


# Secure Compilation of Probabilistic Side-Channel Countermeasures

Gilles Barthe,[1,2] Gaëtan Cassiers,[3,4] Henri Dohmen,[5] Marc Gourjon,[1]
Benjamin Grégoire,[6] Nora Khayata,[5] Thomas Schneider,[5] Finn Schubert[5]
[1]*Max Planck Institute for Security and Privacy, Bochum, Germany {firstname.lastname}@mpi-sp.org*
[2]*IMDEA Software Institute, Madrid, Spain*
[3]*UCLouvain, Louvain-la-Neuve, Belgium* [4]*CryptoExperts, Paris, France gaetan.cassiers@uclouvain.be*
[5]*Technical University of Darmstadt, Germany {lastname}@encrypto.cs.tu-darmstadt.de, finn.schubert@stud.tu-darmstadt.de*
[6]*Inria Sophia-Antipolis, France benjamin.gregoire@inria.fr*

***Abstract*—Probabilistic countermeasures are a common means of protecting programs against side-channels. These countermeasures are hard to implement and are routinely broken by compilers. Proving that a compiler preserves probabilistic side-channel security directly would require reasoning about perfect simulatability of distributions, a challenging technique normally reserved for high-assurance cryptography in frameworks like EASYCRYPT and out of scope for compiler correctness.**

**In this paper, we present a language- and leakage-model-agnostic framework for provably secure compilation of probabilistic side-channel security notions. Our main contribution is formalizing a deterministic leakage-preservation relation that implies preservation of probabilistic security notions, including $d$-NI and $d$-SNI for masking and cryptographic constant-time. Our framework establishes the same guarantees without probabilistic proof obligations, reducing preservation to a per-pass simulation argument on deterministic traces.**

**We use the ROCQ proof assistant to prove that a model of the core pipeline and several optimizing passes of the JASMIN compiler preserve security in the value-probing model with glitches. We also show how to harden against transition leakage with lightweight annotations at the source level. On top of this, we build LIBMASKING, an open-source library of arbitrary-order masked gadgets in JASMIN that securely compiles to ARM and RISC-V assembly, provides C language bindings, and is benchmarked on real hardware. Together, these contributions are a step towards programming environments in which masked code is written or generated, verified at the source level, optimized, and compiled without breaking security.**

## 1. Introduction

Security in real-world systems rests on cryptography, yet the assurance of such deployed cryptographic schemes can be broken by physical side-channel attacks (SCAs). SCAs exploit physical side-effects of computation on concrete devices, e.g., by measuring the consumed power or electromagnetic emanations. Without sophisticated countermeasures, SCAs routinely break cryptographic schemes [1], [2], [3]. Especially in critical domains such as payment, secure elements, and eID devices, physical attacks are explicitly in scope, and sound protection against physical attacks is a core requirement in certification schemes such as Common Criteria, gating market access for billions of devices [4].

The dominant but error-prone countermeasure against SCAs is *masking*: every secret is encoded as $n$ randomized values, called shares, such that every subset of $d < n$ shares is independent of the secret, where $d \geq 1$ is the *masking order*, a security parameter. Every operation on secrets is replaced by a *gadget* that securely operates on shares by keeping the joint distribution of any $d$ intermediates independent of secrets [5], [6]. Implementing gadgets that satisfy this $d$-wise independence is error-prone, frequently leading to flaws both at the algorithmic [7], [8] and implementation level [9], [10], [11], [12]. An active line of research (see §6) verifies formal security properties at source code or executable level to rule out such flaws. At the *algorithmic level*, composable security notions such as $d$-NI [6], $d$-SNI [13], and $d$-PINI [14] are checked automatically on gadgets and their compositions by tools such as MASKVERIF [15], SILVER [16], IRONMASK [17], and EVER [18]. At the *implementation level*, SCVERIF [12] and COCO [19] verify concrete assembly under fine-grained hardware leakage models [20] that capture practically observable leakage arising from transitions and glitches in the microarchitecture of processors. However, the problem with source-level verification is that off-the-shelf compilers like LLVM and `gcc` routinely break the security of masked gadgets during compilation. In §5, we highlight one example where `gcc` breaks masking security at optimization levels `-O1`, `-O2`, and `-O3`. Since compilers may break security of the source, developers are forced to rewrite gadgets in architecture-specific assembly for every new target and then verify each binary's security independently with fragmented verification tooling that supports only a few target architectures [21]. This raises the question:

*How can we prove that an optimizing compiler preserves the security of masked source programs down to assembly?*

Related questions have already been studied in research on secure compilation [22], [23], [24], [25]. However, our focus is on preserving the security of masked implementa-

tions, which mandates proving preservation of probabilistic security properties: One needs to prove that a given compiler pass preserves (independence of) the probability distributions of intermediate variables for any input masked source program. Such arguments about probability distributions are typically left to advanced security proofs of cryptographic schemes in dedicated interactive theorem provers (ITPs) like EASYCRYPT [26] that are out of scope for compiler passes.

The benefits of a dependable compiler are demonstrated by the JASMIN compiler [27], [28], [29], [30], which provably preserves functional correctness as well as constant-time and speculative constant-time security from source to assembly. With it, the open-source cryptographic library LIBJADE was developed and formally verified for correctness and security in EASYCRYPT [26], achieving high speed and end-to-end (E2E) high assurance. This combination motivated its recent integration into the Signal messenger [31]. Recent work [32] proves that JASMIN preserves cryptographic security, which involves probability distributions too. However, the difficulty in masking is to relate the distributions of subsets of leakage at source and target, which is a fundamentally different problem from the one in [32].

## Our Contributions

In this work, we close the gap of a missing secure compilation flow for masking by establishing and applying a framework for formally proving *in a deterministic setting* that optimizing compiler passes preserve the security of masked implementations. We summarize our contributions as follows:

**Formal framework for provably secure compilation of side-channel countermeasures (§4)**: We present a generic notion of leakage-preservation for both timing and power side-channels. This relation allows us to prove in a deterministic setting, i.e., without any probabilistic reasoning, that compiler passes do not introduce leakage beyond what is already observable at the source. We prove that this deterministic relation implies preservation of probabilistic security notions, concretely $d$-PS, $d$-NI, $d$-SNI, and $d$-PINI [5], [13], [14]. Per-pass proof obligations then reduce to much simpler deterministic simulation arguments on traces, with no probabilistic reasoning required. Our leakage-preservation relation is deterministic and agnostic to the involved programming languages and leakage models: The underlying notion of leakage is part of the program semantics: the semantics encode not only what a program computes, but also exactly what leakage it emits and when. The relation itself applies to any deterministic source and target language. Thereby it allows us to prove preservation for fine-grained models of arbitrary leakage behavior encountered in practice.

**Applying our formal framework to prove a realistic compiler pipeline (§4.4)**: We instantiate our framework on a model of the core pipeline and several passes of the JASMIN compiler [27] and prove in the ROCQ theorem prover preservation in the value-probing model with glitches [33]. The formal development is provided as an artifact.[1] As a corollary, we find that preservation in the plain value-probing model is infeasible in general when the compiler has to split a source statement containing complex expressions into multiple instructions. Protection against glitches, a baseline for masked hardware implementations, must therefore also serve as the baseline for masking in software. Going further, we show how lightweight annotations in source code and a small change to the JASMIN compiler allow us to also harden masked code against transition leakage throughout compilation [33].

**Open-source library of masked gadgets LIBMASKING (§3 and §5)**: We provide LIBMASKING,[1] a library of arbitrary-order masked gadgets with hardening against transition leakage that is written in JASMIN, compiles securely to ARM and RISC-V assembly, and provides C language bindings. Our gadgets are accompanied by functional correctness proofs in EASYCRYPT [26]. To confirm our security claims for secure compilation and the hardening against transition leakage, we verify the compiled binaries using SCVERIF [12] as an independent check. Finally, we implement SHA3 and SHAKE using our gadgets, and benchmarking on real hardware shows that our JASMIN implementation has competitive performance with C implementations, demonstrating that strong security guarantees do not come at the price of performance.

Together, these contributions are a step towards programming environments that support research and development by E2E security: masked code is written once, verified at the source level, and subsequently automatically optimized and compiled to architecture-specific assembly in a way that provably maintains security throughout the flow.

## Limitations

Our mechanized preservation results are grounded on a specific leakage model: the value-probing model with glitches. One natural question is whether this leakage model is a reasonable reflection of the physical leakage encountered in practice. The question has possible answers, none of which are carried out in this paper. The first answer is to use our methodology and part of our existing ROCQ infrastructure to reproduce our results in other leakage models. Our estimate is that it can be done with moderate effort. The second answer is to formalize physical leakage models and prove preservation results with respect to such models. Hardware-software contracts [20] would be of significant help here, but this remains a significant project on its own, and outside the scope of this work.

Another concrete limitation is that our leakage preservation results are proven for a core compiler, rather than for the JASMIN compiler itself. Since the connection between our core compiler and JASMIN is informal, we close this gap by treating the JASMIN compiler as untrusted and verifying compiled gadgets using SCVERIF. The ultimate goal, of

1. Artifact containing ROCQ proofs, LIBMASKING, and a ready-to-use development environment at https://doi.org/10.5281/zenodo.22827961.

