

Hardware-Level Mitigation of Micro-Architectural Leakages in Embedded Processors

Niels Wieser^{}, Nicolai Müller^{} and Amir Moradi^{}

Technische Universität Darmstadt, Darmstadt, Germany
`{niels.wieser,nicolai.mueller1,amir.moradi}@tu-darmstadt.de`

Abstract. Cryptographic software, with its inherent flexibility and adaptability, has become essential for meeting today’s rapidly evolving security requirements, despite offering lower performance than dedicated hardware accelerators. However, Side-Channel Analysis (SCA) complicates its secure execution on physical hardware, making robust protection a central challenge. While the masking countermeasure offers provable security guarantees in theory, its effectiveness is often undermined by the processor’s micro-architecture. In response, formal adversary models have been adapted to account for micro-architectural leakage under generic (but over-conservative) assumptions on the hardware. Consequently, achieving security under such a generic model requires compilers to produce less performant software. To bridge this gap, secure hardware-software co-design assumes a precisely specified hardware leakage behavior, enabling the specification of constraints for secure compilation. While this approach reduces the overhead required to counteract these effects, it does not entirely remove it.

This work follows a more fundamental paradigm: instead of managing leakage in software, we aim at eliminating it at its source. Conceptually, we propose architectural changes to general-purpose processors at the Register-Transfer Level (RTL) that remove the root causes of micro-architectural leakages. The application of these changes enables the secure execution of masked software without introducing micro-architectural leakage under the Robust but Relaxed (RR) d -probing model, thereby removing the need for complex, security-aware compilers and hardware-specific workarounds in the software. While our countermeasures can be applied to many micro-architectures, we formally prove the security of each countermeasure individually and integrate them into the Ibex processor, whose security we evaluate as a whole through tool-assisted leakage assessment of the full netlist and physical experiments on a Field Programmable Gate Array (FPGA).

Keywords: micro-architecture · side-channel analysis · implementation · verification

1 Introduction

Modern cryptographic primitives provide provable security within a black-box model, where the adversary is assumed to have access only to the inputs and outputs of the primitive. However, Kocher demonstrated that cryptographic implementations, i.e. the concrete implementation of a cryptographic primitive running on a device, are vulnerable to Side-Channel Analysis (SCA), effectively turning the black-box model into a grey-box setting [Koc96]. Specifically, an adversary can observe data-dependent physical characteristics of the device, e.g. its instantaneous power consumption [KJJ99], to extract secret data. To protect cryptographic implementations against SCA, masking splits each sensitive value into an additive $(d + 1)$ -out-of- $(d + 1)$ secret sharing scheme, thereby achieving security of order d [CJRR99, ISW03]. In particular, a d -order secure masking scheme is

secure in the abstract d -probing model meaning that no d -tuple of values observed during execution gives information on any sensitive value [ISW03]. In practice, the complexity of performing successful SCA grows exponentially with d assuming that every value leaks independently. However, real hardware often deviates from the independent-leakage assumption, as physical imperfections, most notably glitches and transitions, can introduce joint leakages arising from recombined internal variables [MPG05, CGP⁺12, CBG⁺17]. Extensive research has focused on designing masked hardware implementations that remain secure even in the (joint) presence of glitches and transitions [NRR06, GMK16, MMSS19]. The robust d -probing model ultimately strengthens the classical d -probing model by explicitly accounting for the joint leakages introduced by such glitches and transitions, thereby providing a sound and formal framework for verifying the security of masked hardware [FGP⁺18].

For masked software, where a Central Processing Unit (CPU) executes a list of instructions operating on different shares, the same issues arise, as the CPU's underlying hardware may unintentionally recombine different shares. Many of these micro-architectural effects have been documented in the literature and linked to glitches and transitions, for example [PV17, CGD18, MMT20, GPM21, MPW22]. However, identifying their precise root cause remains challenging, as commercial CPUs are closed-source and detailed hardware descriptions, such as netlists, are not publicly available. Consequently, recent masking schemes [GD23], verification tools [ZMM23], and security-aware compilers [ZM24] adopt a worst-case application of the robust d -probing model on software, which comes at the cost of increased compile, execution, and verification time. In contrast, open-source CPUs make it possible to model leakage more precisely, as their actual hardware descriptions are publicly available. Hardware-software co-design and co-verification methods formalize the interaction between the executed software and its underlying hardware to ensure that a particular masked software runs securely on a specific CPU. For instance, [GHP⁺21] modifies the hardware to prevent glitch-induced leakage and shifts the mitigation of transition leakage to a small set of formal constraints that the software has to adhere to. Conversely, [CPW24] eliminates leakage caused by overwriting in hardware, but explicitly considers other micro-architectural leakage, such as glitches, out of scope. Tian et al. [TCG⁺25] follow a two layered approach, in which Boolean shares are additionally encoded with a public linear code, such that registers and memory only hold encoded shares, while the Arithmetic and Logic Unit (ALU) is wrapped by decoders and an encoder. Similar to an encryption, this removes transitional and glitch-related leakage outside of the ALU, e.g., overwriting and vertical leakage in the register file, but leaves the leakage in the decoders, the ALU, and the encoder to requirements on the software. Based on the software constraints imposed by the hardware, security-aware compilers can build secure implementations without resorting to worst-case assumptions. Similarly, [BGG⁺22, HHB24] formally specify a CPU's leakage by defining a leakage contract, which enables security-aware compilation for a given micro-architecture. While both approaches yield more efficient software, they still suffer from treating at least some hardware leakage as a given and attempting to mitigate its effects solely at the software level. In turn, this places a substantial burden on the compiler, which must incorporate intricate platform-specific knowledge and still contend with unavoidable performance trade-offs.

1.1 Our Contributions

We propose a stricter, more fundamental paradigm. Instead of defining a contract to manage and compile for existing leakage, our goal is to avoid micro-architectural leakage entirely. We propose design modifications to several components of modern CPUs by strategically combining and refining established mitigation techniques for micro-architectural leakages found in the literature. We first discuss our contributions concerning related countermeasures.

One-hot Encoded Multiplexer (OH-MUX) Similar to [GHP⁺21], we apply OH-MUXes to counteract glitches.

Double-Clocked Registers (DCRs) We refine previously proposed register designs that avoid transitional leakage, including area-optimized “double-buffer” registers [CPW24], the register duplication approach [EM25], and secure double-rate registers [BBM⁺18]. Our DCR mitigates leakage caused by glitches and transitions, while also supporting the storage of values across multiple cycles.

General-Purpose Register File (GPR) To protect the GPR, we apply a similar construction as in [EM25] but with additional protection against transitional leakage.

ALU We apply a similar approach as given in [GHP⁺21] but additionally avoid transitional leakage on control signals.

Memory We contribute a formal proof that, without constraints on either the memory or the software, no memory adapter can be secure. Additionally, we propose two constructions requiring only an upper bound $d_{\max}$ on the masking order. Both constructions, in contrast to the approaches in [CPW24, EM25], remain secure for arbitrary memories, i.e., under vertical and temporal probe propagation inside the memory.

Our core contributions are as follows:

- We propose design changes in several components of modern CPUs to avoid micro-architectural leakages. In contrast to [GHP⁺21, CPW24, TCG⁺25], we aim to mitigate micro-architectural leakages, i.e. the ones shown in [BGG⁺14, PV17, CGD18, MMT20, GPM21, MPW22], fully in hardware to avoid the need for complex leakage contracts and security-aware compilers.
- As a prototype implementation, we modified the Ibex RISC-V CPU’s Register-Transfer Level (RTL) code from [GitHub](https://github.com/lowRISC/ibex)¹ to incorporate our enhancements². In contrast to [GHP⁺21, CPW24, TCG⁺25], our CPU also mitigates transitional leakage, not only limited to storage elements as in [TCG⁺25] or the overwriting of registers as in [CPW24], but securing the entire micro-architecture. Furthermore, we secured the memory, discussing differences of our approach to those proposed by [GHP⁺21, CPW24, TCG⁺25].
- We formally proved, for each countermeasure individually, that it restricts the probe extension within the Robust but Relaxed (RR) d -probing model [MM24]. We note that none of the countermeasures discussed above has previously been analyzed under this model. Further, we evaluated the complete gate-level netlist of our prototype in the RR 1-probing model using multiple leakage-prone software case studies. To this end, we extended PROLEAD to support the behavioral simulation of generic memory blocks.
- Regarding performance and area overhead, our Ibex core implementation imposes a moderate overhead compared to the baseline Ibex design and to prior work [GHP⁺21, CPW24, TCG⁺25], but avoids the need for complex software constraints. We emphasize that the choice of the memory adapter used by our memory encryption will have a negative impact on performance and area. However, by purely counteracting any leakage in hardware, this approach provides a new and reliable option for secure masked software execution.

¹<https://github.com/lowRISC/ibex>

²Our countermeasures apply to other CPUs as well. We also implemented them on an ARM Cortex-M0, but do not include those results here due to NDA restrictions.

The resulting protected RISC-V CPU, the extended version of PROLEAD³, and the evaluated case studies have been made publicly available on [GitHub](#)⁴. Furthermore, an overview highlighting the modifications applied to each module of the CPU is provided in Figure 1.

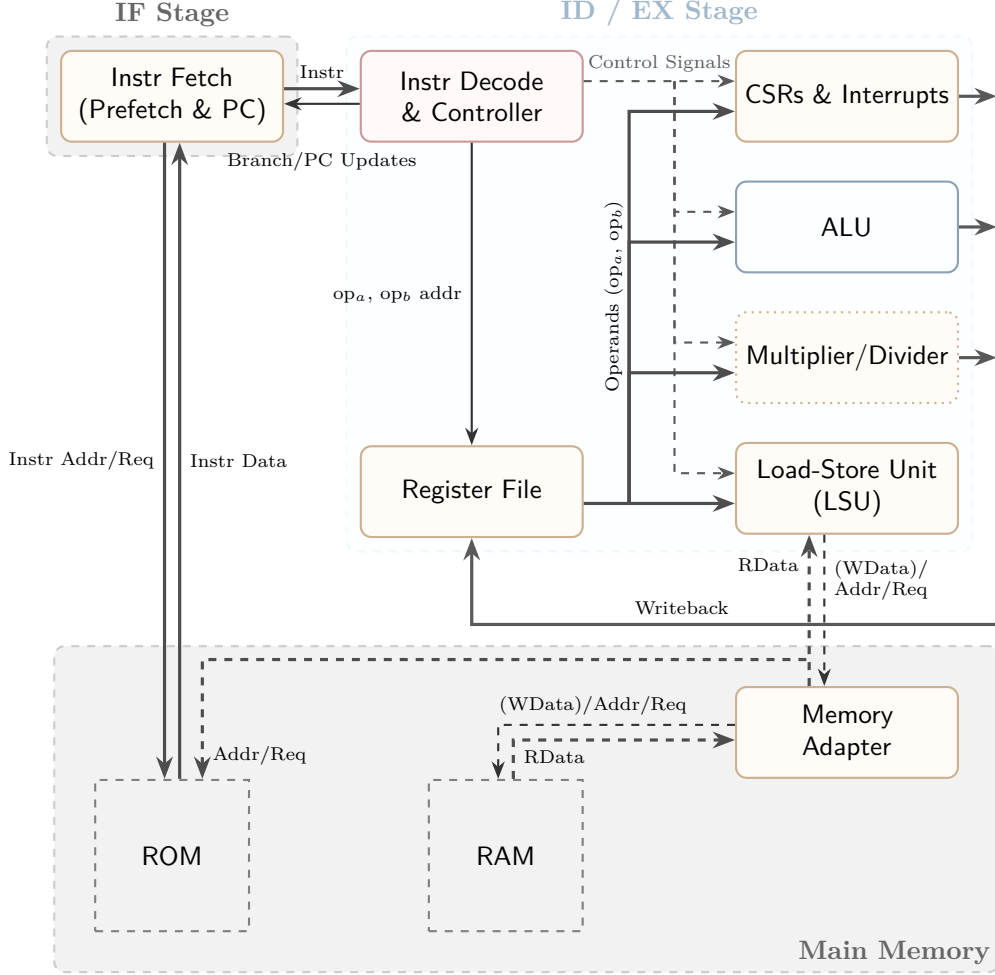


Figure 1: Our secure CPU. Orange modules are protected by DCRs and OH-MUX, pink modules only use DCRs to store control signals while their processed data is not sensitive, blue modules are fully combinational and protected by OH-MUXes, gray denotes peripherals. The dotted multiplier/divider is optionally enabled in the RTL.

