IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–35. DOI:XXXXXXXX

PoMMES: Prevention of Micro-architectural Leakages in Masked Embedded Software

Jannik Zeitschner[1] and Amir Moradi[2]

[1] Ruhr University Bochum, Horst Görtz Institute for IT Security, Bochum, Germany
firstname.lastname@rub.de
[2] Technische Universität Darmstadt, Darmstadt, Germany
firstname.lastname@tu-darmstadt.de

Abstract.
Software solutions to address computational challenges are ubiquitous in our daily lives. One specific application area where software is often used is in embedded systems, which, like other digital electronic devices, are vulnerable to side-channel analysis attacks. Although masking is the most common countermeasure and provides a solid theoretical foundation for ensuring security, recent research has revealed a crucial gap between theoretical and real-world security. This shortcoming stems from the micro-architectural effects of the underlying micro-processor. Common security models used to formally verify masking schemes such as the d-probing model fully ignore the micro-architectural leakages that lead to a set of instructions that unintentionally recombine the shares. Manual generation of masked assembly code that remains secure in the presence of such micro-architectural recombinations often involves trial and error, and is non-trivial even for experts.
Motivated by this, we present `PoMMES`, which enables inexperienced software developers to automatically compile masked functions written in a high-level programming language into assembly code, while preserving the theoretically proven security in practice. Compared to the state of the art, based on a general model for micro-architectural effects, our scheme allows the generation of practically secure masked software at arbitrary security orders for in-order processors. The major contribution of `PoMMES` is its micro-architecture aware register allocation algorithm, which is one of the crucial steps during the compilation process. In addition to simulation-based assessments that we conducted by open-source tools dedicated to evaluating masked software implementations, we confirm the effectiveness of the `PoMMES`-generated codes through experimental analysis. We present the result of power consumption based leakage assessments of several case studies running on a Cortex M0+ micro-controller, which is commonly deployed in industry.

Keywords: Side-Channel Analysis · Compiler · Software · Masking

1 Introduction

In our technologically advancing and increasingly interconnected world, adversaries are constantly seeking opportunities to compromise third-party systems and gain access to protected digital environments. Decades of research and scientific exchange have led to the establishment of highly secure mathematical algorithms that ensure confidentiality, authenticity, and integrity. Therefore, rather than breaking the algorithm itself, adversaries continuously seek for practically feasible attack vectors. Side-channel attacks, which exploit observable but undesired physical effects such as power consumption, timing differences or electromagnetic emanation during the execution of an implementation on a target device belong to such realistic attack scenarios. The burgeoning relevant research in novel machine learning attacks, improved signal processing techniques, and the development of sophisticated side-channel exploitation schemes highlight the importance of protecting against side-channel attacks as part of a multi-layered defense against modern adversaries. Therefore, it is crucial to integrate theoretically proven and practically robust countermeasures into security-enabled devices to ensure the secrecy of citizens' sensitive data.

Among the known countermeasures, masking [CJRR99] – inspired by secret sharing schemes [Sha79] – has received the most attention from the scientific community, as it allows to formally define adversary models and provide proofs for protected implementations. Although different types of masking are known, Boolean masking is the most common scheme used in the implementation of symmetric block ciphers. Due to the transparency of linear Boolean operations over Boolean masking, the majority of research deals with the difficulty of applying Boolean masking to non-linear operations, e.g., substitution boxes. During the emergence of side-channel countermeasures and formal models, it was thought that masking would be easy to apply to software because operations realized by sequential instructions simplify the model by assuming that instructions are well isolated. As a result, the lion's share of relevant research has focused on hardware masking, which was thought to be more challenging due to physical imperfections, such as glitches and parallelism, that complicate the modeling process. As a result, several hardware masking schemes with solid foundations have been introduced in the last decade, enabling the construction of masked hardware circuits that provide the desired security in theory and in practice. Examples include but are not restricted to models [FGP+18, BCP+20], constructions [NRS11, GMK16, GMK17, GIB18, MPZ22], composable modules [CS20, CGLS21, CS21, KM22, KSM22], and tools [KSM20, KMMS22, MM22, GHP+21, BMRT22, BBC+19].

While there is now a high level of understanding and knowledge about hardware masking, looking back at the software domain, it is clear that software masking is significantly less researched and understood. Although there are several publicly available masked software implementations, they are either not thoroughly evaluated in practice, or physical characteristics of the execution platform are ignored in their security proofs. Recent research [BWG+22] has shown that almost every theoretically proven secure masked software implementation exhibits unexpected leakage in practice. The reason for such a shortcoming is often the micro-architecture of a Central Processing Unit (CPU) and its physical behavior, which are commonly ignored. The micro-architecture of processors describes their internal organization and structure. It is usually abstracted away from software developers who interact with the processor through instructions given by the Instruction Set Architecture (ISA). The micro-architecture defines *how* the CPU handles instructions internally, what paths data can take, or where data can be temporarily stored. Thus, micro-architectural effects are physical defaults within the CPU that can lead to the unexpected recombination of masked data due to transitions in storage elements or glitches in combinational logic within the Arithmetic-Logic Unit (ALU), memory bus, or pipeline stages, resulting in an unforeseen leakage. Since software developers can only interact with the CPU via instructions, the micro-architecture and potential sources of leakage within the micro-architecture are a black box, especially when operating on commercial micro-processors where insight into the details and netlist of the CPU is strictly limited.

Currently, we have two options for overcoming this discrepancy between the theoretical and practical security of masked software implementations. The first is to outsource critical operations of the ALU to dedicated hardware by integrating masked hardware modules into the CPU [GMP+20, FBR+22]. The applicability of such a solution is fundamentally limited. First, it is not trivial to identify and isolate all sources of leakages inside the netlist of an ALU. Second, such modifications are beyond the capabilities of a software developer, who has no chance to modify the hardware of the CPU. The other option is to explicitly consider such effects during the generation of masked software. The authors of the recent work [GD23] collected general sources of leakage from the literature and derived specific notions and additional properties when applying Threshold Implementation (TI) [NRR06] in software. Originally developed for hardware platforms, TI allows for first-order secure masking in the presence of glitches. The technique divides a masked function into smaller coordinate functions, each of which takes a non-complete masked input and produces one share as output. The construction of the coordinate functions is based on the three properties *correctness*, *non-completeness* and *uniformity*. However, these properties as defined in [NRR06] do not cover common micro-architectural effects in software. Gaspoz and Dhooghe extended the non-completeness property for registers in micro-controllers with the notions of *horizontal* and *vertical* non-completeness. They also provided a refined definition for register uniformity in the context of software masking to secure sequential compositions. With the applied adaptions, the authors successfully presented practically secure software implementations of some block ciphers.

Another work [ZMM23] has gone one step further by incorporating multiple micro-architectural effects into a more or less general leakage model. The authors traced observable leakage found in the public literature back to the origins of the micro-architecture, such as transitions in developer-hidden storage elements, or glitches in combinational logic of the ALU. Based on the potential leakage behavior of the micro-architecture, they refined the ability of adversaries to obtain additional information induced by physical defaults of the micro-architecture beyond just the probed position. The leakage model is general because it is independent of a particular CPU implementation and is only interested in the potential micro-architectural effects that can cause leakage. This allows examining the security of arbitrary masked software implementations based on worst-case micro-architectural leakages but without specifying a particular CPU. Although implementations in [BC22, GD23] were partially written in assembly to avoid uncontrolled optimizations of the compiler, the authors of [ZMM23] discovered security flaws in some of their implementations. While this generalization handles various sources of leakage, it leads to a conservative and worst-case hypothetical architecture of the CPU to be able to capture as much leakage as possible originating from the micro-architecture. It means that the model is conservative in the sense that it assumes a CPU that contains all the micro-architectural sources of leakage defined within the model. However, a violation originating from a micro-architectural effect defined in this model is not guaranteed to result in observable leakage in practical measurements on a specific CPU. Considering all these additional properties and potential leakage sources when manually developing a masked software implementation is challenging and not easily feasible even for experts. As a consequence, some research has been conducted to automate the generation of secure software in order to reduce the development time and provide a higher level of confidence. It is noteworthy that the currently existing tools [SSB$^+$21, ABB$^+$21] that attempt to address the issues arising from micro-architectural details operate on already compiled assembly code. In other words, the masked binary is already provided by the software developer, which is then modified by such tools, mainly by inserting clean operation between critical instructions or replace critical instructions by handcrafted gadgets.

We tackle this problem one step ahead, at compile time. Compilers are a crucial part of almost every software development workflow. Unfortunately, off-the-shelf compilers are not aware of the concept of masking and do not handle masked data with sufficient awareness and care. This can lead either to the introduction of additional leakages or to constructions with reduced security (e.g. lower security order), even if no optimizations are performed by the compiler.

Our Contributions. In this work, we present a novel and open source tool PoMMES[1] for automated prevention of micro-architectural leakages in masked implementations targeting the ARMv6-M ISA on in-order processors. Given a masked high-level implementation in C, PoMMES generates assembly code that complies with a refined version of the general and CPU-independent leakage model presented in [ZMM23]. The structure of the C code is allowed to have arbitrary control flow, i.e., different kinds of loops, conditions, recursion and internal function calls. To the best of our knowledge, we present the first approach as a register allocation algorithm for compilers that is by design side-channel aware and thus free of all the micro-architectural effects identified in [ZMM23]. Such register allocation algorithms are a fundamental part of the back-end of any compiler, as their role is to allocate the actual registers of the target CPU to the program, which is written using only virtual registers. Further, PoMMES is not limited to any particular security order, and can be applied on higher-order masked software implementations.

We demonstrate the effectiveness of our tool by verifying the security of its generated assembly code with PROLEAD_SW, a simulation-based tool for assessing the security under the CPU-independent leakage model, introduced at CHES 2023 [ZMM23]. We also provide practical evaluation results on the commonly used ARM M0+ micro-processor. To the best of our knowledge, we are the first to provide such a practical evaluation for higher-order masked software implementations against multivariate attacks, and show the validity of not only the CPU-independent leakage model, but also our proposed register allocation algorithm.

Outline. The rest of the paper is organized as follows. Section 2 introduces the theoretical foundations of Boolean masking and recalls the CPU-independent leakage model. This provides the basis for understanding why masking schemes often fail to maintain their security in practice due to micro-architectural effects. Since we address the issue of secure software at compile time, we give an overview of compilers and in particular the register allocation algorithm in this section as well. An overview of works done in this research area is given in Section 3. In Section 4 we present our workflow from the high-level C code to hardened assembly code. This includes the translation to intermediate representation, the taint-tracking procedure and a detailed explanation of the modifications required for register allocation algorithms to be in line with our leakage model. We show simulation-based and practical evaluations of the assembly code generated by PoMMES in Section 5 and outline current limitations and attractive lines of future work in Section 5.5. Finally, we conclude the paper in Section 6.

2 Background

In this section, we review the basic knowledge needed to follow the rest of the paper.

2.1 Boolean Masking

Based on a solid theoretical foundation [CJRR99], masking is a side-channel countermeasure that has been thoroughly researched and widely accepted by both academic and industrial communities. Although it has been applied at various levels of abstraction, algorithm-level approaches have received the most attention. In general, the goal is to prevent the adversary from being able to predict the intermediate values of the circuit when observing the associated side-channel leakages. For this purpose, each sensitive variable $x \in \mathbb{F}_n$ is split into $d+1$ shares $(x_0, \dots, x_d)$, where the first d shares are drawn uniformly at random from $\mathbb{F}_n$. Following Boolean masking, the last share is determined as $x_d = \left(\bigoplus_{i=0}^{d-1} x_i\right) \oplus x$. Algorithmically, all computations that are initially performed on x should now be performed on the shares $(x_0, \dots, x_d)$. Under this definition, linear Boolean operations are trivial to

[1] https://github.com/ChairImpSec/PoMMES

mask, since each share can be processed individually as $\mathsf{L}(x) = \bigoplus_{i=0}^{d} \mathsf{L}(x_i)$. Non-linear operations, on the other hand, require combining different shares and are therefore more complex to secure. An implementation is said to be d-th order secure, if any combination of d intermediate values does not reveal any information about the secret x. The adversaries' ability, i.e. the information they can gain from a device under attack, is commonly abstracted by the probing model explained as follows.

2.2 Probing Model

Adversary models enable the formal representation of an adversary's capabilities and define what informational leakage can be extracted from the target. The advantage of these models is that they allow the generation of systematic masking schemes and formal security verification, assuming that the capabilities of the adversary defined in the underlying security model hold. One of the earliest and most common security models is the d-probing model [ISW03] proposed by Ishai, Sahai and Wagner. In this probing model, the adversary is allowed to observe up to d signals carrying intermediate values during the processing of information. Due to its simplicity and high level of abstraction, this model attracts wide attention in the scientific community, especially for the generation of security proofs [RP10a, Cor14, GSM+19]. However, the d-probing model comes with the assumption that each probed intermediate value leaks independently and that *only* the stable intermediate is observable. Unfortunately, these assumptions are generally not met in practice, since physical defaults such as glitches, transitions and coupling may occur during the operation on a physical device. To achieve a better transition between provably secure masking schemes within an adversary model and practical security, physical defaults must be taken into account.