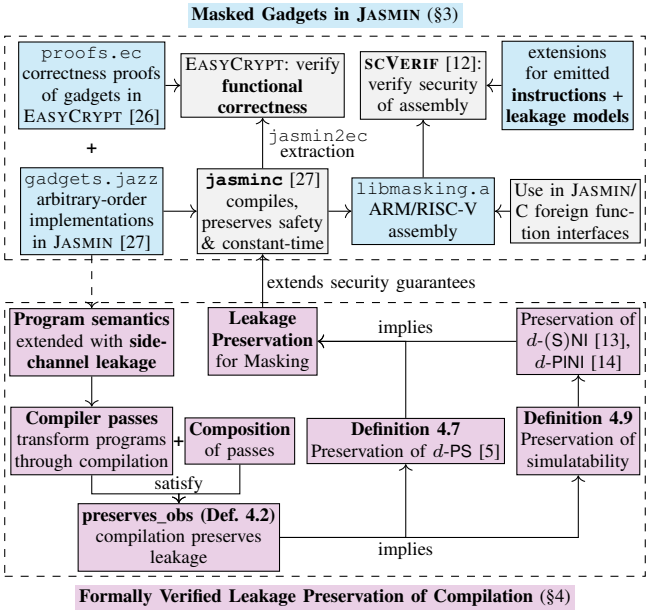



Figure 1. Overview of technical contributions. Colored boxes are ours, gray ones are prior work.

course, is to prove that the JASMIN compiler is trustworthy. Our work provides both a methodology to carry out this work and evidence that all technical challenges can be addressed (and have been at a smaller scale). The only lacking ingredient to conclude the proof was a probabilistic semantics for the JASMIN language, which has recently been released. We view the completion of this task as a significant proof engineering effort.

The final limitation is that our experimental evaluation does not compare with optimized secure implementations and contains a small set of examples. Our reusable artifact is a starting point for studying how JASMIN performs for engineering masked software at scale.

## 2. Technical Overview

We give an overview of our core technical contributions in Fig. 1 and summarize them in this section.

**Masked Gadgets in JASMIN.** On the practical side, we present LIBMASKING, our library of masked gadgets, written in JASMIN (§3). The gadget implementations are generic in the masking order and are developed in a target-independent manner. We compile the gadgets to secure ARM and RISC-V assembly and benchmark them for realistic masking orders ($d = 1, \ldots, 4$). We verify using SCVERIF [12] that the emitted assembly is secure on both targets, and show that it has comparable efficiency to the `gcc`-emitted assembly, which is not reliably secure (cf. §5).

In addition, we developed proofs of functional correctness for our gadgets in EASYCRYPT [26], which are also generic in the masking order. `jasminc` supports automatic extraction to EASYCRYPT, so we use the same JASMIN implementation for functional correctness proofs and assembly generation. The JASMIN gadgets can be used in other languages besides JASMIN via C foreign function interfaces.

**Formally Verified Leakage Preservation of Compilation.** We develop a formal framework for reasoning about security preservation during compilation. We formalize a core language that abstracts JASMIN and prove secure a realistic compilation flow consisting of several core compiler passes that capture essential code transformations performed by JASMIN's compiler. We develop our proofs with the ROCQ [34] interactive theorem prover.

At the heart of our work is an operational program semantics that includes the leakage behavior as part of the execution model. This leakage separates the information that can be leaked through timing side-channels from the value leakage, typically leaked through power or electromagnetic side-channels. We then introduce a core definition of our formalization: leakage preservation for a compiler (cf. Definition 4.2). We demonstrate the relevance of this notion by showing that if a compiler preserves leakage and a source program satisfies threshold probing security or simulatability (two widely used definitions of security for masking schemes), then the compiled program also satisfies the same property. Next, we model a representative set of compiler passes (cf. Table 1) and prove that each of these passes preserves leakage. We also prove that a compiler made of a sequence of leakage-preserving passes is itself leakage-preserving.

The generic notion of leakage preservation decouples formal reasoning about how leakage is deterministically transformed by compiler passes from the proofs of masking security. As a benefit of this decoupling, even though our target security notions like $d$-NI and $d$-SNI [13] are probabilistic, we can work with deterministic program semantics, which simplifies the analysis of the compiler passes.

## 3. Masked Algorithms in JASMIN

In this section we explain how JASMIN can be leveraged for writing high-speed and high-assurance masked implementations of entire cryptographic schemes. The use of JASMIN is primarily motivated by the fine-grained control the programmer retains over compilation results, which lets them leverage high-level language constructs while writing "assembly in the head", i.e., retaining control over the assembly and thus over side-channel leakage.

The overall development flow is to implement architecture-independent and arbitrary-order masked algorithms in the JASMIN language and to rely on the JASMIN compiler to securely compile them to architecture-specific assembly. It is common practice to use composable security notions $d$-(S)NI and $d$-PINI [13], [14] to simplify the protection of large schemes: first, a set of masked primitives, so-called *gadgets*, is developed. These are then composed to implement the function of the to-be-protected algorithm. The security of such a composition rests on $d$-(S)NI or $d$-PINI security of the individual gadgets and on the fact that the composition rules of $d$-(S)NI or $d$-PINI are adhered to, which can be checked manually or automatically. An

intuitive description of the composition rules and a large-scale example appear in [14], [21]. The ideas described here are used later in §5 to develop a library of verifiably secure masked gadgets and masked implementations of SHA3 and SHAKE.

In addition to verified compilation, JASMIN also allows extracting the source code to the EASYCRYPT [26] ITP. This allows us to conduct proofs of functional correctness at the source, which are then preserved E2E during compilation. JASMIN's proofs do not cover the preservation of masking security, which is the subject of §4. We use the extraction mechanism to mechanically prove functional correctness of our gadgets in EASYCRYPT (cf. §5). Future work could integrate existing automated tools for masking to allow source-level security verification, or extend JASMIN's extraction mechanism to emit leakage information. The latter would enable arbitrary-order proofs of $d$-(S)NI and $d$-PINI for concrete implementations in EASYCRYPT, reinforcing the handwritten proofs appearing in the literature.

**Example: AND Gadget.** The JASMIN language is specifically designed for low-level highly optimized implementations. We exemplify some design tricks on an implementation of the ISW AND gadget [5], shown in Listing 1, which computes a bitwise logical AND of two secret inputs. The underlying algorithm is sufficiently complicated to highlight several important features of JASMIN for masking. The AND gadget takes as input `N` shares per sensitive value $a$ and $b$, and yields `N` shares encoding output $c$, such that $c = a \wedge b$. Its implementation `_g_and` receives a buffer for the resulting shares (`c`) in the form of a register pointing to an array of `N` 32-bit values (line 2, `reg ptr u32[N]`) and arrays containing the two input sharings (`a` and `b`).

Many gadgets, among them the AND, require access to uniformly distributed random values to achieve security against probing adversaries. We provide two variants of such gadgets: a "derandomized" version where the caller supplies pre-sampled randomness (shown in Listing 1), and a probabilistic version that samples its own randomness. On embedded devices, the latter can reduce memory pressure while the derandomized variant improves efficiency when the randomness for multiple gadgets is sampled in blocks [12], [21]. The two variants share one implementation: the probabilistic variant samples random values to its stack using the JASMIN intrinsic `x = #randombytes(x);` and then runs the inlined derandomized variant on the collected random values.

Another feature of JASMIN is its close correspondence between the source code and the generated assembly: Variables declared `inline` (e.g., lines 6–7) are fully evaluated and `for`-loops are unrolled at compile time, whereas `reg`-variables and `while`-loops are evaluated at runtime. Furthermore, every assignment that operates on at least one non-`inline` variable is compiled to exactly one assembly instruction. Assignments that cannot be compiled to one instruction of the target architecture cause compilation errors. These features give developers fine-grained control over the generated assembly and thus, over performance and side-channel characteristics. In lines 12–15, we explicitly perform a load-compute-store cycle because ARM and RISC-V

```
1  fn _g_and( // [ISW03] AND gadget functionality
2      reg ptr u32[N] c,   // buffer for output shares
3      reg ptr u32[N] a b, // arrays of input shares
4      reg ptr u32[AND_RANDOMNESS] randomness
5  ) -> reg ptr u32[N] {
6    inline int i, j; // declare compile-time evaluated vars
7    inline int rand_counter = 0;
8    // declare all register variables used in computation
9    reg u32 a_i, a_j, c_i, c_j, ab, b_i, b_j, rand;
10
11   for i = 0 to N { // compute c[i] = a[i] & b[i]
12     a_i = a[i];    // 1. load values to registers
13     b_i = b[i];
14     a_i &= b_i;    // 2. compute on registers
15     c[i] = a_i; }  // 3. store result back in memory
16
17   for i = 0 to N {
18     for j = i+1 to N {
19       rand = randomness[rand_counter];
20       rand_counter += 1;
21       // c[i] = c[i] ^ rand
22       c_i = c[i];
23       c_i ^= rand;
24       c[i] = c_i;
25       // ab = (a[i] & b[j]) ^ rand
26       ab = a[i];
27       b_j = b[j];
28       ab &= b_j;
29       ab ^= rand;
30       // a_j = (a[j] & b[i]) ^ ab
31       a_j = a[j];
32       b_i = b[i];
33       a_j &= b_i;
34       a_j ^= ab;
35       // c[j] = c[j] ^ a_j
36       c_j = c[j];
37       c_j ^= a_j;
38       c[j] = c_j; } }
39   return c; }
```

Listing 1. JASMIN code for a $d$-SNI AND gadget [5], generic in the masking order $d$ with $n = d + 1$ shares.

are load-store architectures (i.e., operands to computational instructions cannot be memory locations), so the equivalent one-line formulation `c[i] = a[i] & b[i]` would be rejected by the compiler.

## 4. Formal Framework

In this section, we introduce the notion of leakage preservation in a more formal setting. We demonstrate how this notion implies preservation of the probabilistic security properties common in masking. Then, we prove preservation of leakage for a model of the JASMIN compiler, and conclude by discussing how this result extends to JASMIN.