2 Background

2.1 Notation

We denote variables by lower-case letters such as $x \in \mathbb{F}_2$. For multi-bit variables we denote their individual bits by subscripts, like $x \in \mathbb{F}_2^3$ with $x = (x_0, x_1, x_2)$. Further, we

³<https://github.com/ChairImpSec/PROLEAD>

⁴<https://github.com/ChairImpSec/Micro-Arch-Protected-Ibex>

denote sets by calligraphic letters, e.g., $\mathcal{S}$. We denote no specific time instances but use primes to denote two consecutive time instances with x' being the preceding value and x the subsequent value. Memory elements are denoted as R and their primary signals are specified according to their port name. Specifically, a register output of R , forwarded by port Q is q_R while its input, captured by port D is D_R .

2.2 Boolean Masking

Boolean masking leverages additive masking [CJRR99, ISW03] as a countermeasure against SCA [CJRR99, GP99]. In a d -order masking scheme, a sensitive value $x \in \mathbb{F}_2$ is split into $d + 1$ shares, $Sh(x) = \{x_0, \dots, x_d\}$, such that $x = \bigoplus_{i=0}^d x_i$. Shares $\{x_0, \dots, x_{d-1}\}$ are sampled uniformly at random, and x_d is computed as $x_d = x \oplus \bigoplus_{i=0}^{d-1} x_i$. This way, any strict subset $M \subset Sh(x)$ reveals no information about x . Once a value is shared, all subsequent computations must be performed on the shares themselves. For any linear function $f : \mathbb{F}_2^n \rightarrow \mathbb{F}_2$ (e.g., xor), the computation is straightforward, as f can be applied to each share individually: $f(a, b, c, \dots) = f(\bigoplus_{i=0}^d a_i, \bigoplus_{i=0}^d b_i, \bigoplus_{i=0}^d c_i, \dots) = \bigoplus_{i=0}^d f(a_i, b_i, c_i, \dots)$. For non-linear functions (e.g., and), this property does not hold and more advanced schemes are required.

2.3 Dual-Rail with Pre-charge Logic (DRPL)

DRPL aims at equalizing the dynamic power consumption of circuits independent of the processed data. In DRPL, every wire is replaced by a pair of wires: one carrying the original signal and the other carrying its inverted version. To operate a circuit implemented in DRPL, each wire pair must alternate between the zero state $(0, 0)$ and a data state, either $(1, 0)$ or $(0, 1)$. Wave Dynamic Differential Logic (WDDL) achieves this by using standard cells that are monotonic positive [TV04b]. The monotonicity ensures that the zero state propagates through the entire circuit, while, to reach the data state, exactly one wire in each pair can toggle. This results in a constant number of toggles independent of the processed data, which equalizes the power consumption. Although WDDL has been shown to be insecure as a standalone countermeasure due to the early-propagation effect [SS06, MLM24], the fact that WDDL circuits are inherently glitch-free can be advantageous when carefully combined with hardware masking [LMW14, LMDM25].

2.4 Adversary Models

Basic Probing Model The foundational model, introduced by Ishai et al. [ISW03], considers an adversary capable of placing up to t probes on arbitrary circuit wires. In this model, the adversary is able to learn the values carried by probed wires during a single but arbitrary point in time. If this information is not sufficient to reveal sensitive information, the circuit is t -probing secure.

Robust d -Probing Model Signals cannot propagate infinitely fast and logic-gates cannot switch infinitely fast. Therefore, signals propagate with a tiny but noticeable delay. Since a logic gate does not wait until all signals have propagated and are stable, but instead potentially switches already once its first input signal arrives, the output of the gate will potentially switch with every arriving input. If the inputs are not stable, and therefore the output switches due to a temporary input signal, this behavior is considered a *glitch*. While different wire lengths can already suffice for this to happen, preceding gates in the combinational path have a much stronger impact on the delay. Faust et al. [FGP⁺18] formalized this via *glitch-extended probes* (cf. Definition 1), which account for the fact that an adversary observing the final output of a circuit will not only learn about the values observed on this wire, but all combinational inputs.

Definition 1 (Glitch-Extended Probe). Let p be a probe on wire w . The glitch-extension of p is the set of probes $\mathcal{P}$ containing all register outputs or primary inputs that provide a combinational path to w . If w is itself a primary input or register output, then $\mathcal{P} = \{p\}$.

Furthermore, because physical transitions depend on both the current and the preceding state of a wire, *transition-extended probes* are used to model leakage stemming from register updates and memory effects as defined in [Definition 2](#).

Definition 2 (Transition-Extended Probe). Let p_t be a probe on wire w at clock cycle t . The transition-extension of p_t is the set $\mathcal{P} = \{p_t, p_{t-1}\}$, i.e. probes recording the current and the preceding clock cycle.

If an attacker cannot reveal any sensitive value using d glitch-and-transition-extended probes, we say that the circuit is robust d -probing secure.

Robust but Relaxed d -Probing Model While the robust d -probing model captures the effects of physical defaults like glitches and transitions, it has been shown in [\[MLM24\]](#) that its generic consideration might be over-conservative, potentially rejecting masking schemes which are secure in practice. More specifically, its over-conservativeness stems from the assumption that glitches propagate independently of the actual signals, whereas in practice such propagation may be suppressed for specific signals. To address this discrepancy, the robust but relaxed d -probing model considers the physical defaults more realistically, by taking the gate behavior and its operands into account [\[MM24\]](#). Instead of unconditionally extending a probe on a gate's output to all inputs of the same gate, this model evaluates each gate individually. A probe is only propagated from a gate's output to a specific input if and only if a glitch on this input is actually capable of affecting the gate's output, defined by [\[MM24\]](#) in [Algorithm 1](#). Whether a probe is extended depends on a Boolean

Algorithm 1 Relaxed probe-extension (`relaxed_extend`) from [\[MM24\]](#)

Input: p_w ▷ A relaxed probe placed by the adversary on wire w
Output: $\mathcal{P}$ ▷ The resulting set of probes

```

1: if  $p_w$  is placed on the output of a combinational gate with two inputs then
2:   if  $F_w = 1$  then
3:      $\mathcal{P} \leftarrow \{\text{relaxed\_extend}(p_x)\} \cup \{\text{relaxed\_extend}(p_y)\}$  ▷  $p_x$  and  $p_y$  are relaxed
       probes on both inputs of the combinational gate
4:   else
5:      $\mathcal{P} \leftarrow \{p_{w'}\} \cup \{p_w\}$  ▷ Do the transition extension
6:   end if
7: else if  $p_w$  is placed on the output of a NOT-gate then
8:    $\mathcal{P} \leftarrow \{\text{relaxed\_extend}(p_x)\}$  ▷  $p_x$  is the relaxed probe on the input of the
       NOT-gate
9: else
10:  if  $p_w$  is placed on an output of a sequential gate or a primary input then
11:     $\mathcal{P} \leftarrow \{p_{w'}\} \cup \{p_w\}$  ▷ Do the transition extension
12:  end if
13: end if
```

function F_w , which evaluates two consecutive input states of every input, such as (x, x') for the gate computing w (cf. [Line 2](#)). Moreover, F_w receives the outcomes of other Boolean functions G_x, G_y indicating whether x and y are already glitchy. A probe extends only if $F_w = 1$, otherwise the probe-extension stops at wire w . For the gates relevant to this work, i.e., $z = x \circ y$ with $\circ \in \{\wedge, \vee, \oplus\}$, we restate F_z and G_z from [\[MM24\]](#), defined over

$\mathcal{I} = (x', x, G_x, y', y, G_y)$, in an equivalent but more compact form:

$$F_z^{\text{AND}}(\mathcal{I}) = (x' \vee x \vee G_x) \wedge (y' \vee y \vee G_y) \quad (1)$$

$$G_z^{\text{AND}}(\mathcal{I}) = (x' \wedge \bar{x} \wedge \bar{y}' \wedge y) \vee (\bar{x}' \wedge x \wedge y' \wedge \bar{y}) \vee ((G_x \vee G_y) \wedge F_z^{\text{AND}}(\mathcal{I})) \quad (2)$$

$$F_z^{\text{OR}}(\mathcal{I}) = (\bar{x}' \vee \bar{x} \vee G_x) \wedge (\bar{y}' \vee \bar{y} \vee G_y) \quad (3)$$

$$G_z^{\text{OR}}(\mathcal{I}) = (x' \wedge \bar{x} \wedge \bar{y}' \wedge y) \vee (\bar{x}' \wedge x \wedge y' \wedge \bar{y}) \vee ((G_x \vee G_y) \wedge F_z^{\text{OR}}(\mathcal{I})) \quad (4)$$

$$F_z^{\text{XOR}}(\mathcal{I}) = (x' \oplus x) \vee (y' \oplus y) \vee G_x \vee G_y \quad (5)$$

$$G_z^{\text{XOR}}(\mathcal{I}) = ((x' \oplus x) \wedge (y' \oplus y)) \vee G_x \vee G_y \quad (6)$$

2.5 Micro-Architecture

A processor’s Instruction Set Architecture (ISA) defines its functional behavior from the perspective of a software programmer. It is the contract that specifies the set of available instructions, register layouts, and memory addressing modes. Any number of processors can adhere to the same ISA (e.g. x86, ARMv6-M, RISC-V) and be capable of correctly executing the same software. However, the term micro-architecture refers to the specific underlying implementation of that ISA. While the ISA defines what an instruction does, the micro-architecture determines how it is executed in hardware. This enables a vast design space for micro-architectures, with countless variations that are all functionally compliant with the same ISA. These variations often stem from trade-offs between metrics like performance, area, and power consumption. These micro-architectural choices can range from high-level design decisions to low-level implementation details. For instance, designers might choose to utilize a deep pipeline with many stages to maximize clock frequency, or a simpler, shorter pipeline to reduce complexity and power consumption. Similarly, they might implement a multiplier as fast but complex parallel version or a slower, iterative one. Even smaller details, such as the precise multiplexer structure used to fetch operands from the register file, are part of the micro-architecture. As with any hardware, executing instructions produces observable side effects, such as variations in power consumption or timing. Critically, while the implementation differences are invisible to the software’s functional correctness, they have profound implications for security. Since these side effects are a direct consequence of the hardware’s activity, two functionally identical processors with different micro-architectures can exhibit completely different side-channel leakage profiles. It is this gap, the discrepancy between the abstract ISA and the concrete micro-architecture that creates the attack surface this work aims to eliminate. Some of these so-called micro-architectural leakages have already been generalized and abstracted into a variant of the robust probing model, which captures such effects independently of the concrete CPU architecture [GD23, ZMM23]. This means that even if the implementation details of the CPU are not publicly available, the model can still be used to reason about the security of a masked software implementation. Specifically, the model considers relevant micro-architectural effects from literature in a generic and conservative manner by defining two additional probe-extension procedures.

Vertical Probes. In a standard CPU, such as ARM or RISC-V, the selection of operands from the register file is performed using multiplexers or any similar combinational logic. According to the robust d -probing model [FGP⁺18], a glitch-extended probe placed on the output of such a multiplexer is not limited to the register selected by the executed instruction, instead, it glitch-extends to all possible register outputs forwarded to the multiplexer. Vertical probes abstract this effect by modeling the joint leakage of a single specific bit position across every register in the register file. Formally, any probe p targeting a bit index i of an operand is expanded to the set of probes $\mathcal{P}$ targeting the same bit index i across the set of all registers $\mathcal{R} = \{R_0, R_1, \dots, R_{31}\}$.

