher than the longest available instruction (≥ 7 cycles). An alternative approach could perform the padding based on examining the interrupted instruction (the address of which is saved on the stack) and determining its execution time. This would make the mitigation agnostic from the timer at the cost of added complexity.

TABLE 6. THE EFFECT OF DIFFERENT APPROACHES TO SECURE INTERRUPT HANDLING ON FUNCTIONALITY AND SECURITY.

Approach	Secure scheduling	Architectural protection	Nemesis mitigation	Untrusted ISRs
Software disable	○	◐	●	○
Hardware disable	○	●	●	○
`SW-IRQ` [7]	◐	●	○	●
`FW-IRQ`	◐	●	●	●

TABLE 7. OVERHEAD OF DIFFERENT INTERRUPT HANDLING APPROACHES ON OPENIPE.

Design	LUTs	FFs	Δ Software
openIPE (baseline)	2,582	1,191	–
Software disable	–	–	8 bytes / 6 cycles
Hardware disable	2,581 (-1)	1,191	–
`SW-IRQ`	2,597 (+15)	1,191	282 bytes / 198 cycles
`FW-IRQ`	2,577 (-5)	1,190 (-1)	674 bytes / 417 cycles

6.5. Evaluation

Table 6 summarizes the different approaches we implemented on openIPE. We see that primitive approaches that completely disable interrupts cannot be used to implement secure scheduling where the scheduler runs in an enclave and can interrupt untrusted code on a compromised device [7], [8], [12]. However, without interruptible enclaves, the Nemesis attack is also impossible to carry out. As explained earlier, disabling interrupts from enclave software is an error-prone process, which cannot necessarily guarantee the protection of IPE memory unless stack switching is implemented in hardware.

A current limitation of all our secure interrupt schemes is that, as on the original openMSP430, interrupts can be disabled from untrusted code and nested interrupts are also not supported. This enables denial-of-service attacks that trigger an untrusted ISR and never return control to the CPU. To address this issue, additional minimal hardware extensions for bounded atomicity [8], [12] could be implemented on openIPE, thereby enabling hard real-time applications.

Software and hardware overhead. Table 7 shows the results of our evaluation of the software and hardware overheads of the different approaches we implemented (without focusing on optimizations) on openIPE. All proposals incur negligible or no additional hardware cost when evaluated with our strategy from Section 5, and interestingly, `FW-IRQ` even outperforms `SW-IRQ`. Our evaluation of `SW-IRQ` gives similar results as those reported in the original paper, +10 LUTs and +2 FFs [7]. This is in contrast to their hardware-based approach, which, while offering weaker security guarantees than `FW-IRQ` (no Nemesis defense), incurs an overhead of +186 LUTs and +34 FFs. Sancus$_V$, the hardware-based Nemesis defense requires +142 LUTs and +260 FFs, mainly due to saving the register state in hardware [33].

While `FW-IRQ` has the smallest hardware overhead, transferring the responsibilities to the firmware is a trade-off in terms of code size and execution time. The software overhead is clearly the lowest for the approaches disabling interrupts completely, as these do not need code to handle the context switches between domains. `SW-IRQ` and `FW-IRQ` duplicate the IVT and add code in the IPE and the untrusted region. `FW-IRQ` additionally adds code for the interrupt padding and dispatching in the firmware. The table lists the worst-case execution latency between the arrival of an interrupt and the start of the ISR's execution. The main reason for the worse performance of `FW-IRQ` is due to the calculation and execution of the interrupt latency padding, which we also did not spend time optimizing. While these latencies are much larger than insecure context switches (6 cycles), they are deterministic, making it possible to apply these mitigations in systems with hard real-time requirements that need knowledge of worst-case latencies.

7. Future work

Our extensible memory isolation framework provides an excellent foundation for prototyping new hardware and software extensions. Below, we outline several directions that demonstrate openIPE's versatile potential for the rapid prototyping of innovative hardware-software co-designs.