Glitches occur within combinational logic and are caused by different path delays between synchronization stages. They can cause unintentional value recombinations within the combinational logic, allowing the adversary to observe not only the stable value at the position probed, but also to observe other intermediate values within the probed combinational logic back to the last synchronization point. The glitch-extended d-probing model [FGP+18] allows up to d glitch-extended probes to be placed. Here, each probe retrieves information about the all stable intermediates that contribute to the probed intermediate.

Transitions occur when storage elements change their containing value. In this case, the adversary is able to observe the incoming and outgoing intermediate value during the transition, i.e. when the value of a storage element changes. The transition-extended d-probing model [FGP+18] allows up to d transition-extended probes to be placed on storage elements. In this model, each probe is able to observe the old and new stable intermediate value of the storage element.

The (g, t, c)-robust probing model[2] introduced by Faust et al. [FGP+18] unifies the extended probing models explained above for hardware circuits. However, the aforementioned physical effects occur not only in hardware implementations but also in the execution of software, since it must run on a dedicated hardware device with its own physical defaults. Indeed, numerous practical evaluations [BWG+22, GOP21] showed that d-probing secure software implementations fail to hold their security in practice.

Similar to the robust probing model for hardware implementations, the CPU-independent leakage model [ZMM23, GD23] acknowledges physical defaults introduced by the micro-architecture inside general purpose CPUs. It derives the ability of adversaries to retrieve additional knowledge based on physical defaults inside a generalized CPU and its micro-architecture, in particular the register file, ALU, pipeline, decoding stages, and memory

[2] g for glitches, t for transitions, and c for coupling. We do not cover coupling in this work as it highly depends on the layout of the circuit.

lanes. All extensions of the probes can be traced back to the origins of transitions and glitches.

2.3 Composability and Gadget-Based Masking

Instead of relying exclusively on practical measurements, e.g., in form of Test-Vector Leakage Assessment (TVLA) [GJJR11], whose security statement is dependent on the utilized setup and environment, the focus of the scientific community has shifted to formally verifying masked implementations in a security model. It allows universal reasoning, which is independent of factors such as execution platforms or experimental equipment. Unfortunately, finding an efficient approach for the direct verification of large implementations remains an open problem. The trivial approach of checking that the distribution of all combinations of d probes are independent of all secrets poses significant efficiency problems with growing number of probes. There exist dedicated tools [BBC+19, KSM20, BGG+21, BMRT22, MM22, ZMM23] for hardware and software implementations that are able to overcome this issue for small implementations and low masking orders. However, larger implementations, e.g., full block ciphers, at higher orders ($d > 1$) are even for them practically infeasible in terms of runtime and/or memory requirements.

A widely acknowledged solution for this limiting problem is called gadget-based masking. It follows a divide-and-conquer approach. The idea is to reason about the security of large implementations, by composing smaller, formally verified and composable building blocks, called gadgets. Gadgets themselves implement small functionalities, e.g., a masked multiplication or XOR operation, and are equipped with properties that allow their secure composition. Now the side-channel resistance can be efficiently verified for each gadget independently as long as the circuit is composed of only such gadgets. A proper composability notion for each gadget then allows the secure composition of gadgets to create the desired, larger functionality. Generally, composability notions define how probe propagation behaves through each gadget and establish sufficient conditions to allow side-channel resistant combination of gadgets. For more relevant information, we refer to the composability notions NI [BBD+15], SNI [BBD+16], and PINI [CS20].

2.4 CPU-Independent Leakage Model

The CPU-independent leakage model [ZMM23] has been presented at CHES 2023 to verify the d-probing security of masked software implementations along with a tool that is in line with ARMv6-M, ARMv7-M and ARMv7E-M ISAs. Compared to other adversary models, the authors considered a wide range of micro-architectural effects collected from the public literature [MPW22, GPM21, PV17]. Such micro-architectural effects play a crucial role in making meaningful inferences about the security of embedded software implementations in practice. Practical evaluations of masked software [BWG+22, BC22] show that probing secure masking schemes alone are not sufficient in software, and that additional attention must be paid with respect to the handling of the shares. Therefore, the authors in [ZMM23] formalized some micro-architectural effects into generic security statements that act as basis for their model. The advantage of such generic statements is that no internal information about the specific design of the CPU is required. This is important in many scenarios, because either the access to the hardware designs of CPUs is restricted by intellectual property, or certain compilation phases are not designed to take this information into account. The task of a compiler is to generate executable code for different processors in a generic way, i.e., without having knowledge of internals other than the information provided by the ISA of the respective CPU.

In the following we recall the micro-architectural effects discussed in [ZMM23]. Consider a CPU that executes a program with a set of instructions $I = \{i_0, \ldots, i_{|I|-1}\}$ and leverages

$|R|$ registers $R = \{r_0, \ldots, r_{|R|-1}\}$, each of which is B bits wide, where we denote a single bit of register $r_f \in R$ at bit position b by r_f^b. Besides registers, the CPU has also a memory area M where each memory location m can be accessed either with byte, half-word, or word granularity.

Transitional Leakage. Transitions describe the changes between successive values stored in a register across different instructions. If a register $r_f \in R$ contains a value that was set by the instruction i_j, and the content of r_f is overwritten by the instruction i_k, then each bit position b with $0 \leq b < B$ leaks the joint information between the old and new bit-value of r_f^b when instruction i_k is executed. If the new and old values contain information about different shares, it leads to an unintended recombination as a transition-extended probe is able to observe both values.

Horizontal Leakage. Inside the ALU, multiple arithmetic and logical operations are always performed simultaneously regardless of the desired operation of the instruction, and later the appropriate result is selected to be stored in the target register [GD23]. Certain operations such as addition, subtraction or multiplication do not operate on the bits individually, but result in interactions between different bits of the given operands. For example, while every single-bit output of an XOR instruction depends only on the corresponding bit of the registers (operands), an addition instruction over two registers requires the interaction of all register bits. For instance, the most significant bit of the addition's result depends on all bits of both operands. As a result, glitch-extended probes at the output of the ALU are able to observe the interaction of these register bits. To capture such interactions between different register bits, the model allows access to all bits of pairwise combinations between two registers in $|R|$ that are the operands of the instruction i_j.

Vertical Leakage. To select which registers are required for the current instruction and must be sent to the ALU, there must be some combinational logic, e.g. in the form of multiplexers, that pass the desired registers to the ALU input. Glitch-extended probes in such combinational logic are able to propagate backwards to multiple registers in the register file and could reveal information about all registers that are connected to that logic. Therefore the model covers this effect in a worst case manner, meaning that at instruction i_j the attacker is able to gain the joint information about the b-th bit of all registers in R.

Memory Overwrite Leakage. Overwrites do not solely occur in registers, but can happen at any storage element. Therefore, the authors of [ZMM23] additionally considered the memory area M. The transition between a value stored at memory address $m \in M$ during instruction i_j and the value that is written to the same address m during instruction i_k gives the attacker the combined information between the old and new value at that memory address by placing a transition-extended probe on m. Similar to the transitional leakage on the registers in R, if both values contain shares of masked data, this may reduce the security order.

Memory Remnant Leakage. During the interaction between the registers and the memory area, combined leakage can be observed not only between the same memory location, but also between two instructions operating on two different memory addresses m and m'. The authors modeled this observation with an additional hidden register that is placed between the memory area M and the register file, called the memory shadow register. In short, this means that if a value is loaded from (respectively stored to) memory address m during instruction i_j, followed by a load from (respectively store to) memory address m' during instruction i_k, an adversary will be able to observe the combined information between the consecutively read or stored values, even if $m \neq m'$. In particular, the adversary gets

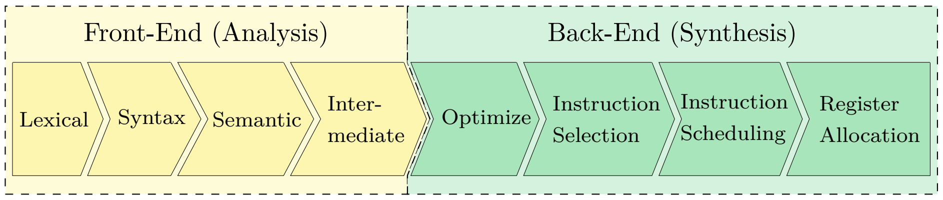


Figure 1: Schematic depiction of compiler phases.

always the combination of the *word-size* values regardless of the specified data-length in the mnemonic. This means that independent of the number of bytes requested by the instruction, e.g., loading one byte with `ldrb` or storing two bytes with `strh`, the shadow register contains the full 4-byte value when the architecture is designed to handle the memory with word (4-byte) granularity. As a side note, we identified a flaw in this approach and altered the model and the tool, which is explained in detail in Section 4.1.

Pipeline Forwarding Leakage. Common processors use an n-stage pipeline that divides the execution of instructions into n subtasks that are separated by pipeline registers. To optimize processor performance, pipelines can have a dedicated forwarding logic, that passes values from later pipeline stages to earlier stages before writing them back to the register file. While such logic reduces the stalls inside the pipeline, glitches inside this logic are able to reveal information about the values in all pipeline stages [GPM21]. The CPU-independent leakage model handles this micro-architectural effect conservatively by giving the adversary joint information about the b-th bit of every register in the register file across the last n instructions.

2.5 Compiler

Software developers frequently use compilers to translate high-level source code into executable software. A compiler is a piece of software that takes human-readable, high-level source code, and translates it into a low-level language that ultimately results in executable software. In the context of embedded software development, the C language enjoys high popularity and widespread use because it has only a narrow abstraction layer from the hardware and provides us with a low computational overhead. Arguably the two most popular open-source compilers for C are `Clang` and `GCC`. In this paper, we focus mainly on the `GCC` compiler, since ARM provides a powerful and widely used compiler for ARM binaries with its ARM-GCC toolchain. Although various compilers differ in their specific behavior and technical handling of inputs, we give a high-level overview, not only because most of them follow similar phases as shown in Figure 1, but also because it provides a better understanding of where our research comes into play.

From a high-level perspective, compilers have a front-end and a back-end module. The main task of the front-end is to analyze the syntax and semantics of the source code. Therefore, the **Lexer** scans the source code character by character and groups them into tokens, i.e., meaningful elements for the compiler. These tokens are passed for **Syntax** analysis to the Parser, which uses a predefined grammar to construct a tree-like data structure and checks the correct combination of tokens. This tree structure is used by the **Semantic** analysis tool to determine whether the source code (respectively the arranged tokens) has a valid meaning. At the end of the front-end pass, the compiler generates an **Intermediate Representation (IR)**. Its representation is chosen to facilitate the translation into the explicit low-level language.

While the front-end is practically platform-independent, the second module is responsible for the synthesis and architecture-specific generation of the code in the low-level language. Besides several (optional) **Optimization** steps, the platform-dependent code generation consists of **Instruction Selection**, **Instruction Scheduling**, and **Register Allocation** and are the main tasks of the back-end. Although the IR passed from the front-end correctly resembles the semantics of the source and destination languages, its representation of instructions does not correspond to real ISA-specific instructions. Therefore, the task of the **Instruction Selection** is to choose a set of instructions provided by the ISA of the underlying processor and replace the abstract IR instructions while maintaining semantic correctness. During **Instruction Scheduling**, the compiler considers further performance-critical properties of the instructions. This includes the order in which the instructions can be arranged among each other. By reordering instructions, the compiler can consider dependencies, which reduces unwanted stalls in the hardware and results in increased performance and minimized runtime. The final step of the back-end, i.e., before the compiler outputs the result, is the **Register Allocation**. Up to this point, all instructions operate on purely virtual registers. To reduce the complexity of the other previously executed steps and to ensure modularity between them, the compiler assigns virtual registers to all source code variables and internally created variables. Since the developer can use any number of source code variables in the high-level language, the number of virtual registers is potentially unlimited. However, since the processor is only able to operate on a limited number of real registers available in its hardware, it is the task of the **Register Allocation** algorithm to ensure that every virtual register is mapped to a register provided by the ISA. Note that for each of the above code generation tasks, there exist different algorithms with different strengths and weaknesses suitable for diverse application scopes.

2.6 Linear Scan Register Allocation Algorithm

The linear scan register allocation algorithm was proposed by Polleto and Sarkar in 1999 [PS99]. It performs the mapping between virtual registers V and real registers R, where possibly $|V| \gg |R|$. It is a global register allocation technique, which means that it operates over the entire function, unlike other techniques that perform the allocation either on basic block level or across functions. The linear scan follows a comparatively simple greedy approach, and captivates through an asymptotically linear runtime, while other global allocation algorithms have quadratic complexity.

