

Control-Flow Balancing for Texas Instruments IPE

Marton Bognar

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

Hans Winderix

hans.winderix@kuleuven.be
DistriNet, KU Leuven
Leuven, Belgium

Alexander Croes

alexander.croes@student.kuleuven.be
DistriNet, KU Leuven
Leuven, Belgium

Jo Van Bulck

jo.vanbulck@kuleuven.be
DistriNet, KU Leuven
Leuven, Belgium

Abstract

Microarchitectural side channels pose a prominent threat to the strong architectural isolation model pursued by trusted execution environments (TEEs). While high-end TEEs typically rely on transforming application code to enforce the constant-time model, disallowing secret-dependent control flow, recent research on prototype systems has demonstrated that careful (potentially compiler-assisted) balancing of secret-dependent control flow can achieve equivalent security guarantees at substantially lower performance overheads.

In this work, we investigate the feasibility of applying a control-flow balancing approach for Intellectual Property Encapsulation (IPE), a specialized TEE deployed on real-world Texas Instruments MSP430 microcontrollers. We show that, even on this ultra-low-end platform, straightforward profiling-based techniques from prior academic work are insufficient. We address this problem through a more in-depth reverse engineering effort to produce efficient, balanced control flow whose security properties can be validated using automated analysis tools.

CCS Concepts: • Computer systems organization → Embedded systems; • Security and privacy → Embedded systems security.

Keywords: Side Channels, Code Transformation, Control-Flow Balancing

ACM Reference Format:

Marton Bognar, Alexander Croes, Hans Winderix, and Jo Van Bulck. 2026. Control-Flow Balancing for Texas Instruments IPE. In *9th Workshop on System Software for Trusted Execution (SysTEX '26)*, April 27–30, 2026, Edinburgh, Scotland Uk. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3805690.3805736>



This work is licensed under a Creative Commons Attribution 4.0 International License.

SysTEX '26, Edinburgh, Scotland Uk

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2607-1/26/04

<https://doi.org/10.1145/3805690.3805736>

1 Introduction

Features such as process isolation and trusted execution environments (TEEs) can provide confidentiality for code and data in computing systems. However, these guarantees are undermined by microarchitectural side-channel leakage [14], which can manifest across all classes of devices, from shared cloud servers to resource-constrained embedded microcontrollers [2–4, 6, 30, 35]. Microarchitectural side-channel attacks exploit secret-dependent behavior in victim code. Hence, the most widely applicable mitigation on high-end platforms with complex microarchitectures is to carefully rewrite applications following “constant-time” coding guidelines [14]. Constant-time coding enforces the principle that secret-dependent data must not influence branch conditions, memory access patterns, or the operands of variable-latency instructions.

Unfortunately, constant-time transformations often incur substantial overhead in both code size and execution time. Control-flow balancing, explored in recent research on low-end platforms [39] with deterministic microarchitectures, is a promising alternative to control-flow linearization [8], the code transformation used to eliminate secret-dependent control flow. Control-flow balancing relies on the key insight that attackers often cannot infer the precise control flow (i.e., the program counter) of a victim program. If secret-dependent behavior is constructed such that its impact on the attacker-visible microarchitectural state remains constant, information leakage can be prevented. However, such balancing must be applied carefully, as overlooked side channels can expose secret-dependent behavior. Recently, MicroProfiler [7] proposed a methodology for principled control-flow balancing, in which a profiling phase analyzes the microarchitectural leakage of each instruction in an instruction set. The profiling results enable a compiler to automatically balance secret-dependent control flow and a binary analysis tool to validate secure balancing. However, this approach has only been demonstrated on a white-box research prototype, based on the open-source openMSP430 [15] software.

In this work, we investigate how feasible control-flow balancing is on a real-world, commercial microcontroller and

whether the MicroProfiler approach is a suitable methodology for this device. We analyze a Texas Instruments (TI) MSP430 microcontroller featuring the Intellectual Property Encapsulation (IPE) TEE, which has been recently shown to contain multiple sources of side-channel leakage [4], including a unified instruction/data cache. Not only does the black-box setting complicate the profiling phase, but, somewhat surprisingly, we find that the microarchitecture of this simple low-end microcontroller is more complex than anticipated in prior work. To be able to accurately profile the side-channel leakage of the device, we reverse engineer the properties of the cache and the microcontroller pipeline. We show that the presence of the cache makes the leakage unbalanceable using the approach from prior work.