Other proprietary TI hardware features. To investigate the security of and propose changes to proprietary TI MSP430 microcontrollers [15], openIPE could be extended with more of their features. This includes the MPU, which was used in IPE Exposure to support the software mitigation framework [19], and which could be used to enforce a no-execute policy on protected data inside IPE. Other interesting features include a hardware AES accelerator and even TI's extended 20-bit MSP430X instruction set. Especially relevant may be the addition of a small cache to prototype mitigations against microarchitectural side channels [19]. Lastly, openIPE-enabled FPGAs could be interfaced with TI's FRAM [46] persistent memory technology to investigate protection against physical attacks, e.g., through the use of a transparent memory-encryption engine similar to Intel SGX [90].

Advanced memory isolation. Future work could also enhance IPE's currently limited access-control policies. This could include supporting multiple protected regions [2], [73], securing I/O devices through dedicated driver enclaves [2], [3], [51], or even enabling support for multi-CPU environments that share the same physical memory space. We anticipate that implementing such refined security policies will be relatively straightforward, as our design already channels all access-control logic decisions through a dedicated openMSP430 peripheral hardware component (cf. Figure 3). Support for multiple protected regions could use the firmware as a secure monitor without requiring additional hardware changes [73].

Compiler extensions. Using compilers to analyze and harden code against security vulnerabilities is a popular practice in research, also on openMSP430 [24], [84]. By integrating the open-source `msp430-gcc` compiler into our framework, we facilitate similar research opportunities for openIPE. Furthermore, future work could involve upstreaming native support for IPE in both the `msp430-gcc` and `clang` compilers, as well as conducting security analyses of the TI toolchain, particularly in light of the issue identified in TI's proprietary disassembler (see Section 3.6). Additionally, incorporating support for memory-safe systems programming languages such as Rust could further enhance the security and reliability of the development process.

Secure direct memory access (DMA). One of the microarchitectural attacks demonstrated on openMSP430-based platforms such as Sancus and VRASED is a DMA-based contention attack [23]. To date, the only available mitigation against this attack is a compiler-based code balancing approach, which incurs a sizeable overhead [24]. We believe that openIPE can serve as a research platform for prototyping hardware- or firmware-based mitigations against this attack. Additionally, openIPE's adaptable memory-isolation model could provide a foundation for creating enclave-aware DMA solutions, enabling trusted devices to access a specific subset of enclave memory.

Side-channel resistance. On TI microcontrollers, three different sources of side-channel leakage were discovered: interrupt latency, cache contention, and MPU violations [19]. In Section 6, we demonstrated how the firmware and the IPE code can cooperate to eliminate side channel leakage through interrupt latency. Future work could investigate whether this protection can be extended to the other two sources of leakage by modifying the trusted software, similar to reset-based switching [40].

Hard real-time support. The secure interrupt case study presented in Section 6 provides an excellent foundation for enhancing openIPE's capabilities in mixed-criticality systems that require hard real-time support. For example, incorporating minimal additional hardware extensions to achieve limited atomicity [8], [12] would facilitate guaranteed real-time execution of security-critical, interrupt-driven functions within the protected IPE region. This enhancement could be particularly beneficial within IPE's deployment model (see Figure 2), ensuring that essential functionalities from the IP vendor remain operational even after the microcontroller is handed over to the end user.

Security testing and verification. Currently, our symbolic validation tool checks attacker-controlled memory operations and register clearing on context switches. To enhance the security guarantees, this tool could be expanded to check additional properties, such as the proper setup of the IPE region and the configuration registers. Another promising direction is to extend our symbolic validation tool to ensure the absence of timing-based side channels [91]. Additionally, employing formal verification tools [4] could assist in validating the correct and secure implementation of the hardware or firmware code, while taking into account model-implementation gaps [23].

Cross-testing TI features. Our goal when building openIPE was to maintain compatibility as much as possible with TI's specification. While we test this using our unit test suite, future work could investigate more thorough ways of ensuring that the two implementations behave identically. Automated testing of TI devices is currently difficult due to the proprietary development toolchain, future work could develop tooling to aid researchers in this task and enable more robust methodologies such as fuzz testing on TI microcontrollers.

Other avenues. We believe that the availability of open-source implementations of commercial security technologies, such as openSGX [26] and openIPE are important to drive research with industry impact. The open instruction set RISC-V already has a number of open-source CPUs with standard security features implemented, but other technologies such as TrustZone [85] would also benefit from such research tools.