Before the actual algorithm is executed, the compiler must generate the necessary data structure called *live intervals* for each virtual register in the IR. Live intervals are continuous ranges that define the start and end of each virtual register. In other words, each virtual register $v \in V$ has exactly one live interval $l = [x, y]$, where x is the earliest instruction that uses v, and y denotes the last instruction that uses v. To construct such live intervals, the representation of the instructions must be in a linear form, i.e., if necessary the IR must be transformed into a linear sequence of instructions where each instruction is numbered in ascending order. For the sake of clarity, we extend the live interval l with additional explicit information. For each virtual register v, we define the data-structure $d = \langle l, v, r, m \rangle$, which is a set containing the live interval, the virtual register v, the real register r, and a memory position m. We call this set of information *live mapping*.

The algorithm then generates two lists. The first list, called *active*, contains all live mappings that have a real register assigned and whose live interval l overlaps with the current instruction position. The second list, called *available*, contains the real registers that are not assigned to any live mapping in *active* and is initialized with all registers permitted by the respective ISA.

A complete illustration of the algorithm is given in Algorithm 1 to Algorithm 3. For now, only the *original* version of the algorithm, i.e., without our modifications, is important and is highlighted in black. Our adaptions, marked in red, will be explained in Section 4.4.

Algorithm 1 shows the main loop of the linear scan register allocation. The algorithm iterates over all live mappings, sorted by the increasing start point of their live intervals l. At the beginning of each iteration, i.e., before assigning a register to the current mapping, the algorithm checks whether the list *active* contains mappings whose interval endpoint is smaller than the start point of the current live interval. This functionality is outsourced to Algorithm 2. If this condition is fulfilled, we can remove the corresponding live mapping from *active* and put its assigned register r back into the *available* list, since this means that all further live mappings that need a register to be assigned will not interfere with the removed live mapping. After that, the algorithm distinguishes between two cases. If there are still registers in *available* (respectively the size of *active* is less than $|R|$), we can simply take a register from *available*, assign it to the current mapping, and add it to *active*. In the second case, there is no register in *available*, i.e., all registers are currently assigned to a live mapping. In this case, the algorithm must store a live mapping to memory instead of a register. This procedure is called *spilling*, and its operational behavior is described in Algorithm 3. To reduce the number of spills, the algorithm selects the mapping that needs to be spilled based on the highest interval endpoint. Hence, either the current mapping or a mapping from *active* should be spilled into memory.

3 Related Works

To the best of our knowledge, no research has been published that presents a micro-architectural side-channel-aware version of a register allocation algorithm. However, there are certain lines of research that either attempt to harden the software directly after compilation or mimic the behavior of a compiler with side-channel protection in mind.

Moss et al. [MOPT12] introduced their own compiler that automatically applies Boolean masking to a given high-level source code in CAO [BNPS05] and generates ARM assembly. First, by annotating the sensitivity to variables in the code and parsing it to an intermediate representation, the propagation of the sensitivity through the program flow is tracked. In a second step, if predefined security violations are found, the tool transforms the program based on repair heuristics. Their prototype is limited to first-order masking and makes the critical assumption that the processor only leaks information associated to each instruction independent of other instructions. It has been shown in [MPW22, GPM21, PV17] that such assumptions do not hold in practice.

Bayrak et al. [BRB+11] introduced a customized compiler that applies Boolean masking to an unprotected AVR assembly code. The authors perform information leakage analysis on the code, identify sensitive instructions and apply the necessary code transformations. Apart from the fact that micro-architectural leakages are completely ignored, similar to the scheme presented in [MOPT12], their approach is limited to first-order masking.

Eldib and Wang [EW14] presented a scheme to take unprotected C code as input, transform the program into the LLVM IR, and mask each instruction based on the scheme given in [BGK04], i.e., the result of computations should become statistically independent of secret data. For practical security, the authors insufficiently considered micro-architectural effects. While they ensure that each instruction itself does not leak information, inter-instruction leakage, i.e., leakage between different instructions, is not considered, which is potentially harmful [MPW22]. Furthermore, after their transformation, the compiler is able to introduce changes to the IR in later phases, such as re-ordering the sequence of instruction whose order is important for the security of the scheme, or assigning registers to instructions that make them prone to transitional leakage of different shares.

Abromeit et al. [ABB+21] used a standard off-the-shelf compiler to compile annotated but unprotected C code. After performing information and control flow analysis, they substitute insecure instructions with device-specific and formally verified gadgets, i.e., side-channel resistant but functionally equivalent versions. In addition to the fact that they can only work on first-order designs, the gadgets, which are the crucial part of this process, must be provided and constructed manually by security experts.

ROSITA, presented by Shelton et al. [SSB$^+$21], uses an iterative approach. A pre-compiled binary of a masked implementation is fed into the leakage emulator ELMO*, a refined version of ELMO [MOW17]. The assembly code of the binary together with the result of the leakage analysis is then passed to ROSITA, a rule-based code-rewrite engine. Its core strategy is to eliminate leakage by inserting instructions that overwrite sensitive data with a random value, i.e., the well-known random precharging technique [MOP07]. The process of leakage analysis and code-rewrite is repeated until no leakage is detected. The accuracy of leakage detection is highly dependent on the emulator and its underlying model. If the emulator is not accurate enough, leakage sources will remain undetected. Furthermore, ROSITA occupies one register that contains the random mask throughout the implementation. This means that either the developers must avoid using this register (if the implementation is written directly in assembly) or the compiler must not use this register by passing a particular flag during compilation (if the implementation is written in a high-level language). This limits the number of available registers, since we have fewer registers to work with than those available in the underlying architecture. As a consequence, the total number of instructions may increase as the register pressure increases. Register pressure describes the conflict between the number of variables that could reside in different registers at the same time and the total number of registers available. Naturally, the register pressure increases as the number of registers is reduced. As a result, values must be stored in memory to make space for other values to use the register. Such a restriction to avoid using certain registers is *not* present in `PoMMES`. We can freely use all registers available in the architecture. Furthermore, we would like to note that ROSITA is only able to add instructions to mitigate leakage, i.e., it repairs the implementations by not dealing with register allocation algorithms or any other steps of the compiling process.

4 Technique

In this section, we describe the workflow of `PoMMES`. First, we modify the CPU-independent leakage model to more closely reflect the behavior of the processor. We then transform the capabilities of an adversary in this model into constraints on the generation of masked assembly, and make adjustments to the linear scan register allocation algorithm to be in line with our propositions.

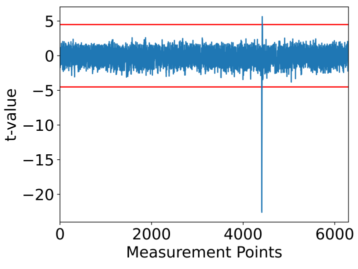
4.1 Refinement of the CPU-Independent Leakage Model

Before we apply the implications for the register allocation algorithm that we will derive from the CPU-independent leakage model, we need to make a subtle yet crucial modification to the memory remnant leakage model. Currently, the leakage model in [ZMM23] (also reviewed in Section 2.4) assumes that values relevant to both load and store instructions (from and to memory) are transferred via the same bus. In their model, the authors introduced *one* hidden shadow register that is overwritten by any value either loaded from memory to a register or stored from a register to memory. This means that only *consecutive* memory operations, regardless of the concrete mnemonic, could potentially leak information through transitional leakage in the hidden shadow register. However, through experimental analysis, we have recognized its inconsistency with the actual leakage behavior of ARMv6-M compliant micro-processors.

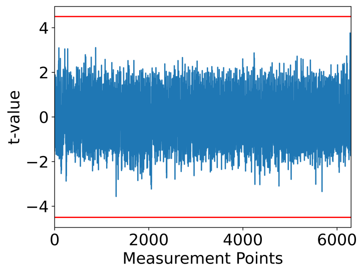
To explain the problem, let us consider the assembly sequence given in Figure 2(a). The first load operation on line 2 loads the first input share from memory into the register `r2` and stores it at a different location in memory on line 7. The load instruction at line 12 loads the second input share into the register `r5`. In the original CPU-independent leakage model, the shadow memory register always contains the value of the last memory operation, regardless of whether it was a store or a load operation. In particular, the first share stored in line 7 and the second share loaded in line 12 would be recombined in the shadow memory register, creating transitional leakage that exposes the unshared input. To avoid this we initially placed a dummy store instruction between these memory operations at line 10, which is highlighted in red. According to the original leakage model, there should be no leakage as there is no more transitional leakage between the shares, since the inserted store instruction has overwritten the shadow memory register with a non-sensitive value. However, when we conducted power consumption based leakage assessment t-test, we noticed clear leakage right when the first share was loaded in line 12, as shown in Figure 2(b). In a second experiment we *only* performed one modification to the assembly code where we switched the dummy store operation to a dummy load operation in line 10. We repeated our evaluation with this modified assembly code, and the results in Figure 2(c) clearly reports the absence of the aforementioned leakage. This behavior cannot be explained with the original leakage model presented in [ZMM23]. This observation has led us to conclude that a single memory shadow register, which holds both the values loaded from memory and the values stored to memory, does not model the actual leakage behavior closely enough.

```
1   [...]
2   ldr r2, [r5]
3   mov r3, r13
4   str r3, [r3]
5   mov r3, r11
6   subs r3, r3, #20
7   str r2, [r3]
8   lsls r2, r4, #2
9   mov r3, r13
10  str r3, [r3]
11  adds r3, r0, r2
12  ldr r5, [r3]
13  [...]
```

(a) Leaking assembly sequence.



(b) T-test results of initial assembly code.



(c) T-test results of assembly code in line with the refined memory remnant effect.

Figure 2: Experimental analyses showing leakage that are not explainable with the original CPU-independent leakage model [ZMM23] leading us to a refinement of the memory remnant effect.

Therefore, we propose to adapt the memory remnant effect as follows. Besides the already existing shadow register, we add two additional memory shadow registers, which we call r_{store} and r_{load}. The content of these hidden registers only change when a corresponding memory instruction is executed. More precisely, when at instruction i_j and i_k values are loaded from memory addresses m and m', and there is no other load instruction between i_j and i_k, the adversary will be able to observe the combined information of the loaded values regardless of any store instruction executed between i_j and i_k. The same principle applies to r_{store}, which deals with instructions that store values into memory. With the introduction of these two additional shadow registers r_{store} and r_{load}, we are able to explain the leakage observed in Figure 2. The first share that is loaded into register `r2` in line 2 resides in the shadow register r_{load}. If we load the second share in line 12 without any further load instructions in between, we get the transitional leakage of both shares in r_{load}, which again reveals the unshared value. This leakage is eliminated when substituting the mnemonic in line 10 from a non-sensitive store operation to a non-sensitive load operation. We argue that our modification is still consistent with a CPU-independent leakage model for two reasons. First, similar to all other effects that are encompassed in this model, they are based on leakage observations and experiments. These observations are the foundations from which general micro-architectural effects (such as transitions in hidden registers) are derived. Second, with our adaption, we only extend the generality of the model to allow not only subsequent memory operations to leak jointly, but additionally subsequent memory operations of the same type.

4.2 GIMPLE

Our modifications only affect the back-end of a compiler, i.e., operations on the respective intermediate representation. The language-independent IR of the `GCC` compiler is called `GIMPLE`[3] and provides simplified expressions and basic operations. The `GIMPLE` representation can be obtained by instructing the `GCC` compiler (via a specific flag) to pipe the IR of the high-level code into a file. We intercept the compilation process right after the IR is generated, so no optimization, instruction selection, or instruction scheduling has taken place. We would like to emphasize that our main focus is the register allocation algorithm. Prior back-end passes, i.e., optimization, instruction selection and instruction scheduling may introduce sources of leakage that the register alocation algorithm cannot adjust. Therefore, we do not perform optimizations on the `GIMPLE` code and use trivial selection and scheduling. More precisely, each final assembly instruction mirrors exactly each `GIMPLE` statement and operates only on the exact data size of the original variables. Furthermore, no instructions are reordered. The `GIMPLE` IR format divides the scope of the given C function into multiple basic blocks, which are atomic instruction sequences that have no entry and exit point other than the beginning and end of the basic block. Each instruction in these basic blocks is a `GIMPLE` statement. The functionality of the statements is generally grouped into 1) simple constant assignments, 2) unary and binary expressions, 3) possible function calls, and 4) memory handling in the form of pointers and arrays. The statements consist of one operation, at most three operands (one destination and two source operands), and are in Static Single-Assignment (SSA) form. The three-address representation already closely mimics the structure of actual instructions, since our target micro-processors typically operate on one target and two source registers as well. The SSA form is an additional property of the IR, where each variable is assigned exactly once and must be defined before use. This form is not only useful (and sometimes required) for compiler optimizations, but also helps us in the information propagation step by making it easier to track different security levels and identify dependencies between instructions and other variables. We parse the generated `GIMPLE` IR and build a Control-Flow Graph (CFG), which is used in the information flow analysis phase explained below.