After establishing the precise leakage behavior of the device, we use this information to harden applications. First, similar to prior work, we extend a binary analysis tool to model the side-channel leakage of code running in the TEE to identify potentially insecure code patterns. We also formulate innovative principles for secure control-flow balancing in the presence of a cache on our target device, and we apply these to selected microbenchmarks from the literature. Our results show that even in the presence of side channels that require more careful balancing, the performance benefits over constant-time code are significant. In a broader perspective, our work provides motivation to investigate similar approaches in higher-end devices with predictable side-channel patterns and similar attacker models.

Contributions. Our main contributions are as follows:

- We reverse engineer the side-channel behavior of IPE-enabled MSP430 microcontrollers and profile the leakage of individual instructions in a black-box setting.
- We propose a novel balancing algorithm that prevents secret leakage on systems with an instruction cache, outperforming constant-time code on multiple benchmarks while preserving the same security guarantees.
- We extend a binary validation tool to ensure the security of enclave programs.

Open science. To support reproducibility and encourage future research, all our experiment code and data are released at <https://github.com/martonbognar/ipe-balancing>.

2 Background

MSP430 in research. MSP430 is an instruction set architecture (ISA) and a line of microcontrollers by TI. MSP430 has a long history in industry and has been used in research, either directly [16] or by building on openMSP430 [15], the open-source softcore implementation of the instruction set. Notably, this platform has been used for building trusted execution technologies [11, 24] and software hardening techniques for the code running on them [7, 39].

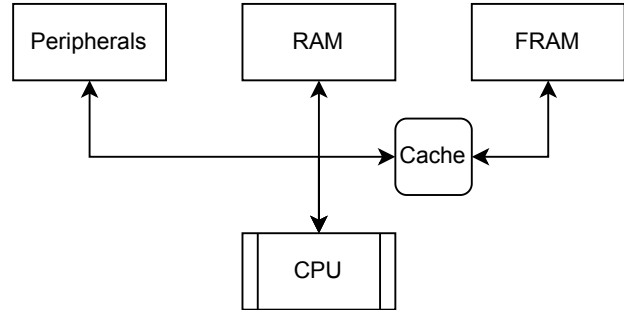


Figure 1. Architecture of the analyzed MSP430 devices.

The 16-bit MSP430 instruction set is relatively simple, featuring 27 instructions with seven addressing modes, capable of operating directly on memory values. Each instruction is encoded in 1–3 words and can have between 0 and 2 operands. Notably, the ISA specification lists the *static* latency of each instruction, which has been exploited by interrupt-latency attacks such as the Nemesis [35] attack.

MSP430 devices. While there are many generations of MSP430 microcontrollers, this work focuses on the devices that feature the IPE technology,¹ explained later in this section. TI has sold millions of devices that feature this TEE technology [34]. IPE is present on microcontrollers that are equipped with Ferroelectric Random Access Memory (FRAM), a special persistent low-power memory technology. Importantly, the operating frequency of FRAM is lower than that of the microcontroller itself at its highest frequency, so a small cache has been included between the core and the FRAM (cf. Figure 1). According to the specification [33] and prior research [4], this cache is made up of two sets with two ways per set, each containing four 16-bit words. In the device’s von Neumann architecture, FRAM stores code and data, both of which can be cached.

IPE. The FRAM-equipped microcontrollers feature multiple security features. Besides tools like the memory protection unit (MPU) that can selectively enable read/write/execute permissions on memory segments, the most advanced is Intellectual Property Encapsulation (IPE). IPE enables the isolation of selected code and data from the rest of the system, functioning as a TEE. Unfortunately, recent studies [5, 31] have pointed out architectural and microarchitectural shortcomings of current generations of IPE. For our work, we assume correct architectural isolation on the device, e.g., by applying fixes to the hardware [5], and focus on mitigating the side-channel leakage. Specifically, IPE is vulnerable to three side channels: interrupt latency [35], cache timing [26], and controlled-channel attacks [41] through the onboard MPU. Interrupt latency leaks how many cycles a given instruction takes to execute, which depends on the instruction

¹All experiments for this paper were conducted on the MSP430FR5969.

and operand types. The cache leaks whether a given (code or data) memory location has been loaded recently and reveals one address bit. The MPU allows the memory to be partitioned into 1 kB pages and to restrict read/write/execute permissions to these pages, enabling an attacker to monitor each type of access at the granularity of the pages.

