Toward Secure Compilation: Leakage Detection for Masked Implementations in Jasmin

Nicolai Schmitt[1], Sven Wroblewski[2], Fabio Campos[3], Andreas Heinemann[4]

Hochschule Darmstadt - University of Applied Sciences, Darmstadt, Germany
member of EUt+ (European University of Technology), first.last@h-da.de[1,2,4], campos@sopmac.de[3]

Preprint. Accepted at ICICS 2026, Fukui, Japan.

Abstract. Masking is a well-established software countermeasure against side-channel attacks, yet even algorithmically correct masked implementations can leak on real hardware once the compiler has performed instruction selection, register allocation, and stack allocation. Existing approaches either rely on leakage simulation, which is tied to a specific power model and computationally expensive, or on formal verification of the source program, which does not capture the effects introduced by the subsequent compilation stages. We address this gap from within the compiler and propose a leakage detection pass for the Jasmin language, integrated into its formally verified pipeline and operating on the intermediate representation before register and stack allocation. Rather than simulating power traces, the pass implements a configurable, microarchitecture-oriented leakage model that tracks the contact between shares, secrets, random values, and public values, making the root causes of the detected leakage explicit and enabling the detection of masking-order reductions. We validate the pass on 60 dedicated Jasmin test snippets covering all considered leakage sources and category combinations, and intend it as the foundation for a subsequent compiler stage that automatically removes the detected leakage, thus constituting a first step toward secure compilation.

Keywords: Side-channel analysis · Masking · Secure compilation · Jasmin · Leakage detection.

1 Introduction

Side-channel attacks pose a significant risk to the security of computing systems, as they can undermine a system that is secure at the algorithmic and cryptographic level. The risk originates from attack vectors on the hardware itself rather than from algorithmic weaknesses. By monitoring environmental parameters such as electromagnetic emissions or the power consumption of a device, an attacker can infer private information such as cryptographic key material [13].

Because embedded and IoT devices are frequently deployed in physically accessible environments, they are particularly exposed to this class of attacks.

Mitigating this risk requires countermeasures at multiple levels, and several approaches exist to harden devices against side-channel attacks and to reduce side-channel leakage. Countermeasures can be implemented in hardware or software, though hardware-based solutions are not always feasible, for instance, when a system is built on commodity off-the-shelf processors whose design cannot be modified. A well-known countermeasure that can be applied purely in software is masking, which splits every secret value into several randomized shares such that any proper subset of the shares remains statistically independent of the secret. However, even a masked implementation that is correct at the algorithmic level can still leak on real hardware because the device may exhibit leakage that differs from the behavior expected at the architectural level. Such deviations originate from microarchitectural implementation details such as hidden registers, register reuse, pipeline stages, glitches in combinational logic, and shared memory buses, which can recombine shares that the masking scheme was designed to keep separate [15], making security guarantees difficult to establish in practice.

Verifying whether a masked implementation is secure on a given target is therefore particularly challenging for two reasons. First, the compiler sits between the source program and the executed machine code. Compiler transformations such as instruction selection, register allocation, and stack allocation can cause values to share a register, pipeline stage, or memory bus, reintroducing leakage that the masked implementation was designed to prevent. Second, the relevant leakage behavior is specific to the microarchitecture, so a result obtained for one processor does not necessarily carry over to another. Existing tools address both challenges either through leakage simulation, which is tied to a particular power model and can be computationally expensive, or through formal verification of the source program, which does not capture the effects introduced by subsequent compilation stages. Neither approach fully accounts for the compiler- and microarchitecture-induced leakage described above.

This work approaches the problem from within the compiler. We propose a leakage detection pass operating on the intermediate representation before architecture-specific lowering and register allocation. The pass implements a configurable, microarchitecture-oriented leakage model that accounts for register write-backs, the recombination of operands in the ALU, the memory bus, memory overwrites, and the CPU pipeline. By tracking the interaction between shares, secrets, random values, and public values throughout the program, the analysis captures masking-order reductions that are difficult to detect with simulation-based approaches alone. The configurable model can be tuned to the available knowledge of the target hardware, balancing analysis precision against computational cost. Beyond detection, the pass is designed to drive a subsequent compiler stage that automatically eliminates the identified leakage, with the goal of fully automated secure compilation.

We instantiate this approach in the Jasmin compiler [3], whose formally verified compilation pipeline and close correspondence to the underlying hardware make it a natural fit for the analysis described above.

Contributions. In summary, this work makes the following contributions:

1. We propose a configurable, microarchitecture-oriented leakage model that captures the leakage sources relevant for masked implementations, including register write-backs, ALU recombination, the memory bus, memory overwrites and pipeline forwarding.
2. We implement this model as a compiler pass within Jasmin's formally verified pipeline, operating on the intermediate representation before register and stack allocation.
3. Beyond the model, we employ a formal analysis that tracks the interaction between shares, secrets, and random values, enabling the detection of masking-order reductions and making their root causes traceable.
4. We evaluate the detection pass on known leakage cases, confirming soundness and practical applicability on realistic masked implementations.
5. We make the implementation publicly available in a repository[1] and thereby contribute to the open-source community.

The remainder of this paper is structured as follows. Section 2 provides background on the Jasmin language and compiler. Section 3 surveys related work on compiler-assisted and software-level side-channel countermeasures. Section 4 introduces the leakage model and attacker model that underpin the analysis. Section 5 describes the implementation of the leakage detection pass within the Jasmin compiler pipeline and validates it against a structured set of test cases. Section 7 concludes the paper and outlines directions for future work.

2 Background

This section provides the necessary background on masking and the Jasmin framework, both of which form the technical foundation of this work.

2.1 Masking

Masking is an algorithmic countermeasure that eliminates the statistical dependency between physical side-channel leakage and sensitive intermediate variables. The principle relies on secret sharing: a secret value x is split into $d+1$ shares (x_i) satisfying $x = x_0 \star x_1 \star \cdots \star x_d$ for some algebraic group operation $\star$, where d is the masking order. The shares are drawn uniformly at random, ensuring that any subset of at most d of them is statistically independent of x.