Compilers often handle programs written in different programming languages: source, target, or intermediate representations. Our formalism does not specify the languages in detail, which enables relating programs of distinct languages. A programming language is a set $\mathcal{P}$ of programs. The execution of a program takes an input from a set $\mathcal{I}$ (which, e.g., models the arguments of its entry point), and produces an output from a set $\mathcal{O}$ (e.g., the return value). The semantics of programs is defined by a set $\mathcal{S}$ of execution states, sets $\mathcal{L}_t$ and $\mathcal{L}_v$ of timing and value leakage, respectively, an initialization function $\mathsf{init} \colon \mathcal{P} \times \mathcal{I} \to \mathcal{S}$, a single-step relation $\sigma \xrightarrow[v]{\tau} \sigma'$ between states $\sigma$ and $\sigma'$, where $\tau$ represents (cache-)timing leakage and the so-called *value*

leakage $\upsilon$ represents power side-channel observations. Both leakage behaviors are defined in the program semantics and left abstract for the moment. We work with a deterministic language: for any $\sigma \in \mathcal{S}$ there is at most one tuple $(\sigma', \tau, \upsilon)$ such that $\sigma \xrightarrow[\upsilon]{\tau} \sigma'$. The states $\sigma$ that admit no such tuple are known as *final states* $\mathcal{S}_F \subset \mathcal{S}$, and the output function $\mathsf{output} \colon \mathcal{S}_F \to \mathcal{O}$ produces the execution result from a final state. Abstracting inputs $\mathcal{I}$ and outputs $\mathcal{O}$ is essential for preserving leakage when compiling between two distinct languages, e.g., in linearization. We also consider a multi-step relation that terminates in a final state $\twoheadrightarrow$, defined by the following rules:

$$\text{NIL}\ \frac{\sigma \in \mathcal{S}_F}{\sigma \xrightarrow[\cdot]{\cdot} \sigma} \qquad \text{CONS}\ \frac{\sigma \xrightarrow[\upsilon]{\tau} \sigma' \qquad \sigma' \xrightarrow[\vec{\upsilon}]{\vec{\tau}} \sigma''}{\sigma \xrightarrow[\upsilon \cdot \vec{\upsilon}]{\tau \cdot \vec{\tau}} \sigma''}.$$

We say that a state $\sigma$ is *terminating* iff there exist $\vec{\tau}$, $\vec{\upsilon}$, and $\sigma'$ such that $\sigma \xrightarrow[\vec{\upsilon}]{\vec{\tau}} \sigma'$, and that an input $x$ for a program $p$ is terminating if $\mathsf{init}(p, x)$ is terminating.

A *compiler* is a partial function $C$ that translates programs from a source language $\mathcal{P}_s$ to a target language $\mathcal{P}_t$. Compilers are typically organized as a series of compiler passes: $C = C_m \circ \cdots \circ C_1$, where each $C_i$ is formally a compiler.

A compiler is *correct* if a compiled program always produces the same output as the source program, given the same input.

**Definition 4.1** (Compiler correctness)**.** *For all $p_s \in \mathcal{P}_s$ with $p_t = C(p_s)$, for all inputs $x$, states $\sigma_s$, and leakage $\vec{\tau}_s$ and $\vec{\upsilon}_s$ such that $\mathsf{init}(p_s, x) \xrightarrow[\vec{\upsilon}_s]{\vec{\tau}_s} \sigma_s$, there exist $\sigma_t, \vec{\tau}_t, \vec{\upsilon}_t$ such that $\mathsf{init}(p_t, x) \xrightarrow[\vec{\upsilon}_t]{\vec{\tau}_t} \sigma_t$ and $\mathsf{output}(\sigma_s) = \mathsf{output}(\sigma_t)$.*

$$\begin{array}{ccc} \mathsf{init}(p_s, x) & \xrightarrow[\vec{\upsilon}_s]{\vec{\tau}_s} & \sigma_s \\ {\scriptstyle p_t = C(p_s)}\,\| & & \|\,{\scriptstyle \mathsf{output}(\sigma_s) = \mathsf{output}(\sigma_t)} \\ \mathsf{init}(p_t, x) & \xrightarrow[\vec{\upsilon}_t]{\vec{\tau}_t} & \sigma_t \end{array}$$

Note that we use a slightly less general definition than the one typically employed for compiler correctness. Usually it is formulated with a relation $\phi$ to relate initial states and a relation $\psi$ to relate final states. Here, these relations are

$$\begin{array}{rcl} \phi(\sigma_s, \sigma_t) & = & \exists x,\ \sigma_s = \mathsf{init}(p_s, x) \wedge \sigma_t = \mathsf{init}(p_t, x), \\ \psi(\sigma_s, \sigma_t) & = & \mathsf{output}(\sigma_s) = \mathsf{output}(\sigma_t). \end{array}$$

Furthermore, compiler correctness is usually expressed as a backward simulation: if the target program evaluates, then the source program also evaluates. However, it is well known [35] that backward simulation is equivalent to forward simulation for all *safe* programs when the target language is deterministic. The advantage of using a forward simulation is that it implies preservation of safety.

## 4.1. Leakage Modeling and Preservation

Each instruction produces both timing and value leakage. *Timing leakage* $\tau$ is unbounded: the adversary is able to observe all the timing variations in a program execution. *Value leakage* $\upsilon$ is harder to observe, and the widely used threshold $d$-probing model allows the adversary to observe an arbitrary subset of size $d$ of value leakage [5].[2]

Informally, preservation expresses that a compiled program should not leak "more" than the corresponding source program, and that a $d$-probing secure source program compiles to a $d$-probing secure target program. We state our main preservation definition as a simulation: all leakage of the target program should be simulatable from the leakage of the source program. First, we require preservation of cryptographic constant-time security: the whole target timing leakage trace (marked blue) must be simulatable from the source timing leakage trace. For the value leakage (marked red), we have a finer-grained definition: the leakage of any instruction in the target execution must be simulatable from the leakage of an instruction in the source program (along with the source timing leakage).

**Definition 4.2** (Leakage preservation)**.** *A compiler $C$ preserves leakage if there exists a timing leakage simulator $\mathsf{sim}_\tau \colon \mathcal{P}_s \times \mathcal{L}_t^* \to \mathcal{L}_t^*$, a leakage index map $\mathsf{sim}_{\mathsf{idx}} \colon \mathcal{P}_s \times \mathcal{L}_t^* \times \mathbb{N} \to \mathbb{N}$ and a value leakage simulator $\mathsf{sim}_\upsilon \colon \mathcal{P}_s \times \mathcal{L}_t^* \times \mathbb{N} \times \mathcal{L}_\upsilon \to \mathcal{L}_\upsilon$ such that for any $p_s \in \mathcal{P}_s$ and $p_t = C(p_s)$, any input $x$, if $\mathsf{init}(p_s, x) \xrightarrow[\vec{\upsilon}_s]{\vec{\tau}_s} \sigma_s$, for some $\sigma_s$, $\vec{\tau}_s$, $\vec{\upsilon}_s$, then there exist $\sigma_t$, $\vec{\tau}_t$, and $\vec{\upsilon}_t$ such that $\mathsf{init}(p_t, x) \xrightarrow[\vec{\upsilon}_t]{\vec{\tau}_t} \sigma_t$, $\vec{\tau}_t = \mathsf{sim}_\tau(p_s, \vec{\tau}_s)$ and, for any $i_t \in \mathbb{N}$, $\vec{\upsilon}_t[i_t] = \mathsf{sim}_\upsilon(p_s, \vec{\tau}_s, i_t, \vec{\upsilon}_s[i_s])$ where $i_s = \mathsf{sim}_{\mathsf{idx}}(p_s, \vec{\tau}_s, i_t)$.*[3]

$$\begin{array}{ccccccc} \mathsf{init}(p_s, x) & \xrightarrow[\vec{\upsilon}_s[1]]{\vec{\tau}_s[1]} & \cdots & \xrightarrow[\vec{\upsilon}_s[i_s]]{\vec{\tau}_s[i_s]} & \cdots & \xrightarrow[\vec{\upsilon}_s[n]]{\vec{\tau}_s[n]} & \sigma_s \\ {\scriptstyle p_t = C(p_s)}\,\| & & & & & \|\,{\scriptstyle \vec{\upsilon}_t[i_t] = \mathsf{sim}_\upsilon(p_s, \vec{\tau}_s, i_t, \vec{\upsilon}_s[i_s])} & \\ \mathsf{init}(p_t, x) & \xrightarrow[\vec{\upsilon}_t[1]]{\vec{\tau}_t[1]} & \cdots & \xrightarrow[\vec{\upsilon}_t[i_t]]{\vec{\tau}_t[i_t]} & \cdots & \xrightarrow[\vec{\upsilon}_t[m]]{\vec{\tau}_t[m]} & \sigma_t \end{array}$$

The RocQ development formalizes Definition 4.2 in `preserves_obs`. In this definition, besides the timing leakage and value leakage simulators, we use $\mathsf{sim}_{\mathsf{idx}}$ to map indices of steps in the target executions to indices of steps in the source executions needed for the simulation. This mapping is needed in order for this definition to be satisfied by compiler passes that change the number of instructions (e.g., dead-code elimination). Later, we require that source programs are cryptographic constant-time to ensure that the mapping only depends on the public part of the input, so that no sensitive information can leak through the mapping.

Leakage preservation, along with compiler correctness, is preserved by composition of compiler passes, as shown by the following lemma.

2. This model and its variants are often used for analyzing the security of masking schemes. Other leakage models such as the random probing model also limit the amount of leakage that is observed by the adversary.

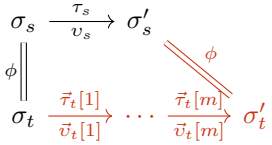
3. We also denote by $l[i]$ the element of the list $l$ at index $i$ (defined to $\bot$ for out-of-bounds indices).