3 Threat model

In this work, we focus on mitigating the software-observable side channels that have been identified on the target device [4]. The attacker is allowed to control all untrusted code and data on the device, which notably allows scheduling and measuring the execution time of interrupts (also during victim execution); modifying the cache state with untrusted data; and setting up MPU access rules—covering all three side channels. Importantly, memory accesses are observable only at cache set granularity; the attacker cannot directly observe the memory addresses or instructions executed by the victim. This assumption is consistent with deployment settings that do not permit shared memory between attacker and victim. For instance, the presence of shared memory as exploited in Flush+Reload [42] would violate this assumption, as it enables the attacker to test access to a specific shared cache line, significantly strengthening the side-channel signal. On our device, we verified that memory accesses rejected by IPE’s access control neither depend on the cache state nor cause any observable modifications to it.

Side channels that require physical access and measurement equipment, such as power consumption [18] or EM emissions [28], are out of scope for this paper.

4 Challenges

To detect and mitigate side-channel leakage on low-end embedded TEEs, prior work proposed an approach that profiles the side-channel leakage of individual instructions [7]. This approach was demonstrated on the Sancus research TEE [24], a prototype based on the openMSP430 softcore. Since Sancus is available as a white-box design, profiling could be performed entirely in simulation with full visibility into the internal behavior of the core. Although the authors suggested that this approach could be performed in a black-box setting by using the same measurement techniques available to an attacker, this has not yet been demonstrated. In this work, we show that applying this methodology to commercial TI MSP430 devices is not as straightforward. In particular, we encountered two main challenges.

First, microarchitectural side channels are caused by microarchitectural optimizations. Prior work [38] classifies these optimizations into those that yield balanceable observations and those that yield unbalanceable observations, arguing that the latter require hardware support to mitigate. Unbalanceable observations mean that leakage on one side

of a branch cannot be mitigated simply by inserting compensating instructions on the other side, as is possible for balanceable observations. Microcontrollers with IPE feature a cache storing instructions, which is an example of a microarchitectural optimization that produces unbalanceable observations. This is because instruction caches maintain state as a function of instruction addresses, causing the program counter to leak. Libra [38] addresses this issue by introducing hardware support combined with constraints on the code memory layout. In this paper, we show that for IPE, secure balancing of secret-dependent control flow can be achieved solely by constraining the memory layout without requiring additional hardware support, in a different manner than proposed in Libra [38].

Second, because we operate in a black-box setting, we do not have access to the RTL code of the design. As a result, we had to reverse engineer the behavior of the cache to understand how to place code in memory securely. Furthermore, to perform instruction-level profiling in a black-box setting, we could not rely on simulating the hardware to identify leakage. Instead, we measured the execution time of each instruction using two independent measurement methods and compared the results with the latency values reported in the device documentation. The two methods are (1) using interrupts as in a real-world Nemesis attack, and (2) placing the measured instruction between time measurements. Instead of relying on LLVM to generate all possible instructions [7], we generated them based on the list of instructions and operand types supplied by the documentation [33].

Interestingly, we discovered multiple deviations from the documentation. Even in the simple case of a deactivated cache (low operating frequency), the real behavior of instructions deviated from the documented execution times. The most notable exception is the omission of the constant generator register from the timing information, which reduces the latency of each instruction that operates on a common constant value by one cycle. The impact of the cache on instruction timings is also not explained. These deviations could potentially cause problems for applications that rely on the provided timing values for real-time operations, and certainly for balancing approaches that would use this information to mask secret-dependent timing results.

5 Reverse engineering

This section presents an overview of the microcontroller’s microarchitectural behavior revealed through our reverse engineering effort. Most of the analysis relates to the cache, as various architectural and microarchitectural features, such as CPU pipelining and interrupt handling, affect its state, which in turn can influence other side channels, such as interrupt latency. Our experiments were conducted by placing the measurement code in the device’s RAM to avoid polluting the cache state. When code from FRAM is executed, we

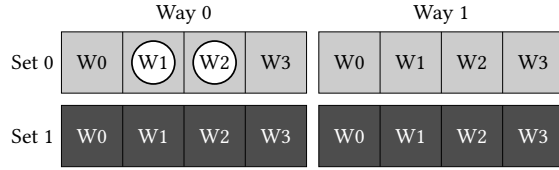


Figure 2. Structure of the cache. In figures, we use different shades of gray to mark the two cache sets. Circles in cache blocks indicate the words that a given instruction occupies.

control this precisely using direct jumps and interrupts to avoid additional instructions affecting the cache state.