[1] https://gitlab.com/ATHENE-LEAK/Jasmin-Leakage-Detection-Extension

2.2 Jasmin language

High-assurance cryptography aims to provide formally verified guarantees about the correctness and security of cryptographic implementations, bridging the gap between mathematical proofs and efficient executable code. Jasmin [3,4] is a high-level assembly language, often described as "assembly-in-the-head", designed specifically for this purpose. It provides zero-cost abstractions such as variables and arrays, while retaining fine-grained programmer control over low-level details including instruction selection and loop compilation strategies. The Jasmin compiler formally guarantees that both the semantics [3] and the constant-time properties [10,7] of safe source programs are preserved in the compiled output. Beyond correctness and timing security, the language further offers primitives for implementing Selective Speculative Load Hardening (selSLH) with a type system to verify correct deployment of Spectre v1 countermeasures [6], as well as compiler support for stack zeroization to ensure secrets are effectively erased upon function return [8]. The applicability of Jasmin to high-speed, high-assurance cryptographic primitives has been demonstrated through several case studies, including ML-KEM [5,2], where the implementation was proven in EasyCrypt [9] to securely refine a formal IND-CCA specification.

3 Related Work

A substantial body of work has investigated software-level countermeasures against power side-channel leakage, spanning compiler-based transformations, register-allocation strategies, masking-aware code transformations, and architecture-level leakage analysis tools.

ROSITA [19] is a code rewrite engine that takes leakage estimates from ELMO*, a power consumption emulator for ARM processors, and iteratively rewrites assembly code until residual leakage is minimized. ROSITA++ [18] extends this approach to higher-order leakage, evaluating it against second- and third-order masked cipher implementations. Along similar lines, Zeitschner et al. [22] propose PoMMES, a register-allocation-based countermeasure grounded in a hardware-independent leakage model and implemented in GCC for ARM, targeting microarchitectural leakage introduced by the compiler's register assignment decisions. Wang et al. [21] pursue a comparable goal within the LLVM compilation pipeline, constraining the register allocator to avoid operand patterns that produce exploitable Hamming distance leakage, with an evaluation on x86. In [20], the authors build on this direction with SecCG, a constraint-based secure compilation mechanism in LLVM that combines register promotion with leakage-aware code generation and has been evaluated on ARM Thumb and MIPS32, producing optimized secured code that runs up to eight times faster than its non-optimized counterpart.

Turning to the RISC-V platform, Gigerl et al. [11] harden the SweRV [12] core against side-channel attacks and derive a set of software rules for minimizing microarchitecture-aided leakage, offering a systematic characterization of how pipeline effects and operand reuse interact with masked software. Sajadi et

al. [16] complement this with an experimental comparison of Boolean masking, instruction shuffling, and dummy instruction insertion on RISC-V-based SoCs. Their TVLA-based results confirm that the effectiveness of software masking depends heavily on microarchitectural factors such as pipeline behavior and register reuse. In a similar vein, Schmitt et al. [17] propose a simulation-based leakage evaluation methodology for the RISC-V platform, enabling software-level leakage assessment without requiring physical measurements. Mainka and Papagiannopoulos [14] further investigate the combination of masking and shuffling on a 32-bit RISC-V platform using Ascon as a case study, systematically quantifying the combined security gains, performance overhead, and randomness costs of the two techniques.

Most closely related to leakage detection and root-cause analysis, Adhikary et al. present ARCHER [1], an architecture-level simulator for side-channel analysis tailored to RISC-V processors. Taking a binary as input, ARCHER pairs TVLA-based leakage assessment with data flow analysis to pinpoint leakage locations and explain their microarchitectural causes, producing interactive visualizations and detailed reports. To the best of our knowledge, ARCHER is the first architecture-level side-channel analysis tool designed specifically for RISC-V, with the explicit goal of guiding the development of cryptographic implementations resistant to power side-channel attacks.

Our method differs from the approaches above in two key respects. First, unlike simulation- and measurement-based tools, it requires neither a power model nor physical measurements, operating instead through a formal share-contact analysis on the compiler intermediate representation. Second, unlike register-allocation-based approaches such as PoMMES and SecCG, it intervenes before register and stack allocation, reasoning about leakage independently of the target architecture and enabling subsequent compiler passes to automatically eliminate the detected leakage.

4 Concept

This section introduces the leakage model and the attacker model that underpin the proposed detection pass.

4.1 Leakage Model

The leakage model implemented in this work serves two principal goals. First, it preserves computational efficiency by making properties that depend on the microarchitecture configurable. For instance, if the target hardware leaks Hamming weights, random precharging may not be considered a sufficient protection for secrets, whereas if the hardware predominantly leaks Hamming distances, random precharging can be employed as a protective mechanism. This configurability accounts for the fact that the target hardware may already implement countermeasures at the hardware level. Second, the model leverages formal analysis techniques rather than simulation-based approaches. Through share-contact

tracking, the model captures masking-order reductions, which would be difficult to capture through simulation alone. This makes the model suitable for reasoning about higher-order masked implementations.

Leakage Sources. The leakage model developed as part of this work takes the following microarchitecture oriented sources of leakage into account:

Register writeback Leakage can occur when registers are overwritten. The leakage could be the Hamming distance between the previous value in the register and the new value written to the register. This type of leakage leads to recombination of both values, which could be critical for masked implementations. Depending on the concrete implementation, also leakage in form of the Hamming weight of the new value could occur.

Calculation inputs When instructions are executed and calculations are performed in the ALU, combinatorial leakage of the input values is expected. Also, depending on the concrete implementation, the Hamming weight of the inputs could leak.

Pipeline forwarding Most CPU implementations, such as the Ibex RISC-V core[2], feature a pipeline design, often consisting of an instruction-fetch, decode+execute and writeback stage. The pipeline design can produce leakage when values are transferred between pipeline stages and hidden registers are overwritten.

Memory bus When data is written to or read from the RAM, leakage can occur from the memory bus. This leakage can be the Hamming weight of the transferred value or the Hamming distance between the transferred value and the previous read or written value overwritten in hidden registers in the memory architecture. The actual leakage depends strongly on the implementation of the memory bus. We assume combinatorial leakage between read and read transactions, write and write transactions, as well as between read and write transactions and between the data and address of a memory access, which is configurable in our model. On the hardware side, this assumes multiple hidden registers of which some are shared between read and write and some are dedicated to only read and only write transactions. The combinatorial leakage between address and data accounts for the detection of unintended data-dependent memory accesses.