**Lemma 4.3** (Compiler composition)**.** *Let $C_1, \ldots, C_m$ be correct and leakage-preserving compilers. Then, the compiler $C_m \circ \cdots \circ C_1$ is correct and leakage-preserving.*

*Proof.* We prove the case $m = 2$ in our RocQ development as `compile_correct_compose` (for correctness) and `preserves_obs_compose` (for leakage-preservation). The general case then follows directly from the case where $m = 2$. Intuitively, leakage preservation is obtained by composition of the simulators as appropriate, in order to get the simulators for the composition of the compilers. The correctness of the composition of compilers follows from the transitivity of equality of $\mathsf{output}(\cdot)$ on states. □

**Leakage preservation from single-step.** Let us now turn to a single compiler pass. As we work with small-step semantics, we will prove leakage preservation of our compiler using forward simulation, where a single step in the source program corresponds to any number of steps in the target. For this purpose, we first define an *abstract program pointer*, which intuitively corresponds to the program itself, along with an instruction pointer. Formally, given a state $\sigma$, the evolution of an abstract program pointer $\mathsf{PC}(\sigma)$ during execution is uniquely determined by the timing leakage: if $\sigma_1 \xrightarrow[\upsilon_1]{\tau} \sigma_1'$ and $\sigma_2 \xrightarrow[\upsilon_2]{\tau} \sigma_2'$, then $\mathsf{PC}(\sigma_1) = \mathsf{PC}(\sigma_2)$ implies $\mathsf{PC}(\sigma_1') = \mathsf{PC}(\sigma_2')$. The abstract program pointer of initial states is fully determined by the program: for any program $p$ and inputs $x_1, x_2$, $\mathsf{PC}(\mathsf{init}(p, x_1)) = \mathsf{PC}(\mathsf{init}(p, x_2))$.

**Definition 4.4** (Single-step leakage simulation)**.** *A compiler $C$ has single-step leakage (forward) simulation with respect to a relation $\phi$ between source and target states if there exist single-step timing and value leakage simulators $\mathsf{sim}_\tau^1$, $\mathsf{sim}_\upsilon^1$ such that for all source states $\sigma_s$ and target states $\sigma_t$ such that $\phi(\sigma_s, \sigma_t)$, if $\sigma_s \xrightarrow[\upsilon_s]{\tau_s} \sigma_s'$, then there exists $\vec{\tau}_t$, $\vec{\upsilon}_t$, and $\sigma_t'$ such that $\sigma_t \xrightarrow[\vec{\upsilon}_t]{\vec{\tau}_t}{}^{*} \sigma_t'$, $\phi(\sigma_s', \sigma_t')$, and further $\vec{\tau}_t = \mathsf{sim}_\tau^1(\mathsf{PC}(\sigma_s), \tau_s)$ and $\vec{\upsilon}_t = \mathsf{sim}_\upsilon^1(\mathsf{PC}(\sigma_s), \tau_s, \upsilon_s)$.*




**Lemma 4.5** (Leakage-preservation from single-step simulation)**.** *For a compiler $C$, let $\phi$ be a relation between source and target states. We say that $\phi$ relates initial states if $\phi(\mathsf{init}(p_s, x), \mathsf{init}(C(p_s), x))$ for all source programs $p_s$ that can be compiled and all inputs $x$, and that it preserves final states if for any $\sigma_s \in \mathcal{S}_F^s$, $\phi(\sigma_s, \sigma_t)$ implies $\sigma_t \in \mathcal{S}_F^t$. If $C$ has single-step leakage simulation with respect to a relation $\phi$ that relates initial states and preserves final states, then $C$ preserves leakage.*

*Proof.* The proof is in `lift_step_preserves_obs` in the RocQ development. Intuitively, the proof and the construction of the leakage simulators are performed by induction over the number of steps in the source execution. In the base case (no step), the initial state is final, and there is no step in the target since $\phi$ preserves final states. Furthermore, the leakage is empty. Otherwise, we decompose the source execution as a single step, followed by an arbitrary number of steps. Using the single-step leakage simulation we get the timing and value leakage simulators for the first step, as well as new related states. These single-step leakage simulators compound (by induction) to form the multi-step simulators, and the index simulator is built by summing the number of target steps. Note that the number of target steps does not depend on the state $\sigma_t$. □

## 4.2. Security Notions

We now show how our notion of leakage preservation implies preservation of classic masking security notions such as threshold probing security ($d$-PS) and (Strong) Non-Interference ($d$-(S)NI). We first define what an adversary can observe in a program execution, i.e., its *view*: given an input, the adversary observes the whole timing leakage (modeling simple power analysis), a subset of value leakage (corresponding to probes), and some part of the output. To simplify notation, let $\vec{v}_{|X} = (\vec{v}[i])_{i \in X}$ denote a restriction of elements. For a program $p$, given an input $x \in \mathcal{I}$, a set of value leakage indices $O \subset \mathbb{N}$ and a set of output indices $\chi$, if $\mathsf{init}(p, x) \xrightarrow[\vec{\upsilon}]{\vec{\tau}} \sigma$, the view is

$$\mathcal{V}_O^\chi(p, x) = (\vec{\tau}, \vec{\upsilon}_{|O}, \mathsf{output}(\sigma)_{|\chi}),$$

and $\mathcal{V}_O^\chi(p, x) = \bot$ if execution does not terminate. We omit $\chi$ if empty.

Further, we need to consider probabilistic programs, which we model by providing a randomness tape as part of the program input. Formally, given a randomness symbol domain $\mathcal{R}$ (e.g., $\{0, 1\}$), we denote by $\mathcal{I}'$ the deterministic program input domain and let $\mathcal{I} = \mathcal{I}' \times \mathcal{R}^{\mathbb{N}}$ denote the entire input domain. Programs are provided with a uniformly distributed random tape $\boldsymbol{R} \leftarrow\!\!\$\ \mathcal{R}^{\mathbb{N}}$. We denote by $\mathcal{D} \equiv \mathcal{D}'$ point-wise equality of probability distributions $\mathcal{D}, \mathcal{D}'$ and use probabilistic simulators $\mathsf{S}$ that perfectly replicate probability distributions (expressed by $\equiv$). These are in contrast to the compiler-centric deterministic simulations $\mathsf{sim}_\tau$ and $\mathsf{sim}_\upsilon$.

As a first example, we consider preservation of threshold probing security.

**Definition 4.6** (Threshold probing security)**.** *Given an encoder $\mathsf{Enc}$ that maps a secret $x$ into a randomized sharing, a program $p$ is $d$-threshold probing secure if there exists a simulator $\mathsf{S}$ such that for any secret $x$ and any set of probes $O \subset \mathbb{N}$ such that $|O| \leq d$,*

$$\mathcal{V}_O^\emptyset(p, (\mathsf{Enc}(x), \boldsymbol{R})) \equiv \mathsf{S}_O().$$

Note that this definition requires simulating timing leakage (part of the view $\mathcal{V}_O$) and therefore implies independence between timing leakage and secrets, i.e., constant-time security in addition to "classic" probing security.

**Lemma 4.7** (Preservation of threshold probing security)**.** *Let $p_s$ be a $d$-probing secure program for an encoder* Enc *such that the execution of $p_s$ on a random input $(\mathsf{Enc}(x), \boldsymbol{R})$ terminates with probability 1 for any secret $x$. If $p_t = C(p_s)$ for a leakage-preserving compiler $C$, then $p_t$ is $d$-probing secure for* Enc*.*

*Proof.* Given independent random variables $\boldsymbol{Z} \leftarrow \mathsf{Enc}(x)$ and $\boldsymbol{R} \leftarrow\!\$\ \mathcal{R}^{\mathbb{N}}$, we have (with probability 1) a corresponding execution for the source $p_s$ and for the target $p_t$. Further, we know that for any set of probes in the source $O_s$ of size at most $d$, the distribution of the view in the source execution $\mathcal{V}_{O_s}(p_s, (\boldsymbol{Z}, \boldsymbol{R}))$ does not depend on $x$. We have to show that the same holds for the target execution $\mathcal{V}_{O_t}(p_t, (\boldsymbol{Z}, \boldsymbol{R}))$. We denote by $\vec{\boldsymbol{\tau}}_s$, $\vec{\boldsymbol{v}}_s$, $\vec{\boldsymbol{\tau}}_t$ and $\vec{\boldsymbol{v}}_t$ the random variables that represent the leakage of the executions of $p_s$ and $p_t$ with input $(\boldsymbol{Z}, \boldsymbol{R})$.

Let us first note that the assumption implies that the marginal distribution of $\vec{\boldsymbol{\tau}}_s$ is independent of $x$, as well as the conditional distribution $\vec{\boldsymbol{v}}_{s|O_s} \mid \vec{\boldsymbol{\tau}}_s$ (for any $O_s$ of admissible size). Further, thanks to the leakage preservation, the target timing leakage is a function of the source timing leakage: $\vec{\boldsymbol{\tau}}_t = \mathsf{sim}_\tau(p_s, \vec{\boldsymbol{\tau}}_s)$. We then define the set of source indices needed to simulate the target value leakage as $\boldsymbol{O}_s = \mathsf{sim}_{\mathsf{idx}}(p_s, \vec{\boldsymbol{\tau}}_s, O_t)$, such that this leakage becomes a function of the value leakage at the selected indices and the source timing leakage: $\vec{\boldsymbol{v}}_{t|O_t} = \mathsf{sim}_v(p_s, \vec{\boldsymbol{\tau}}_s, \vec{\boldsymbol{v}}_{s|\boldsymbol{O}_s})$. By probing security of the source, these last two equalities imply that the conditional distribution $\vec{\boldsymbol{v}}_{t|O_t} \mid \vec{\boldsymbol{\tau}}_s$ is independent of the secret $x$. Combining this with the target leakage simulator, we obtain that the distribution $\vec{\boldsymbol{\tau}}_t, \vec{\boldsymbol{v}}_{t|O_t} \mid \vec{\boldsymbol{\tau}}_s$ is independent of $x$ as well. Since $\vec{\boldsymbol{\tau}}_s$ is independent of secrets too, this means that the adversary's view $\mathcal{V}_{O_t}(p_t, (\boldsymbol{Z}, \boldsymbol{R}))$ at target can be simulated (by $\mathsf{S}_O$) without access to secrets. □

Next, we treat security properties based on a notion of simulatability such as $d$-NI, $d$-SNI, $d$-PINI, etc.