Horizontal Probes. Bit-wise instructions, such as xor or and, over 32-bit words are defined as $z_i = x_i \circ y_i, \forall 0 \leq i < 32$ where x_i, y_i , and z_i denote the i -th bit of the 32-bit words x, y , and z , respectively, and $\circ$ denotes the corresponding bit-wise operator. Arithmetic instructions, such as add, also combine bits x_i and y_j with $i \neq j$. For example, if add runs a ripple-carry adder, it holds that $z_i = x_i \oplus y_i \oplus c_{i-1}, c_i = x_i y_i \vee c_{i-1}(x_i \oplus y_i), \forall 0 < i < 32$ meaning that glitch-extending a probe on z_i observes all $(x_j, y_j), \forall j \leq i$ through the carry chain. However, choosing another type of adder, such as a Kogge-Stone adder, would lead to glitch-extensions to a different set of input bits. To conservatively abstract these cross-bit dependencies, horizontal probes assume that every probe targeting a single bit b_i within an n -bit word $W = (b_0, b_1, \dots, b_{n-1})$ is expanded to a set of probes $\mathcal{P}$ covering all bit positions within that word.

3 Mitigation of Micro-Architectural Leakage

As our countermeasures often find use in more than one place, we will explain the principle on selected cases, but want to reiterate that we covered the entire Ibex in our work, even though not all modules are listed below. All changes are outlined in Figure 1 and a full list of changes can be observed in our artifact⁵.

3.1 Double-Clocked Register (DCR)

Whenever an instruction writes to a storage element, such as its destination register, a transition-extended probe placed on the element’s output can observe both the previously stored value and the newly written value. Therefore, it must be ensured, either in software or hardware, that no memory overwrite leaks any sensitive information. A natural approach to mitigate transitional leakage is by following a pipelining strategy as discussed in [CS21] where only independent values are written consecutively to the same storage element. In software, this can be achieved by clearing registers with additional write operations or by reordering independent instructions. Although this method is efficient, as it targets only critical transitions, it shifts the entire responsibility for avoiding transitional leakage onto the software developer. Instead, we mitigate transitional leakage at registers by modifying the hardware in a way that all values stored inside of a register are securely overwritten by zero before a new value is stored in this register. To this end, we could try using contributions proposed in other works, but to our knowledge, all of them either do not allow us to hold values for multiple cycles [CPW24, BBM⁺18], or are simply not secure in our security model, as they only cover transitions on the register itself due to overwriting its value, but not leakage due to glitches and transitions on output signals [EM25]. A different approach is taken in [TCG⁺25], where registers and memory only hold encoded shares, such that a transition between two stored values reveals no relation between the underlying shares. In their design, every Boolean share is encoded individually, such that each register holds the encoding of a single share, a setting required by their encoding rather than by masked software in general. Shares only appear decoded within the decoders, the ALU, and the encoder, where their software requirement excludes related shares in consecutive instructions. Hence, transitions on control signals need not be considered separately there. In contrast, as our work does not encode the data in the rest of the micro-architecture, we do need to secure the control signals as well, for which we use our DCRs.

While our construction (as well as the constructions in [CPW24, BBM⁺18, EM25]) may be reminiscent of DRPL [TV04a], as both DRPL and our proposal use a pre-charge phase, we do not use a dual-rail approach. The fundamental difference is that DRPL is designed to provide power equalization or to be glitch-free by having two consecutive phases: an evaluation and a pre-charge. The pre-charge phase between every two evaluation phases

⁵<https://github.com/ChairImpSec/Micro-Arch-Protected-Ibex>

inherently avoids transitional leakage. Instead, we only protect against transitions and do not deal with power-equalization and only require glitch-freeness for a small subset of modules (multiplexers), leaving the rest to be mitigated by masking.

We replace selected registers using custom Double-Clocked Registers (DCRs), as shown in Figure 2, and leave the remaining combinational logic, such as the ALU, mostly unmodified. At the rising edge (resp. falling edge) of each clock cycle, a new value of x

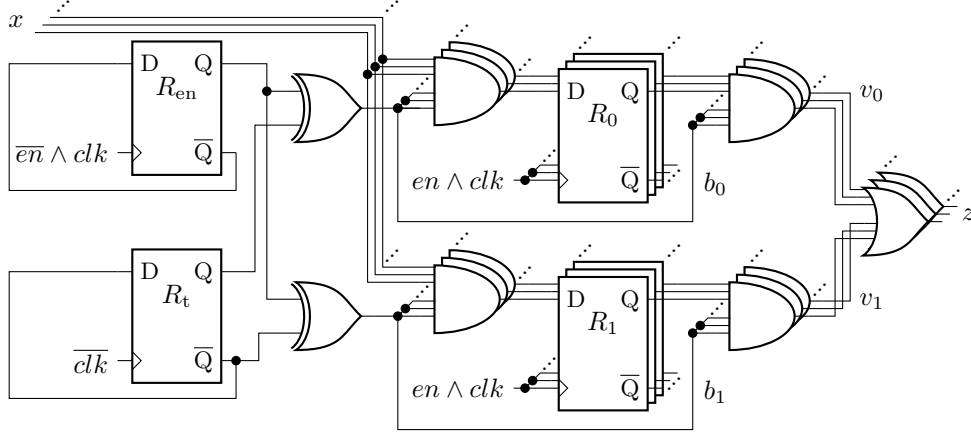


Figure 2: A DCR with n -bit primary signals $x, z \in \mathbb{F}_2^n$. The data path logic is replicated n times, whereas the control signals are generated by a single control-logic instance. This results in a total of $2 + 2n$ registers, $4n$ AND gates, n OR gates, and two XOR gates per n -bit register.

can be stored in one of the two data registers, R_0 or R_1 , depending on the blinding signals $q_{R_t}, \bar{q}_{R_t}$, alternating at every falling edge. Consequently, whenever the DCR writes a new value into R_0 , it simultaneously resets R_1 , and vice versa. The data registers forward their stored values to the output only until the next falling edge (resp. rising edge) of the clock. We show an example waveform of a DCR with rising-edge configuration in Figure 3. To allow DCRs to preserve their state across multiple clock cycles, $q_{R_{en}}$ provides an additional alternating signal that neutralizes q_{R_t} while we use clock-gating to enable and disable the clock of R_0 and R_1 by a global enable signal. We remark that clock gating alone is insufficient, as the required enable signal would have to be valid already after half a cycle. Otherwise, we would misalign the register selection with the register values due to a switch in the toggle signal. This way, the computation of the clock enable signal can take up to a full clock cycle. This is especially important, since we have to consider that some results can only be computed in the second half of any clock cycle, due to a dependency on the results of the current operation.

Security Analysis. We analyze the probe-propagation through a DCR under the RR d -probing model [MM24]. For simplicity, we assume $x, z \in \mathbb{F}_2$, i.e., a single-bit DCR; the analysis extends directly to the n -bit case. Moreover, while the original formulation of the model treats an entire clock cycle as a single time instance, we adapt this notion to circuits that evaluate at both clock edges, so that each half of a clock cycle constitutes one time instance. We place no restrictions on x : across two consecutive time instances, x may take different values, and $G_x = 1$ can hold, i.e., x may be glitchy. As glitches on x can propagate to D_{R_0} or D_{R_1} , a RR probe on D_{R_0} or D_{R_1} behaves as a robust probe according to [FGP⁺18] and extends back to x , $q_{R_{en}}$, and q_{R_t} . As $q_{R_{en}}$ and q_{R_t} are

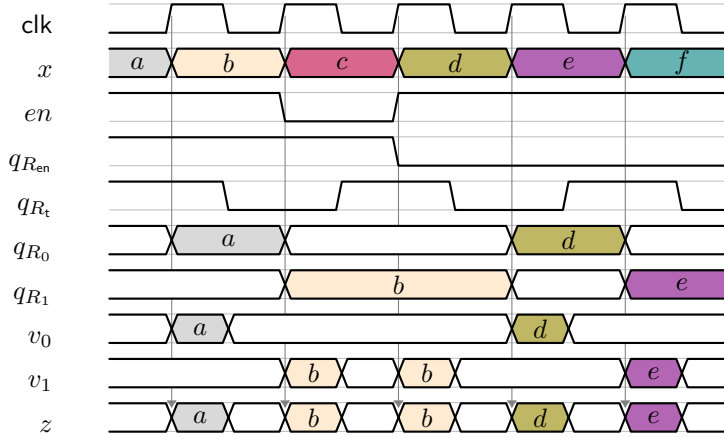


Figure 3: Waveform of a rising-edge DCR, applied to consecutive inputs $a, \dots, f$.

control signals independent of any sensitive data, we ignore them and assume that any robust probe on D_{R_0} or D_{R_1} extends to a robust probe on x . All register outputs are glitch-free, i.e., $A_1 : \forall R \in \{R_{\text{en}}, R_t, R_0, R_1\} : G_{q_R} = 0$. As each time instance is delimited by a single clock edge and R_{en} and R_t are sensitive to different clock edges, it holds that $A_2 : (q'_{R_{\text{en}}} = q_{R_{\text{en}}}) \vee (q'_{R_t} = q_{R_t})$ for any two consecutive time instances. Hence, for $b_0 = q_{R_{\text{en}}} \oplus q_{R_t}$ and $b_1 = q_{R_{\text{en}}} \oplus \bar{q}_{R_t}$, applying Equation 6 yields

$$A_3 : G_{b_0}^{\text{XOR}}(q'_{R_{\text{en}}}, q_{R_{\text{en}}}, G_{q_{R_{\text{en}}}}, q'_{R_t}, q_{R_t}, G_{q_{R_t}}) \stackrel{A_1}{=} (q'_{R_{\text{en}}} \oplus q_{R_{\text{en}}}) \wedge (q'_{R_t} \oplus q_{R_t}) \stackrel{A_2}{=} 0 \wedge \\ G_{b_1}^{\text{XOR}}(q'_{R_{\text{en}}}, q_{R_{\text{en}}}, G_{q_{R_{\text{en}}}}, \bar{q}'_{R_t}, \bar{q}_{R_t}, G_{\bar{q}_{R_t}}) \stackrel{A_1}{=} (q'_{R_{\text{en}}} \oplus q_{R_{\text{en}}}) \wedge (\bar{q}'_{R_t} \oplus \bar{q}_{R_t}) \stackrel{A_2}{=} 0$$

which proves that b_0 and b_1 are glitch-free. As R_0 , R_1 , and R_t are sensitive to different clock edges, and R_{en} is clocked only while R_0 and R_1 are gated, it holds that $A_4 : (q'_{R_i} = q_{R_i}) \vee (b'_i = b_i)$ for any two consecutive time instances and $i \in \{0, 1\}$. Hence, for $v_i = q_{R_i} \wedge b_i$ with $i \in \{0, 1\}$, applying Equation 2 yields

$$A_5 : G_{v_i}^{\text{AND}}(q'_{R_i}, q_{R_i}, G_{q_{R_i}}, b'_i, b_i, G_{b_i}) \stackrel{A_1, A_3}{=} (q'_{R_i} \wedge \bar{q}_{R_i} \wedge \bar{b}'_i \wedge b_i) \vee (\bar{q}'_{R_i} \wedge q_{R_i} \wedge b'_i \wedge \bar{b}_i) \stackrel{A_4}{=} 0$$

Finally, consider $z = v_0 \vee v_1$. It holds that $A_6 : v'_0 = v_0 \vee v'_1 = v_1$ and also $A_7 : (v'_0 = 0 \vee v_0 = 0) \wedge (v'_1 = 0 \vee v_1 = 0)$, and Equation 3 and Equation 4 simplify to

$$F_z^{\text{OR}}(v'_0, v_0, G_{v_0}, v'_1, v_1, G_{v_1}) \stackrel{A_5}{=} (\bar{v}_0' \vee \bar{v}_0) \wedge (\bar{v}_1' \vee \bar{v}_1) \stackrel{A_7}{=} 1 \\ G_z^{\text{OR}}(v'_0, v_0, G_{v_0}, v'_1, v_1, G_{v_1}) \stackrel{A_5}{=} (v'_0 \wedge \bar{v}_0 \wedge \bar{v}_1' \wedge v_1) \vee (\bar{v}_0' \wedge v_0 \wedge v'_1 \wedge \bar{v}_1) \stackrel{A_6}{=} 0$$

Hence, we can assume in the following proofs that all outputs of DCRs are glitch-free and, if we conservatively assume $F_{v_i}^{\text{AND}}(q'_{R_i}, q_{R_i}, G_{q_{R_i}}, b'_i, b_i, G_{b_i}) = 1$ for $i \in \{0, 1\}$ that a RR probe on z observes the set $\{q'_{R_0}, q_{R_0}, q'_{R_1}, q_{R_1}\}$ (excluding control signals b_i). However, as $(q'_{R_0} = 0 \vee q_{R_0} = 0) \wedge (q'_{R_1} = 0 \vee q_{R_1} = 0)$ leading to $z' = 0 \vee z = 0$, no additional information gets leaked due to transitional leakage.