Memory Overwrite When memory is overwritten, the Hamming distance between the old and new value can be leaked, which can result in recombinations of shares in masked implementations.

In comparison to the *CPU-Independent leakage model* used in PROLEAD_SW [23] and PROLEAD_SW_RV [17], our model is more relaxed in regards to leakage from the register-file. We do not expect that an attacker can extract useful data through glitches occurring when addressing the register file. Therefore, our model lacks a leakage source that can be directly compared to the *Vertical Probe* of the PROLEAD_SW leakage model.

[2] https://github.com/lowrisc/ibex

In order to detect leakage, in principle data are classified into different security categories (PUBLIC, RANDOM, SHARE and SECRET) and possible paths of data flow are then analyzed. Thereby, two forms of contact (*overwrite* and *calculation*) are tracked slightly different, which are explained in the following. During the execution of one instruction, multiple contacts can occur through different sources of leakage.

Overwrite Contact. Overwrite contacts occur, whenever a register, that may be an architectural or a hidden register, is overwritten. In particular, this contact is tracked during writeback to a register, in the memory bus hidden states, in the pipeline hidden states and during overwrite of a memory cell. Contacts between both instances of categories are tracked per category instance, which results in category-dependent effects. In addition the category instance at the destination (register, memory-cell or hidden register) is replaced by the category instance it is overwritten with. These effects are described in the following.

PUBLIC - PUBLIC No leakage effect. Reason: both values are already known to the attacker.

PUBLIC - RANDOM The random category instance gets added a public contact. Reason: the overwrite of a random with a public or a public with a random could result in a measurable Hamming distance and therewith leak information about the random value. If the random category instance reaches public-random-contact exhaustion, it is marked as leaked. Random values marked as leaked further behave similar to public. Thereby, the not adequately protected data are reported.

PUBLIC - SHARE The share category instance gets added a public contact. If the share category instance reaches public-share-exhaustion, the share is set to leaked. Reason: similar to the *PUBLIC - RANDOM* case, information about the share can leak.

PUBLIC - SECRET The secret category instance gets added a public contact. If secret public contacts are exhausted, the secret is set to leaked. Reason: similar to the *Public - Random* case, information about the Secret can leak.

RANDOM - RANDOM No leakage effect.

RANDOM - SHARE No leakage effect.

RANDOM - SECRET No leakage effect.

SHARE - SHARE On share share contacts the effect depends on the share groups. If the share category instance is the same, then there is no leakage effect. If the share group is the same, but the share index differs, then share-share contacts are added to both category instances. If the shares are from different share groups, no leakage effect takes place. Reason: based on the Hamming distance, combinatorial leakage could leak information about a shared value.

SHARE - SECRET No leakage effect.

SECRET - SECRET No leakage effect.

Besides this normal behavior, the leakage model can be configured to discourage any unmasked processing of secrets, which means that it can be set to

random precharging is sufficient / insufficient to protect a secret. If set to insufficient, any contact with a secret category instance will set the secret to leaked. This is also true for the *Secret - Secret* case.

Calculation Contact. Unlike overwrite contacts, which only generate category-dependent effects, calculation contacts additionally determine an output category. The resulting category from a calculation contact is then the input to an overwrite contact. The determination of the output category of calculations is described in the following.

Public - Public Output: new public category instance. Reason: The output depends on two values known to the attacker, which makes it also known to the attacker.

Public - Random Output: the random category instance. Reason: When an operation has at least one random input, the output is expected to be random.

Public - Share Output: the share category instance. Reason: The output requires the same protection as the input share. An example could be a modulo or shift operation.

Public - Secret Output: the secret category instance. Reason: The output requires the same protection as the input secret. An example could be a modulo or shift operation.

Random - Random Output: new random category instance. Reason: The output depends on two random values. Such calculations are expected when random values are derived from random values as e.g., a pseudo random value from a seed.

Random - Share Output: the random category instance. Reason: When an operation has at least one random input, the output is expected to be random.

Random - Secret Output: the random category instance. Reason: When an operation has at least one random input, the output is expected to be random.

Share - Share Output: depends on the share category instances. If both data are classified identical (same share group and same share index), then this share class is returned. Reason: The operation only depends on this share. However, this could potentially result in misclassifications, when the operation clears the variable, as further explained in Section 5.3. If both shares are in the same share group, but are not the same share, a new share with a new share index is added to the share group. Each share gets added share contacts with the other two shares. The new share category instance is returned. Reason: The output should further allow tracking of the exhaustion of the share group. If both shares are from different groups, a new random category instance is returned. Reason: Individual shares from different share groups are random.

Share - Secret Output: the share category instance. Reason: An individual share behaves random. The computational output of an operation between

a secret and a share further remains subject to the security classification of the share.

Secret - Secret Output: a new secret category instance. Reason: The result of a computation with two secrets further underlies the security classification of a secret.

By incorporating these contact rules, the leakage model embodies random precharging and randomness reuse in a configurable manner.

Random Precharging. When a register contains a value known to the attacker, designated as public, a random value can be loaded into the register before loading a secret value. The random value masks the Hamming distance between the public and the secret value. Random precharging can be disabled in the model. If disabled, an unmasked secret loaded into a register is directly considered leaking based on its Hamming weight. The use of random precharging should be carefully chosen based on the properties of the target hardware and is enabled by default.

Randomness Reuse. When a random value comes into contact with a public value, either during processing or through an overwrite, an attacker could gain knowledge about the random value. This leads to a degradation of the protective property of the random value. Therefore, the contact points of random inputs with public values are tracked throughout the analysis. The more leakage sources are enabled, the more frequently random values come into contact with public values. A threshold of public contacts per random value can be configured. The threshold shall be set carefully in conjunction with the enabled leakage sources. A higher threshold can also be used to set a tradeoff between security and resource consumption, especially if randomness is limited on the target platform. Beyond the amount of public contacts per random value, it is also possible to set the amount of allowed public contacts per share and per secret.