**Definition 4.8** (Simulatability)**.** *A program $p$ is called $\phi$-simulatable if there exists a simulator* S *that never outputs $\perp$ such that for any input $x$, observation set $O$, and output observation $\chi$, there exists an input set $I$ such that $\phi(|O|, \chi, I)$ and*

$$\mathcal{V}_O^\chi(p, (x, \boldsymbol{R})) \equiv \mathsf{S}_O^\chi(x_{|I}).$$

Let us note that $\phi$ only depends on the cardinality of $O$, and that the input $x$ contains the shares of the input secrets (public values would be included in $I$).

For example, the notion of $d$-SNI can be defined using

$$\phi(|O|, \chi, I) \coloneqq |O| + |\chi| \leq d \Rightarrow |I| \leq |O|.$$

**Lemma 4.9** (Preservation of simulatability)**.** *If a program $p_s$ is $\phi$-simulatable and $p_t = C(p_s)$ for a leakage-preserving compiler $C$, then $p_t$ is $\phi$-simulatable.*

*Proof.* The proof proceeds similarly to that of Definition 4.7 for the simulation of timing and value leakage. For the outputs selected by $\chi$, compiler correctness implies that the source and target program outputs are identically distributed, therefore a simulator for the target can rely on the source output value simulation. □

$$
\begin{aligned}
\mathbf{c} &::= \varepsilon \mid \mathbf{i}; \mathbf{c} \\
\mathbf{i} &::= \mathbf{x} := \mathbf{e} \mid \mathbf{x} := a[\mathbf{e}] \mid a[\mathbf{e}] := \mathbf{x} \mid \mathbf{x}^* := \#\mathbf{mop}(\mathbf{x}^*) \\
&\quad \mid \texttt{if}\ \mathbf{e}\ \texttt{then}\ \mathbf{c}\ \texttt{else}\ \mathbf{c} \mid \texttt{while}\ \mathbf{e}\ \texttt{do}\ \mathbf{c} \\
\mathbf{e} &::= \mathbf{n} \mid \mathbf{b} \mid \mathbf{x} \mid (\neg \mathbf{e}) \mid (-\mathbf{e}) \mid (\mathbf{e} + \mathbf{e}) \mid (\mathbf{e} == \mathbf{e}) \\
&\quad \mid (\mathbf{e}\ \&\ \mathbf{e}) \mid (\mathbf{e} < \mathbf{e}) \\
\mathbf{mop} &::= \texttt{neg} \mid \texttt{not} \mid \texttt{add} \mid \texttt{and} \mid \texttt{cmp}
\end{aligned}
$$

Figure 2. Syntax of our language.

## 4.3. Concrete Model Language

In this section, we introduce a language for which we prove leakage preservation of selected compiler passes. The language is designed as a simple model for JASMIN [27], with a focus on the parts of the language that are relevant to side-channel leakage. In particular, our core language contains basic instructions such as assignments, load- and store instructions, conditionals, while loops, and machine operations. These machine operations, i.e., architecture-specific instructions, are modeled by the $\mathbf{x}^* := \#\mathbf{mop}(\mathbf{x}^*)$ instruction in our language, where $\mathbf{x}^*$ denotes a list of registers and $\mathbf{mop}$ is one of multiple machine operations. The formal syntax of our language appears in Fig. 2; its semantics is based on the single-step definition in Fig. 3 and the multi-step extension introduced previously.

For *timing leakage* we follow the classic notion of cryptographic constant-time: timing leakage contains all memory access addresses and branch directions. Formally, it is null for assignments and machine operations $(-)$, the branch condition for branch instructions $(\mathrm{branch}\ b)$, and the accessed array and index for loads and stores $(\mathrm{addr}\ a\ j)$. *Value leakage* contains other data-dependent leakage, e.g., all intermediate values computed during execution. For expressions, this (recursively) includes the leakage of all subexpressions, which for expression $e$ is denoted by $\mathcal{L}_\rho(e)$:

$$\mathcal{L}_\rho(e) = \begin{cases} [\![e]\!]_\rho & \text{if } e \text{ is a variable or constant,} \\ \mathcal{L}_\rho(e_1), \ldots, \mathcal{L}_\rho(e_n), [\![e]\!]_\rho & \text{otherwise,} \end{cases}$$

where, in the second case, $e = o(e_1, \ldots, e_n)$ for an $n$-ary operator $o$. This recursive descent into subexpressions is crucial to prove some of the compiler passes. For machine operations $\vec{x} := \#m(\vec{o})$, we leave the leakage abstract and denote it as $\mathcal{L}^{\mathrm{mop}}(\vec{v}, m)$. This allows us to instantiate hardware-dependent leakage models (e.g., including glitch leakage). Our proofs hold for any function $\mathcal{L}^{\mathrm{mop}}$.

## 4.4. Proving Preservation of Compiler Passes

Based on the language defined in the previous section, we can formally specify and analyze the following compiler passes, which model the main code transformations performed by a compiler such as `jasminc`:

**Dead-Code Elimination** removes dead assignments, machine operations, and loads.

$$\text{assign}\ \frac{i = x := e \quad \rho' = \rho[x \mapsto [\![e]\!]_\rho]}{\langle i; c, \rho, \mu\rangle \xrightarrow[\mathcal{L}_\rho(e)]{-} \langle c, \rho', \mu\rangle}$$

$$\text{mop}\ \frac{i = \vec{x} := \#m(\vec{o}) \quad [\![\#m(\vec{o})]\!]_\rho = \vec{v} \quad \rho' = \rho[\vec{x} \mapsto \vec{v}]}{\langle i; c, \rho, \mu\rangle \xrightarrow[\mathcal{L}^{\text{mop}}(\vec{v}, m)]{-} \langle c, \rho', \mu\rangle}$$

$$\text{load}\ \frac{\begin{array}{c} i = x := a[e] \\ [\![e]\!]_\rho = j \qquad j \in [0, |a|) \\ \mu(a, j) = v \quad \rho' = \rho[x \mapsto v] \end{array}}{\langle i; c, \rho, \mu\rangle \xrightarrow[\mathcal{L}_\rho(e), v]{\text{addr } a\ j} \langle c, \rho', \mu\rangle}$$

$$\text{store}\ \frac{\begin{array}{c} i = a[e] := x \quad [\![x]\!]_\rho = v \\ [\![e]\!]_\rho = j \qquad j \in [0, |a|) \\ \mu' = \mu[(a, j) \mapsto v] \end{array}}{\langle i; c, \rho, \mu\rangle \xrightarrow[\mathcal{L}_\rho(e), v]{\text{addr } a\ j} \langle c, \rho, \mu'\rangle}$$

$$\text{if}\ \frac{\begin{array}{c} i = \texttt{if}\ e\ \texttt{then}\ c_t\ \texttt{else}\ c_f \\ [\![e]\!]_\rho = b \end{array}}{\langle i; c, \rho, \mu\rangle \xrightarrow[\mathcal{L}_\rho(e)]{\text{branch } b} \langle c_b; c, \rho, \mu\rangle}$$

$$\text{while}_\text{T}\ \frac{i = \texttt{while}\ e\ \texttt{do}\ c_t \quad [\![e]\!]_\rho = \texttt{true}}{\langle i; c, \rho, \mu\rangle \xrightarrow[\mathcal{L}_\rho(e)]{\text{branch true}} \langle c_t; i; c, \rho, \mu\rangle}$$

$$\text{while}_\text{F}\ \frac{i = \texttt{while}\ e\ \texttt{do}\ c_t \quad [\![e]\!]_\rho = \texttt{false}}{\langle i; c, \rho, \mu\rangle \xrightarrow[\mathcal{L}_\rho(e)]{\text{branch false}} \langle c, \rho, \mu\rangle}$$

Figure 3. Single-step semantics with timing and value leakage for each instruction.

**Loop Peeling** peels an iteration of a `while` loop by rewriting it as an `if` statement whose true branch is the loop body.
**Constant Propagation** substitutes variables that contain compile-time constants by their values.
**Constant Folding** evaluates a constant expression if *all* subexpressions are constant.
**Dead-Branch Elimination** removes conditionals with constant guards and loops with false guards.
**Flattening and Common Subexpression Elimination** rewrites code into three-address code by splitting expressions appropriately.
**Register Allocation** renames register variables with target registers given a register map.
**Stack Allocation** improves stack usage via **Array Concatenation** of smaller arrays into a bigger one, and **Array Reuse** to minimize additional allocations.

Since our goal is to prove that compiler transformations preserve leakage properties, it is often unnecessary to formalize every detail of the compiler algorithms themselves. For instance, in the case of register allocation, the specific renaming chosen by the allocator is irrelevant, provided that it preserves the program's semantics.