Application. We replaced every register that either handles sensitive values itself or controls logic that handles sensitive data with a DCR. Additionally, we added DCR-based buffer stages into the combinational circuits, as the outputs of the first DCR stage will become invalid in the second half of the cycle. Fortunately, as the CPU often selects inputs in the first half, this buffer stage is often much smaller than the first stage.

3.2 One-hot Encoded Multiplexer (OH-MUX)

In a typical execution flow, a multiplexer-tree selects the instruction's operands from the GPR before forwarding them to the ALU. It was shown in [PV17] that such a construction suffers from the Neighbor Leakage Effect (NLE), i.e. recombination of different register values due to glitches. Therefore, a glitch-extended probe placed on the multiplexer output extends to all registers in the GPR, abstracted as vertical leakage in [GD23, ZMM23].

We apply OH-MUXes as presented in Figure 2b of [GHP⁺21] combined with a DCR. Select signals are encoded via one-hot encoding, i.e. an n -bit signal for an n -to-1 OH-MUX. The general construction is shown in Figure 4. A DCR buffers s resulting in q alternating

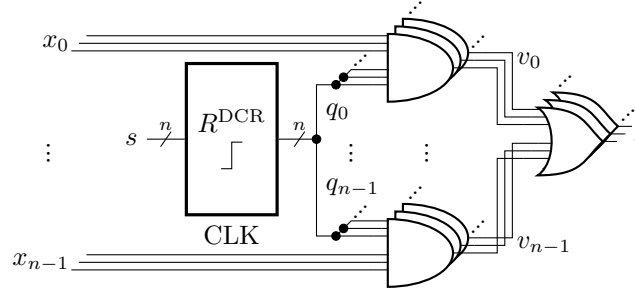


Figure 4: OH-MUX that selects $z \in \mathbb{F}_2^m$ from n m -bit inputs $x_i \in \mathbb{F}_2^m$ for all $i \in [0, n)$, based on an n -bit one-hot encoded select signal $s \in \mathbb{F}_2^n$.

between $\mathbf{0}^n$ and a valid one-hot encoded select signal. Hence, only one x_i gets forwarded to v_i through its respective AND gate while all other inputs are blinded and z is computed by a subsequent OR gate with fan-in n .

Security Analysis. We assume $(x_0, \dots, x_{n-1}) \in \mathbb{F}_2^m$, i.e., $m = 1$; since $q = (q_0, \dots, q_{n-1})$ is identically distributed for every bit of an m -bit operand, all arguments apply bit-wise and our analysis holds for arbitrary m . We assume that s is one-hot encoded (including $\mathbf{0}^n$), such that $s \in \mathcal{S}$ with $\mathcal{S} = \{s = (s_0, \dots, s_{n-1}) \in \mathbb{F}_2^n \mid \sum_{i=0}^{n-1} s_i \leq 1\}$. Aside from this, we impose no restrictions on the inputs: they may change at every time instance and may exhibit glitches. Since q is the output of a DCR, it holds that $B_1 : q \in \mathcal{S}$ and $B_2 : \forall i \in [0, n) : (G_{q_i} = 0) \wedge (\bar{q}_i' \vee \bar{q}_i)$ as established in Section 3.1. Applying Equation 2 yields $\forall i \in [0, n) : G_{x_i} = 1 \wedge (q_i' \vee q_i) \implies G_{v_i}^{\text{AND}} = 1$ stating that a glitch on x_i propagates to v_i if q_i is set. Conversely, if q_i is not set it holds that $B_3 : \bar{q}_i' \wedge \bar{q}_i$ and $\forall i \in [0, n) : G_{x_i} = 1 \wedge B_3 \implies G_{v_i}^{\text{AND}} = 0$, i.e. no propagation of the glitches on x_i . If we apply Equation 1 and Equation 2 on all v_i , it holds that $B_1 \wedge B_2 \implies (\forall i \in [0, n) : G_{v_i}^{\text{AND}} = F_{v_i}^{\text{AND}}) \wedge (\exists i \in [0, n) : G_{v_i}^{\text{AND}} = 1)$. This matches the intuition of a multiplexer as only one x_i (including its glitches) gets forwarded to the output while all q_j with $j \neq i$ are constant zero and therefore blind their respective x_j . For a RR probe on v_i , this means that $F_{v_i}^{\text{AND}} = 1$ extends to RR probes on x_i, q_i while all probes on v_j with $j \neq i$ are not extended. The glitches propagated from x_i to v_i further propagate to z as $(\sum_{i=0}^{n-1} q_i = 1 \wedge \forall i \in [0, n) : \bar{v}_i' \vee \bar{v}_i) \wedge \exists i \in [0, n) : G_{v_i}^{\text{AND}} = 1 \implies G_z^{\text{OR}} = 1$ according to Equation 4. In any case, a RR probe on z extends to RR probes on all v_i as $\forall i \in [0, n) : \bar{v}_i' \vee \bar{v}_i \implies F_z = 1$ (cf. Equation 3). However, at most one v_i contains the forwarded x_i and only the probe observing this v_i extends to a RR probe on x_i . All other v_j with $j \neq i$ are constant zero and therefore blind their respective x_j . The probe-extension stops here ($F_{v_j} = 0$) meaning that a RR probe on z cannot extend to probes on multiple inputs of the OH-MUX.

to the correct R_i , since the previous index i was retained in R_w . The value for the new destination register is then again captured in $R_{\text{stg}}^{\text{DCR}}$. If the CPU reads from $R_0, \dots, R_{31}$, an OH-MUX forwards the respective value to the output. To allow reading of values that are still staged in $R_{\text{stg}}^{\text{DCR}}$, which occurs for consecutive write and read instructions targeting the same destination register, the output of $R_{\text{stg}}^{\text{DCR}}$ is also connected to the OH-MUX. Hence, instead of replacing all registers $R_0, \dots, R_{31}$ with dedicated DCRs, a single 32-bit DCR $R_{\text{stg}}^{\text{DCR}}$ together with an additional 66-bit DCR $R_{\text{buf}}^{\text{DCR}}$ used solely to synchronize intermediate results, is sufficient to eliminate the micro-architectural leakages discussed above.

Security Analysis. We assume that only x contains sensitive information, such as processed data, whereas s_r and s_w are public, as they merely encode the source and destination register addresses. We impose no further constraints on x , s_r , or s_w and analyze a single arbitrary bit of each 32-bit value, since our arguments apply uniformly to all bits. As shown in Section 3.1, probes on outputs of DCRs yield no information due to probe extension caused by glitches or transitions. The same holds for the outputs of $R_0, \dots, R_{31}$, as each register is reset to zero before being overwritten. For any two consecutive time instances, it holds that $\forall i \in [0, 32) : q'_{R_i} \neq q_{R_i} \implies q'_{R_i} = 0 \vee q_{R_i} = 0$. Since an OH-MUX forwards one of $q_{R_0}, \dots, q_{R_{31}}$ to v , a probe on v extends only to a single q_{R_i} (cf. Section 3.2). Moreover, the data inputs $q_{R_0}, \dots, q_{R_{31}}$ originate from registers so that $\forall i \in [0, 32) : G_{q_{R_i}} = 0$ while the select signal is a one-hot-encoded output of $R_{\text{buf}}^{\text{DCR}}$. As shown in [MM24], such a construction implies $G_v = 0$. The same reasoning applies to z , which is the output of an additional 2-to-1 OH-MUX. Its select signal is again a one-hot-encoded DCR output, and its data inputs originate from $R_{\text{buf}}^{\text{DCR}}$ as well. Consequently, $G_z = 0$, and a probe on z extends only to a single element of $q_{R_0}, \dots, q_{R_{31}}, q_{R_{\text{stg}}^{\text{DCR}}}$. We therefore conclude that the circuit shown in Figure 5 does not exhibit transitional or vertical leakage.

3.4 Secure Arithmetic and Logic Unit (ALU)

We focus on always-active ALUs as they are usually used in, e.g., ARM and RISC-V CPUs. This means that the ALU performs its supported operations in parallel, such as AND, XOR, ADD, or SUB, and selects the required result by a multiplexer. Such an architecture suffers from horizontal leakage since arithmetic operations, like ADD, or SUB, recombine different bits from the same operand. Formally, a glitch-extended probe placed on the multiplexer's output extends back to the outputs of the ALU and, therefore, to multiple bits from the same operand. Notably, this occurs regardless of the specific instruction being executed, since an always-active ALU performs all computations anyway. The recombination of operand bits is inherent to arithmetic instructions and required for correct execution. Consequently, arithmetic operations may leak information when their operands depend on multiple shares. However, a d -probing-secure masked software mitigates such leakages. Similar to [GHP⁺21] our hardware countermeasures aim to isolate the individual ALU components responsible for different instructions to prevent horizontal leakage when no arithmetic instruction is executed. Notably, our proposed implementation additionally covers transitional leakage due to glitches on control signals, which is not covered by the constraints in [GHP⁺21] and [CPW24]. As a consequence, our ALU allows consecutive instructions to operate on shares of the same secret, which is excluded by the software requirements of [GHP⁺21, CPW24, TCG⁺25]. In [TCG⁺25], secure multiplexers as in COCO [GHP⁺21] are utilized inside the read ports of the register file, while no gating of the internal ALU operations is described. Instead, leakage of always-active ALU operations is covered by their software requirement, which excludes shares of the same secret in consecutive instructions. In contrast, our ALU removes the need for such a requirement, by isolating the individual operations and clearing their inputs between

instructions. To illustrate horizontal leakage, we briefly discuss an example.

Example 1 (Horizontal Leakage in Always-active ALUs). Let $a, b, c, e \in \mathbb{F}_2^8$ be sensitive variables while $(a_0, a_1), \dots, (e_0, e_1) \in \mathbb{F}_2^8 \times \mathbb{F}_2^8$ denote their respective sharings with two shares. Now assume that the ALU computes an instruction such as `xor  $R_1, R_1, R_2$` on two 32-bit registers with $R_1 \leftarrow (a_0, a_1, b_0, b_1)$ and $R_2 \leftarrow (c_0, c_1, e_0, e_1)$. If the ALU computes only this instruction, it would only combine shares with the same share index, i.e. it computes $a_i \oplus c_i$ and $b_i \oplus e_i$ for $i \in \{0, 1\}$. However, an always-active ALU simultaneously computes `add  $R_1, R_1, R_2$` which recombines both shares.

We use an n -bit one-hot encoded select signal $s = (s_0, \dots, s_{n-1})$ to blind the inputs of all n individual operations performed by the ALU. This mirrors the blinding applied to the inputs of $R_0, \dots, R_{31}$ in Figure 5. Specifically, each input x of an operation with index i is pre-processed as $y_i = x \wedge s_i$, ensuring that y_i rather than x is forwarded to operation i . As the select signal comes from its own n -bit DCR it is necessary to determine it at an earlier time than in the original implementation. If the DCR is configured for the rising edge, the entire computation needs to happen one cycle earlier while configuring it for the falling edge increases only the propagation delay. In our case, while moving the operand address to the previous clock cycle similar to [GHP⁺21] was rather straightforward, restricting the decoder logic to the first half of the clock cycle meant that we were required to bypass or replace dependencies from the ALU, since these results can otherwise not be computed at this time. At the end, an n -to-1 OH-MUX selects the required output of the ALU based on s (cf. Figure 6a).