Attacker Model. The attacker model assumes that the attacker can re-run the target. Every time a variable comes into contact with a public value, either through processing or through an overwrite, the attacker may gain information about that variable. Assuming secrets remain constant across executions, they are considered leaked whenever they come into contact with a public value. Depending on the hardware and the configurable options described above, the attacker can learn from Hamming distances and optionally from Hamming weights. The attacker cannot influence the source of randomness. The cryptographic security of implementations is not considered within this model.

5 Implementation

The leakage detection mechanism described in Section 4 is implemented as an additional compilation pass of the Jasmin compiler[3]. The pass is invoked

[3] Repository: https://github.com/jasmin-lang/jasmin, (Commit 09fdb46 (January 5, 2026)).

through the binary `jasmin-pl` and operates on the intermediate representation of the Jasmin compiler, after type checking and before assembly generation. The implementation is written in `OCaml` and integrates with the existing unit layout of the Jasmin compiler. The complete leakage detector is contained in a single unit, `Pl_checker_forward`, which exposes the entry point `Evaluator.evaluateFunctions` that receives the typed program together with a list of functions to analyze and returns the corresponding leakage reports.

5.1 Integration into the Jasmin Compiler

The Jasmin compiler is a multi-stage pipeline that progressively transforms the source program toward the target architecture. The leakage detection pass is placed in the same position as the existing speculative constant-time (SCT) checker, i.e., before the lowering of architecture-specific operations and before stack allocation and register allocation. At this stage the program is fully resolved on the semantic level but not yet bound to concrete hardware registers, which preserves the symbolic reasoning required by the leakage model while still capturing the data flow that the later compilation stages will translate to assembly.

Function annotations expressing the security category of each input and output (cf. Section 4) are attached to the Jasmin source through the standard annotation mechanism of the language and are interpreted by the dedicated `AnnotationManager` module of the checker.

5.2 Data Model

The data model is organized into a set of `OCaml` modules whose dependencies are summarized in Fig. 1 in the Appendix. The model is centered around two notions: the *category* of a value and the *reason* that led to the value being assigned to that category.

Categories. A category describes the role a value plays with respect to the leakage model. The implementation distinguishes between a homogeneous category (`Hcategory`), which groups all variables that share a base class (one of Secret, Share, Public, Random) and a user-provided group name, and an individual category (`Icategory`), which represents an individual member of a homogeneous group together with bookkeeping data such as the number of public contacts observed for that value and the indices of shares of the same group that have been combined with it. Categories are not bound to a particular variable but are referenced from variables through their history items, keeping the model independent of the control-flow path on which the values were produced.

Variable Histories. For each variable that has been touched by the program, the checker maintains a chronologically ordered list of state items, organized in a `VarHistory` object. Each item, `VarHistoryItem`, references an input reason list and an output reason that together characterize how the value of the variable

came to be. Histories are kept per variable identifier and per scope. The scope distinguishes between the value that is currently held in a variable seen as a register (SC_DIRECT) and the value that is reachable through a variable seen as a pointer (SC_PTR_DEST). The variable identifier itself is a tagged union (`VarID`) that captures the different ways a value can be addressed in Jasmin, in particular as a register variable, as an immediate, and as a register/immediate combination expressing a pointer with an index.

Reasons. Whenever an intermediate operation creates or modifies a value, the checker records a `ReasonItem`. A reason carries the probe type and the leakage type observed at that operation, the resulting individual category, a list of input reasons, the categories whose security state is degraded by the operation, the categories which gain public contacts through it, and the categories which gain share-share contacts. The location in the original source code at which the reason was created is attached to the reason as well, in order to allow human-readable findings to be emitted later on. Top-level reasons are collected into the manager's flattened reasons list, which is used to derive the final report.

History Manager. The `VarHistoryManager` module is the central state container of the checker. In addition to the per-variable histories described above, it holds the histories that model the read, write and shared paths of the memory bus (each modeled as a dedicated synthetic `VarHistory`), the current set of homogeneous and individual categories, the configuration controlling the leakage model (cf. Section 5.6), the flattened list of unique reasons that will be inspected for findings, and the data structure used to model the CPU pipeline (cf. Section 5.5). The manager exposes operations to perform read- and write-transactions, to combine two reasons via the central `processIntermediate` routine, and to introduce, look up, and update categories on demand.

5.3 Limitations

Our implementation tracks data flow through computations abstractly, without considering the specific operation performed. This leads to two limitations. First, when a computation is performed on two identical operands, the output category may be misclassified. For instance, when a secret is XORed with or subtracted from itself, the result is always zero and should be classified as public; however, the implementation assigns an output category based solely on the input categories, not the operation, and would classify the result as secret. Mitigating this limitation requires an operation-dependent evaluation that accounts for the specific semantics of each instruction. Second, when an array element is accessed using a computed index, the ability to identify the target element is limited. A computed index is resolved through the output category instance of the computation that produced it, which is abstract from the actual value. This can cause two computations yielding the same value to be assigned distinct category instances, and therefore to resolve different memory locations, or conversely, two computations yielding different values to share a category instance and resolve

the same location. When memory that has not been previously written is read or overwritten, its content is classified as public. Mitigating this limitation likewise requires a more precise, value-aware evaluation of index computations. Both limitations are rooted in the abstraction of computations from their concrete values. As the implementation already tracks all changes to every variable, extending it to a more precise evaluation is feasible and is left as future work.

5.4 Evaluation Procedure

The evaluation procedure traverses each function of the analyzed program. For each function, an initial `VarHistoryManager` is created and pre-populated by the `AnnotationManager` with annotation-derived categories for the function's input variables. The traversal then walks the body of the function statement by statement using the `eva_cmd` routine, which dispatches on instruction kind to `eva_expr` for right-hand-side expressions and `eva_lval` for left-hand-side targets.

Expression Evaluation. `eva_expr` handles the recursive evaluation of right-hand-side expressions. Variable accesses retrieve the latest history item from the corresponding `VarHistory`. Constants are mapped to the public category. Binary and unary arithmetic operations call `processIntermediate` on the reasons of their operands, which determines the resulting category as well as any side effects on the security state of the operands. Memory accesses through a pointer variable trigger a read transaction on the corresponding memory bus history. Array accesses with a non-constant index optionally couple the address bus reason with the data bus reason, depending on the configuration described in Section 5.6.