To abstract away such implementation details, several compiler passes are modeled as having access to analysis oracles. Rather than specifying the internal workings of these oracles, we characterize them through the properties they must satisfy to ensure the correctness of the compilation pass. These properties serve as assumptions in our proofs and are checked by the compiler before applying the corresponding transformation; compilation fails whenever the required conditions are not met. This approach allows us to focus on the leakage-preservation guarantees of the transformations while avoiding unnecessary formalization of auxiliary analyses.

The proofs of the passes are done using Definition 4.5 by proving that the pass satisfies single step leakage simulation (Definition 4.4), which requires defining the invariant $\phi$ relating source and target states and the $\mathsf{sim}^1_\tau$ and $\mathsf{sim}^1_v$ leakage simulators. For all passes, the invariant can be decomposed into two parts. The first expresses that the code of the target state corresponds to the compilation of the code of the source state. The second relation captures the correspondence between the data contained in the source and target states (the variable map $\rho$ and memory $\mu$, as well as, when necessary, the analysis information used to perform the transformation). This additional information allows us to establish the validity of the analysis results on which the transformation relies.

All the proofs presented in this paper have been formally verified in RocQ and are included in the artifact (cf. Appendix C). For ease of reference, the name of each compiler pass coincides with the name of the corresponding file in the formal development.

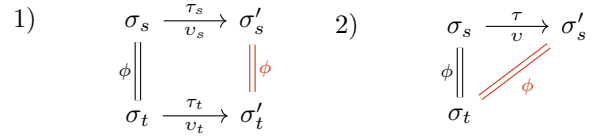
**Dead-Code Elimination.** This pass removes instructions that are identified as dead, namely assignments, loads, and machine operations whose results are never used. Formally, for each program point, an oracle provides a set of live variables. The compiler then removes dead instructions, i.e., instructions that do not define any live variable.

The liveness oracle is considered correct if every variable accessed by a live (i.e., non-eliminated) instruction is itself marked as live.

We now define the invariant $\phi$ used in Definition 4.4 to relate source and target program states. As explained above, $\phi$ encodes the assumption that the target program is obtained from the source program by applying the Dead-Code Elimination pass. In addition, $\phi$ requires that the memories of the two states are identical and that both states agree on the values of all variables marked as live by the analysis at the current program point.

As a technical detail, the compiler not only transforms the program but also verifies the consistency of the liveness information provided by the oracle. For example, consider an assignment $x := e$. If $x$ is marked as live after the execution of the instruction, the compiler checks that every variable occurring in $e$ is marked as live before the instruction. It also verifies that every variable that is live after the instruction, except $x$, is live before the instruction. Compilation fails whenever one of these conditions is violated.

It therefore suffices to consider a single step of the source program. There are thus two cases to be handled separately: 1) when the instruction is live, and 2) when the instruction is dead, which we illustrate as:




When the next instruction is live, the pass does not alter the instruction in any way, so the steps in the source and the target are equal and emit the same leakage, and the relation $\phi$ is preserved. When the next instruction is dead, the pass removes the instruction, so the target takes no step and stays in the same state. In this case, no leakage is emitted.

**Constant Propagation.** This pass replaces register variables with their values if these values are known at compile time. Knowing which variables contain constants requires prior code analysis, which we again shift to an analysis oracle. Formally, the compiler pass gets as input the result of the analysis, which for each program point maps constant variables to their values. For correctness, we ensure that this mapping is consistent with the variable map during execution, i.e., the analysis result and the variable map agree on all variables that contain constant values; as for dead-code elimination, this is enforced by the invariant $\phi$.

Since Constant Propagation only replaces all occurrences of a constant variable's name with the stored value, it does not change the observable value leakage and it does not add or remove instructions. The observed timing and value leakage from source and target are identical for this compiler pass.

**Constant Folding.** This pass simplifies expressions by evaluating purely constant expressions at compile time. This enables other passes, such as Dead-Branch Elimination, to optimize the program even further. As this pass only transforms the expressions, it neither adds nor removes any instruction. Correctness of the Constant Folding transformation immediately implies timing leakage preservation, as the timing leakage for expression evaluation is preserved. The value leakage, however, is modified for all instructions that evaluate expressions. Given an expression $e$, let $e'$ be the result of constant-folding $e$. Then, the value leakage $\mathcal{L}_\rho(e')$ is similar to $\mathcal{L}_\rho(e)$, up to some constant values corresponding to the folded expressions. Therefore, we can define a value leakage simulator $\mathsf{sim}^e_v$ that recomputes $\mathcal{L}_\rho(e)$ from $\mathcal{L}_\rho(e')$.

**Loop Peeling.** This pass peels off one iteration of a `while` loop into an `if` instruction:

$$\texttt{while } e \texttt{ do } c \quad \rightsquigarrow \quad \begin{array}{l}\texttt{if } e \texttt{ then}\\ \quad c\texttt{; while } e \texttt{ do } c\\ \texttt{else } \varepsilon\end{array}$$

Rewriting a loop into a conditional and the same loop again does not change the semantics of the program. Additionally, evaluating an `if` has the same leakage behavior as a `while`, so the observable timing and value leakage stays the same after loop peeling.

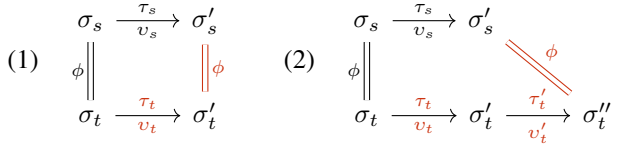
**Dead-Branch Elimination.** After running Loop Peeling, Constant Propagation, and Constant Folding, the resulting code may contain loops or conditionals where the conditions evaluate to compile-time constants. At this stage, Dead-Branch Elimination removes `while` loops where the guard evaluates to `false`, and replaces `if` statements with the code of the appropriate branch, i.e., the code of the `true` branch if the conditional evaluates to `true`, and the code of the `false` branch if the conditional evaluates to `false`.

Similarly to Dead-Code Elimination, this pass removes some instructions in the target execution, while the other instructions remain identical between the source and target. The proof of leakage preservation for Dead-Branch Elimination is therefore very similar to the one for Dead-Code Elimination. Since Dead-Branch Elimination does not change the values of variables, we can use a slightly stronger invariant to relate the source and target executions (equality of memories and variable maps).

**Flattening and Common Subexpression Elimination.** Flattening rewrites nested expressions into three-address code by introducing temporaries for subexpressions: e.g., $x := a + (b + c)$ becomes $t := b + c;\ x := a + t$. It is a prerequisite for instruction selection (lowering high-level expressions to target instruction set architecture (ISA) operations), since machine instructions operate on at most a few operands. Common subexpression elimination (CSE) applies a similar transformation to optimize code: a subexpression occurring in several instructions is computed once into a temporary and then reused.

We capture both passes with a single rewrite, parameterized by three oracles: a liveness analysis, an oracle choosing a dead or fresh temporary variable $t$, and an oracle choosing a subexpression $s$. At each instruction $i$, the oracle proposes an expression $s$; if $s$ is a subexpression of $i$, the pass inserts $t := s$ for temporary variable $t$ and substitutes $t$ for $s$ in $i$. Our preservation proof is independent of the oracle implementations, and the invariant $\phi$ is similar to the one used for the Dead-Code Elimination pass.

We distinguish (1) the case where the oracle proposes an expression that is *not* a subexpression of $i$ (no rewrite), and (2) the case where it is, i.e., one source step is matched by two target steps. In the diagrams below, black arrows and equalities are given; red marks what the proof establishes.




Case (1), handled in `non_flattening_step`, is immediate by the source step, equality of starting states, and determinism. Case (2), handled in `flattening_step`, requires simulators that, from the source value leakage $v_s$, reconstruct (a) the leakage $v_t$ of $t := s$ and (b) the leakage $v'_t$ of $x := e[s/t]$. The key observation is that in the value leakage model with glitches, evaluating an expression leaks recursively over all its subexpressions, yielding a tree

(encoded as a list). The leakage of $s$ is therefore already *contained* in $\upsilon_s$ and can simply be peeled off to produce $\upsilon_t$. For $\upsilon'_t$, the simulator replaces each subtree in $\upsilon_s$ corresponding to $s$ with the leakage of reading $t$, which itself is part of the simulated leakage of step one. Timing simulators are trivial; preserving the state invariant $\phi$ relies on functional correctness under substitution.

Preservation in this pass crucially depends on glitch leakage being modeled. In the value leakage model without glitches, a source value leakage corresponds to the single value of $e$ and carries no information about subexpression $s$; no simulator for $\upsilon_t$ can exist, so this pass cannot be proven leakage-preserving in the glitch-free value-probing model.

**Register Allocation.** This pass performs variable renaming with respect to a given mapping of variables to target registers. For each program point, the renaming oracle provides a partial map from source variables to target registers, and the pass renames all register variables accordingly.

For correctness, the invariant $\phi$ requires that at each program point source and target memories are equal and if the oracle maps a source variable $x$ to a target register $r$, then the value of $x$ is equal to the value of $r$. Regarding leakage behavior, we prove that all steps in source and target emit the same timing and value leakage, and leakage preservation follows immediately.

**Array Concatenation.** This pass merges multiple arrays into a single larger array that subsumes their storage. The purpose is to model the allocation of a function's stack frame, in which local arrays are laid out within a single contiguous memory block. The larger array thus serves as an abstract stack frame containing the data of the original arrays. Choosing an appropriate array layout requires prior analysis. In practice, such analyses are common, for instance to enforce specific memory-alignment constraints on data.

Formally, the pass takes as input the result of this analysis, which maps each source array $a_s$ to a slot within a target array $a_t$. A slot is represented by a pair consisting of the target array $a_t$ and the offset at which the contents of $a_s$ are stored within $a_t$. For correctness, we require that the slot assigned to $a_s$ is large enough to accommodate all of its elements, and that the slots assigned to distinct source arrays are pairwise disjoint, ensuring that their contents never overlap.