4.3 Sensitivity Tracking

We perform our information flow analysis statically based on the CFG generated in the previous step, i.e., without running the program. To track the propagation of sensitive information through a function, we need to annotate the inputs. We do this by defining a C macro in the source file called `PoMMES_INPUT_<function_name>`, followed by some annotations according to the number of inputs of the function, each of which can be either `PUB` or `SEC` (for public or secret). Since the function that `PoMMES` operates on is able to call other functions, we also have to annotate (if necessary) the sensitivity of the inputs and possible outputs for the internal function calls. For this, we use the macro `PoMMES_FUNCTION_CALL_<function_name>` with the same possible annotations as described above. All internally used variables are initially set to `PUB`, but their sensitivity may change during the tracking process.

We have chosen Kildall's Algorithm [Kil73] for our information analysis. Each basic block in our CFG gets two lists, called *in* and *out*, which track the sensitivity of all

[3] https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html

incoming, respectively outgoing, variables. We initialize a worklist with the entry basic block of the function and fill *in* with the input variables and their sensitivities. The algorithm removes one basic block from the worklist and performs the analysis sequentially on each instruction of the basic block. Therefore, we need to define how secret information is able to propagate. Generally speaking, if at least one source operand in any assignment or arithmetic instruction is marked `SEC`, its destination should be marked `SEC` as well. For memory instructions, it holds that loading from a memory address containing a secret value taints the destination as `SEC`, while storing secret values to a memory address marks the content of the address `SEC`. Note that the address that contains sensitive information does not necessarily have to be sensitive as well. Our tool is able to distinguish between the case where the address itself contains secret related information, e.g., in a table lookup, or just points to a secret value without being secret dependent, e.g., a pointer to input shares. Furthermore, we allow only one sensitivity annotation for each variable, i.e., variables marked as `SEC` are not able to change their sensitivity back to `PUB`. At the end of the basic block, *out* contains the updated sensitivity of all variables that have been detected. If the algorithm identifies changes in sensitivity between *in* and *out*, we add the subsequent basic blocks to the worklist. When there are no more basic blocks in the worklist, the algorithm terminates, and we have the final sensitivity assignment for each variable in the function. For the final annotation, we introduce the sensitivity level s, which will be a new entry in the live mapping d of every virtual register (see Section 2.6). We call a live mapping *sensitive*, if s is set to `SEC`.

4.4 Our Linear Scan Register Allocation Algorithm

Considering the possible micro-architectural leakage sources explained in Section 2 and Section 4.1, our goal is to derive properties that must be satisfied to ensure that such micro-architectural effects do not violate any security assumption that is considered by the underlying masked implementation. We use these properties in the next steps to modify the linear scan algorithm to suite our needs.

Proposition 1 (Leakage-Free Transitions)**.** *A real register that is marked as sensitive, i.e., is or was assigned to a live mapping of a sensitive virtual register, must not be assigned to a live mapping whose virtual register is also marked as sensitive.*

Proposition 2 (Leakage-Free Memory Overwrites)**.** *A memory address that is marked as sensitive, i.e., containing a sensitive value, must not be overwritten with another sensitive value.*

Proposition 3 (Leakage-Free Memory Remnants)**.** *Two sensitive live mappings must not be loaded subsequently, and two sensitive live mappings must not be stored subsequently to ensure no combined leakage in our refined definition of memory remnant leakage. Furthermore, there must not be two subsequent, arbitrary memory operations of sensitive live mappings. Note that subsequent operations are not necessarily consecutive operations.*

Proposition 4 (Leakage-Free Pipeline Forwarding)**.** *No two sensitive live mappings must be in n consecutive instructions, where n is the pipeline-depth.*

Proposition 5 (Leakage-Free Vertical)**.** *The live intervals of any two sensitive live mappings must not overlap in order to ensure that only one secret value is in the register file at a time.*

Proposition 6 (Leakage-Free Horizontal)**.** *At most one sensitive value must be stored in one register at any time.*

```
for(int i=0;i<2;++i)
{
   state[i]^=key[i];
}
```

(a) High level

```
mov rA, #0
.loop:
ldrb rB, [rC,rA]
ldrb rD, [rE,rA]
eor  rB, rB, rD
strb rB, [rC,rA]
add rA, #1
cmp rA, #2
blt .loop
```

(b) Unoptimized assembly

```
ldrh rB, [rC]
ldrh rD, [rE]
eor  rB, rB, rD
strh rB, [rC]
```

(c) Optimized assembly

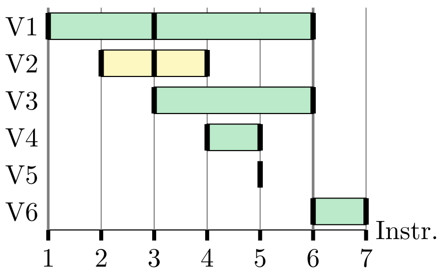
Figure 3: Instruction selection and optimization may lead to horizontal leakage.

Note that Proposition 6 is not solely achievable with a modification of the register allocation algorithm. In fact, it defines restrictions on the optimization pass and the instruction selection algorithm (see Figure 1), since it selects the exact ISA instruction to transform the IR. For clarification, let us consider the example shown in Figure 3(a), where `state` is represented by a byte array containing two shares. The goal is to update `state` by adding `key`, which is also represented by a byte array containing two shares. In the high-level code, we loop over both indices and add the key to each array index individually. If the compiler does not optimize the code and selects the instructions based on the size of each operand, the assembly code may look like Figure 3(b). However, to minimize the number of instructions, the compiler may apply modifications as long as it ensures the correctness of the final result. In this particular example, it may choose not to handle each byte separately, but to perform the operation directly on both indices, since they fit together in one register and the same operation is applied to each of them (see Figure 3(c)). It can be seen that regardless of the chosen specific register, the instruction itself always loads both shares at once into the register, causing horizontal leakage.

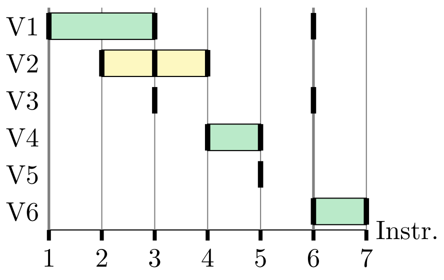
4.4.1 Leakage-Free Vertical Modifications

Generation of Live Mappings. To ensure vertical separation between live intervals of sensitive live mappings, we start our modifications at the generation of live intervals. We make use of the sensitivity attribute s (introduced in Section 4.3) in the live mappings and check for each combination of sensitive live mappings whether their live intervals overlap. If this is the case, we divide the intervals of these mappings into the minimal number of disjoint ranges. To generate these ranges, it is not enough to save only the start and the end of each interval, because we need to define the boundaries of the new ranges, which are based on the instructions that make use of the sensitive virtual registers. Therefore, we introduce the list u to our live mapping, which collects all *use points*, i.e., all instruction indices where the corresponding virtual register is used. This means that all elements in u are between the start and end of the initial live interval.

A visual representation of the transformation of live intervals is given in Figure 4. The live intervals of the sensitive virtual registers V1, V3, V4, V5, V6 and the non-sensitive virtual register V2 in Figure 4(a) are computed as in the original algorithm. With our sensitivity level s, we identify that the live intervals of the sensitive virtual registers V1, V3 and V4 overlap. This means that the linear scan would assign them to different registers, leading to the presence of multiple sensitive values in different registers and thus violating Proposition 5. Therefore, we split the intervals of these virtual registers so that they do not overlap (see Figure 4(b)). Note that instruction 6, for example, uses the sensitive virtual registers V1, V3 and V6, which effectively results in three sensitive values in three registers in the same instruction. We cannot prevent this tangent use of sensitive registers as the function provided by the user requires the sensitive values to be in the registers to execute this instruction. We should highlight that if such an instruction leads to a leakage,



(a) Original live intervals



(b) Non-overlapping of sensitive intervals

Figure 4: Live intervals with sensitive virtual registers V1, V3, V4, V5, V6 (marked as 🟩 green), while V2 is not annotated as sensitive (marked as 🟨 yellow). The *use points* are marked with a black vertical line.

it originates from the original masked C implementation given by the user. In other words, if the algorithm of the given masked C implementation is correct with respect to security (e.g., fulfills probing security requirements), it is ensured that such instructions combining multiple sensitive values do not lead to leakage.

When we split the sensitive live intervals, we create gaps between the ranges (see V1 in Figure 4(b)). To ensure the correctness of the program, we either have to ensure that the content of the register is not overwritten until we arrive at the next range of the live mapping or save the content by writing it to memory, i.e., spill the mapping. Holding the value in the register is not an option for sensitive live mappings, because it would violate Proposition 5, since other sensitive intervals are active in those gaps. Therefore, we mark these mappings during the generation of the live mappings as spilled and assign a stack location m in memory to each of them.

In general, Proposition 5 is not fully satisfied when sensitive live intervals do not have any overlap. The live intervals only mark the duration where the virtual registers are in *active*, i.e., in use during the execution of the program. Afterwards, during the expiration of intervals (Algorithm 2), the algorithm only moves the real register back to *available*. This does *not* mean that the value of the register is cleared, it just notifies the algorithm that this register can be assigned to another live mapping. This may lead either to transitional leakage of sensitive values if the register is assigned to another sensitive live mapping, or to vertical leakage if the sensitive value resides in this register while other sensitive mappings are in *active*. To attain sufficiency for Proposition 5, we apply two adjustments. First, we modify the list *available* and place constraints on the selection of available registers in the linear scan algorithm. Second, we adjust the spill procedure during the algorithm. These two adjustments are explained below.

Selection of available registers. Instead of just storing the free real registers r in *available*, we store a tuple (r, s) that tracks whether an available register was previously assigned to a sensitive live mapping. Depending on the sensitivity of the next live mapping to which a register should be assigned, we distinguish between the following two cases.

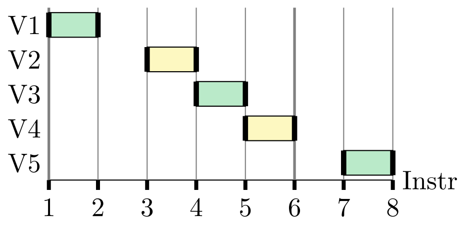
- In the first case, the next live mapping is non-sensitive. We check in *available* if we identify a tuple, whose sensitivity s is set to `SEC`. If we find a sensitive available register, i.e., a register that still contains secret information, we assign the register to the non-sensitive live mapping. This removes the sensitive value from the register before the next sensitive live mapping becomes active. Trivially, if there is no sensitive register in *available*, we assign any available register to the live mapping. This modification reflects lines 12 to 16 highlighted in red in Algorithm 1.

- In the second case, the next live mapping, that needs a register assignment, is sensitive. Before we check if there is a suitable register in *available*, we make the following important observation.
  - If the virtual register of the sensitive live mapping starts as a destination register, and we detect a sensitive live mapping in *active*, then we know by construction that the two sensitive live mappings are used in the same instruction. Since we split all sensitive intervals into disjoint ones during live interval generation, only the necessity of multiple sensitive values at the same instruction can lead to an occurrence of a new sensitive live mapping while there is a sensitive live mapping in *active*. Consequently, we can assign the real register of the sensitive live mapping in *active* to the current live mapping, which removes the content of the active live mapping before another unrelated sensitive value will be written into any other real register. This modification will result in the highlighted lines 8 through 11 in Algorithm 1.

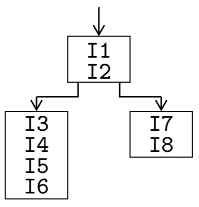
    Let us take a look at the sensitive virtual register V3 in Figure 4(b). When V3 starts at instruction 3, the sensitive virtual register V1 is active at the same instruction. If the provided function is correctly masked at the algorithmic level, we do not observe leakage between V1 and V3 because they need to be combined at the same instruction. Furthermore, V1 will no longer be in *active* after the live interval of V3 has started, because we have split the intervals into disjoint ranges. Therefore, we assign the real register of V1 to V3, ensuring that the sensitive value of V1 will be not present in the real register anymore, when an unrelated sensitive live mapping is active.
  - If either the virtual register of the next live mapping does not start as destination register, or there is no sensitive live mapping in *active*, we now check if there is a sensitive register in *available*. We have to assign the sensitive available register to the live mapping, because if we would write it to a different available register, we would violate Proposition 5.
  - Finally, if no sensitive register is in *available*, we assign an arbitrary available register.