Security Analysis. We assume that single-cycle operations are implemented entirely in combinational logic, whereas multi-cycle operations store intermediates in registers. For these registers, we guarantee that they are never shared across different operations and that they are reset to zero after each operation. Since the GPR forwards the operands to the ALU, the ALU inputs $(x_0, \dots, x_{m-1})$ satisfy $\forall j \in [0, m) : G_{x_j} = 0 \wedge (x'_j = 0 \vee x_j = 0)$. Hence, $y_{i,j} = x_j \wedge s_i$ leads to the same properties $\forall (i, j) \in [0, n) \times [0, m) : G_{y_{i,j}} = 0 \wedge (y'_{i,j} = 0 \vee y_{i,j} = 0)$. For each operation $z_i = F_i(y_{i,0}, \dots, y_{i,m-1})$, we assume that bit-wise operations are implemented as independent single-bit functions, thereby avoiding horizontal leakage. This means that a probe on bit l of z_i only results in probes on bit l of all inputs $(y_{i,0}, \dots, y_{i,m-1})$. For arithmetic operations we assume no such restrictions on horizontal leakage meaning that probing bit l of z_i results in probing all bits of all inputs. Hence, our CPU would leak whenever different shares are processed by an arithmetic operation. However, this can only occur if the executed software is not d -probing secure, which contradicts our initial assumption that the software itself satisfies d -probing security. Finally, a probe on bit l of the ALU output z extends through the OH-MUX to bit l of a single z_i namely the one selected by s . We therefore conclude that no probe in the ALU gains additional information through transitional leakage, and that horizontal leakage only plays a role when the executed software is already insecure in the d -probing model.

We emphasize that the proposed countermeasures are not limited to always-active ALUs. They can equally be applied to designs in which specific operations can be selectively deactivated. Always-active ALUs represent a worst-case scenario, as every operation must be protected by our countermeasures. In contrast, if certain operations can be disabled by design, the overall protection overhead can be reduced accordingly. Furthermore, transitional leakage can originate not only from operands but also from the operations, even when only one of two consecutive instructions processes sensitive data. This happens when the ALU inputs do not alternate between zero and their valid states, a behavior we enforce by taking all inputs from DCRs. In this case, a subsequent operation may still process the preceding inputs, resulting in a probe on z that gains information on both consecutively executed operations on the same operands. This becomes relevant if

an arithmetic operation follows a bit-wise operation while one operand of the bit-wise operation contains multiple shares. Notably, this form of leakage is not covered by the software constraints of COCO [GHP⁺21], which only restrict consecutive operations on shares of the same secret, and lies outside the scope of eLIMInate [CPW24], which targets overwriting leakage only.

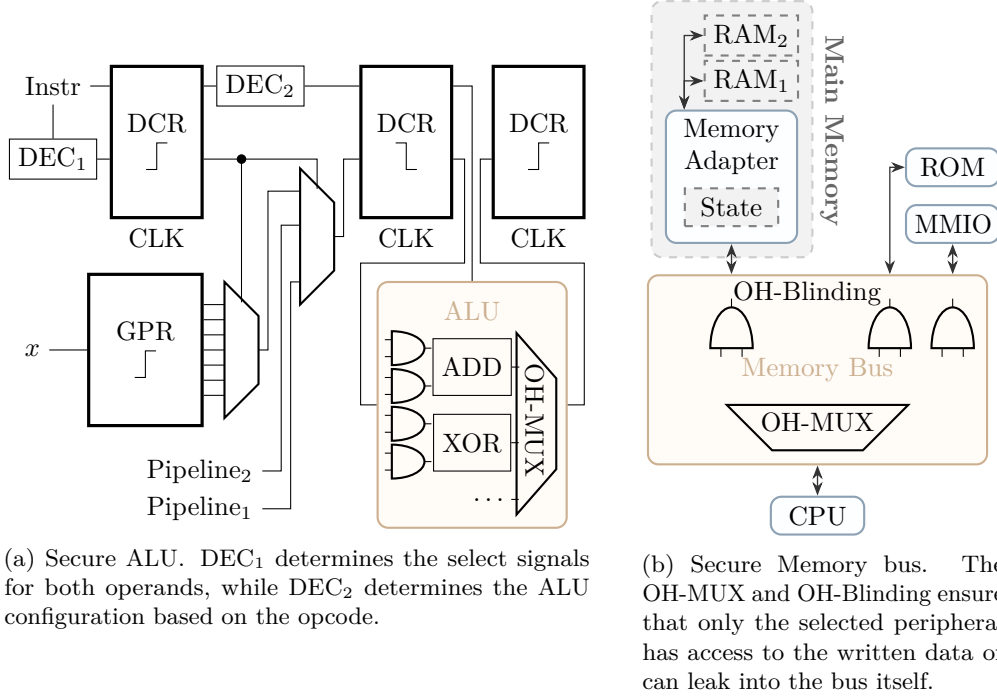


Figure 6: Our secure ALU and our secure bus.

3.5 Bus

The CPU communicates with various memory components through a shared bus. Such a shared bus system can suffer from transitional leakage if the communication is not properly cleared. To initiate a communication between the CPU and an individual component, a simple request/valid handshake activates a single endpoint by unblinding the respective inputs (cf. Figure 6b). Moreover, the request signal and a DCR blind all other bus-related signals. Since this process only focuses on the select signals for the various AND-gates, the bus system propagates probes due to glitches and transitions from the CPU to the selected module. However, our bus ensures that the CPU transmits data only to a single peripheral and assumes each peripheral to provide its own countermeasures. In Figure 6b, the CPU is using the bus to connect to Memory Mapped I/O (MMIO) and ROM elements, as well as the secure main memory, which is explained below.

3.6 Secure Memory Components

One major consideration was the recombination of shares inside the memory. Since we want to allow the software masking to be of arbitrary order d , we have to ensure that the memory will not reduce this security order. In this work, we propose two solutions:

- Custom memories allow secure constructions, but will most likely impose an area and performance overhead. Additionally, this solution also forces the integration of the

entire memory into the hardware design of the CPU, which is not always desirable.

- We can interface arbitrary memories through an adapter. For the CPU and memory this should be completely transparent, but such construction allows for modifications on the adapter like using memory encryption, which scales much more favorably (ideally constant) with the memory size and security order.

As the custom-memory solution follows the same logic as the GPR (Section 3.3), we want to focus on solutions using closed-source, but possibly constrained memory. While these constructions allow for many solutions, they all impose some assumptions or constraints on the memory, which we will discuss in the following part of this section.

Vertical Propagation. Since memories have to be able to retrieve all stored words, we have to assume some sort of multiplexer selecting the requested word. Due to this unknown combinational circuit, we have to assume that a probe on the memory output can propagate to all words simultaneously.

Temporal Propagation. While registers will only leak the transition of values, we have to assume larger hierarchies inside the main memory. Commonly a cache is used in combination with a slower, but larger memory backend. Here, we have to consider that a write-back strategy can be selected for the cache. In this case, a write is not guaranteed to reach every layer at a defined time. In other words, if we try overwriting a secret with a dummy value, no matter how often we perform this write operation, it is possible that the original secret value is still present in some memory layer. Therefore, we have to assume that any secrets written to memory, at any time, can be observed through a single probe propagating to all memory layers.

Leakage Assumptions. To reason about arbitrary memory, we must adopt a worst-case leakage scenario. A single probe on the memory propagates to all stored data, across all addresses, present and past.

3.6.1 Impossibility in the Unrestricted Scenario

We model the adapter between memory and CPU as a finite-state transducer with state S . As inputs we assume arbitrarily many fresh random bits, the address, and the data. Any state not part of S needs to be stored inside the probed memory and is therefore known to the attacker using a single probe p . We also require the adapter to be invertible, in other words, read operations need to return the last written value at this address.

Stateless adapters. We claim that there exists no stateless adapter which can secure memories for arbitrary d -probing secure software. For $d = 1$, this requires temporal probe propagation, while for $d \geq 2$ the claim holds even if temporal probe propagation is bounded.

Proof (Proof). Let $M : \mathcal{A}^{\text{CPU}} \times \mathcal{D}^{\text{CPU}} \rightarrow \mathcal{A}^{\text{MEM}} \times \mathcal{D}^{\text{MEM}}$ be the mapping performed by the memory adapter. Specifically, $M(s, x) = (v, z)$ maps a tuple of an address s and a data input x requested by the CPU to a modified address v and modified data z handled by the memory. As M is deterministic and must be invertible, $M(s, \cdot)$ is injective for every address s , i.e., $M(s, x) = M(s, x') \iff x = x'$. Hence, even if an adversary cannot learn x from observing z , which is the case if M implements memory encryption, comparing two values stored at the same address reveals whether the corresponding values in the CPU domain are equal. Let $(x_0, \dots, x_d)$ be a Boolean sharing of a secret x .

Case $d = 1$. Assume the software stores x_0 and afterwards x_1 at the same address s . Due to temporal propagation, a single probe on the memory observes $M(s, x_0)$ and $M(s, x_1)$ and therefore learns whether $x_0 = x_1$, i.e., whether $x = 0$.

Case $d \geq 2$. Here, we do not require temporal propagation. Instead, the adversary places one probe on the memory at time t_0 and a second one at a later time t_1 . Due to vertical propagation, each probe observes all addresses, so the adversary learns for every address whether its stored value changed between t_0 and t_1 . Consider software that samples fresh randomness $r_1, \dots, r_{d-1}$ and stores at the addresses $s_0, \dots, s_{d-1}$ the values

$$(x_0, r_1, \dots, r_{d-1}) \text{ at } t_0 \quad \text{and} \quad (x_1 \oplus r_1, x_2 \oplus r_2, \dots, x_{d-1} \oplus r_{d-1}, x_d) \text{ at } t_1.$$

As each r_i only occurs in r_i and $x_i \oplus r_i$, recovering x requires all $2d$ values, hence storing them is d -probing secure. But if the stored data z changed at no address between t_0 and t_1 , injectivity of M implies $x_0 = x_1 \oplus r_1$, $r_1 = x_2 \oplus r_2$, $\dots$, $r_{d-1} = x_d$, and thus $x_0 = x_1 \oplus \dots \oplus x_d$, i.e., $x = 0$. Hence, the observation of two (i.e., at most d) probes depends on x , although the software is d -probing secure.

Consequently, no stateless adapter is secure for $d \geq 2$, and for $d = 1$ only if temporal probe propagation is bounded. $\square$

Note that under these worst-case assumptions, one-time-pad constructions with a constant blind per address, i.e., $M(s, x) = x \oplus b_s$ as used in [CPW24, MGH19], are affected even more strongly. Whenever a known value, e.g., a public constant or a zero-initialized word, is stored at address s , a single probe on the memory reveals b_s for all such addresses. Afterwards, every further probe on the memory reveals all values subsequently stored at these addresses in plain, independent of the probing security of the Pseudo Random Function (PRF) generating b_s or of its seed.

A related case are randomized but keyless encodings, such as the code-based encoding of Boolean shares proposed in [TCG⁺25]. Here, every stored word is a fresh encoding of a share, so two values stored at the same address cannot be compared as in the proof above. However, as the encoding is public and invertible without any secret, a probe observing a stored word can simply decode it, and, due to vertical and temporal propagation, a single probe on the memory decodes all shares ever stored. We note that recovering complete words from a single trace is generally hard in practice, and [TCG⁺25] explicitly targets the noisy leakage of individual bits in storage elements. In the probing model with vertical and temporal propagation, however, this is exactly the observation granted to the adversary, which is why our constructions rely either on a secret key or on independently leaking memories.

Stateful adapters. To break the dependency between $M(s, x)$ and $M(s, x')$, the adapter applies fresh randomness or a non-repeating state r , leading to a probabilistic mapping $W : \mathcal{A}^{\text{CPU}} \times \mathcal{R}^{\text{CPU}} \times \mathcal{D}^{\text{CPU}} \rightarrow \mathcal{A}^{\text{MEM}} \times \mathcal{R}^{\text{MEM}} \times \mathcal{D}^{\text{MEM}}$ with $W(s, r, x) = (v, u, z)$ such that $W(s, r_0, x) = W(s, r_1, x') \not\Rightarrow x = x'$ if $r_0 \neq r_1$. To ensure correct inversion, $W^{-1}(v, u, z)$ needs access to the correct u . We distinguish two cases. If the adapter holds no secret, u cannot be stored on the memory, as any probe on the memory observing z would also observe u . An adversary could equally observe u by probing its registers inside the adapter, leading to the same information gain as for stateless adapters but with one additional probe. Therefore, u must be stored as a sharing with d shares, so z and the sharing of u form a $(d+1)$ -sharing of x stored in $(d+1)$ independently leaking memories. Alternatively, the adapter holds a secret key k , stored as a $(d+1)$ -sharing, and computes z as a d -probing secure encryption of x under k and a tweak (s, u) . In this case, u is a public, non-repeating nonce that may be stored in any memory, but security relies on the computational security of the encryption. We claim that no adapter with finite state ensures d -probing security for unbounded d .