5.1 Cache policies

The cache of the device (Figure 2) is known to have two sets and two ways per set, with each cache line holding four 16-bit words of data [33]. Our experiments confirmed that the set is selected by the 4th lowest bit of the accessed address, as the lowest bit selects the byte inside a word, while bits 2 and 3 select the word in the cache line. The replacement policy favors evicting the least recently used line of the two per set. Memory store instructions do not place data into the cache; however, if a store targets a line already present in the cache, that line gets invalidated. Interestingly, this invalidation does not seem to affect the replacement policy; the cache will still evict the least recently used line in a set, even if the more recently used line was invalidated by a store.

5.2 Code execution

The exact mechanism of how the microcontroller executes code is relevant to its side-channel leakage. The Nemesis [35] attack measures the interrupt latency of instructions, which is directly related to how they are executed in the processor pipeline. As the cache holds both code and data, we need to understand how the cache state is affected by code execution. During our experiments, we made two notable observations.

First, the device likely features a three-stage pipeline. When interrupting execution right after instruction retirement, the following two instruction words can be found in the cache. This suggests that during one instruction’s execution, two other pipeline stages already start fetching and decoding the following instruction words.

Second, branching instructions influence the cache state differently depending on their outcome. The first instruction word following the branch is always cached, just as with other instructions. Notably, the target of the branch is also always cached, suggesting that after decoding the branch target, one cycle after fetching the branch, the pipeline always fetches the next instruction from this location. The second instruction word following the branch (executed in the non-taken case) is only fetched when the condition evaluates to false. This behavior was observed for forward and backward branches alike.

5.3 Interrupts

Interrupts are a crucial tool for conducting side-channel attacks on this device. The Nemesis attack [35] measures the interrupt latency of instructions directly, but the effects of the cache can also be most accurately measured by the execution time of instructions. The best approach to measure the execution time of individual instructions in the victim code is to schedule precise interrupts before and after the targeted instruction, filtering out the effects of other instructions as done in prior work [35].

The addresses of the interrupt service routines (ISRs) are stored in the interrupt vector table (IVT), located at a fixed address in FRAM. For each interrupt, the address of the corresponding ISR needs to be fetched from the IVT. The mapping between interrupt sources and IVT entries is fixed, which means that for each interrupt, it can be calculated which cache set will be affected. Additionally, if the ISR is located in FRAM, its execution will further change the cache state and affect the measured interrupt latency.

6 Binary analysis

MicroProfiler [7] adapted a binary analysis tool [27] to detect programs with sensitive side-channel leakage based on the results of instruction profiling. While our findings indicate that automated profiling is difficult with more complex sources of side channels, the binary analysis tool can still be applied for detecting leakage based on our reverse engineering efforts. We adapted SCF-MSP [27], also used in MicroProfiler [7], to detect control-flow leakage through the cache state and MPU regions, marking an important difference compared to prior studies [7, 39]. Previously, leakage was static, depending only on the instruction’s leakage class; in our model, it also depends on the instruction’s address in memory, requiring knowledge of the binary layout for accurate detection.

The MPU side channel is relatively simple, as it can be used to assign different permissions to 1 kB blocks in memory. As long as a secret-dependent branch region is contained within a single block, and balanced data accesses are constrained to the same blocks, the control flow does not leak.

To track the control-flow leakage in the cache, we built a model of the cache based on our reverse engineering. At the start of the analysis, we initialize an empty cache state, then update it for each instruction while analyzing the possible control flows. If the cache state evolves differently on the two sides of a secret-dependent branch, a violation is reported.

Finally, for interrupt latency, we currently rely on the instruction latencies obtained during our simple profiling step with the cache disabled. A more complete approach should investigate the possible impact of the cache on instruction latency given different initial states and instruction operands.

This is important, as two instructions that have identical timing with the cache disabled could, in theory, exhibit different latencies with a carefully crafted initial cache state.

Using our adapted tool, we confirmed that certain benchmark programs that were only balanced for interrupt latency [27, 39] exhibit secret leakage through the cache.

7 Control-flow balancing

We aim to avoid costly control-flow linearization by balancing secret-dependent branches as done in prior work [7, 38, 39] to eliminate control-flow leakage. Based on our profiling and reverse engineering results, we devised an algorithm for balancing IPE programs against the three known side channels of the device. Our approach builds on the concept of *dummy instructions*, introduced by seminal work [1] on control-flow balancing. Dummies are no-ops inserted on one side of a branch that produce the same side-channel observations as the corresponding (useful) instructions on the other side. Once a branch is balanced, the two paths may comprise any combination of useful and dummy instructions, as long as they produce indistinguishable side-channel observations and preserve program semantics.