Left-Hand-Side Evaluation. `eva_lval` writes the reason produced by the evaluation of the right-hand-side expression into the variable, array slot or memory location named on the left-hand side. For array writes with non-constant index the address bus probe can again be enabled or disabled through configuration. For pointer writes the write is also propagated through the memory bus write history in order to enable the memory overwrite probe.

Control Flow. Branching constructs (`if`, `for`, `while`) are handled by walking the branches independently and accumulating the resulting per-branch states. A dedicated `RecursionContainer` module owns the currently evaluated state together with a list of side-branches and provides a deep-copy operation that is used at every branch point. `for` loops are unrolled into three representative cases of zero, one and two iterations to capture the loop-carried effects without sacrificing performance. `while` loops are handled analogously. Function calls are inlined into the analysis: the called function is evaluated within a fresh sub-state of the manager, with the lazy transition flag enabling the deferred propagation of register-level reasons across the call boundary.

5.5 Probe Models

The checker models the leakage sources identified in Section 4 as a family of probes. The probe type that produced a given reason is recorded inside the reason and is later reflected in the corresponding finding.

Register and Calculation Probes. Reasons produced by direct write-backs into a destination register carry the PT_REGISTER probe type. Reasons produced by the recombination of two operands inside an arithmetic operation carry the PT_CALCULATION probe type. The combination logic is parameterized by the probe type so that the same arithmetic kernel can be used both for register transitions and for arithmetic recombinations.

Memory Bus Probes. Three synthetic histories model the memory bus: a read history, a write history and a shared history. Read transactions update the read history and the shared history; write transactions update the write history and the shared history. The shared history captures the leakage that may occur on hardware where read and write paths share routing resources. Each probe is reported under a dedicated probe type (PT_MEM_BUS_R, PT_MEM_BUS_W, PT_MEM_BUS_RW). An additional probe type PT_MEM_BUS_ADDR is used to report leakage from the coupling between memory addresses and data on indexed memory operations.

Memory Overwrite Probe. Whenever a memory cell is overwritten through a pointer write, the reason of the previous content is combined with the reason of the new content to detect leakage caused by the transition. The resulting findings are reported under the PT_MEM_OVER probe type.

Pipeline Probe. The CPU pipeline is modeled by the `PipelineProbe` module. The probe consists of a configurable number of pipeline stages, each of which holds the list of input reasons and the list of output reasons of the instructions that have been issued in that stage. On every modeled cycle the pipeline is shifted, discarding the oldest stage and introducing a new empty stage at the front. Four independent comparisons are then performed between the values in the freshly populated stage and the values in the older stages: an outputs-to-outputs comparison, an inputs-to-inputs comparison, and the two cross-comparisons of inputs-to-outputs and outputs-to-inputs. Each comparison is reported under its own probe type (PT_PIPELINE_OUTPUTS, PT_PIPELINE_INPUTS, PT_PIPELINE_IN_OUT, PT_PIPELINE_OUT_IN). All four comparisons can be enabled or disabled independently, which allows the pipeline probe to be adapted to the specific microarchitecture of the target CPU.

5.6 Configurability

The behavior of the checker is controlled by the `Configuration` record. Its options can be grouped into four groups.

Randomness budgeting. The fields `max_public_random_contact`, `max_public_share_contact` and `max_public_secret_contact` bound the number of times a random, share, or secret category may come into contact with public values before being considered exhausted.

Leakage model selection. The flags `allow_rp_mask`, `primary_share_public_degradation` and `primary_share_public_high_contact_report` select the leakage model variant described in Section 4, specifically the random-precharge masking option and the handling of shares in contact with public values.

Probe selection. The flags `probe_address_data_bus_coupling`, `probe_pipeline_outputs`, `probe_pipeline_inputs`, `probe_pipeline_cross_out_in` and `probe_pipeline_cross_in_out` enable or disable the corresponding probe families, allowing the model to be aligned with the microarchitecture of the targeted hardware. The integer field `probe_pipeline_length` controls the depth of the pipeline probe.

The default configuration is conservative in the sense that, besides `probe_address_data_bus_coupling`, which is a less common leakage source [15], all probes are enabled and a single public contact is sufficient to exhaust a random or a share. This default reflects the assumption that the target hardware microarchitecture is not known in detail; once detailed information about the target is available, individual probes can be disabled to better match the hardware behavior.

5.7 Findings and Report

After the function bodies have been traversed, two post-processing stages produce the report. The *incident* stage walks the flattened list of unique reasons collected by the manager and emits a finding for each reason associated with a detected leakage. The *exhaustion* stage iterates over the individual categories and emits a finding whenever a category has exceeded its contact budget or all shares in its group have been recombined. Each finding is represented by the `Finding` record and contains the finding type, the base category and the variable group concerned, the probe and leakage types, the location of the operation that triggered the finding, the branch information identifying the control-flow path in which the finding was observed, and a human-readable description that is generated from the underlying reason. The `Report` module serializes findings as structured text.

6 Validation

To validate the implementation we developed an extensive set of test cases consisting of Jasmin code snippets that are sorted into the following test groups:

functional The functional test group verifies the functional correctness of the static analysis in the Jasmin intermediate language representation.

probetypes The probetypes test group verifies whether leakage is captured correctly by all leakage sources taken into account by our leakage model.

baseclass/calculation The baseclass/calculation test group verifies whether the calculation contact logic operates correctly for all possible combinations of base categories.

baseclass/exhaustion The baseclass/exhaustion test group verifies whether the exhaustion logic works as expected for all base categories.

baseclass/overwrite The baseclass/overwrite test group verifies whether the overwrite contact logic operates correctly for all possible combinations of base categories.

A full list of the test cases is given in the Appendix in List 1.