Proof. We showed in the [proof for stateless adapters](#) that no stateless adapter is secure. Hence, we focus on stateful adapters. Without a secret, each address v needs an associated random value u_v stored as a sharing with d shares inside the adapter, which leads to an overhead factor of d in the memory size (previously $|u| \cdot |\mathcal{A}^{\text{MEM}}|$) of the adapter. With a secret key, the key must be stored as a $(d + 1)$ -sharing inside the adapter. In both cases, the state of the adapter grows with d . As d is unbounded, the state must also be unbounded in size, which cannot be ensured by a synthesized design. Therefore, no secure construction is possible under these assumptions. $\square$

3.6.2 Our Construction

We propose two variants: a basic construction optimized for latency, and a memory-optimized construction based on encryption, whose number of memories is independent of $d_{\max}$. For both constructions, we require the following constraint:

Property 1. There exists an upper bound $d_{\max} \in \mathbb{N}$ such that every executed software is d -probing secure with $d \leq d_{\max}$.

Basic construction. Here, we enforce [Property 1](#) by specifying an upper bound $d_{\max}$ at synthesis time. Our variant requires $32 d_{\max}$ bits of fresh randomness per write operation to encode every x written to the memory as $(d_{\max} + 1)$ -sharing. Every share is stored in its own independently leaking memory requiring $d_{\max}$ memory blocks for the adapter in addition to the main memory. Hence, the overhead depends on the choice of $d_{\max}$, while the security does not depend on a secret key, but on the quality of the fresh randomness. An adversary with $d \leq d_{\max}$ probes can only access an incomplete set of shares by either probing several shares in the adapter or the last share stored in memory.

Memory-optimized construction. As the memory requirements of the basic construction scale with $d_{\max} \cdot |\mathcal{A}^{\text{MEM}}|$, the memory-optimized construction only stores a $(d_{\max} + 1)$ -sharing of a secret key k inside the adapter. In addition, every address s is assigned a public nonce n_s , e.g., a w -bit counter, which is stored in a separate memory. As n_s is public, this memory does not need to be protected and can again be an arbitrary memory. On every write to s , the adapter increments n_s and stores $z = x \oplus F_k(s, n_s)$ for a $d_{\max}$ -probing secure masked PRF F . Alternatively, any $d_{\max}$ -probing secure tweakable block cipher with tweak (s, n_s) can be used. Instead of a separate memory, n_s can also be stored together with the data in a single $(32 + |n_s|)$ -bit word, similar to the encoding in [\[TCG+25\]](#), but keyed, i.e., $z = E_k(s, n_s \| x)$ for a $d_{\max}$ -probing secure tweakable block cipher E with tweak s and a block size of $32 + |n_s|$ bits. If no tweak (s, n_s) is used twice, all values ever stored in memory are pseudorandom and independent of each other. Hence, neither vertical nor temporal probe propagation inside the memory reveals any relation between stored values, and in contrast to stateless adapters, no bound on temporal propagation is required. Similarly, combining a probe on the memory with a probe on the bus only reveals a blind that is never reused for any other value.

Of course, this choice comes with trade-offs. First, n_s must never repeat under the same key, so w must be chosen such that no address is written more than 2^w times, or the adapter has to re-key and re-encrypt the memory on overflow. Second, the memory overhead is reduced from $32 d_{\max}$ to w bits per word, which is only beneficial if $w < 32 d_{\max}$, i.e., the basic construction remains preferable for small $d_{\max}$. Third, every access requires n_s , and writes have to read it first, so each write effectively becomes a read-modify-write, independent of the chosen primitive. Consequently, byte-enabled writes do not cause an additional round trip, as the old word can be read together with n_s . Here, the One-Time-Pad (OTP) variant can reduce, but not avoid, the resulting latency. Since $n_s + 1$ is known as soon as n_s has been read, both keystreams $F_k(s, n_s)$ and $F_k(s, n_s + 1)$ can be computed

without waiting for the data, leaving only XOR operations on the data path. In contrast, a block cipher has to decrypt the old word and encrypt the modified word sequentially on the data path.

3.7 Application to Other Micro-Architectures

As most of our countermeasures are not limited to the Ibex, but are generally applicable to any micro-architecture, we want to distinguish which countermeasures apply easily to other micro-architectures, which do not and how to transfer them in a systematic manner.

Memory. As a first decision, the memory will have to be adjusted. This change transfers easily from our implementation to others, but will dictate the latency or critical path depending on the memory choice. Here, there are three main options. If the memory is to be implemented on the same chip as the CPU, a custom memory solution similar to the one in COCO [GHP⁺21] can be used. As the memory will have to be designed regardless of the security aspects, a custom memory will often be suitably small and secure for arbitrary d . If the memory can be supplied externally, choosing an adapter-based solution is often required. Here we recommend going for the basic version using $d_{\max} + 1$ memories for $d_{\max} = 1$ or for small memory sizes below roughly 9 kbit (FF) or 50 kbit (Static Random-Access Memory (SRAM)). Otherwise memory encryption will offer suitable security if the additional latency and the bounded number of writes per address are acceptable.

GPR. The GPR will transfer without major changes to any other CPU. Here the most relevant aspects are the register count and access patterns. As our implementation is easily adjustable for those parameters, this should pose no issues.

DCR. While a DCR can replace any register, its impact on propagation delay and the inherent pipelining of the processor often impose serious considerations when designing the micro-architecture. Here we want to reiterate the difficulties when control signals are normally computed in the same cycle as they are used. In this case, the entire decoding has to be performed in one half, while the output control signals are used in the next half. Sometimes this can require duplicating some logic. For regular pipeline stages and forwarding stages, the DCR and OH-MUX pose a simple way to counteract glitch and transition related micro-architectural leakages.

MUX and Combinational Logic. Protecting the ALU or replacing multiplexers is a straightforward process if glitch-free control signals are used, e.g. provided by DCR. The OH-MUX, as already used by other implementations [GHP⁺21], can simply be used as a drop-in replacement. In the ALU, the inputs of each operation are blinded by AND gates with its own glitch-free select signal, separating bit-wise operations (e.g., XOR) from arithmetic ones (e.g., ADD).

Specifics. Our countermeasures target micro-architectural leakage, i.e., leakage that is a side product of the implementation and not required by the specified behavior. However, some processors specify behavior that inherently combines data. For example Advanced RISC Machine (ARM) processors offer zero and negative result flags about previous ALU operations, which would inherently cause leakage between different bit positions similar to ADD instructions in the ALU. Architectural design choices like this will require specific adjustments on the micro-architecture, to control when or if these flags are generated. Similarly, any out-of-order and speculative execution might recombine shares unintended by the user or caches may introduce timing-based side channels, which are not addressed by our countermeasures.

4 Case Studies

While we formally prove the security of each countermeasure individually in Section 3, we do not provide a formal proof covering the composition of all modules in the complete CPU. Instead, we complement our component-level proofs with an evaluation of the complete gate-level netlist. For this purpose, we integrated all countermeasures described in Section 3 into the RISC-V Ibex core and conducted several case studies using two complementary methods: tool-assisted leakage evaluation with an extended version of PROLEAD [MM22], and experimental power analysis on a Sakura-X⁶ Field Programmable Gate Array (FPGA) platform.

4.1 Setup

Targeted CPU Platform. Our target Ibex CPU is based on the baseline RV32IC(M) variant, configured similarly to the default “small” Ibex configuration with 32 registers and support for compressed instructions, but only a minimal pipeline, no cache, no branch predictor, and no error-correction. The security was verified with both the fast multiplier and no multiplier, to test multi-cycle ALU operations. A depiction of the processor can be found in Figure 1.

Tool-Assisted Evaluation Framework. Our approach requires a tool capable of analyzing masked software which is executed on a custom CPU. The first tool capable of performing such co-verification is COCO [GHP⁺21] which takes masked assembly code as well as the RTL code of the targeted CPU to perform leakage verification based on approximated Fourier coefficients. While COCO provides an important foundation for co-verification, it is not fully suitable for our purposes as we evaluated all countermeasures in the RR d -probing model. While COCO accounts for blinding techniques by tracking the stability of signals in a given execution trace, this approach only applies to signals that are independent of the actual execution. As a result, it does not fully align with the robust but relaxed d -probing model. As most required features are already implemented in the latest version of PROLEAD, we choose PROLEAD for our case studies. For evaluation, PROLEAD requires the gate-level netlist of our CPU and a library describing the behavior of each cell used in the netlist. Using this information, PROLEAD simulates the CPU on a per clock cycle basis and statistically analyzes the simulated intermediates by creating a contingency table using the simulated signal values. While this methodology is very memory-demanding, similar to experimental tests, the security of any digital circuit can be evaluated on a purely statistical basis. The only missing feature is the support for optimized memory blocks for instructions and data, such as BRAM blocks. While this can be technically circumvented by replacing memory blocks with registers, its support improves the performance of PROLEAD significantly by avoiding the simulation of thousands to millions of registers together with their combinational logic while simultaneously providing a convenient way to initialize memory with a specified firmware. By also using a memory efficient internal data-structure like a binary tree, even larger memories can be supported, since memory only has to be allocated on write, reducing the memory overhead of the simulator to a manageable size. By initializing this memory block with a compiled binary, true hardware-software co-evaluation on the gate-level netlist is possible. To support a wide range of memory types, we added the memory block as a register-like cell to PROLEAD’s cell library, which supports a parameterizable address and data width. The configurable number of ports then provides a generic interface for various memory architectures, using single cycle latency read and write requests. By offering this generic memory block, we support a wide range of memories. Naturally, this would require a unique leakage model for every memory

⁶<https://sato.h.cs.uec.ac.jp/SAKURA/hardware/SAKURA-X.html>

type (e.g. DRAM, SRAM), if we tried to report internal leakages accurately. Since this would far exceed the requirements for our verification, we instead decided to treat the memory block similar to a large, addressable register, allowing probes only at the inputs and outputs.

Experimental Evaluation Setup. For the experimental evaluation, we synthesized our secure Ibex core for the AMD (formerly Xilinx) xc7k160tfbg676-1 (Kintex-7) FPGA on the Sakura-X SCA evaluation board using Vivado 2025.2. The collected measurements were continuously recorded using a Teledyne LeCroy WavePro 404HD Oscilloscope, sampling the power consumption using a 1Ω shunt resistor, inserted into the V_{DD} path of the target FPGA at 100 MS/s, with an increased vertical resolution through the use of a 5 dB operational amplifier. The 6 MHz clock used by the target FPGA was controlled by the controller FPGA. Similarly, the masked input signals were generated on the controller FPGA, while the randomness was computed on the target FPGA using a Linear-Feedback Shift Register (LFSR), randomly seeded through the controller. The leakage analysis was performed on the recorded traces utilizing a non-specific (fixed-vs-random) t-test [SM15], to detect potential information leakage.