8. Conclusion

Responding to the pressing need for specialized memory isolation mechanisms in low-end embedded microcontrollers, we proposed openIPE, an extensible openMSP430-based implementation and enhancement of Texas Instruments' proprietary IPE specification. By implementing mature hardware support and a configurable firmware layer, openIPE offers a unified approach to enhance security while ensuring extensibility for future research. Our evaluation of the base design, as well as a case study into secure interrupt handling serve as a validation of openIPE's potential as a unified research platform to explore flexible hardware-software co-designs for future research proposals on trusted execution. Apart from extensions to the security functionality, we would also like to further improve a mature testing and verification framework built around this design that can be reused for future extensions.

Acknowledgment

This research was partially funded by the ORSHIN project (Horizon Europe grant agreement #101070008), the Research Fund KU Leuven, and the Cybersecurity Research Program Flanders.

References

[1] Microchip. Differences between mcu and mpu development. https://developerhelp.microchip.com/xwiki/bin/view/products/mcu-mpu/32bit-mpu/differences-between-mcu-and-mpu-development/, 2024.

[2] 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):7:1–7:33, 2017.

[3] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: Secure and minimal architecture for (establishing dynamic) root of trust. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[4] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED: A verified hardware/software co-design for remote attestation. In *USENIX Security Symposium*, pages 1429–1446, 2019.

[5] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: a security architecture for tiny embedded devices. In *EuroSys*, pages 10:1–10:14, 2014.

[6] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. TyTAN: Tiny trust anchor for tiny devices. In *Design Automation Conference (DAC)*, pages 34:1–34:6, 2015.

[7] Ruan de Clercq, Frank Piessens, Dries Schellekens, and Ingrid Verbauwhede. Secure interrupts on low-end microcontrollers. In *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 147–152, 2014.

[8] Fritz Alder, Jo Van Bulck, Frank Piessens, and Jan Tobias Mühlberg. Aion: Enabling open systems through strong availability guarantees for enclaves. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1357–1372, 2021.

[9] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. Toward remotely verifiable software integrity in resource-constrained iot devices. *IEEE Communications Magazine*, 62(7):58–64, 2024.

[10] Fatemeh Arkannezhad, Justin Feng, and Nader Sehatbakhsh. IDA: Hybrid attestation with support for interrupts and TOCTOU. In *Network and Distributed System Security Symposium (NDSS)*, 2024.

[11] 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)*, pages 146–151, 2016.

[12] Ramya Jayaram Masti, Claudio Marforio, Aanjhan Ranganathan, Aurélien Francillon, and Srdjan Capkun. Enabling trusted scheduling in embedded systems. In *Annual Computer Security Applications Conference (ACSAC)*, pages 61–70, 2012.

[13] Raoul Strackx, Frank Piessens, and Bart Preneel. Efficient isolation of trusted subsystems in embedded systems. In *Security and Privacy in Communication Networks - 6th Iternational ICST Conference, SecureComm*, pages 344–361, 2010.

[14] Olivier Girard. openmsp430. https://github.com/olgirard/openmsp430/blob/master/doc/openMSP430.pdf, 2017.

[15] Texas Instruments. MSP code protection features. https://www.ti.com/lit/an/slaa685/slaa685.pdf, 2015.

[16] Microchip. Codeguard security: Protecting intellectual property in collaborative system designs. http://ww1.microchip.com/downloads/en/DeviceDoc/70179a.pdf, 2006.

[17] Microchip. Selective code protection - microchip developer help. https://microchipdeveloper.com/xc8:selective-code-protection, 2021.

[18] 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.

[19] Marton Bognar, Cas Magnus, Frank Piessens, and Jo Van Bulck. Intellectual property exposure: Subverting and securing intellectual property encapsulation in Texas Instruments microcontrollers. In *USENIX Security Symposium*, 2024.

[20] Prakhar Sah and Matthew Hicks. Ripencapsulation: Defeating IP encapsulation on TI MSP devices. In *WOOT Conference on Offensive Technologies*, pages 117–132, 2024.

[21] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *ACM Conference on Computer and Communications Security (CCS)*, pages 178–195, 2018.

[22] Jo Van Bulck, David F. Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1741–1758, 2019.

[23] Marton Bognar, Jo Van Bulck, and Frank Piessens. Mind the gap: Studying the insecurity of provably secure embedded trusted execution architectures. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1638–1655, 2022.