In addition, the test snippets, their configurations, and the corresponding findings reports produced by each test snippet will be made available alongside the implementation in a public repository[4]. The following example demonstrates how leakage is found and how it can be prevented when overwriting a register. The JASMIN snippet `reg_over.jazz`, which is given in Listing 1 is analyzed using `jasmin-pl --print-all ./reg_over.jazz`.

```
1  #[pl = "nDval1 * nDval1 * rDval * rDval -> pDval"]
2  export fn demo (reg u32 s1d1, reg u32 s2d1,
3          reg u32 rnd1, reg u32 rnd2) -> reg u32
4  { reg u32 temp_a;
5    reg u32 temp_b;
6    temp_a=rnd1;
7    temp_a=s1d1;
8    temp_b= rnd2;
9    temp_a=s2d1; //leak. through overw. contact
10   return rnd1; }
```

Listing 1: Overwrite example

```
1  #[pl = "nDval1 * nDval1 * rDval * rDval -> pDval"]
2  export fn branchtest_a (reg u32 s1d1, reg u32 s2d1,
3             reg u32 rnd1, reg u32 rnd2) -> reg u32
4  { reg u32 temp_a;
5
6    temp_a=rnd1;
7    temp_a=s1d1;
8    temp_a= rnd2;
9    temp_a=s2d1; // no leak. through overw. contact
10   return rnd1; }
```

Listing 2: Fixed overwrite example

```
1  Incidents (2):
2  [INCIDENT_SHARE_SHARE] SHARE group=nDval1 idx=1 probe=REGISTER leak=HD
3  public-contacts=0 share-share-contacts=, 0  loc=row: 9, col: 246,
4  fname: demo/reg_over.jazz . branch= -> 0  reason=
5  [INCIDENT_SHARE_SHARE] SHARE group=nDval1 idx=0 probe=REGISTER leak=HD
6  public-contacts=0 share-share-contacts=, 1  loc=row: 9, col: 246,
7  fname: demo/reg_over.jazz . branch= -> 0  reason=
8  Exhaustions (1):
9  [EXHAUSTED_SHARES] SHARE group=nDval1 idx=0 probe=NONE leak=NONE public-
10 contacts=0 share-share-contacts= loc=row: -1, col: -1, fname:
   . branch= -> 0  reason=
```

Listing 3: Findings of register overwrite leakage example.

The snippet can produce leakage findings as shown in Listing 3, indicating an overwrite contact at Line 9. This leak is attributed to both shares of the share group "nDval1", which is subsequently marked as exhausted. The issue can be fixed by writing randomness into the variable "temp_a" before writing the second share into it, as demonstrated in the Listing 2. This fix removes potential leakage from register overwrites, as the destination variable is now cleared between writing two shares. The remaining test cases follow a similar structure and test different aspects and leakage sources. Comparable fixes can be applied in many cases.

[4] https://gitlab.com/ATHENE-LEAK/Jasmin-Leakage-Detection-Extension

In total, 60 test cases were developed across the five test groups, covering all leakage sources and all base category combinations for both contact types. The pass produces the expected findings in every case, confirming the correctness of the contact logic and the completeness of the leakage source coverage. The full test case list, configurations, and findings reports are provided in Appendix 1.

7 Conclusion and Future Work

We propose a configurable, hardware-aware leakage model designed to analyze masked implementations within the Jasmin compiler pipeline, directly exposing the root causes of detected leakage. The model can be tuned to reflect available knowledge of the target hardware, balancing security against resource consumption. We implement the model as a static analysis pass to the Jasmin compiler and validate its correctness against a structured set of test cases. Two directions for future work follow naturally from this foundation. First, reducing the abstraction of computations from their concrete values would address the current misclassification limitations and improve analysis precision. Second, consuming the pass output directly within the compiler would enable automatic elimination of detected leakage, allowing hardened assembly to be generated without manual intervention and representing a concrete step toward fully automated secure compilation.

References

1. Adhikary, A., Basurto-Becerra, A., Batina, L., Buhan, I., Chatterjee, D., van Hoek, S., Sanfelix Gonzalez, E.: ARCHER: Architecture-Level Simulator for Side-Channel Analysis in RISC-V Processors. In: Savaş, E., Moradi, A., Leander, G. (eds.) Lightweight Cryptography for Security and Privacy – 6th International Workshop, LightSec 2025. pp. 157–177. Lecture Notes in Computer Science, Springer, Istanbul, Türkiye (2025). https://doi.org/10.1007/978-3-032-15541-2_9
2. Almeida, J.B., Arranz-Olmos, S., Barbosa, M., Barthe, G., Dupressoir, F., Grégoire, B., Laporte, V., Léchenet, J.C., Low, C., Oliveira, T., Pacheco, H., Quaresma, M., Schwabe, P., Strub, P.Y.: Formally Verifying Kyber – Episode V: Machine-Checked IND-CCA Security and Correctness of ML-KEM in EasyCrypt. In: Reyzin, L., Stebila, D. (eds.) Advances in Cryptology – CRYPTO 2024. pp. 384–421. Lecture Notes in Computer Science, Springer, Santa Barbara, CA, USA (2024). https://doi.org/10.1007/978-3-031-68379-4_12
3. Almeida, J.B., Barbosa, M., Barthe, G., Blot, A., Grégoire, B., Laporte, V., Oliveira, T., Pacheco, H., Schmidt, B., Strub, P.Y.: Jasmin: High-Assurance and High-Speed Cryptography. In: Thuraisingham, B., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1807–1823. ACM, Dallas, TX, USA (2017). https://doi.org/10.1145/3133956.3134078
4. Almeida, J.B., Barbosa, M., Barthe, G., Grégoire, B., Koutsos, A., Laporte, V., Oliveira, T., Strub, P.Y.: The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. In: 2020 IEEE Symposium on Security and Privacy