4.2 Results

4.2.1 Micro-Benchmarks

The initial firmware evaluation focused on “cherry-picked” assembly sequences. These handcrafted snippets were designed to trigger specific, common micro-architectural leakages to ensure the CPU and memory behave as intended. The following interaction patterns were explicitly tested:

Consecutive Loads This test determines if sequential (consecutive) load operations on two shares cause leakage within the micro-architecture. Due to our implementation, no leakage should be observed here, as the values will be stored securely inside the memory and then enter the register file, with one empty cycle in between, as load operations will take at least two clock cycles. One cycle for the request, one for the response.

```
lw    t0, 0(a0)      # load share 0 (a0: address of the shares)
lw    t1, 4(a0)      # load share 1
```

Leakage in Memory We tested for leakage during memory transitions by writing two shares of the same secret to the same address. Regardless of our memory solution, overwriting the same location is often prone to transitional leakage. Similarly, this is a recurring scenario, as the stack, including function arguments, is often written to the same address.

```
sw    t0, -4(sp)     # store share 0 to address A
sw    t1, -4(sp)     # store share 1 to address A (overwrites share 0)
```

Overwriting Registers Writing one share directly over another in the same register is a common leakage source. Our hardware mitigates this via a staging DCR that captures new values and only forwards them to the destination register after they have been reset to zero.

```
add   t2, t0, zero   # write share 0 to t2
add   t2, t1, zero   # overwrite t2 with share 1
```

Bit Permutations Since shares are not always 32-bit wide, in some cases multiple shares are stored in one 32-bit word. If in this parallelized setting probes can propagate across

multiple bits, this would invalidate probing security. While this will always be true for instructions with dependencies on all bits like multiplications, PROLEAD detected no leakage for this sequence, namely bit-wise, shift and load/store operations did not propagate probes across bit positions, maintaining the independence of the shares.

```
# a1 = 0xFFFF0000, a2 = 0x0000FFFF, a3 = 0xAAAAAAAA
and t2, t0, a1      # t2 = {share0[31:16], 0}
and t3, t1, a2      # t3 = {0, share1[15:0]}
or t2, t2, t3       # t2 = {share0[31:16], share1[15:0]}
xor t3, t2, a3      # t3 = t2 xor 0xAAAAAAAA
li t3, 7            # shift distance = 7
sll t2, t2, t3      # t2 = t2 << 7
```

The handcrafted firmware was first analyzed using PROLEAD with 4 million simulations for each individual clock cycle. Here no significant leakage was reported, as $-\log_{10}(p)$ stayed below the threshold of 5, see Figure 7d. For the physical experimental evaluation, we recorded 3 million traces on the Sakura-X board. The results for the first two statistical orders are depicted in Figure 7.

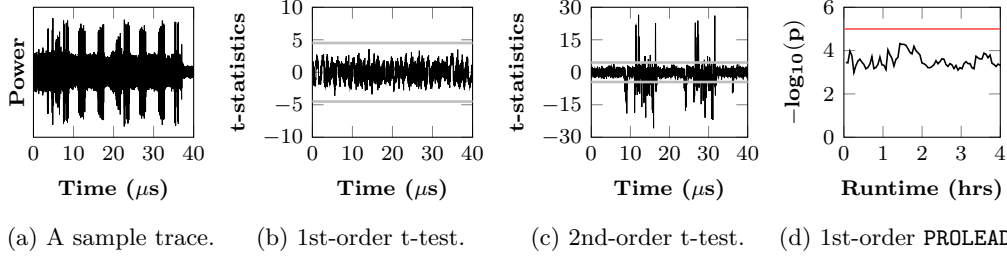


Figure 7: Experimental analysis of the handcrafted firmware using 3 million traces during physical experiments or 4 million simulations in PROLEAD.

The results show no evidence of first-order leakage, as the 4.5 threshold is never surpassed in Figure 7b. As the software was not specifically hardened against higher-order attacks, the leakage observed in the second-order analysis (Figure 7c) is expected and consistent with our security model.

4.2.2 Advanced Encryption Standard (AES)

Finally, we evaluated a masked AES execution. Due to the long execution time, we limited our experiment to the first round only. We base our experiment on the masked AES implementation provided by Rivain and Prouff [RP10], instantiated for $d = 1$. As the underlying masking scheme is provably 1-probing secure, any leakage detected in the first order would have to originate from the micro-architecture, which allows us to evaluate our countermeasures without additional software based countermeasures. Due to the significant execution length, an evaluation with PROLEAD was infeasible due to the time required to simulate that many clock cycles and store all required observations in contingency tables. Instead, we focused on an experimental evaluation on our FPGA platform. The results can be observed in Figure 8, confirming the effectiveness of our countermeasures. The 4.5 threshold is never exceeded in the first-order results (cf. Figure 8b), whereas second-order leakage is detected at several sample points during execution (cf. Figure 8c). We note that the total number of traces is limited to 10 million, due to the substantial measurement time required.

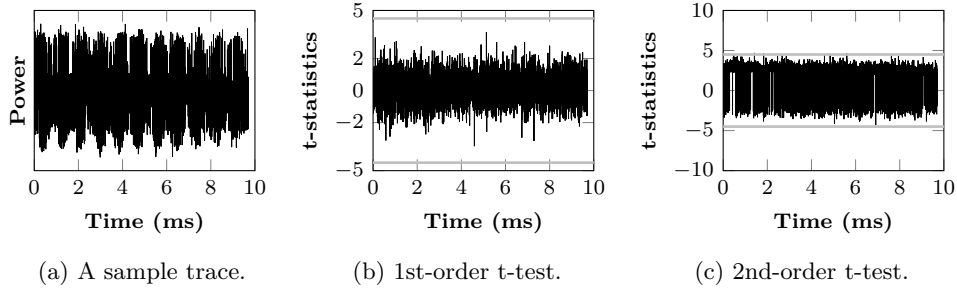


Figure 8: Experimental analysis of the masked AES (first-round) using 10 million traces.

5 Comparison

5.1 Area Overhead

We evaluated the area overhead of our implementation against various other implementations based on the Application-Specific Integrated Circuit (ASIC) netlist generated by Synopsys Design Compiler 2024.09 using the Nangate 45 cell library. The detailed comparison of the respective area overheads can be found in Table 1. For the memory adapter, we present the area usage using the basic variant, separating the memory overhead itself as the implementations differ quite significantly. Here we want to note that this module is completely interchangeable, when used in accordance with our findings in Section 3.6.1. For other implementations, like COCO [GHP⁺21], this overhead has been moved to the software, if their proposed custom memory is not used, or in case of eLIMInate [CPW24] the construction does not satisfy the conditions in Section 3.6.1, as they use a stateless adapter, which cannot be secure for any $d \geq 2$ under our memory assumptions. To evaluate these interchangeable implementation differences, we listed the area requirements for memory adapters separated from the rest of the system and provided some area requirements for different memories in various sizes.

For the memory, we want to highlight the flexibility of supporting arbitrary memories. COCO [GHP⁺21] secures the memory by a one-hot encoded address decoding, which is synthesized by us as an FF-based memory using their publicly available design. We synthesized this memory with the same flow as our designs, resulting in a moderate overhead compared to a plain FF-based memory, as shown in Table 2. As one-hot encoding does not scale to larger address spaces, the authors additionally propose to integrate SRAM blocks using partially one-hot encoded addresses [GHP⁺21, Section 4]. However, both proposed variants impose additional software constraints on where shares may be stored, and the variant without restrictions within an SRAM block requires internally stable word lines, for which the authors note that “it is generally not easy to tell if, or to what extent, an off-the-shelf SRAM block fulfills this requirement since they are full custom and partially analog blocks” [GHP⁺21]. Consequently, such a memory has to be custom designed or carefully characterized, and cannot be derived directly from a standard synthesis or memory compiler flow, which is why we only report the FF-based variant. In contrast, our adapter makes no assumptions on the internal structure of the memory, so any off-the-shelf SRAM, e.g., generated by a memory compiler such as OpenRAM or supplied as an external chip, can be used directly. As shown in Table 2, custom FF-based memories remain preferable for small memory sizes, being smaller and secure for arbitrary d , while for larger memories the adapter-based solution using SRAM scales considerably better, offering a compelling alternative for small $d_{\max}$.

The area required for random number generation depends on the exact Pseudo Random Number Generator (PRNG) and its parameters. As 32 random bits per instruction are

Table 1: Resource usage comparison in GE for the complete micro-architecture. The synthesis was performed by us using Synopsys with Nangate 45. We additionally synthesized the other designs, as the authors of COCO [GHP⁺21] used a different synthesis flow, namely Cadence with f130LL, and the authors of eLIMInate [CPW24] only stated FPGA resource usage in their work. The relative increase is in all cases reported in relation to the baseline variant. Main memory and the PRNG are excluded for all designs. Memory is compared separately as μm^2 per bit in Table 2 (our basic variant requires $d_{\max}$ additional memory blocks), and all designs require a PRNG for masked software, so including it would add the same overhead to all designs.

Area Comparison in GE							
Module	Baseline Ibex	Ours		COCO [GHP ⁺ 21]		eLIMInate [CPW24]	
ALU	1151	1839	+60%	1153	+0%	1109	-4%
Register File	9817	13,293	+35%	10,348	+5%	13,135	+34%
Memory Adapter	–	1282		–		1044	
Total (all modules)	21,108	32,050	+52%	22,630	+7%	27,033	+28%

typically provided for the software masking regardless of the chosen implementation for the memory adapter, we attribute the first 32-bit random word of each store instruction to the software masking, implying the overhead due to randomness for a first-order secure memory is zero. For higher security orders, $32(d_{\max} - 1)$ with $d \leq d_{\max}$ additional random bits are required per store instruction. The PRNG (unrolled Trivium [CMM⁺24]) used in our first-order implementation was synthesized using about 3500 Gate Equivalent (GE).

To provide a rough break-even point between both variants, we estimate the memory-optimized construction as OTP using a 32-bit counter as nonce, a $d_{\max}$ -probing secure Keccak- $f[200]$ as PRF [ZSS⁺21], and our unrolled Trivium as PRNG for its fresh randomness. For $2 \leq d_{\max} \leq 5$, the break-even point lies at about 9–12 kbit for FF-based memories and 50–65 kbit for SRAM, as the PRF area grows with roughly $(d_{\max} + 1)^2$, while the saved memory grows with $d_{\max} - 1$. For $d_{\max} = 1$, both variants store two bits per data bit, so the basic variant is always smaller. The code-based encoding of [TCG⁺25] comes with a comparable storage overhead, as each 32-bit word holds only 16 bits of data, but only for data processed by the ISE, while our adapter applies to every stored word. In contrast to our memory-optimized construction, the encoding requires no secret key and hence no masked PRF, which reduces the area overhead and suffices for the noisy leakage of individual bits considered in [TCG⁺25], but not under the worst-case memory assumptions of Section 3.6.1.

5.2 Latency Overhead

In terms of performance, the proposed implementation maintains the same Instructions Per Cycle (IPC) as the baseline Ibex for the basic memory adapter, while the latency of the memory-optimized variant depends on the nonce read, the chosen cipher or PRF and $d_{\max}$. Additionally, the integration of our countermeasures impacts the critical path delay, thereby reducing the maximum achievable clock frequency. Two primary modules potentially compete for the critical path. First, if the memory-optimized version is chosen as memory adapter, the masked cipher or PRF can introduce a significant propagation delay. Although the cipher or PRF may use full clock cycles, we still expect it to dominate the overall performance, if the implementation remains unrolled. As the cipher can be exchanged, we compare the overhead for the basic variant. Here, the system becomes limited by the half-cycle path through the ALU. In our architecture, the timing budget for the ALU is reduced by 50% compared to the baseline. To accurately frame this effect, it must be noted that while the timing budget is halved, the relevant logic is simultaneously

Table 2: Area comparison for the different memory variants. The FF-based variants were synthesized with Synopsys DC using the Nangate 45 nm library, while the SRAM based variant is a single-port (1RW) macro, port-matched to the FF-based variants, generated with OpenRAM using FreePDK45. The baseline and COCO variants correspond to the unprotected and the secure FF-based data memory of their public design [GHP⁺21], synthesized by us with the same flow. The COCO overhead at 32 and 64 kbit stems from a different cell mapping of the write enables by the synthesis tool. The conversion from GE to area was done using the Nangate 45 nm specifications. As this will omit the area required for routing, the area requirements for the FF-based variants are a lower-bound approximation. While not an exact measurement, as Nangate 45 is based on FreePDK45, the comparison should provide realistic proportions.