[24] Marton Bognar, Hans Winderix, Jo Van Bulck, and Frank Piessens. Microprofiler: Principled side-channel mitigation through microarchitectural profiling. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 651–670, 2023.

[25] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. Keystone: an open framework for architecting trusted execution environments. In *EuroSys*, pages 38:1–38:16, 2020.

[26] Prerit Jain, Soham Jayesh Desai, Ming-Wei Shih, Taesoo Kim, Seong Min Kim, Jae-Hyuk Lee, Changho Choi, Youjung Shin, Brent ByungHoon Kang, and Dongsu Han. Opensgx: An open platform for SGX research. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[27] Intel. Xucode: An innovative technology for implementing complex instruction flows. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/xucode-implementing-complex-instruction-flows.html, 2021.

[28] Fritz Alder, Lesly-Ann Daniel, David F. Oswald, Frank Piessens, and Jo Van Bulck. Pandora: Principled symbolic validation of intel SGX enclave runtimes. In *IEEE Symposium on Security and Privacy (S&P)*, pages 4163–4181, 2024.

[29] Tobias Cloosters, Michael Rodler, and Lucas Davi. Teerex: Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves. In *USENIX Security Symposium*, pages 841–858, 2020.

[30] Pedro Antonino, Wojciech Aleksander Woloszyn, and A. W. Roscoe. Guardian: Symbolic validation of orderliness in SGX enclaves. In *CCSW@CCS: Cloud Computing Security Workshop*, pages 111–123, 2021.

[31] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy (S&P)*, pages 138–157, 2016.

[32] Gert-Jan Goossens and Jo Van Bulck. Principled symbolic validation of enclaves on low-end microcontrollers. In *8th Workshop on System Software for Trusted Execution (SysTEX)*, 2025.

[33] 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 *IEEE Computer Security Foundations Symposium (CSF)*, pages 262–276, 2020.

[34] Matteo Busi, Riccardo Focardi, and Flaminia L. Luccio. Bridging the gap: Automated analysis of sancus. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 233–248, 2024.

[35] Matteo Busi, Pierpaolo Degano, Riccardo Focardi, Letterio Galletta, Flaminia Luccio, Frank Piessens, and Jo Van Bulck. Exceptions prove the rule: Investigating and resolving residual side channels in provably secure interrupt handling. In *Workshop on Program Analysis and Verification on Trusted Platforms (PAVeTrust)*, September 2024.

[36] G. Edward Suh, Dwaine E. Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *International Conference on Supercomputing (ICS)*, pages 160–171, 2003.

[37] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for tcb minimization. In *EuroSys*, pages 315–328, 2008.

[38] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.

[39] Zhihao Yao, Seyed Mohammadjavad Seyed Talebi, Mingyi Chen, Ardalan Amiri Sani, and Thomas E. Anderson. Minimizing a smartphone's TCB for security-critical programs with exclusively-used, physically-isolated, statically-partitioned hardware. In *International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 233–246, 2023.

[40] Anish Athalye, Adam Belay, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Notary: a device for secure transaction approval. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 97–113, 2019.

[41] Texas Instruments. FRAM FAQs. https://www.ti.com/lit/wp/slat151/slat151.pdf, 2014.

[42] Texas Instruments. Introduction to MSP430FR5969. https://www.youtube.com/watch?v=QRJ0r-Zx2Hk, 2014.

[43] AspenCore. The current state of embedded development. https://www.embedded.com/wp-content/uploads/2023/05/Embedded-Market-Study-For-Webinar-Recording-April-2023.pdf, 2023.

[44] Stephan Nolting. The neo430 processor. https://github.com/stnolting/neo430, 2020.

[45] Texas Instruments. MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx family user's guide. https://www.ti.com/lit/ug/slau367p/slau367p.pdf, 2012.

[46] Texas Instruments. Closing the security gap with TI's MSP430 FRAM-based microcontrollers. https://www.ti.com/lit/wp/slay035/slay035.pdf, 2014.

[47] Olivier Girard. openmsp430: the road trip to space of an open-source uc core. https://opencores.org/articles/1478437499, 2016.

[48] Jan Tobias Mühlberg, Job Noorman, and Frank Piessens. Lightweight and flexible trust assessment modules for the internet of things. In *European Symposium on Research in Computer Security (ESORICS)*, pages 503–520, 2015.