7.1 Balancing safe instructions

Prior work [38] defines an instruction operand as *unsafe* if the instruction’s timing behavior depends on the value of that operand (cf. Section 7.2). For instructions with safe operands, we can rely on the results of the side-channel profiling [7], which classifies instructions into leakage classes such that instructions within the same class induce identical side-channel observations. In our profiling results, instructions in a given leakage class exhibit identical size and latency, with each class containing a suitable dummy instruction. Balancing then reduces to ensuring that corresponding instructions across branches belong to the same leakage class.

7.2 Balancing unsafe instructions

The cache of IPE devices makes instructions that access memory unsafe, since their latency depends on whether the accessed data is cached. Thus, to avoid leakage, corresponding instructions that access memory must access the same memory locations. The most secure option for dummy data accesses is to do this at the granularity of a cache line. Dummy accesses avoid modifying program behavior, e.g., by loading into an unused temporary or constant register, or by storing the existing value to memory.

In cases where this is not feasible, or would cause invasive code changes, an alternative solution could place dummy values in congruent cache lines (i.e., mapping to the same cache set), but this brings additional complications, which are discussed in Section 7.4.

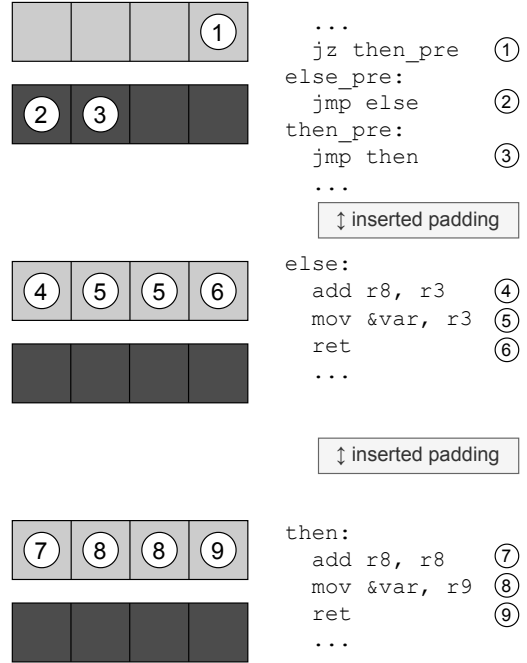


Figure 3. Outline of a balanced branch and the corresponding cache mappings. The numbered circles indicate how certain instructions map to words in the two cache sets.

7.3 Balancing branch instructions

Because the instruction cache maintains state as a function of the instruction’s address, the program counter leaks directly (i.e., balancing instructions is not effective). Prior work [38] refers to this type of optimization as a source of *unbalanceable* observations. Consequently, achieving control-flow balancing requires additional measures.

Per our reverse engineering results, branch instructions have a cache footprint that depends on the outcome of the branch condition. It is thus crucial to make this dependency invisible to an attacker by arranging branch instructions, their targets, and the fallthrough paths in memory in a way that the cache footprint becomes indistinguishable.

We propose the memory layout shown in Figure 3. First, the branch instruction (`jz then_pre`) is placed at the end of a cache line, ensuring that the next block is always fetched into the cache regardless of the branch outcome. The instruction following the branch and the instruction at the branch target are placed in the next cache line, and both are chosen to be unconditional jumps (`jmp else` and `jmp then`). The targets of these two jumps contain the actual instructions of the two sides of the branch, and these do not have to fall into the same cache line anymore, as long as they (and the consecutive instructions) map to the same set with the same word offset (e.g., instructions 4 and 7 mapping to the first word of the light gray set in Figure 3). To be balanced, corresponding

instructions across branches must belong to the same leakage class as explained before.

7.4 Additional considerations

When balancing memory accesses, it is important to consider what the impact of prior accesses might be. While an attacker in our threat model cannot distinguish between accesses to two congruent cache lines if neither was cached before, this is not the case for consecutive accesses. By observing the timing of individual instructions, an attacker might be able to detect whether the same cache line is accessed in different iterations of a loop, or even across different executions of the enclave software if the cache state is large enough. For this reason, it is important to carefully consider the effects of using dummy accesses that use congruent cache lines instead of accessing the same cache line. Furthermore, a similar effect can be visible if instructions from a prior execution of one side of a secret-dependent branch are still in the cache when the branch is executed again, revealing whether the branch condition was the same. This can leak secrets directly if the attacker can control the branch condition of an execution by invoking the enclave software with specifically chosen parameters. A straightforward solution, especially for small caches, is to start the enclave execution with setting up the cache state in a known empty state.