Area Comparison in $\mu\text{m}^2/\text{bit}$					
Memory Variant	Baseline	COCO		SRAM	
1 kbit	8.01	8.82	+10.1%	8.83	+10.3%
4 kbit	7.86	8.60	+9.5%	3.61	-54.0%
16 kbit	7.81	8.53	+9.3%	1.76	-77.5%
32 kbit	7.79	9.39	+20.5%	1.57	-79.8%
64 kbit	7.78	9.35	+20.1%	1.43	-81.7%

shortened. For instance, the entire instruction decoder is removed from this specific path. Consequently, the final frequency is dictated by the adder or multiplier logic. Synthesis results from Synopsys Design Compiler indicate a maximum frequency of 285 MHz (3.50 ns) for our implementation, compared to 400 MHz (2.50 ns) for the baseline, which corresponds to a performance reduction of 28.6%.

In contrast, other secure Ibex implementations ([GHP⁺21], [CPW24]) are more difficult to compare with, due to their combination of hardware and software countermeasures. Not only can those countermeasures be dynamically applied when required, they also can range in their efficiency, making direct comparisons difficult. If we only consider performance overhead stemming from a reduction in maximum clock frequency, which would require that the implementation is actually running at maximum frequency in hardware with no overhead due to software countermeasures, we can report a maximum clock frequency of 222 MHz (4.50 ns) for eLIMInate, caused by the critical path running through both Keccak [BDPA15] (0.99 ns) and the ALU (2.32 ns) plus additional logic (1.20 ns). For COCO, we optimistically assume the baseline speed of 400 MHz (2.50 ns). For the code-based ISE of Tian et al. [TCG⁺25], the authors report an area overhead of 8 % (resp. 10 %) in GE, including register gating and their PRG, at a maximum frequency of 83 % (resp. 98 %) of their baseline for the area-optimized (resp. frequency-optimized) encoder. As we could not find a publicly available implementation and these results stem from a different synthesis flow (Yosys with Nangate45 and a different baseline configuration), we cannot reproduce them with our flow and refrain from a direct comparison.

5.3 Security

When comparing the security of our implementation with other strategies, we can see similarities, but also differences. The primary advantage of our countermeasures is the protection against transitional leakage, while in other solutions this has to be taken care of in software. Notably, our implementation is also secure against instruction-opcode related transitions, while the constraints of other implementations do not seem to cover this source of leakage. Specifically, the first core constraint for COCO [GHP⁺21] and eLIMInate [CPW24] states that “Shares of the same secret must not be accessed within two successive instructions” [GHP⁺21]. While this covers most interactions, implementations using byte-granular shares instead of 32-bit words are vulnerable to horizontal probing, when

preceding or following instructions use additions or any other instruction with horizontal probe propagation (e.g. load instructions, due to their address computation).

The code-based ISE of Tian et al. [TCG⁺25] keeps all shares encoded outside of the ALU, which, similar to an encryption, protects registers, pipeline stages, and memory against transitional and glitch-related leakage at a low area cost, and reduces the Signal-to-Noise Ratio (SNR) of each share to about 2% of the SNR on their unprotected base core. For the decoders, the ALU, and the encoder, the security relies on a software requirement similar to COCO's, which is stated in a strict variant, i.e., the same requirement as in COCO, used for their formal verification, and in a variant relaxed to consecutive instructions of the ISE for practical implementations. Moreover, [TCG⁺25] considers each register to hold the encoding of a single Boolean share, such that the decoders and encoder may combine all bits of a word, whereas multiple shares within one word, as discussed above, are outside the setting considered there. As [TCG⁺25] targets Boolean masking, its ISE provides bit-wise logical and shift instructions on 16-bit data per 32-bit register, as the remaining 16 bits are reserved for the randomness of the encoding, while other operations, e.g., for arithmetic masking, are not covered by the ISE. Finally, [TCG⁺25] additionally aims to increase the attack complexity by reducing the SNR of each share. In contrast, we prove each countermeasure individually in the RR d -probing model and do not directly aim to decrease the SNR.

Regarding software constraints, compared to other implementations [GHP⁺21, CPW24, TCG⁺25], we only constrain the d -probing secure software to be upper bounded by $d \leq d_{\max}$ at synthesis time, which we proved to be necessary for arbitrary memory solutions. Otherwise, we do not impose any work on the software like rotating keys or nonce values in order to maintain security, as this is performed by the adapter in hardware.

6 Conclusion

We developed and refined hardware countermeasures designed to mitigate micro-architectural leakages across multiple components of a CPU. First, we replace selected registers with DCRs, which clear their registers in the second half of each clock cycle to eliminate transitional leakage. These DCRs effectively pipeline the entire core at twice the clock frequency. Second, using those DCRs, we created pre-charged control signals to disable all logic that is not required for the specified instruction. For example, this results in a secure multiplexer for the operand and instruction selection. Third, we secure the main memory by proving the necessity of a single constraint and then providing example implementations for our proposed solutions. As a result, we developed a fully-protected prototype CPU based on the Ibex RISC-V CPU which can execute d -probing secure software without degradation in the security order caused by the various micro-architectural effects known to us. First, we formally proved the security of our individual countermeasures and then evaluated the security of our prototype with an adjusted version of PROLEAD. Specifically, we extended PROLEAD to efficiently simulate the behavior of larger memory blocks, allowing us to simulate the entire CPU. In its evaluation of our prototype, PROLEAD detected no leakage under the RR 1-probing model while executing several leakage-prone software case studies. Finally, we conducted an experimental analysis on an FPGA board, revealing no first-order leakage for the first round of a masked AES implementation. Evaluating the synthesis results, we conclude that the performance of our CPU after applying our countermeasures, in terms of both area and latency, imposes only a moderate overhead compared to other state-of-the-art approaches that rely on compiler constraints to ensure that the executed masked software does not cause micro-architectural leakage. Hence, our prototype simplifies the development of masked software, as the software no longer needs to ensure that specific micro-architectural leakage conditions are avoided.

Acknowledgments

The work described in this paper has been supported in part by the German Research Foundation (DFG) through the project 535533866 (SSIMA), and by the National Research Center for Applied Cybersecurity ATHENE.

References

- [BBM⁺18] Davide Bellizia, Simone Bongiovanni, Pietro Monsurrò, Giuseppe Scotti, Alessandro Trifiletti, and Francesco Bruno Trotta. Secure double rate registers as an RTL countermeasure against power analysis attacks. *IEEE Trans. Very Large Scale Integr. Syst.*, 26(7):1368–1376, 2018.
- [BDPA15] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. *IACR Cryptol. ePrint Arch.*, 2015:389, 2015.
- [BGG⁺14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In Marc Joye and Amir Moradi, editors, *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, Lecture Notes in Computer Science, pages 64–81. Springer, 2014.
- [BGG⁺22] Roderick Bloem, Barbara Gigerl, Marc Gourjon, Vedad Hadzic, Stefan Mangard, and Robert Primas. Power contracts: Provably complete power leakage models for processors. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 381–395. ACM, 2022.
- [CBG⁺17] Thomas De Cnudde, Begül Bilgin, Benedikt Gierlichs, Ventzislav Nikov, Svetla Nikova, and Vincent Rijmen. Does coupling affect the security of masked implementations? 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*, Lecture Notes in Computer Science, pages 1–18. Springer, 2017.
- [CGD18] Yann Le Corre, Johann Großschädl, and Daniel Dinu. Micro-architectural power simulator for leakage assessment of cryptographic software on ARM Cortex-M3 processors. In Junfeng Fan and Benedikt Gierlichs, editors, *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*, Lecture Notes in Computer Science, pages 82–98. Springer, 2018.
- [CGP⁺12] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, and Praveen Kumar Vadnala. Conversion of security proofs from one leakage model to another: A new issue. In Werner Schindler and Sorin A. Huss, editors, *Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings*, Lecture Notes in Computer Science, pages 69–81. Springer, 2012.
- [CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International*

- Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.
- [CMM⁺24] Gaëtan Cassiers, Loïc Masure, Charles Momin, Thorben Moos, Amir Moradi, and François-Xavier Standaert. Randomness generation for secure hardware masking - unrolled Trivium to the rescue. *IACR Commun. Cryptol.*, 1(2):4, 2024.
- [CPW24] Hao Cheng, Daniel Page, and Weijia Wang. eLIMInate: a leakage-focused ISE for masked implementation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(2):329–358, 2024.
- [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.
- [EM25] Mathieu Escouteloup and Vincent Migliore. A hardware design methodology to prevent microarchitectural transition leakages. In Matthieu Rivain and Pascal Sasdrich, editors, *Constructive Approaches for Security Analysis and Design of Embedded Systems - First International Conference, CASCADE 2025, Saint-Etienne, France, April 2-4, 2025, Proceedings*, volume 15952 of *Lecture Notes in Computer Science*, pages 478–501. Springer, 2025.
- [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 Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1469–1486. USENIX Association, 2021.
- [GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In Begül Bilgin, Svetla Nikova, and Vincent Rijmen, editors, *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*, page 3. ACM, 2016.
- [GP99] Louis Goubin and Jacques Patarin. DES and differential power analysis (the "duplication" method). In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1999.
- [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*, *Lecture Notes in Computer Science*, pages 3–32. Springer, 2021.

- [HHB24] Johannes Haring, Vedad Hadzic, and Roderick Bloem. Closing the gap: Leakage contracts for processors with transitions and glitches. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(4):110–132, 2024.
- [ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.
- [KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
- [Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.
- [LMDM25] Daniel Lammers, Nicolai Müller, Siemen Dhooghe, and Amir Moradi. Constant-cycle hardware private circuits. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(4):172–214, 2025.
- [LMW14] Andrew J. Leiserson, Mark E. Marson, and Megan A. Wachs. Gate-level masking under a path-based leakage metric. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 580–597. Springer, 2014.
- [MGH19] Elke De Mulder, Samatha Gummalla, and Michael Hutter. Protecting RISC-V against side-channel attacks. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, page 45. ACM, 2019.
- [MLM24] Nicolai Müller, Daniel Lammers, and Amir Moradi. A deep analysis of two glitch-free hardware masking schemes SESYM and LMDPL. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(3):76–98, 2024.
- [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.
- [MM24] Nicolai Müller and Amir Moradi. Robust but relaxed probing model. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(4):451–482, 2024.
- [MMSS19] Thorben Moos, Amir Moradi, Tobias Schneider, and François-Xavier Standaert. Glitch-resistant masking revisited or why proofs in the robust probing model are needed. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(2):256–292, 2019.
- [MMT20] Lauren De Meyer, Elke De Mulder, and Michael Tunstall. On the effect of the (micro)architecture on the development of side-channel resistant software. *IACR Cryptol. ePrint Arch.*, 2020:1297, 2020.

- [MPG05] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-channel leakage of masked CMOS gates. In Alfred Menezes, editor, *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, volume 3376 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2005.
- [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.
- [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*, *Lecture Notes in Computer Science*, pages 529–545. Springer, 2006.
- [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*, *Lecture Notes in Computer Science*, pages 282–297. Springer, 2017.
- [RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010.
- [SM15] Tobias Schneider and Amir Moradi. Leakage assessment methodology - A clear roadmap for side-channel evaluations. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, *Lecture Notes in Computer Science*, pages 495–513. Springer, 2015.
- [SS06] Daisuke Suzuki and Minoru Saeki. Security evaluation of DPA countermeasures using dual-rail pre-charge logic style. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 255–269. Springer, 2006.
- [TCG⁺25] Qi Tian, Hao Cheng, Chun Guo, Daniel Page, Meiqin Wang, and Weijia Wang. A code-based ISE to protect Boolean masking in software. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(2):293–332, 2025.
- [TV04a] Kris Tiri and Ingrid Verbauwhede. A dynamic and differential CMOS logic style to resist power and timing attacks on security IC's. *IACR Cryptol. ePrint Arch.*, 2004:66, 2004.
- [TV04b] Kris Tiri and Ingrid Verbauwhede. A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France*, pages 246–251. IEEE Computer Society, 2004.

- [ZM24] Jannik Zeitschner and Amir Moradi. PoMMES: Prevention of micro-architectural leakages in masked embedded software. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(3):342–376, 2024.
- [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.
- [ZSS⁺21] Sara Zarei, Aein Rezaei Shahmirzadi, Hadi Soleimany, Raziye Salarifard, and Amir Moradi. Low-latency Keccak at any arbitrary order. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):388–411, 2021.