Spilling of live mappings. If we have no available register, i.e., all registers are currently assigned to a live mapping, a mapping must to be spilled. Regardless of the sensitivity level of the next live mapping, we check if *active* contains sensitive live mappings. If the virtual register of the next live mapping is used as a destination at its first use point, and the endpoint of the active live mapping matches the first use point of the next live mapping, we assign the real register of the active live mapping to the new live mapping. This means that the live intervals do not overlap, except in one instruction where the destination register and one source register share the same real register. This modification is also highlighted in red in Algorithm 3. The argument why this satisfies Proposition 5 follows the same reasoning as in Section 4.4.1. Even if the next live interval is sensitive and we assign it the register of a sensitive live mapping from *active*, because both intervals are used in the same instruction, and assuming that the function is correctly masked, it does not lead to any leakage even if the intervals share the same register. If either there is no sensitive interval in *active* or the start point of the next interval differs from the end point of the mapping in *active*, we follow the standard algorithm where we spill the non-sensitive mapping with the longest live interval.

Note that we never spill a sensitive live mapping during the actual register allocation. During the generation of the live mappings, we already ensured that the intervals do not overlap. Further spillings of sensitive live mappings do not change this property, but only introduce load and store operations of sensitive values, which need to be handled to fulfill



(a) Live intervals after sensitivity annotation. V1, V3 and V5 are sensitive.



(b) Control-Flow Graph of the provided function.

Figure 5: Linearized description of the program can not ensure leakage free memory remnants.

Proposition 3 while creating further overhead. If at any point in time all registers are simultaneously assigned to live mappings, the maximum number of sensitive live mappings in *active* is two (two source registers of an instruction). Since in the ARMv6-M ISA the number of real registers is greater than two, we can be sure that there will always be a non-sensitive live mapping to spill.

4.4.2 Leakage-Free Memory Modifications

The original algorithm imposes no restrictions on the assignment of memory addresses to spilled live mappings. However, to be consistent with Proposition 2 and also Proposition 6, we concretize the memory assignments. For Proposition 2, it is important that each spilled sensitive live mapping gets its own stack location even if such mappings could share a memory location due to temporal separation. To comply with Proposition 6 during memory operations, we always place (sensitive) data at a word-aligned memory address m, regardless of its actual size. This ensures that each memory operation does not place multiple secret values in any of our modeled memory shadow registers, as according to the model, each memory shadow register always gets the word-size value at the specified address. This specification is especially important during the generation of live mappings, where sensitive mappings may be split and assigned a memory address.

To fulfill Proposition 3, we have to ensure that no subsequent loads (respectively stores) of two sensitive live mappings appear in our final output. Sensitive memory operations can only occur while a sensitive live mapping is active, either because the mapping is marked as spilled and thus has to be loaded from (respectively stored to) memory, or there is a non-spilling memory instruction during the interval. While there can be multiple sensitive memory operations within a sensitive live mapping, they do not cause memory remnant leakage because they belong to the same sensitive live mapping and no other sensitive live mapping is active during this interval. Since internal memory operations do not interact between two different sensitive live mappings, for Proposition 3 we consider only the first (respectively last) load and store operation of each sensitive live mapping.

As we consider functions with arbitrary control flow, the current description of the program in form of a linearized code and live mappings does not give us enough information to check all possible violations of Proposition 3. We indeed miss the control flow, which may jump to later or earlier instructions, effectively not following the program in a linear fashion. Let us take a closer look at Figure 5(a) with the sensitive live mappings V1, V3 and V5 and assume that all live mappings V1 through V5 are spilled, i.e., at the beginning of each interval the value is loaded from memory into a register and at the end stored from the register to memory. If we would have only this representation, we would assume that there is no memory remnant leakage observable, since the spilled non-sensitive mappings V2 and V4 separate each sensitive mapping. However, the actual control flow in Figure 5(b) reveals a direct control flow path between instructions `I2` and `I7`. Consequently, we have two subsequent loads and stores between the sensitive live mappings V1 and V5, which breaks Proposition 3. Therefore, we check the property of leakage-free memory remnant in each control flow path. We do this by transforming the linearized code back into the CFG. Afterwards, we perform a reverse Depth-First Search (DFS) for every sensitive live mapping. We start the search at the first load (respectively store) of the sensitive live mapping and check if there exists a path with a preceding sensitive load (respectively store) from a different sensitive live mapping. If such a path exists, we have found a violation of Proposition 3 and have to insert an overwrite instruction that clears the content of the corresponding memory shadow register. We insert the instruction at the position in the path that is closest to the beginning of the function, since it may be beneficial to the next sensitive live intervals in the CFG. Let us take another look at the example in Figure 5 and assume that only the sensitive live mappings V1, V3 and V5 are spilled. In our reverse DFS we would detect a violation of Proposition 3 between V1 and V3. We insert the clear instruction in the *first* basic block after `I2`. Now V5 also benefits from the clear instruction, as the path between V1 and V5 does not contain any subsequent sensitive memory operations, and we do not need to insert an additional clear between V1 and V5.

4.4.3 Leakage-Free Transition Modifications

An advantage of our changes for Proposition 5 is that we have already implicitly covered multiple potential pitfalls regarding harmful transitional leakage. If the algorithm finds registers in *available* that we can assign, only assigning a sensitive live mapping to a sensitive available register may cause a harmful transition. This constellation tells us that the expired sensitive live mapping and the new sensitive live mapping are unrelated, i.e., they have no overlap at an instruction, as otherwise the previous sensitive live mapping would still be in *active*. Hence, we cannot assume under this constellation that the combined leakage of the old and new sensitive values does not lead to leakage-free transitions.

Every other transition of registers between live mappings leads to a non-harmful transitional leakage between live mappings. This means, if we have a non-sensitive live mapping that gets assigned to a sensitive available register, we overwrite the sensitive content of the register with a non-sensitive value. Otherwise, we detect a sensitive live mapping in *active* while handling the next sensitive live mapping. In this case, we know that their live intervals are tangent and that they share the same instruction. Hence, their information jointly leaks due to their combined use at the same instruction. Again, if the function is correctly masked at the algorithmic level, this joint leakage does not reveal any information. Trivially, if we have only non-sensitive registers in *available*, no harmful transitional leakage occurs.

If all registers are assigned to live mappings at the same time, we need to spill a live mapping to memory. Here, our modification checks if there is a sensitive live mapping in *active*, whose end point overlaps with the start point of the next live mapping. If so, we assign the register from the active live mapping to the new live mapping. If the new live mapping is sensitive as well, we know that their intervals overlap at the same instruction. Hence, the same reasons given above for the non-harmful transitions in the non-spilling case, apply here as well. If the new live mapping is non-sensitive, we overwrite the sensitive value in the register with a non-sensitive value. Thus, there is no transitional leakage during spilling.

However, similar to that for Proposition 3, the linear description during the register allocation algorithm is not sufficient to account for all possible transitions for functions with arbitrary control flow. Referring back to Figure 5, let us assume that the sensitive mappings V1 and V5 are assigned the same register. Again, in its linearized form in Figure 5(a), we do not detect any potential violations of Proposition 1. However, the CFG in Figure 5(b) reveals a direct path between V1 and V5 leading to a transition between these two sensitive live mappings. In this case, analogous to handling Proposition 3, we need to insert an additional instruction that clears the content of the register and breaks the transition between the two sensitive live mappings. Similar to Section 4.4.2, we place the instruction at the position between two sensitive live mappings that is closest to the start of the function.

Algorithm 1 Linear Scan Register Allocation

```
Input: $D = \{d_0, \ldots, d_{|D|-1}\}$, where $d_i = \langle l, v, r, m, s, u \rangle$      ▷ list of live mappings of all virtual registers
Output: $D$                                        ▷ Updated live mappings with assigned registers
 1: $active \leftarrow \emptyset$
 2: $available \leftarrow \{(r_0, s_0), \ldots, (r_{|R|-1}, s_{|R|-1})\}$
 3: for $\forall d \in D$, in order of increasing start point of $d_l$ do
 4:     ExpireOldInterval($d$, $active$, $available$)
 5:     if $|active| = R$ then                                  ▷ cardinality of $active$
 6:         SpillAtInterval($d$, $active$)
 7:     else
 8:         if $(d_s = \texttt{SEC}) \land (\exists a \in active : a_s = \texttt{SEC}) \land (d_v$ is destination at $d_{l_x})$ then
 9:             $d_r \leftarrow a_r$
10:             $active \leftarrow active \setminus \{a\}$
11:         else
12:             if $\exists av \in available : av_s = \texttt{SEC}$ then
13:                 $d_r \leftarrow av_r$
14:                 $available \leftarrow available \setminus \{av\}$
15:                 $active \leftarrow active \cup \{d\}$
16:             else
17:                 $av \leftarrow$ arbitrary element from $available$
18:                 $d_r \leftarrow av$                          ▷ assign real register to mapping
19:                 $available \leftarrow available \setminus \{av\}$   ▷ remove assigned register
20:                 $active \leftarrow active \cup \{d\}$        ▷ add updated mapping to $active$
```

Algorithm 2 Expire Old Interval

```
Input: $d = \langle l, v, r, m, s, u \rangle$, $active$, $available$
Output: $active$, $available$                                  ▷ Updated sets
 1: for $\forall a \in active$, in order of increasing end point of $a_l$ do
 2:     if $a_{l_y} \geq d_{l_x}$ then return    ▷ mapping can be removed from $active$ if the end point in interval is smaller than start point of current mapping.
 3:     $active \leftarrow active \setminus \{a\}$              ▷ remove expired mapping from $active$
 4:     $available \leftarrow available \cup \{(a_r, a_s)\}$    ▷ add register from expired mapping
```

5 Case Studies

We thoroughly examine the security of our presented scheme and conduct a two-stage evaluation of four different masked software implementations. The first stage consist of a simulation-based evaluation of the generated assembly code with PROLEAD_SW to ensure that the output of PoMMES is in line with the CPU-independent leakage model. We then show that our implementations provide practical security by collecting power traces from an ARM Cortex-M0+ processor and performing first-order and second-order t-test evaluations using 100 000 measurements. Our case studies aim to demonstrate the versatility of PoMMES and consist of four implementations. The first three case studies cover

Algorithm 3 Spill At Interval

```
Input: $d = \langle l, v, r, m, s, u\rangle$, $active$
Output: $active$                                                    ▷ Updated set
 1: if $(\exists a \in active : a_s = \texttt{SEC}) \wedge (d_{l_x} = a_{l_y}) \wedge (d_v$ is destination at $d_{l_x})$ then
 2:     $d_r \leftarrow a_r$
 3:     $active \leftarrow active \setminus \{a\}$
 4:     $active \leftarrow active \cup \{d\}$
 5: else
 6:     $s \leftarrow$ last live mapping in $active$
 7:     if $s_{l_y} > d_{l_y}$ then                      ▷ first case: spill mapping from active
 8:         $d_r \leftarrow s_r$                         ▷ hand over register from spilled mapping
 9:         $s_m \leftarrow$ new stack location          ▷ spilled mapping gets memory location
10:         $active \leftarrow active \setminus \{s\}$   ▷ remove spilled mapping
11:         $active \leftarrow active \cup \{d\}$        ▷ add current interval
12:     else                                             ▷ second case: spill current mapping
13:         $d_m \leftarrow$ new stack location
```

gadgets that implement a refresh of shares, a multiplication in $\mathbb{F}_{2^n}$, and a bitwise AND operation. The last case study is a more complex implementation, i.e., a masked AES Sbox. For each case study, we generated both first- and second-order implementations.

We choose these case studies as the provided functions are commonly found in masked software, either in symmetric cryptography or Post-Quantum Cryptography (PQC), and cover different properties. We show that `PoMMES` is able to handle different masking domains as it is able to generate secure software for the multiplication gadget which uses arithmetic masking, while the other examples use Boolean masking. Second, we can handle different security notions, such as Probe Isolating Non-Interference (PINI) for the multiplication and refresh gadget, and Strong Non-Interference (SNI) for the AND gadget (see Section 2.3). `PoMMES` is able to not only implement linear functions securely, e.g., a refresh gadget, but also to handle non-linear functions, which are more difficult to implement securely as different shares need to be combined. With our generated masked AES Sbox we highlight that `PoMMES` is also able to handle more complex implementations and can uphold the security across function borders, since our AES Sbox implementation is a concatenation of different functions.

5.1 Setup

5.1.1 PROLEAD_SW

Initially, we start by verifying that the assembly code generated by `PoMMES` follows the general leakage model represented in [ZMM23]. We adapt `PROLEAD_SW` to handle the refined memory remnant effect as explained in Section 4.1. We tested all first-order designs and evaluated them with 10 000 simulations in a fixed vs. random setting. For the second-order designs, we were not able to perform a bivariate evaluation with `PROLEAD_SW` due to the large size of the case studies and as a result the exponential growth of probing sets, which made the analysis infeasible in terms of runtime and memory consumption. The randomness required by the gadgets is sampled from the internal randomness source of `PROLEAD_SW`. We ran the simulations on a standard computer with an Intel i7 CPU at 1.8 GHz, 32 GB of RAM, and a maximum of 6 threads. We choose `PROLEAD_SW`'s compact mode for its statistical evaluation to identify leakage for large probing sets with fewer number of simulations. The confidence threshold for the false-positive probability was set to 10^{-5}.