Secret-dependent regions containing loops, function calls, and nested branches can also be balanced. Prior work [1] showed that the worst-case increase in code size due to balancing nested control-flow constructs scales linearly with their nesting depth, rather than exponentially. Loops cannot have a secret-dependent number of iterations, but static loops inside a secret-dependent branch can be balanced by a dummy loop in the other branch. Similarly, function calls can be balanced by a call to a dummy function exhibiting the same leakage (or, when applicable, a useful function with the same leakage).

8 Evaluation

Lacking compiler support, the hardening of vulnerable programs must be performed manually. Accordingly, we selected a subset of the benchmark programs from prior work [7, 27, 39] for evaluation. We ported these programs to our device and manually balanced their control flow based on our approach outlined in Section 7, starting from compiler-balanced versions from prior work [39].

We compared the binary size and execution time of our balanced variants (Bal.) to the vulnerable baseline (Vuln.) and a secure version linearized (Lin.) using a method from the literature [21]. The results in Table 1 show that in terms of execution time, control-flow balancing consistently outperforms control-flow linearization in our benchmarks and is often more competitive in code size. While these are preliminary results on small benchmarks, we believe that they

Table 1. Evaluation results: binary size is measured in bytes, execution time in number of cycles. The last row contains the geometric mean of the relative overheads compared to the fastest vulnerable baseline version.

Name	Binary size			Execution time		
	Vuln.	Lin.	Bal.	Vuln.	Lin.	Bal.
bsl	162	270	238	821–881	1296	1054
mulhi	166	308	354	492–790	1640	1278
mulmod	232	424	520	61–349	494	382
sharevalue	230	252	276	3294–3307	3524	3443
switch16	184	670	348	29–42	430	44
switch8	184	670	348	29–42	430	44
overhead		2.08x	1.76x		4.64x	1.92x

provide evidence that control-flow balancing can be effective and efficient for securing critical pieces of code.

Additionally, we ran the security analysis tool developed in Section 6 and confirmed that the vulnerable versions of the benchmarks show side-channel leakage, while the linearized and balanced versions do not.

9 Discussion and related work

In this work, we set out to show the feasibility of control-flow balancing as a countermeasure against control-flow leakage attacks on real-world microcontrollers. Prior work has demonstrated this approach for research architectures [7, 38, 39], showing that it often carries performance benefits over constant-time methods that eliminate secret-dependent control flow altogether [8, 21, 32, 36, 40], while offering the same security guarantees on low-end microcontrollers.

Notably, our analysis revealed that even simple microcontrollers can have more complex leakage models than research prototypes, with the black-box nature of the analysis further complicating exhaustive and accurate leakage profiling. Nonetheless, we believe that our results show the efficiency and effectiveness of the approach, and hope to inspire more elaborate future work that explores control-flow balancing approaches to protect sensitive applications, such as those in TEEs. Future work could investigate how to integrate our mitigation approach into an automated tool, using either binary rewriting or compiler passes. Some approaches might be inherently too fragile to apply in our scenario, as the security depends on the precise addresses used during runtime, which might not always be readily available.

Learning techniques [9] or contract derivation approaches making use of fuzzing techniques [23, 25] or formal methods [12, 13, 17, 22, 37] provide promising alternatives for discovering the leakage of instructions. A leakage contract constructed using these techniques could also be used as a blueprint for constructing balancing mitigations.

Beyond control-flow balancing and control-flow linearization, several alternative approaches have been proposed to

mitigate control-flow leakage. Masking-based cryptographic techniques [10] transform sensitive data using random masks, ensuring that intermediate computations remain statistically independent of the underlying secrets. Transaction-based mechanisms [29] execute all possible paths of a secret-dependent branch within transactions, committing only the architecturally correct path. Unfortunately, both of these approaches introduce substantial overhead. Finally, instruction randomization techniques [19, 20] aim to obscure side-channel signals by introducing randomness at the execution level, such as inserting delays or injecting noise. These methods generally provide weaker security guarantees and are often insufficient against sophisticated adversaries.

We believe that in applications that closely align with our threat model (cf. Section 3), such as TEEs, confidential VMs, and even regular process isolation, control-flow balancing is an underexplored alternative to linearization approaches for preventing control-flow leakage.