(SP 2020). pp. 965–982. IEEE, San Francisco, CA, USA (2020). https://doi.org/10.1109/SP40000.2020.00028
5. Almeida, J.B., Barbosa, M., Barthe, G., Grégoire, B., Laporte, V., Léchenet, J.C., Oliveira, T., Pacheco, H., Quaresma, M., Schwabe, P., Séré, A., Strub, P.Y.: Formally verifying Kyber Episode IV: Implementation correctness. IACR Transactions on Cryptographic Hardware and Embedded Systems **2023**(3), 164–193 (2023). https://doi.org/10.46586/tches.v2023.i3.164-193
6. Ammanaghatta Shivakumar, B., Barthe, G., Grégoire, B., Laporte, V., Oliveira, T., Priya, S., Schwabe, P., Tabary-Maujean, L.: Typing High-Speed Cryptography against Spectre v1. In: 44th IEEE Symposium on Security and Privacy (SP 2023). pp. 1094–1111. IEEE, San Francisco, CA, USA (2023). https://doi.org/10.1109/SP46215.2023.10179418
7. Ammanaghatta Shivakumar, B., Barthe, G., Grégoire, B., Laporte, V., Priya, S.: Enforcing Fine-grained Constant-time Policies. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 83–96. ACM, Los Angeles, CA, USA (2022). https://doi.org/10.1145/3548606.3560689
8. Arranz-Olmos, S., Barthe, G., Gonzalez, R., Grégoire, B., Laporte, V., Léchenet, J.C., Oliveira, T., Schwabe, P.: High-assurance zeroization. IACR Transactions on Cryptographic Hardware and Embedded Systems **2024**(1), 375–397 (2024). https://doi.org/10.46586/tches.v2024.i1.375-397
9. Barthe, G., Grégoire, B., Heraud, S., Zanella-Béguelin, S.: Computer-Aided Security Proofs for the Working Cryptographer. In: Rogaway, P. (ed.) Advances in Cryptology – CRYPTO 2011. pp. 71–90. Lecture Notes in Computer Science, Springer, Santa Barbara, CA, USA (2011). https://doi.org/10.1007/978-3-642-22792-9_5
10. Barthe, G., Grégoire, B., Laporte, V., Priya, S.: Structured Leakage and Applications to Cryptographic Constant-Time and Cost. In: Kim, Y., Kim, J., Vigna, G., Shi, E. (eds.) CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security. pp. 462–476. ACM, Virtual Event, Republic of Korea (2021). https://doi.org/10.1145/3460120.3484761
11. Gigerl, B., Hadzic, V., Primas, R., Mangard, S., Bloem, R.: Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs. In: Bailey, M.D., Greenstadt, R. (eds.) 30th USENIX Security Symposium (USENIX Security 2021). pp. 1469–1468. USENIX Association (2021), https://www.usenix.org/conference/usenixsecurity21/presentation/gigerl
12. Gigerl, B., Primas, R., Mangard, S.: Secure and Efficient Software Masking on Superscalar Pipelined Processors. In: Tibouchi, M., Wang, H. (eds.) Advances in Cryptology – ASIACRYPT 2021. pp. 3–32. Lecture Notes in Computer Science, Springer, Singapore (2021). https://doi.org/10.1007/978-3-030-92075-3_1
13. Kocher, P., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) Advances in Cryptology — CRYPTO '99. Lecture Notes in Computer Science, vol. 1666, pp. 388–397. Springer, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-48405-1_25
14. Mainka, L., Papagiannopoulos, K.: Combined Masking and Shuffling for Side-Channel Secure Ascon on RISC-V. In: Rivain, M., Sasdrich, P. (eds.) Constructive Approaches for Security Analysis and Design of Embedded Systems. pp. 451–477. Springer Nature Switzerland, Cham (2026). https://doi.org/10.1007/978-3-032-01405-4_19
15. Marshall, B., Page, D., Webb, J.: MIRACLE: MIcRo-ArChitectural Leakage Evaluation: A study of micro-architectural power leakage across many devices. IACR

Transactions on Cryptographic Hardware and Embedded Systems pp. 175–220 (Nov 2021). https://doi.org/10.46586/tches.v2022.i1.175-220
16. Sajadi, A., Zidaric, N., Stefanov, T., Mentens, N.: A Systematic Comparison of Side-channel Countermeasures for RISC-V-based SoCs. In: 2024 IEEE Nordic Circuits and Systems Conference (NorCAS). pp. 1–7. IEEE (Oct 2024). https://doi.org/10.1109/NorCAS64408.2024.10752477
17. Schmitt, N., Zeitschner, J., Heinemann, A.: Simulation-Based Software Leakage Evaluation for the RISC-V Platform. In: Cha, S.K., Park, J. (eds.) Information Security – 28th International Conference, ISC 2025. pp. 175–195. Lecture Notes in Computer Science, Springer, Seoul, South Korea (2025). https://doi.org/10.1007/978-3-032-08124-7_9
18. Shelton, M.A., Chmielewski, L., Samwel, N., Wagner, M., Batina, L., Yarom, Y.: Rosita++: Automatic Higher-Order Leakage Elimination from Cryptographic Code. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. pp. 685–699. Association for Computing Machinery, Virtual Event, Republic of Korea (2021). https://doi.org/10.1145/3460120.3485380
19. Shelton, M.A., Samwel, N., Batina, L., Regazzoni, F., Wagner, M., Yarom, Y.: Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers. In: Proceedings 2021 Network and Distributed System Security Symposium. Internet Society, Virtual (2021). https://doi.org/10.14722/ndss.2021.23137
20. Tsoupidi, R.M., Castañeda Lozano, R., Troubitsyna, E., Papadimitratos, P.: Securing Optimized Code Against Power Side Channels. In: 36th IEEE Computer Security Foundations Symposium (CSF 2023). pp. 340–355. IEEE, Dubrovnik, Croatia (2023). https://doi.org/10.1109/CSF57540.2023.00016
21. Wang, J., Sung, C., Wang, C.: Mitigating power side channels during compilation. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 590–601. ACM, Tallinn Estonia (Aug 2019). https://doi.org/10.1145/3338906.3338913
22. Zeitschner, J., Moradi, A.: PoMMES: Prevention of Micro-architectural Leakages in Masked Embedded Software. IACR Transactions on Cryptographic Hardware and Embedded Systems **2024**(3), 342–376 (Jul 2024). https://doi.org/10.46586/tches.v2024.i3.342-376
23. Zeitschner, J., Müller, N., Moradi, A.: PROLEAD_SW: Probing-Based Software Leakage Detection for ARM Binaries. IACR Transactions on Cryptographic Hardware and Embedded Systems **2023**(3), 391–421 (Jun 2023). https://doi.org/10.46586/tches.v2023.i3.391-421