Transforming $x := a[e]$ and $a[e] := x$ instructions is done by replacing the accessed source array with the mapped target array and wrapping the index expression inside an addition with the mapped offset. Other instructions remain unchanged. While this retains the amount of leakage, it changes the content of both timing and value leakage. As depicted in Fig. 3, the $x := a[e]$ instruction leaks $\mathrm{addr}\ a\ j$ as timing leakage and $\mathcal{L}_\rho(e), v$ as value leakage, where $\rho$ is the current variable map, $\mu$ is the current memory state, $j = [\![e]\!]_\rho$ and $v = \mu(a, j)$. After performing array concatenation, the leakage produced by the transformed instruction is $\mathrm{addr}\ a'\ j'$ and $\mathcal{L}_\rho(e + o), v$, where $(a', o)$ is the slot that the analysis assigns to $a$ and $j' = [\![e + o]\!]_\rho$. The leakage of the $a[e] := x$ instruction is transformed in the same way, except that $v$ is equal to $[\![x]\!]_\rho$ instead of $\mu(a, j)$. The difference between source and target leakage is therefore only dependent on the original leakage, the source array, and the result of the analysis. Since we have access to the result of the analysis when defining $\mathsf{sim}_\tau$ and $\mathsf{sim}_\upsilon$, and the address of the source array can be obtained from the current source timing leakage, we can prove leakage preservation for Array Concatenation.

**Array Reuse.** This pass optimizes a program by reusing dead arrays to reduce stack space. An array is dead at an instruction if it is no longer accessed in any reachable instruction. A dead array can therefore be reused by an array of the same length. In that sense, this pass is very similar to the Register Allocation pass.

Array Reuse only changes timing leakage, leaving any value leakage unchanged. Since we have access to the result of the analysis when defining $\mathsf{sim}_\tau$, we can simply transform the timing leakage of array accesses accordingly, allowing us to prove leakage preservation for the Array Reuse pass.

## 4.5. JASMIN Compiler

In this section, we discuss the relation between the leakage preservation proofs of §4.4 on our core language and the JASMIN compiler [27]. The core language we describe in §4.3 is an abstraction of JASMIN: an imperative language with value-based semantics, scalar and array values, as well as explicit machine operations.

The main difference between the languages is that JASMIN has functions, while our language does not. Since calling a function in JASMIN is simply compiled to a $\#\mathbf{call}(\cdot)$ machine operation, it does not involve any data leakage. Note that the JASMIN compiler does not spill variables or move registers on function calls; it instead adds constraints on a global cross-function register allocation. Furthermore, only static function calls are supported in JASMIN (no function pointers), so the leakage that arises from calling a function and returning from it (excluding the function body itself) is constant and will not be a problem for leakage preservation.

We have designed our core compiler to accurately reflect the non-trivial effects on leakage in JASMIN's main code transformations. Our development therefore addresses the main technical challenges of a proof for the actual JASMIN compiler. Because our proofs target detailed models of JASMIN's passes rather than idealized ones, the correspondence between our pipeline and JASMIN's is close, which gives us confidence that these JASMIN passes preserve security. This correspondence is summarized in Table 1. We next argue that the three remaining passes preserve the leakage as well (using essentially the same arguments as the ones already developed in the previous section).

First, *Instruction Selection* transforms each assignment of an expression to a variable into a single machine operation; for example, $x := x_1 + x_2$ is transformed into $x := \#\mathbf{add}(x_1, x_2)$. This transformation does not change timing leakage. For the value leakage, we once again crucially rely on the leakage of expressions $\mathcal{L}_\rho(e)$ containing all the operands of the (sub-)expressions. Indeed, the machine

TABLE 1. OVERVIEW OF JASMINC PASSES AND THEIR EFFECT ON LEAKAGE, COMPARED TO THE PASSES PROVEN WITH ROCQ IN THIS WORK. NON-TRIVIAL JASMINC PASSES HIGHLIGHTED IN ITALICS DO NOT CORRESPOND TO PROVEN TRANSFORMATIONS OF §4.4.

| JASMIN pass | Effect on timing and value leakage |
|---|---|
| Dead-Code Elimination | Same as our Dead-Code Elimination. |
| Function Pruning | None (removes unused functions). |
| Remove Unused Results | Similar to Dead-Code Elimination (removes unused return values). |
| Inlining | Removes function call instructions. |
| Constant Propagation | Same as our Constant Propagation, Folding, and Dead-Branch Elimination. |
| Inline Propagation | Same as Constant Propagation (for inline variables). |
| Unrolling | Same as repeated Loop Peeling. |
| Stack Allocation | Same as our Array Reuse and Array Concatenation. |
| Register Renaming | Same as our Register Allocation. |
| *Instruction Selection* | Replaces assignments with machine operations. |
| *Linearization* | Replaces if/while with conditional branches, inserts unconditional branches. |
| *Tunneling* | Collapses chains of consecutive unconditional branches. |

operation leakage $\mathcal{L}^{\text{mop}}(\vec{v}, m)$, which is an arbitrary function of the values of these operands $\vec{v}$, can then be simulated from the expression leakage. Let us note that such flexibility in the leakage of concrete operations is needed in some cases where the machine operation performs more computations than specified by the expression. As a concrete example, some instructions always compute flags: the above addition could then be compiled to $(c, x) := \#\mathbf{add}(x_1, x_2)$, where $c$ is the carry.

Next, *Linearization* replaces structured control flow (`if`/`while`) with conditional and unconditional branches. In this transformation, each execution of a conditional branch in the target program corresponds to the evaluation of an `if` or `while` with the same condition (or its negation), which preserves the leakage (with both instructions including the Boolean in their timing and value leakage). Regarding the unconditional branches, these cause only constant leakage and their addition or removal therefore preserves the leakage. Let us finally note that this last argument also applies to the *Tunneling* pass.

## 4.6. Securing and Preserving Transition Leakage

So far, we considered the value-probing model with glitches but without transitions as a starting point, since common gadgets, e.g., [5], [36], are secure in the value-probing model. As we move towards deploying gadgets on hardware, transition leakage, another ubiquitous source of value leakage, may now occur whenever a hardware register is overwritten. Adding protection against transition leakage is typically a separate step after implementation in the base leakage model, as it is device-specific. We show, as a proof of concept, how to extend the JASMIN compiler and our gadgets with added measures to protect against register transitions with glitches. To account for register transitions, we assume any write to variable $v$ in an assignment, load, or machine operation leaks $[\![v]\!]_\rho$ concatenated with the instruction's value leakage as modeled in our semantics (cf. Fig. 3).

The main challenge with respect to transitions is to prevent variables with content depending on distinct shares from overwriting each other in registers. Therefore, the main compiler pass we are concerned with is register allocation, but we note that dead-code elimination should also be considered since it may remove "dummy" instructions that are commonly used to harden the source code against transitions [37], [38]. We propose a solution to this challenge with JASMIN source code annotations that control how these two compiler passes operate. First, the `#[inplace]` annotation on an assignment forces the register allocation to use the same register for the assigned value and for the first operand. Second, the `#[keep]` annotation on an instruction prevents its removal by dead-code elimination. Jointly, these lightweight compiler features[4] allow a JASMIN program to prevent transitions with a sensitive variable $x$ by resetting its register to 0 with an instruction `#[keep, inplace] x = x ^ x;` (other optimized transition prevention techniques are also enabled by judicious use of these annotations).

We extend our XOR gadget implementation with the necessary annotations to ensure $d$-NI security in the value-probing model with transitions and present the resulting code in Listing 2. It computes a share-wise XOR, i.e., `c[i] = a[i] ^ b[i]`. We must avoid combining two shares of the same input, e.g., `a[i]` with `a[j]`, or `c[i]` with `a[j]` for $i \neq j$. We achieve this by precisely tracking which variable contains these shares: we first load `a[i]` and `b[i]` into register-allocated variables `x` and `y` (lines 7–8), ensure the computation of the result on line 11 reuses the same register with `#[inplace]`, and write the result back to memory (line 12). Afterwards, since we have a guarantee that only two registers contain share-dependent data, we can securely clear them on lines 16–17. In summary, the two core principles to harden our source are 1) to clear registers explicitly with `#[inplace]` and `#[keep]` annotations, and 2) to control the destination registers with `#[inplace]` annotations.

Let us note that verification tools like SCVERIF [12] could in principle verify that the program is secure in a model that includes all transition leakage that is not prevented by the annotations. We conjecture that the leakage in such a strong transition model is preserved by the JASMIN compiler (proving this would be an interesting goal for future work), which would imply that the compiled programs are secure against architectural transition leakage.

4. `#[keep]` was already available in the JASMIN compiler, and adding the necessary changes for `#[inplace]` required only a 14-line patch to the compiler [39].

```
fn _g_xor(reg ptr u32[N] c a b) -> reg ptr u32[N] {
    // allow compiler to move vars to different registers
    c = c; a = a; b = b;
    inline int i;
    reg u32 x y;
    for i = 0 to N {
        x = a[i];
        y = b[i];
        // inplace: allocate result value to the same
        //          register as the first operand
        #[inplace] x = x ^ y;
        c[i] = x;
        // keep: do not eliminate instruction
        // "_" : no logical value stored,
        //       only impacts target register
        #[keep, inplace] _ = x ^ x;
        #[keep, inplace] _ = y ^ y; }
    return c; }
```

Listing 2. JASMIN $d$-NI XOR gadget generic in the masking order $d = N - 1$, additionally hardened against transition leakage.