10 Conclusion

In this work, we proposed a control-flow balancing approach for IPE enclave software to avoid side-channel leakage. Despite difficulties in applying a side-channel profiling approach from prior work in this realistic black-box setting, our microarchitectural analysis allowed us to establish leakage patterns. Our benchmarks show that control-flow balancing outperforms constant-time (i.e., linearized) code, encouraging future research in this direction on other platforms with similar threat models.

Acknowledgments

This research is partially funded by the Internal Funds KU Leuven and the Cybersecurity Research Program Flanders.

References

- [1] Johan Agat. 2000. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. doi:10.1145/325694.325702
- [2] André Barbosa, Cristiano Rodrigues, Tiago Gomes, and Sandro Pinto. 2024. WiP Paper: BUSted Second Stop! A First Step for Breaking Cryptographic Applications on MCU-based IoT Devices. In *Proc. of IEEE EWSN*.
- [3] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales B. Paiva, Prasanna Ravi, and Goutam Tamvada. 2025. KyberSlash: Exploiting Secret-Dependent Division Timings in Kyber Implementations. In *Proc. of IACR TCHES*.
- [4] Marton Bogнар, Cas Magnus, Frank Piessens, and Jo Van Bulck. 2024. Intellectual Property Exposure: Subverting and Securing Intellectual Property Encapsulation in Texas Instruments Microcontrollers. In *USENIX Security Symposium*. <https://www.usenix.org/conference/usenixsecurity24/presentation/bognar>
- [5] Marton Bogнар and Jo Van Bulck. 2025. openIPE: An Extensible Memory Isolation Framework for Microcontrollers. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. 1104–1120. doi:10.1109/EUROSP63326.2025.00067
- [6] Marton Bogнар, Jo Van Bulck, and Frank Piessens. 2022. Mind the Gap: Studying the Insecurity of Provably Secure Embedded Trusted Execution Architectures. In *IEEE Symposium on Security and Privacy (S&P)*. 1638–1655. doi:10.1109/SP46214.2022.9833735
- [7] Marton Bogнар, Hans Winderix, Jo Van Bulck, and Frank Piessens. 2023. MicroProfiler: Principled Side-Channel Mitigation through Microarchitectural Profiling. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. 651–670. doi:10.1109/EUROSP57164.2023.00045
- [8] Pietro Borrello, Daniele Cono D’Elia, Leonardo Querzoni, and Cristiano Giuffrida. 2021. Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization. In *CCS ’21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. 715–733. doi:10.1145/3460120.3484583
- [9] Matteo Busi, Riccardo Focardi, and Flaminia L. Luccio. 2024. Bridging the Gap: Automated Analysis of Sancus. In *IEEE Computer Security Foundations Symposium (CSF)*. 233–248. doi:10.1109/CSF61375.2024.00023
- [10] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. 1999. Towards Sound Approaches to Counteract Power-Analysis Attacks. In *Advances in Cryptology - CRYPTO*. 398–412. doi:10.1007/3-540-48405-1_26
- [11] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. 2019. VRASED: A Verified Hardware/Software Co-Design for Remote Attestation. In *USENIX Security Symposium*. 1429–1446. <https://www.usenix.org/conference/usenixsecurity19/presentation/de-oliveira-nunes>
- [12] Sushant Dinesh, Madhusudan Parthasarathy, and Christopher W Fletcher. 2024. Conjunct: Learning inductive invariants to prove unbounded instruction safety against microarchitectural timing attacks. In *IEEE Symposium on Security and Privacy (S&P)*.
- [13] Sushant Dinesh, Yongye Zhu, and Christopher W Fletcher. 2025. H-HOUDINI: Scalable invariant learning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
- [14] Qian Ge, Yuval Yarom, David A. Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.* 8, 1 (2018), 1–27. doi:10.1007/S13389-016-0141-6
- [15] Olivier Girard. 2017. openMSP430. <https://github.com/olgirard/openmsp430/blob/master/doc/openMSP430.pdf>.
- [16] Michele Grisafi, Mahmoud Ammar, Marco Roveri, and Bruno Crispo. 2022. PISTIS: Trusted Computing Architecture for Low-end Embedded Systems. In *USENIX Security Symposium*. 3843–3860. <https://www.usenix.org/conference/usenixsecurity22/presentation/grisafi>
- [17] Yao Hsiao, Nikos Nikoleris, Artem Khyzha, Dominic P Mulligan, Gustavo Petri, Christopher W Fletcher, and Caroline Trippel. 2024. RTL2MμPATH: Multi-μPATH synthesis with applications to hardware security verification. In *MICRO*.
- [18] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *CRYPTO*.
- [19] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO*, Vol. 1109. 104–113. doi:10.1007/3-540-68697-5_9
- [20] JongHyeok Lee and Dong-Guk Han. 2020. Security analysis on dummy based side-channel countermeasures—Case study: AES with dummy and shuffling. *Applied Soft Computing* (2020).
- [21] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. 2005. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3935)*. 156–168. doi:10.1007/11734727_14