5.1.2 Power Measurements

We conducted our practical measurements on an STM32L081CB micro-controller with an ARM Cortex-M0+ processor that uses an ARMv6-M architecture. The supply voltage is set to 1.8 V and the clock for all implementations was set to 8 MHz. For the acquisition of power traces, we used a PicoScope 5000 series oscilloscope with a sampling rate set to 250 MS/s. We performed a non-specific t-test in a fixed vs. random setting. The required randomness during the execution of the gadgets was sampled on the PC and sent to the board via UART. For the t-test calculation, we used SCAlib [CB23]. For each case study listed below we instantiated the gadget with 2 (respectively 3) shares to generate the first-order (respectively second-order) masked implementations. Each of these implementations was generated separately with `PoMMES` by adjusting the number of shares in the high-level C code, but without needing to further modify the resulting assembly code in any form. The power traces for each case study were measured one after each other but independently, i.e., for each first-order and second-order designs we collected 100 000 traces.

5.2 Results

5.2.1 Locality Refresh Gadget

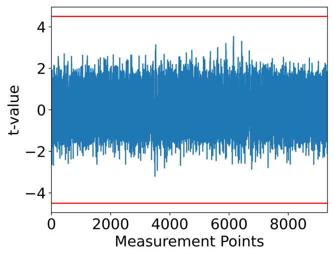
The first gadget, that we evaluate, is a mask refreshing scheme presented by Coron et al. [CGZ20] which is proven to be secure under the PINI [CS20] notion. Refreshing is an essential part of masking schemes as it allows to reduce the number of required shares, which in turn reduces runtime and code size. Furthermore, the mask refreshing does not only act as a methodology to lower the number of shares [BBD$^+$16], but certain masking schemes [RP10a] rely on mask refreshing to guarantee the security. The simulation-based results with `PROLEAD_SW` did not reveal leakage with all effects enabled. Furthermore, the t-test results of the assembly code generated by `PoMMES` is visualized in Figure 6. The first-order results in Figure 6(a) stay well below the 4.5 threshold. For the second-order refresh gadget, we performed a first-order and a second-order multivariate t-test shown in Figure 6(b) and Figure 6(c). In both cases, `PoMMES` was able to generate assembly code which upholds the desired security level.

5.2.2 ISW-AND Gadget

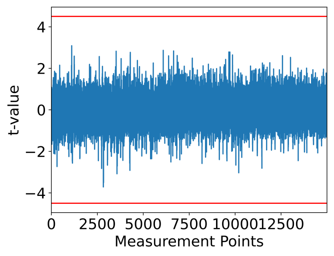
As the second case study, we use the ISW-AND gadget presented in [ISW03]. It is proven secure in the d-probing model and operates – like the refresh gadget – over Boolean masking. Such side-channel secure non-linear operations are commonly found in the confusion layer of masked symmetric ciphers and bitsliced implementations. This gadget is significantly more complex than the refresh gadget, as it has to combine shares from different share domains. We show that `PoMMES` is able to generate assembly code that upholds the desired security level in practice, even in such complicated situations. Similar to the refresh gadget, all t-values stay below the threshold as depicted in Figure 7, showing the expected results for both first- and second-order implementations. The experimental results are backed up by `PROLEAD_SW` for the ISW-AND gadget as no leakage was detected within 10 000 simulations and all effects enabled.

5.2.3 PINI Multiplication Gadget

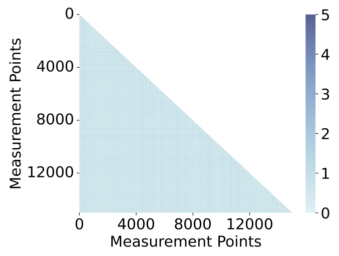
The third case study utilizes the PINI multiplication gadget presented by Cassiers [Cas22]. This gadget can operate on arithmetic shares, contrary to the first two gadgets. Such arithmetic gadgets increasingly absorb the attention of the scientific community with regard to masking PQC schemes, where many operations are performed arithmetically.



(a) First-order t-test of the first-order implementation

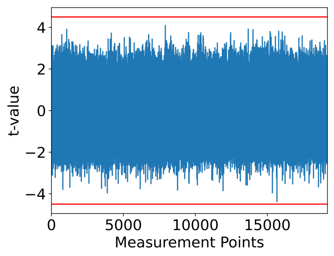


(b) First-order t-test of the second-order implementation

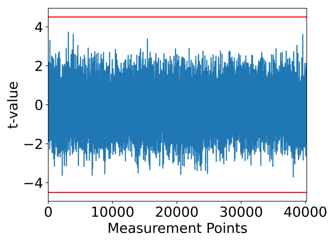


(c) Second-order t-test of the second-order implementation

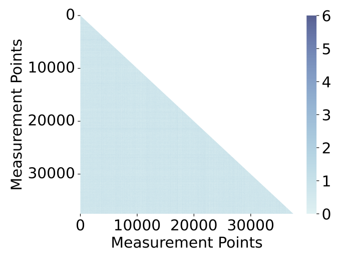
Figure 6: Fixed vs. random t-test of `PoMMES` refresh gadget using 100 000 traces.



(a) First-order t-test of the first-order implementation

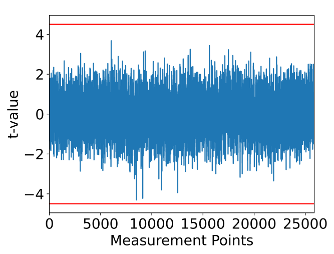


(b) First-order t-test of the second-order implementation

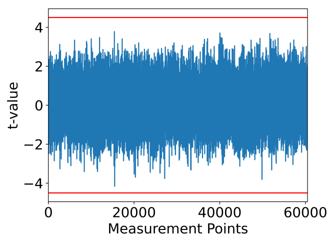


(c) Second-order t-test of the second-order implementation

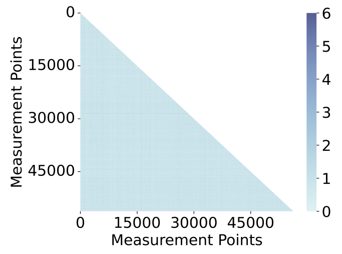
Figure 7: Fixed vs. random t-test of `PoMMES` ISW-AND gadget using 100 000 traces.



(a) First-order t-test on the first-order implementation



(b) First-order t-test of the second-order implementation



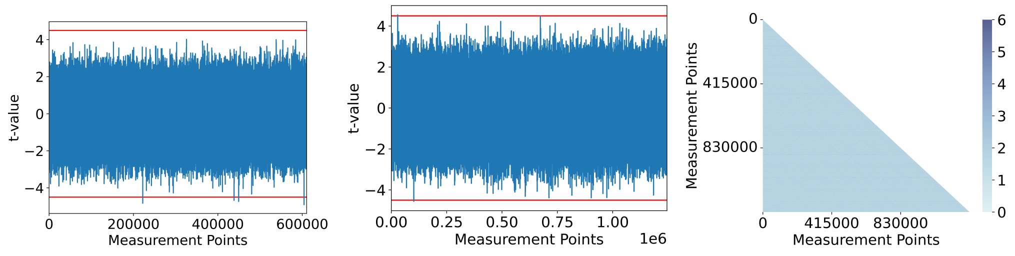
(c) Second-order t-test of the second-order implementation

Figure 8: Fixed vs. random t-test of `PoMMES` PINI multiplication gadget using 100 000 traces.

Our intention is to show that `PoMMES` is widely applicable for different scenarios and different masking techniques. The corresponding first-order as well as multivariate second-order t-test results are depicted in Figure 8. Similar to the previous case studies, all t-values stay within the 4.5 threshold and `PROLEAD_SW` confirms the security within the refined CPU-independent leakage model or the first-order design as no leakage is found.

5.2.4 Rivain-Prouff AES Sbox

The Rivain-Prouff AES Sbox [RP10b] serves as our fourth case study. The authors provided an algorithm to perform masked AES Sbox computations at any security order and have proven the security under the d-probing model. They expressed the inversion of an element in $\mathbb{F}_{2^8}$ as the exponentiation to the power of 254. This exponentiation



(a) First-order t-test of the first-order implementation (b) First-order t-test of the second-order implementation (c) Second-order t-test of the second-order implementation

Figure 9: Fixed vs. random t-test of PoMMES Rivain-Prouff AES Sbox using 100 000 traces.

can be efficiently computed with common exponentiation techniques, such as the square-and-multiply algorithm. To secure the exponentiation the authors provided methods to perform a masked squaring operation, which operates on each share separately, and a masked field multiplication, which is a generalization of the ISW-AND scheme for any finite field $\mathbb{F}_{2^n}$. Furthermore, the exponentiation includes refresh of masks as the inputs to the multiplications need to be mutually independent.

The concrete implementation of this scheme is taken from GitHub[4]. The implementation of this Sbox consists of five different functions, where two functions are responsible for squaring (respectively two functions for multiplication), and the fifth function acts as a wrapper and performs the exponentiation. We created all of these functions with PoMMES. Initially, we perform the security analysis with PROLEAD_SW under all micro-architectural effects and 10 000 simulations. Under the consideration of all available micro-architectural effects in PROLEAD_SW our implementation holds the security in the CPU-independent leakage model. Our experimental t-test results in Figure 9 further underline the simulation results and highlight that PoMMES is able to generate secure software even for larger implementations and also for a concatenation of multiple functions.

5.3 Comparison

To evaluate the performance of PoMMES we compare our case studies presented in Section 5.2.1 to Section 5.2.4 with different optimization levels of the off-the-shelf GCC compiler and handcrafted assembly implementations under four different metrics. These metrics include the code size, cycle count, instruction count, as well as the compile time, and the t-test result. The code size represents the number of bytes within the binary that belong to each case study. To get the cycle counts we used the SysTick timer, which is a 24-bit decrement counter, as our processor does not include a dedicated register that can accurately give the cycle count for each function. We have synchronized the SysTick timer with the processor clock and read the value of the timer before and after the execution of the function under test. Based on these values, we compute the number of cycles. The instruction count represents the number of instructions executed, and the compile time describes the time it took in seconds to generate the output of each case study.

For the results with the GCC compiler, we used the same C functions that we passed as input to PoMMES and compiled them with the arm-none-eabi-gcc compiler. The handcrafted implementations were created in an iterative process, where we implemented the behavior of each gadget as efficient as possible and performed power measurements. If the results indicated leakage, we identified the problematic instruction based on the leaking measurement points. We then made educated guesses how the assembly instructions should

[4]https:https://github.com/coron/htable/blob/master/src/aes_rp.c

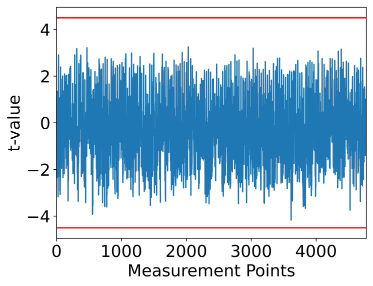
be modified and re-evaluated the implementation. We repeated this until no leakage was detectable with 100 000 traces. Note that this approach was non-trivial and required several iterations, despite having a good understanding of the underlying platform. The benefit of this approach is an overall improvement in code size and cycle count compared to all other implementations, including those that were compiled with `GCC-O3`. However, there is no guarantee that these handcrafted assembly implementations generated by the iterative approach preserve the same practical security when run on the same processor produced by another vendor.

Starting with security evaluations which we limited only to the first-order designs, none of these implementations (including the handcrafted ones) were identified as secure by `PROLEAD_SW`. We further provided the corresponding t-test results in Figure 10 to Figure 13. As an interesting point, no leakage is observed by the t-test conducted on the refresh gadget compiled with `GCC-O0`. This elucidates two things. While we do observe leakage with `PROLEAD_SW`, we would like to stress that the CPU-independent leakage model of `PROLEAD_SW` over-approximates in the sense that leakage in the model does not inevitably lead to leakage in practice, i.e., worst-case scenario. Second, the refresh gadget is extremely simple. It consists of mainly two XOR operations and a few memory operations. This reduces potential pitfalls for the `GCC` compiler, since only a few instructions need to be compiled. In this case, the compiler was able to generate *by chance* a practically secure assembly for this gadget. Note that it is not guaranteed that this behavior holds for other functions, as we can see this for the other case studies, for slight modifications to the same function, or even for the same function under different `GCC` versions.