[49] Jan Tobias Mühlberg, Sara Cleemput, Mustafa A. Mustafa, Jo Van Bulck, Bart Preneel, and Frank Piessens. An implementation of a high assurance smart meter using protected module architectures. In *Information Security Theory and Practice*, pages 53–69, 2016.

[50] Carol Suman Pinto. Optimization of physical unclonable function protocols for lightweight processing. Master's thesis, Virginia Tech, 2016.

[51] Jo Van Bulck, Jan Tobias Mühlberg, and Frank Piessens. VulCAN: Efficient component authentication and software isolation for automotive control networks. In *Annual Computer Security Applications Conference (ACSAC)*, pages 225–237, 2017.

[52] Stien Vanderhallen, Jo Van Bulck, Frank Piessens, and Jan Tobias Mühlberg. Robust authentication for automotive control networks through covert channels. *Computer Networks*, 193, 2021.

[53] Mahmoud Ammar, Bruno Crispo, Ivan De Oliveira Nunes, and Gene Tsudik. Delegated attestation: scalable remote attestation of commodity CPS by blending proofs of execution with software attestation. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, pages 37–47, 2021.

[54] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. Tiny-cfa: Minimalistic control-flow attestation using verified proofs of execution. In *Design, Automation & Test in Europe (DATE)*, pages 641–646, 2021.

[55] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. DIALED: Data integrity attestation for low-end embedded devices. In *Design Automation Conference (DAC)*, pages 313–318, 2021.

[56] Aurélien Francillon, Quan Nguyen, Kasper Bonne Rasmussen, and Gene Tsudik. A minimalist approach to remote attestation. In *Design, Automation & Test in Europe (DATE)*, pages 1–6, 2014.

[57] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. APEX: A verified architecture for proofs of execution on remote devices under full software compromise. In *USENIX Security Symposium*, pages 771–788, 2020.

[58] Xavier Carpent, Gene Tsudik, and Norrathep Rattanavipanon. ERASMUS: Efficient remote attestation via self-measurement for unattended settings. In *Design, Automation & Test in Europe (DATE)*, pages 1191–1194, 2018.

[59] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security Symposium*, pages 479–494, 2013.

[60] Johannes Götzfried, Tilo Müller, Ruan de Clercq, Pieter Maene, Felix C. Freiling, and Ingrid Verbauwhede. Soteria: Offline software protection within low-cost embedded devices. In *Annual Computer Security Applications Conference (ACSAC)*, pages 241–250, 2015.

[61] Gianluca Scopelliti, Sepideh Pouyanrad, Job Noorman, Fritz Alder, Christoph Baumann, Frank Piessens, and Jan Tobias Mühlberg. End-to-end security for distributed event-driven enclave applications on heterogeneous tees. *ACM Trans. Priv. Secur.*, 26(3):39:1–39:46, 2023.

[62] Adam Caulfield, Norrathep Rattanavipanon, and Ivan De Oliveira Nunes. ASAP: reconciling asynchronous real-time operations and proofs of execution in simple embedded systems. In *Design Automation Conference (DAC)*, pages 721–726, 2022.

[63] Avani Dave, Nilanjan Banerjee, and Chintan Patel. RARES: Runtime attack resilient embedded system design using verified proof-of-execution. *CoRR*, abs/2305.03266, 2023.

[64] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. On the TOCTOU problem in remote attestation. In *ACM Conference on Computer and Communications Security (CCS)*, pages 2921–2936, 2021.

[65] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Youngil Kim, and Gene Tsudik. CASU: Compromise avoidance via secure update for low-end embedded systems. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 146:1–146:9, 2022.

[66] Ivan De Oliveira Nunes, Seoyeon Hwang, Sashidhar Jakkamsetti, and Gene Tsudik. Privacy-from-birth: Protecting sensed data from malicious sensors with VERSA. In *IEEE Symposium on Security and Privacy (S&P)*, pages 2413–2429, 2022.

[67] Adam Caulfield, Norrathep Rattanavipanon, and Ivan De Oliveira Nunes. ACFA: Secure runtime auditing & guaranteed device healing via active control flow attestation. In *USENIX Security Symposium*, pages 5827–5844, 2023.