8 Appendix

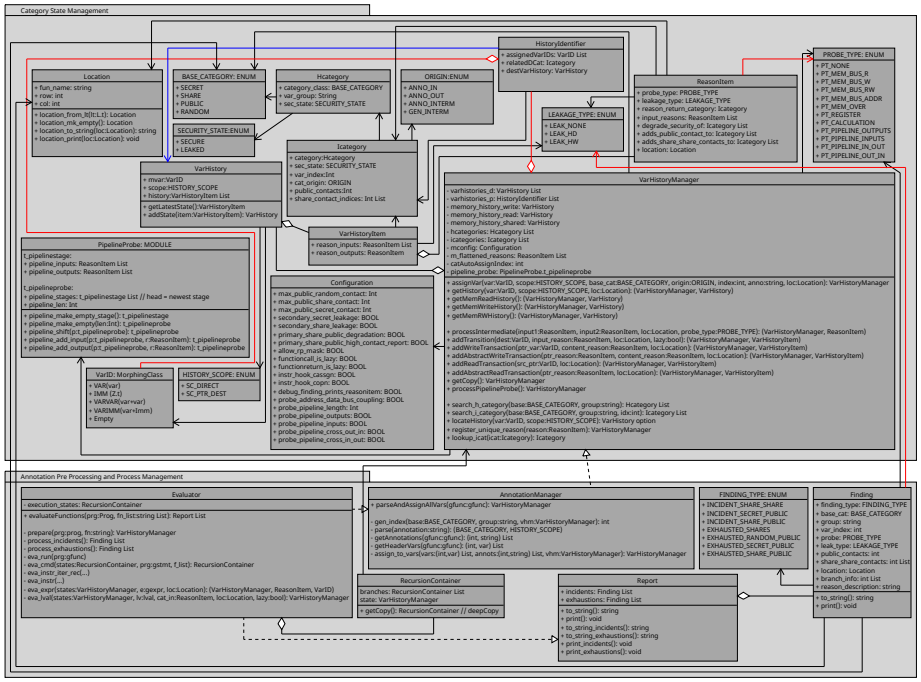

Fig. 1: Simplified UML diagram of the data model and its module dependencies. Colors are for improved readability.

Test group	Snippet	Description
functional	branch_a	Combinatorial leakage in first branch
functional	branch_b	Combinatorial leakage in second branch
functional	branch_ab	Absence of combinatorial leakage between branches
functional	branch_cross_negative	Absence of combinatorial leakage before/after branches
functional	branch_cross_positive	Combinatorial leakage before/after branches
functional	fcall_enter	Overwrite leakage in functioncall
functional	fcall_enter1	Calculation leakage in functioncall
functional	fcall_exit	Overwrite leakage on return of functioncall
functional	loop_pre	Combinatorial leakage on loop enter
functional	loop_post	Combinatorial leakage on loop exit
functional	loop_cond	Combinatorial leakage in loop condition
functional	loop_in	Combinatorial leakage on loop mult. iterations
probetypes	memory_bus_addr_data	Leakage between memory address and data
probetypes	memory_bus_read	Memory bus hidden state read after read
probetypes	memory_bus_write	Memory bus hidden state write after write
probetypes	memory_bus_read_write	Memory bus hidden state read then write
probetypes	memory_bus_write_read	Memory bus hidden state write then read
probetypes	memory_overwrite_negative	Memory overwrite without leakage
probetypes	memory_overwrite_positive	Memory overwrite with leakage
probetypes	pipeline_forwarding_negative	Pipeline probe enabled no leakage
probetypes	pipeline_forwarding_positive	Pipeline probe enabled with leakage
probetypes	register_overwrite	Register writeback leakage
probetypes	register_calculation	Leakage through calculation
baseclass/calculation	public_public	Calculation with two public leads to public
baseclass/calculation	public_random	Public-random calculations with exhaustion
baseclass/calculation	public_random1	Exhausted random behaves as public
baseclass/calculation	public_secret	Public-secret calculation, showing output class secret
baseclass/calculation	public_share	Public-share calculation, showing output class share
baseclass/calculation	random_random	Random-random calculation, showing output class is random
baseclass/calculation	random_random1	Random-random calculation, showing new random exhaustable
baseclass/calculation	random_share	Random-share calculation
baseclass/calculation	random_share1	Exhausted random-share calculation
baseclass/calculation	secret_random	Secret-random calculation, showing output class random
baseclass/calculation	secret_share	Secret-share calculation, showing output share
baseclass/calculation	secret_secret	Secret-secret calculation, showing output is secret class
baseclass/calculation	share_share_ingroup	Share-share contact in calculation
baseclass/calculation	share_othergroup	Share with share of other group contact in calculation
baseclass/calculation	share_same	Share with same share contact in calculatio
baseclass/overwrite	public_public	Overwrite public with public
baseclass/overwrite	public_random	Overwrite public with random
baseclass/overwrite	public_secret	Overwrite public with secret
baseclass/overwrite	public_share	Overwrite public with share
baseclass/overwrite	random_public	Overwrite random with public
baseclass/overwrite	random_random	Overwrite random with random
baseclass/overwrite	random_secret	Overwrite random with secret
baseclass/overwrite	random_share	Overwrite random with share
baseclass/overwrite	secret_public	Overwrite secret with public
baseclass/overwrite	secret_random	Overwrite secret with random
baseclass/overwrite	secret_secret	Overwrite secret with secret
baseclass/overwrite	secret_share	Overwrite secret with share
baseclass/overwrite	share_public	Overwrite share with public
baseclass/overwrite	share_random	Overwrite share with random
baseclass/overwrite	share_secret	Overwrite share with secret
baseclass/overwrite	share_share_ingroup	Overwrite share with share of same group
baseclass/overwrite	share_othergroup	Overwrite share with share of other group
baseclass/overwrite	share_same	Overwrite share with itself
baseclass/exhaustion	random_public	Random-public contact exhaustion
baseclass/exhaustion	secret_public	Secret-public contact exhaustion
baseclass/exhaustion	share_public	Share-public exhaustion
baseclass/exhaustion	share_share	Share-share exhaustion

Table 1: Validation Jasmin snippets