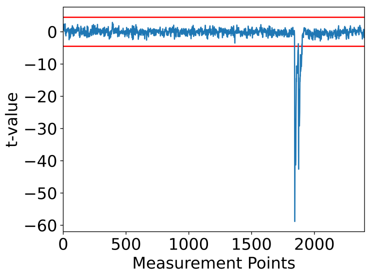
The comparative performance results are bundled in Table 1. As expected, we observed different overheads for different case studies and different optimization levels. While optimizations are beneficial in terms of cycle count and instruction count, they fully ignore the security aspect, which can lead to the unintended leakage, as can be seen in Figure 10 to Figure 13. As `PoMMES` does *not* yet implement any optimization, we focus our comparisons with the cases that were compiled with no optimization, i.e., `GCC-O0`. Here, the overhead in terms of cycle count for the gadgets ranges from a factor of 1.34 for the first-order PINI multiplication to 2.09 for the second-order refresh gadget. The cycle count overhead for the Rivain-Prouff Sbox ranges from 3.38 for the first-order implementation to 3.5 for the second-order implementation. Similarly, the code size between `PoMMES` and `GCC-O0` is in the range of 1.46 for the second-order multiplication gadget and 3.41 for the first-order refresh gadget. The overhead in terms of code size for the Rivain-Prouff AES Sbox is around 2.75. Although `PoMMES` has to execute additional steps to generate assembly code which is in-line with the CPU-independent leakage model, we do not see significant overhead in the compile time, i.e., the time it takes for `PoMMES` to produce the output, compared to the GCC compiler. For all gadget implementations the compile time is close to the GCC compile time. While it does take slightly longer for `PoMMES` to generate the assembly code for the AES Sbox, the overhead is imperceptible in real-time.

5.4 Security vs. Efficiency

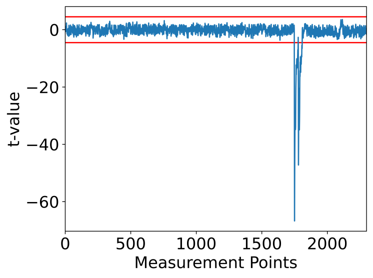
After discussing the performance differences between `PoMMES`, `GCC`, and handcrafted assembly implementations, we now delve into the reasons for the performance penalty and, more generally, the trade-off between security and efficiency. We identify two main sources that explain the efficiency gap. Naturally, the runtime is negatively affected by the conservativeness of our leakage model and the constraints derived from it. `PoMMES` is based on the assumption that software developers do not have detailed information about the CPU's netlist. This is especially true for ARM processors, which are the focus of this paper. Although ARM provides access to netlists of specific processors to academic



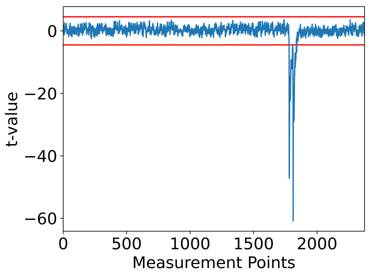
(a) with `GCC-O0`



(b) with `GCC-O1`

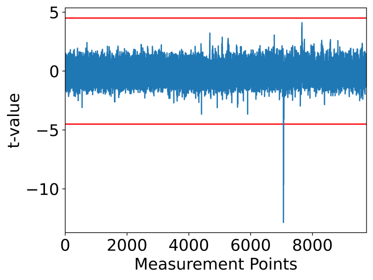


(c) with `GCC-O2`

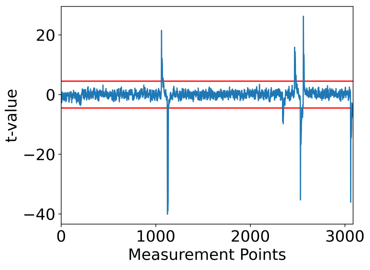


(d) with `GCC-O3`

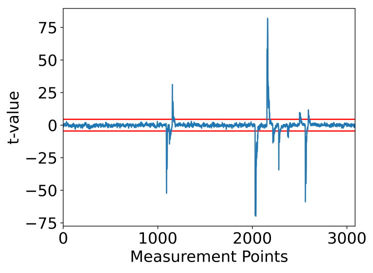
Figure 10: First-order fixes vs. random t-test of refresh gadget compiled with four different optimization levels using 100 000 traces.



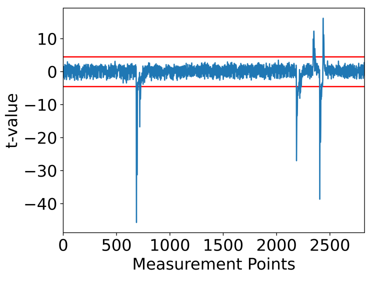
(a) with `GCC-O0`



(b) with `GCC-O1`

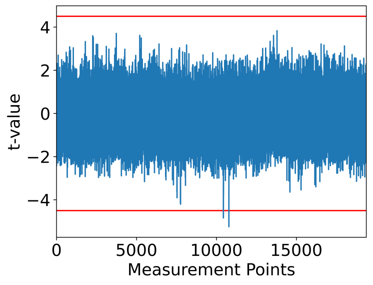


(c) with `GCC-O2`

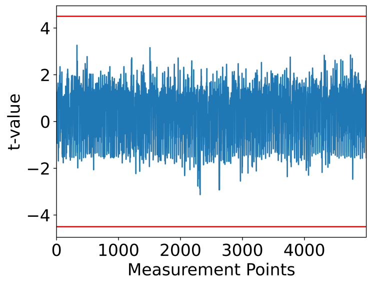


(d) with `GCC-O3`

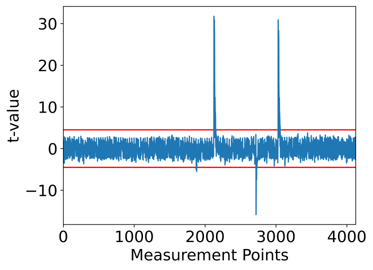
Figure 11: First-order fixed vs. random t-test of ISW-AND gadget compiled with four different optimization levels using 100 000 traces.



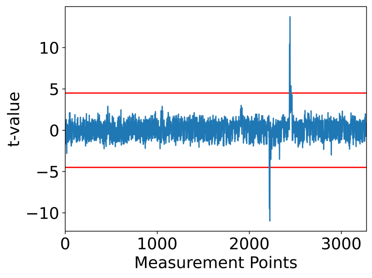
(a) with `GCC-O0`



(b) with `GCC-O1`

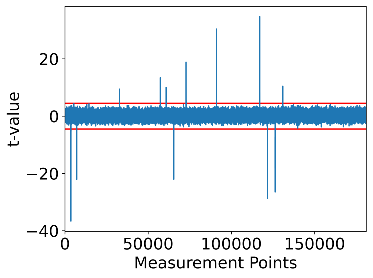


(c) with `GCC-O2`

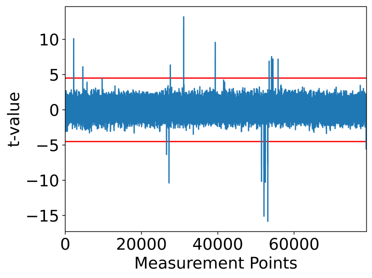


(d) with `GCC-O3`

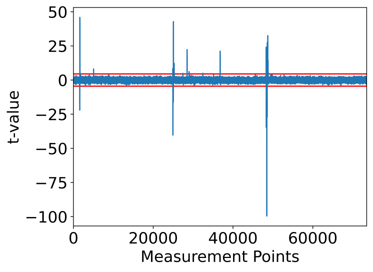
Figure 12: First-order fixed vs. random t-test of PINI multiplication gadget compiled with four different optimization levels using 100 000 traces.



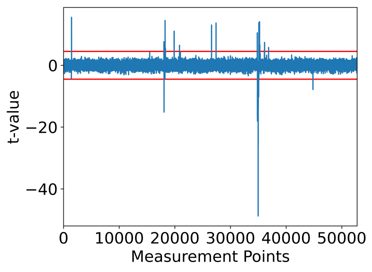
(a) with `GCC-O0`



(b) with `GCC-O1`



(c) with `GCC-O2`



(d) with `GCC-O3`

Figure 13: First-order fixed vs. random t-test of Rivain-Prouff AES Sbox compiled with four different optimization levels using 100 000 traces.

Table 1: Comparison of our case studies regarding their code size in bytes, cycle counts, instruction count and compile time in seconds between PoMMES, GCC with four different optimization levels and handcrafted implementations.

Tool	**Design**	**Code Size**		**Cycle Count**		**Instr. Count**		**Compile Time**	
		$d = 1$	$d = 2$	$d = 1$	$d = 2$	$d = 1$	$d = 2$	$d = 1$	$d = 2$
PoMMES	Refresh	348	356	282	464	178	306	0.02	0.02
	ISW-AND	632	632	598	1270	405	927	0.03	0.03
	PINI Mult.	662	674	812	1918	579	1404	0.03	0.03
	R-P Sbox	2742	2742	19594	39839	13122	26715	0.11	0.11
GCC-O0	Refresh	102	138	136	222	71	122	0.01	0.01
	ISW-AND	220	258	296	614	172	354	0.01	0.01
	PINI Mult.	436	460	603	1384	419	1089	0.02	0.02
	R-P Sbox	996	996	5791	11373	3786	7633	0.03	0.03
GCC-O1	Refresh	44	76	59	93	27	46	0.01	0.02
	ISW-AND	86	152	85	270	42	166	0.02	0.02
	PINI Mult.	158	224	141	493	85	355	0.02	0.03
	R-P Sbox	720	654	2515	4923	1564	3007	0.04	0.04
GCC-O2	Refresh	44	76	57	91	27	46	0.02	0.02
	ISW-AND	88	156	86	271	43	173	0.02	0.03
	PINI Mult.	132	200	115	407	71	293	0.03	0.03
	R-P Sbox	702	690	2339	4833	1447	2657	0.06	0.05
GCC-O3	Refresh	44	76	60	94	27	46	0.02	0.02
	ISW-AND	88	112	84	239	43	158	0.02	0.03
	PINI Mult.	96	228	87	225	46	130	0.03	0.03
	R-P Sbox	580	1210	1648	3962	1229	3119	0.06	0.06
Handcrafted	Refresh	24	-	41	-	12	-	0.01	-
	ISW-AND	52	-	64	-	26	-	0.01	-
	PINI Mult.	60	-	68	-	30	-	0.01	-

institutions through the ARM Academic Access[5], the netlists are generally not open source or accessible to arbitrary organizations or software developer. Therefore, to be independent of the underlying netlist while accounting for multiple micro-architectural leakage sources, we require such a conservative model at the expense of performance loss. It has been shown in several publications [MPW22, GPM21, PV17] that such micro-architectures are exploitable and affect a wide variety of processors. Arguably, a more fine-grained approach would be beneficial to reduce the overhead imposed. One may think of focusing on the leakage points and patch only those positions, similar to what we did for our comparative handcrafted implementations. This approach may be feasible for small functions such as the gadgets in our case studies, since they show only a few leakage points. However, for larger implementations such as the masked AES Sbox, we see that leakage occurs throughout the whole power trace, and identifying all of these leakages in the source code quickly becomes a tedious task. Furthermore, we did not follow a fine-grained approach for three reasons.

[5]https://www.arm.com/resources/research/enablement/academic-access

- First, if we identify leakage and decide to fix it locally, i.e., to focus solely on the leaking instructions, we may inadvertently introduce new leaks, depending on how we modify the code to remove the original leak.
- The second reason is that fixing the leakage based on the observed leakage points is tailored to the specific processor design and its realization. While this customization helps to improve the performance, it prohibits porting the code to another platform as a different processor may exhibit different leakage behavior. The strength of our model is that it removes leakage independent of a specific processor and focuses on a generalized CPU model.
- Third and most importantly, a more fine-grained approach requires either the processor netlist or a gray-box model. While we already discussed the restrictive access to the ARM netlists, the gray-box approaches rely heavily on the accuracy of the underlying setup and model to effectively identify the sources of leakage. Micro-architectural leakages can be very subtle, non-intuitive, and spread across multiple instructions [ZMM23]. A gray-box model must be sophisticated enough to reliably handle such scenarios.

The second major aspect contributing to the performance gap is the difference in complexity between `GCC` and `PoMMES`. Currently, our tool focuses mainly on the register allocation step during compilation. We do only incorporate trivial instruction selection and instruction reordering, and have not yet implemented any compiler optimization steps. Hence, `PoMMES` suffers from performance penalties that more sophisticated compilers like `GCC` do not have. `PoMMES` uses linear scan as its register allocation algorithm. It is a rather simple algorithm, and modern compilers rely on more complex algorithms that provide better code quality. For example, `GCC` uses a combination of local register allocation and graph coloring register allocation, which results in a longer analysis runtime but better performance. In addition, optimization passes within compilers are crucial for the performance of software. Under common optimization flags such as `-O1`, `-O2` or `-O3`, the compiler groups several optimization techniques together[6]. Certain optimization techniques under these flags can greatly reduce the security of the implementation for example by reordering security-critical instructions or inlining and merging functions. The negative impact of optimizations on the security is shown in the t-test results of the implementations generated by the `GCC` compiler under different optimization levels (Figure 10 to Figure 13). The more aggressive the level of optimization is, the more opportunities for leakage are created. While it is not certain that a higher level of optimization certainly introduces leakage, off-the-shelf compilers can never guarantee that they do *not* introduce leakage, since their main focus is performance, not Side-channel Analysis (SCA) security. It is difficult to predict which particular optimization pass or combination of optimizations has a negative impact on security. We show in Figure 13 that the masked AES Sbox has significant leakage throughout the function, even without any optimization, i.e., `-O0`. We argue that the primary goal of masked software is to be secure against SCA. While `PoMMES` undoubtedly cannot currently match the efficiency of commercial compilers, it is the first step toward providing security in the face of several harmful micro-architectural effects.