[68] Esmerald Aliaj, Ivan De Oliveira Nunes, and Gene Tsudik. GAROTA: Generalized active root-of-trust architecture (for tiny embedded devices). In *USENIX Security Symposium*, pages 2243–2260, 2022.

[69] Liam Tyler and Ivan De Oliveira Nunes. Untrusted code compartmentalization for bare metal embedded devices. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 43(11):3419–3430, 2024.

[70] Daniel Dinu, Archanaa S. Krishnan, and Patrick Schaumont. SIA: Secure intermittent architecture for off-the-shelf resource-constrained microcontrollers. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 208–217, 2019.

[71] Archanaa S. Krishnan, Charles Suslowicz, and Patrick Schaumont. Secure and stateful power transitions in embedded systems. *J. Hardw. Syst. Secur.*, 4(4):263–276, 2020.

[72] Archanaa S. Krishnan and Patrick Schaumont. Benchmarking and configuring security levels in intermittent computing. *ACM Trans. Embed. Comput. Syst.*, 21(4):36:1–36:22, 2022.

[73] Taylor Hardin, Ryan Scott, Patrick Proctor, Josiah D. Hester, Jacob Sorber, and David Kotz. Application memory isolation on ultra-low-power mcus. In *USENIX Annual Technical Conference (ATC)*, pages 127–132, 2018.

[74] Michele Grisafi, Mahmoud Ammar, Marco Roveri, and Bruno Crispo. PISTIS: Trusted computing architecture for low-end embedded systems. In *USENIX Security Symposium*, pages 3843–3860, 2022.

[75] Michele Grisafi, Mahmoud Ammar, Marco Roveri, and Bruno Crispo. Flashadow: A flash-based shadow stack for low-end embedded systems. *ACM Trans. Internet Things*, 5(3):19:1–19:29, 2024.

[76] Travis Goodspeed. Practical attacks against the msp430 bsl. In *Twenty-Fifth Chaos Communications Congress*, 2008.

[77] Travis Goodspeed and Aurélien Francillon. Half-blind attacks: Mask ROM bootloaders are dangerous. In *WOOT Conference on Offensive Technologies*, pages 1–6, 2009.

[78] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the difficulty of software-based attestation of embedded devices. In *ACM Conference on Computer and Communications Security (CCS)*, pages 400–409, 2009.

[79] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep K. Khosla. SCUBA: Secure code update by attestation in sensor networks. In *ACM Workshop on Wireless Security*, pages 85–94, 2006.

[80] Texas Instruments. MSP code protection features: Source code. http://www.ti.com/lit/zip/slaa685, 2015.

[81] 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.

[82] Cristiano Rodrigues, Daniel Oliveira, and Sandro Pinto. Busted!!! microarchitectural side-channel attacks on the MCU bus interconnect. In *IEEE Symposium on Security and Privacy (S&P)*, pages 3679–3696, 2024.

[83] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The severest of them all: Inference attacks against secure virtual enclaves. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, pages 73–85, 2019.

[84] Hans Winderix, Jan Tobias Mühlberg, and Frank Piessens. Compiler-assisted hardening of embedded software against interrupt latency side-channel attacks. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 667–682, 2021.

[85] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. https://web.archive.org/web/20070415022456/http://www.arm.com/pdfs/TZ%20Whitepaper.pdf, 2004.

[86] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium*, pages 463–478. USENIX Association, 2013.

[87] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.

[88] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1789–1806, 2017.

[89] Marc Schink and Johannes Obermaier. Taking a look into execute-only memory. In *WOOT Conference on Offensive Technologies*, 2019.

[90] Shay Gueron. Memory encryption for general-purpose processors. *IEEE Secur. Priv.*, 14(6):54–62, 2016.

[91] Sepideh Pouyanrad, Jan Tobias Mühlberg, and Wouter Joosen. Scf$^{\mathrm{msp}}$: static detection of side channels in MSP430 programs. In *International Conference on Availability, Reliability and Security (ARES)*, pages 21:1–21:10, 2020.

Appendix A. Data availability

All our code, including the openIPE and case study implementations, unit tests, and symbolic execution tool are published on GitHub and integrated in a CI pipeline: https://github.com/martonbognar/openipe