- [22] Elvira Moreno, Tiziano Marinaro, Ryan Williams, Marco Patrignani, Roberto Guanciale, Hamed Nemati, and Marco Guarnieri. 2026. Talk: Automated Synthesis of Instruction-Centric Leakage Contracts. In *Proceedings of the Microarchitecture Security Conference*.
- [23] Hamed Nemati, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Swen Jacobs. 2020. Validation of abstract side-channel models for computer architectures. In *International Conference on Computer-Aided Design (ICCAD)*.
- [24] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix C. Freiling. 2017. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. *ACM Transactions on Privacy and Security* 20, 3 (2017), 7:1–7:33. doi:10.1145/3079763
- [25] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. 2022. Revizor: testing black-box CPUs against speculation contracts. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 226–239. doi:10.1145/3503222.3507729
- [26] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3860)*. 1–20. doi:10.1007/11605805_1
- [27] Sepideh Pouyanrad, Jan Tobias Mühlberg, and Wouter Joosen. 2020. SCF^{MSP}: static detection of side channels in MSP430 programs. In *International Conference on Availability, Reliability and Security (ARES)*. 21:1–21:10. doi:10.1145/3407023.3407050
- [28] Jean-Jacques Quisquater and David Samyde. 2001. ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards. In *Smart Card Programming and Security*.
- [29] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing digital {Side-Channels} through obfuscated execution. In *USENIX Security Symposium*.
- [30] Cristiano Rodrigues, Daniel Oliveira, and Sandro Pinto. 2024. BUSTed!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect. In *IEEE Symposium on Security and Privacy (S&P)*. 3679–3696. doi:10.1109/SP54263.2024.00062
- [31] Prakhar Sah and Matthew Hicks. 2024. RIPencapsulation: Defeating IP Encapsulation on TI MSP Devices. In *WOOT Conference on Offensive Technologies*. 117–132. <https://www.usenix.org/conference/woot24/presentation/sah>
- [32] Luigi Soares, Michael Canesche, and Fernando Magno Quintão Pereira. 2023. Side-channel elimination via partial control-flow linearization. *ACM Transactions on Programming Languages and Systems* (2023).
- [33] Texas Instruments. 2012. MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide. <https://www.ti.com/lit/ug/slau367p/slau367p.pdf>.
- [34] Texas Instruments. 2014. Introduction to MSP430FR5969. <https://www.youtube.com/watch?v=QRJ0r-Zx2Hk>.
- [35] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *ACM Conference on Computer and Communications Security (CCS)*. 178–195. doi:10.1145/3243734.3243822
- [36] Daan Vanoverloop, Hans Winderix, Lesly-Ann Daniel, and Frank Piessens. 2024. Compiler Support for Control-Flow Linearization Using Architectural Mimicry. In *PriSC*.
- [37] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. 2025. Synthesis of Sound and Precise Leakage Contracts for Open-Source RISC-V Processors. In *ACM Conference on Computer and Communications Security (CCS)*.
- [38] Hans Winderix, Marton Bognar, Lesly-Ann Daniel, and Frank Piessens. 2024. Libra: Architectural Support For Principled, Secure And Efficient Balanced Execution On High-End Processors. In *ACM Conference on Computer and Communications Security (CCS)*. doi:10.1145/3658644.3690319
- [39] Hans Winderix, Jan Tobias Mühlberg, and Frank Piessens. 2021. Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. 667–682. doi:10.1109/EuroSP51992.2021.00050
- [40] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. 15–26. doi:10.1145/3213846.3213851
- [41] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 640–656. doi:10.1109/SP.2015.45
- [42] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 719–732. <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom>

Open Science

All our materials are open source under an MIT license at <https://github.com/martonbognar/ipe-balancing>. This includes the following:

- The programs used for profiling and reverse engineering the microcontroller's behavior.
- The benchmark programs.
- The extended version of the SCF-MSP tool supporting real-world side channel detection.