5.5 Limitations and Future Work

While we think `PoMMES` helps software developers to avoid micro-architectural leakages in masked software, there is still room for improvement in both `PoMMES` and the broader research field. Contrary to compilers such as `CompCert` [Ler09] or `Jasmin` [ABB+17], `PoMMES` is not formally verified. Having a formal proof of `PoMMES` and the underlying algorithm would be helpful to increase the trust in `PoMMES`.

[6]`https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

While `PoMMES` is able to handle arbitrary control-flows, i.e., (nested) loops, recursions, function calls, and conditions, `PoMMES` is currently not able to handle the full C functionality, e.g., C structs are not supported at the moment. This restriction is only due to the current limitations of the utilized parser.

It would be interesting to see how other register allocation algorithms behave under the restrictions we presented in this paper. While other register allocation algorithms have a higher runtime complexity, they tend to generate better results in term of code quality. Therefore, we leave the comparison of performance and overhead between different SCA-aware register allocation approaches as an interesting line of research for the future.

Furthermore, there are different optimization ideas regarding our presented technique. Our approach, as well as other works, categorize inputs and variables as either *secret* or *public*. Thus, any combination of secret variables leads to a violation. It would be beneficial to have a more fine-grained distinction of sensitive values. For instance, let us consider two sensitive variables a and b. Each of these variables has two shares $a = a_0 \oplus a_1$ and $b = b_0 \oplus b_1$. Currently, if we detect for example a transition between a_0 and b_0 in a register, we consider this as harmful since both are marked as sensitive, even though their combination does not necessarily provide the attacker with useful information if a and b are independently shared. We believe that classifying variables beyond the two groups of *public* and *secret*, e.g., their share domain, would lead to a significant reduction in overhead. We expect that tackling sensitivity at the information analysis step, by allowing different levels of sensitivity for the same variable throughout the execution flow, would have a positive impact on runtime, but possibly complicate the register allocation process even further.

`PoMMES` is currently limited to handling ARM processors, specifically the `GIMPLE` IR, and generating assembly code that is inline with the ARMv6-M ISA. However, our technique in its general form is independent of specific compilers, i.e. `PoMMES` or related tools can be extended to handle different IRs and generate code for different processor models.

6 Conclusions

Correctly applying algorithmic-level masking to programs written in high-level programming languages hardly provides the expected security in practice, since micro-architectural effects are not taken into account during machine code generation. To counteract the security reduction of such effects we introduced `PoMMES`, which – given a masked C code and security annotations of the inputs – transforms the program into ARMv6-M assembly code that is secure under a refined version of the CPU-independent leakage model. We particularly focused on the register allocation algorithm used during compilation. This last step of the compilation process is responsible for assigning an unlimited number of virtual registers to a limited number of real registers, and also decides which values reside in memory and at which location. Based on the ability of an attacker considered in our adversary model to place various transition- and glitch-extended probes inside the processor, we have adjusted and adapted the linear scan register allocation algorithm. To the best of our knowledge, we are the first to propose an SCA-aware register allocation algorithm covering micro-architectural effects. We conducted a two-stage validation of our approach with four different masked software case studies. First, we verified the result of our approach, i.e., the assembly code generated by `PoMMES`, using `PROLEAD_SW` to evaluate its security based on the CPU-independent leakage model covering several micro-architectural effects. In a subsequent step, we performed an experimental analysis using power consumption measurements of the same software running on actual ARM processors, confirming the expected security levels in practice.

Acknowledgments

The work described in this paper has been supported in part by the German Research Foundation (DFG) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, and by the Federal Ministry of Education and Research of Germany through the Project DevToSCA (16KIS1603).

References

[ABB$^+$17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1807–1823, 2017.

[ABB$^+$21] Arnold Abromeit, Florian Bache, Leon A. Becker, Marc Gourjon, Tim Güneysu, Sabrina Jorn, Amir Moradi, Maximilian Orlt, and Falk Schellenberg. Automated masking of software implementations on industrial microcontrollers. In *DATE 2021*, 2021.

[BBC$^+$19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskverif: Automated verification of higher-order masking in presence of physical defaults. In *ESORICS 2019*, volume 11735 of *LNCS*, pages 300–318. Springer, 2019.

[BBD$^+$15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In *EUROCRYPT 2015*, volume 9056 of *LNCS*, pages 457–485. Springer, 2015.

[BBD$^+$16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In *CCS 2016*, pages 116–129. ACM, 2016.

[BC22] Olivier Bronchain and Gaëtan Cassiers. Bitslicing arithmetic/boolean masking conversions for fun and profit with application to lattice-based kems. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):553–588, 2022.

[BCP$^+$20] Sonia Belaïd, Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Abdul Rahman Taleb. Random Probing Security: Verification, Composition, Expansion and New Constructions. In *CRYPTO 2020*, volume 12170 of *LNCS*, 2020.

[BGG$^+$21] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. Masking in fine-grained leakage models: Construction, implementation and verification. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):189–228, 2021.

[BGK04] Johannes Blömer, Jorge Guajardo, and Volker Krummel. Provably secure masking of AES. In Helena Handschuh and M. Anwar Hasan, editors, *Selected Areas in Cryptography, 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10, 2004, Revised Selected Papers*, volume 3357 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2004.

[BMRT22] Sonia Belaïd, Darius Mercadier, Matthieu Rivain, and Abdul Rahman Taleb. Ironmask: Versatile verification of masking security. In *IEEE S & P 2022*, pages 142–160. IEEE, 2022.

[BNPS05] Manuel Barbosa, Richard Noad, Daniel Page, and Nigel P. Smart. First steps toward a cryptography-aware language and compiler. *IACR Cryptol. ePrint Arch.*, page 160, 2005.

[BRB$^+$11] Ali Galip Bayrak, Francesco Regazzoni, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. A first step towards automatic application of power analysis countermeasures. In Leon Stok, Nikil D. Dutt, and Soha Hassoun, editors, *Proceedings of the 48th Design Automation Conference, DAC 2011, San Diego, California, USA, June 5-10, 2011*, pages 230–235. ACM, 2011.

[BWG$^+$22] Arthur Beckers, Lennert Wouters, Benedikt Gierlichs, Bart Preneel, and Ingrid Verbauwhede. Provable secure software masking in the real-world. In *COSADE 2022*, volume 13211 of *LNCS*, pages 215–235. Springer, 2022.

[Cas22] Gaëtan Cassiers. *Composable and efficient masking schemes for side-channel secure implementations.* PhD thesis, Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2022.

[CB23] Gaëtan Cassiers and Olivier Bronchain. Scalib: A side-channel analysis library. *Journal of Open Source Software*, 8(86):5196, 2023.

[CGLS21] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware private circuits: From trivial composition to full verification. *IEEE Trans. Computers*, 70(10):1677–1690, 2021.

[CGZ20] Jean-Sébastien Coron, Aurélien Greuet, and Rina Zeitoun. Side-channel masking with pseudo-random generator. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part III*, volume 12107 of *Lecture Notes in Computer Science*, pages 342–375. Springer, 2020.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO 1999*, volume 1666 of *LNCS*, pages 398–412. Springer, 1999.

[Cor14] Jean-Sébastien Coron. Higher order masking of look-up tables. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 441–458. Springer, 2014.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. Trivially and Efficiently Composing Masked Gadgets With Probe Isolating Non-Interference. *IEEE Trans. Information Forensics and Security*, 15:2542–2555, 2020.

[CS21] Gaëtan Cassiers and François-Xavier Standaert. Provably secure hardware masking in the transition- and glitch-robust probing model: Better safe than sorry. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):136–158, 2021.

[EW14] Hassan Eldib and Chao Wang. Synthesis of masking countermeasures against side channel attacks. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 2014.

[FBR$^+$22] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked accelerators and instruction set extensions for post-quantum cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022.

[FGP$^+$18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):89–120, 2018.

[GD23] John Gaspoz and Siemen Dhooghe. Threshold implementations in software: Micro-architectural leakages in algorithms. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(2):155–179, 2023.

[GHP$^+$21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco: Co-design and co-verification of masked software implementations on cpus. In *USENIX Security Symposium*, pages 1469–1468. USENIX Association, 2021.

[GIB18] Hannes Groß, Rinat Iusupov, and Roderick Bloem. Generic low-latency masking in hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):1–21, 2018.

[GJJR11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, 2011.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In *TIS@CCS 2016*, page 3. ACM, 2016.

[GMK17] Hannes Groß, Stefan Mangard, and Thomas Korak. An efficient side-channel protected AES implementation with arbitrary protection order. In *CT-RSA 2017*, volume 10159 of *LNCS*, pages 95–112. Springer, 2017.

[GMP$^+$20] Johann Großschädl, Ben Marshall, Dan Page, Thinh Hung Pham, and Francesco Regazzoni. An instruction set extension to support software-based masking. *IACR Cryptol. ePrint Arch.*, 2020.

[GOP21] Si Gao, Elisabeth Oswald, and Dan Page. Reverse engineering the micro-architectural leakage features of a commercial processor. *IACR Cryptol. ePrint Arch.*, page 794, 2021.

[GPM21] Barbara Gigerl, Robert Primas, and Stefan Mangard. Secure and efficient software masking on superscalar pipelined processors. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part II*, volume 13091 of *Lecture Notes in Computer Science*, pages 3–32. Springer, 2021.

[GSM$^+$19] Hannes Groß, Ko Stoffelen, Lauren De Meyer, Martin Krenn, and Stefan Mangard. First-order masking with only two random bits. In Begül Bilgin, Svetla Petkova-Nikova, and Vincent Rijmen, editors, *Proceedings of ACM Workshop on Theory of Implementation Security, TIS@CCS 2019, London, UK, November 11, 2019*, pages 10–23. ACM, 2019.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, 2003.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In Patrick C. Fischer and Jeffrey D. Ullman, editors, *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, pages 194–206. ACM Press, 1973.

[KM22] David Knichel and Amir Moradi. Low-latency hardware private circuits. In *CCS 2022*, pages 1799–1812. ACM, 2022.

[KMMS22] David Knichel, Amir Moradi, Nicolai Müller, and Pascal Sasdrich. Automated generation of masked hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):589–629, 2022.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In *ASIACRYPT 2020*, volume 12491 of *LNCS*, pages 787–816. Springer, 2020.

[KSM22] David Knichel, Pascal Sasdrich, and Amir Moradi. Generic hardware private circuits: Towards automated generation of composable secure gadgets. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):323–344, 2022.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[MM22] Nicolai Müller and Amir Moradi. PROLEAD A probing-based hardware leakage detection tool. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):311–348, 2022.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards.* Springer, 2007.

[MOPT12] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. Compiler assisted masking. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 58–75, 2012.

[MOW17] David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. In *USENIX 2017*, pages 199–216. USENIX Association, 2017.

[MPW22] Ben Marshall, Dan Page, and James Webb. MIRACLE: micro-architectural leakage evaluation A study of micro-architectural power leakage across many devices. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):175–220, 2022.

[MPZ22] Maria Chiara Molteni, Jürgen Pulkus, and Vittorio Zaccaria. On robust strong-non-interferent low-latency multiplications. *IET Inf. Secur.*, 16(2):127–132, 2022.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings*, volume 4307 of *Lecture Notes in Computer Science*, pages 529–545. Springer, 2006.

[NRS11] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *J. Cryptol.*, 24(2):292–321, 2011.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 1999.

[PV17] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the gap: Towards secure 1st-order masking in software. In Sylvain Guilley, editor, *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*, volume 10348 of *Lecture Notes in Computer Science*, pages 282–297. Springer, 2017.

[RP10a] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In *CHES 2010*, volume 6225 of *LNCS*, pages 413–427. Springer, 2010.

[RP10b] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of aes. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 413–427. Springer, 2010.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22, 1979.

[SSB$^+$21] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. In *NDSS 2021*. The Internet Society, 2021. https://www.ndss-symposium.org/ndss-paper/rosita-towards-automatic-elimination-of-power-analysis-leakage-in-ciphers/.

[ZMM23] Jannik Zeitschner, Nicolai Müller, and Amir Moradi. Prolead_sw - probing-based software leakage detection for ARM binaries. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(3):391–421, 2023.