TABLE 2. LIBMASKING'S GADGETS AND MASKED ALGORITHMS.

| Name | Operation | Prop. |
|---|---|---|
| lm_xor | $z \leftarrow x \oplus y$ | $d$-NI |
| lm_xor_inplace | $x \leftarrow x \oplus y$ | $d$-NI |
| lm_not | $y \leftarrow \neg x$ | $d$-NI |
| lm_not_inplace | $x \leftarrow \neg x$ | $d$-NI |
| lm_shl | $y \leftarrow x \ll c$ | $d$-NI |
| lm_shl_inplace | $x \leftarrow x \ll c$ | $d$-NI |
| lm_shr | $y \leftarrow x \gg c$ | $d$-NI |
| lm_shr_inplace | $x \leftarrow x \gg c$ | $d$-NI |
| lm_and_derand | $z \leftarrow x \wedge y$ | $d$-SNI |
| lm_ref_derand | $y \leftarrow \mathsf{Ref}(x)$ | $d$-SNI |
| lm_keccakf1600 | $S \leftarrow$ KECCAK-$f[1600](S)$ | $d$-SNI |
| lm_sha3_$D$ | $H \leftarrow$ SHA3-$D(M)$ | $d$-SNI |
| lm_shake128 | $H \leftarrow$ SHAKE128$(M)$ | $d$-SNI |
| lm_shake256 | $H \leftarrow$ SHAKE256$(M)$ | $d$-SNI |

## 5. Evaluation and LIBMASKING

To evaluate the applicability of our results to the JASMIN compiler and to demonstrate JASMIN's usability for implementing masked software, we conduct a case study: we build LIBMASKING, a software library for masked cryptography. LIBMASKING consists of two parts: a collection of arbitrary-order masked gadgets and cryptographic implementations built as a composition of those gadgets. We release LIBMASKING and all accompanying files under an open-source license (cf. Appendix C). LIBMASKING's masked gadgets implement primitive operations over 32-bit Boolean words at *generic* masking order $d$. The gadgets cover linear operations like XOR, NOT, and constant shifts at $d$-NI [6], as well as a masked AND gadget and two refresh gadgets (ISW refresh and the quasilinear refresh of [36]) at $d$-SNI. The $d$-SNI [13] gadgets are available in a randomized variant, which samples its own randomness, and a derandomized variant (_derand), where the caller supplies the randomness. Table 2 gives an overview of the gadgets provided in our library.

The security claim of LIBMASKING's gadgets is primarily based on pen-and-paper security proofs (for the value leakage model), our analysis of §4.6 for transitions, and the informal correspondence between the secure compiler passes of our ROCQ development and the JASMIN compiler (cf. §4.5). However, the security guarantees of LIBMASKING's gadgets do not solely rely on pen-and-paper reasoning or on our ROCQ development: we verified the security properties of all (_derand) gadgets at the assembly level for masking orders $d = 1, \ldots, 4$ in the register-probing model with transitions using SCVERIF [12].

TABLE 3. ASSEMBLY-LEVEL SECURITY OF OUR MASKED GADGETS IN LIBMASKING (JASMIN) AND THEIR NAÏVE C COUNTERPARTS COMPILED WITH GCC AT VARIOUS OPTIMIZATION LEVELS. GADGETS THAT COMPILED TO SECURE ASSEMBLY ARE MARKED BY ✓ AND GADGETS WHERE COMPILATION INTRODUCED A SECURITY FLAW ARE MARKED BY ✗. SECURITY WAS CHECKED USING SCVERIF [12] AT ORDER $d = 1$.

| Gadget | JASMIN | gcc | | | |
|---|---|---|---|---|---|
| | (Ours) | -O0 | -O1 | -O2 | -O3 |
| Ref ($d$-SNI) | ✓ | ✓ | ✗ | ✗ | ✗ |
| AND ($d$-SNI) | ✓ | ✓ | ✗ | ✗ | ✗ |
| XOR ($d$-NI) | ✓ | ✓ | ✓ | ✓ | ✓ |

For this, we develop for each target architecture a generic SCVERIF leakage model. These generic models do not capture microarchitectural effects of a specific processor implementation executing the software. Instead, they exactly reflect the register-probing model (with and without transitions) and thereby corroborate our informal preservation conjecture on any verified gadget.

We also verify the functional correctness of the gadgets. For this purpose, the JASMIN compiler is used to extract the semantics of the gadgets to the EASYCRYPT proof assistant, in which we can write functional correctness proofs that, together with JASMIN's verified compiler, establish E2E functional correctness down to the assembly level.

Built on top of the gadget library, LIBMASKING provides masked implementations of the KECCAK-$f[1600]$ permutation, of the SHA3 family of hash functions, and of the SHAKE128 and SHAKE256 extendable-output functions. These implementations are manual compositions of the verified gadgets; their security is justified by the gadgets' $d$-NI and $d$-SNI composability properties and is not covered by any formal guarantees, such as those provided by masking compilers (cf. §6). Again, the implementations are generic in the masking order $d$ and are compiled to assembly by JASMIN for the specified order. The functional correctness of these implementations is validated by tests.

**Performance.** To demonstrate that JASMIN carries no inherent overhead compared to C, we reimplement our gadgets and the hardened KECCAK-$f[1600]$ function in C, which allows us to compare the performance of similar C and JASMIN implementations. The C gadgets are a naïve attempt at probing-secure gadgets that closely resemble our JASMIN gadgets. Note that we do not compare to optimized masked implementations; instead, we only assess how JASMIN inherently impacts performance when implementing

TABLE 4. RUNTIME IN µs OF MASKED KECCAK-$f[1600]$ (AVERAGED OVER 1000 RUNS) FOR MASKING ORDERS $d = 1, \ldots, 4$ COMPARING LIBMASKING'S IMPLEMENTATION (WITH VERIFIABLY TRANSITION-SECURE GADGETS) TO THE NAÏVE C IMPLEMENTATION. THE C GADGETS ARE NAÏVE IMPLEMENTATIONS OF PROBING-SECURE GADGETS AND COMPILED WITHOUT OPTIMIZATIONS (`gcc -O0`) AS OPTIMIZATIONS BREAK SECURITY GUARANTEES OF THE C GADGETS.

| Order | Naïve C (`gcc -O0`) | LIBMASKING (ours) | Speedup (C / ours) |
|---|---|---|---|
| $d = 1$ | 7160.9 | 1425.5 | 5.02 |
| $d = 2$ | 10 548.5 | 2317.8 | 4.55 |
| $d = 3$ | 14 011.3 | 3344.9 | 4.19 |
| $d = 4$ | 18 798.7 | 5458.8 | 3.44 |

masked algorithms.

To keep the comparison fair, we compile the gadgets with `gcc` and evaluate their security using SCVERIF in the register-probing model without transitions (see Table 3). We observe that, for $d = 1$, the gadgets are secure at the optimization level `-O0`, but they are insecure when optimizations are enabled (`-O1`, `-O2`, or `-O3`). Analyzing those failures for the AND gadget at `-O1` (see Appendix A), we find that these are caused by an optimization that exploits the associativity of XOR, essentially rewriting the computation $a_0b_1 \oplus (a_1b_0 \oplus r)$ into $(a_0b_1 \oplus a_1b_0) \oplus r$, which produces a secret-dependent intermediate value. We can therefore conclude that, as is already well known among practitioners, securely implementing masked gadgets in C is very challenging and ends up "fighting" the compiler, even when simplifying the problem by ignoring the transition leakage issue.

For the performance evaluation, we benchmark the masked KECCAK-$f[1600]$ implementation of LIBMASKING against the naïve C reference implementation. We run the implementation on an RP2350 ARM Cortex-M33 microcontroller clocked at 150 MHz. We report in Table 4 the performance of the C implementation compiled without optimizations (`gcc -O0`) to avoid violating security, and we measure performance on-device. The results in Table 4 indicate that our JASMIN implementation outperforms C compiled with `gcc -O0` by a significant amount: $5.02\times$ speedup at $d = 1$, $4.55\times$ speedup at $d = 2$, $4.19\times$ speedup at $d = 3$, and $3.44\times$ speedup at $d = 4$. To evaluate the performance ceiling, we also measure the performance of the insecure compilation of the C code with `gcc -O3`. The resulting performance is similar to LIBMASKING's (12% speedup at $d = 1$, $\pm 0\%$ speedup/slowdown at $d = 2$, 10% slowdown at $d = 3$, and 40% slowdown at $d = 4$).

Overall, we conclude that JASMIN enables us to write masked gadgets with strong security guarantees, without performance overhead compared to implementations in other languages (such as C).

## 6. Related Work

**Masking compilers.** Ishai et al. [5] introduce the $d$-probing model and present a compiler, known as the ISW compiler, that transforms every straight-line program into a $d$-probing secure program. This seminal work was followed by many proposals that improve over the original ISW compiler in efficiency, attacker model, or realism, e.g., [40], [41], [42]; the last of these inserts masking in source code, albeit without provable guarantees.

Another line of work develops domain-specific languages (DSLs) and compilers such as USUBA [43], TORNADO [44], TIGHTPROVE [45], and bitsliced higher-order masking toolchains [46], [47], which synthesize or generate masked code under source-level models. Both lines are complementary to our preservation-focused approach.

**Verified and secure compilation.** Formal verification of compilers is a long-standing challenge within the programming languages community. A prominent example of a verified compiler is COMPCERT [35], a moderately optimizing C compiler that comes with a proof of functional correctness. Motivated by frictions between security and compilation (reported in [48], [49], [50]), many other projects explore preservation of security, rather than functional correctness [51], [52], [53]. Work in this area, known as secure compilation, follows two different threads. A main thread of work explores the definition of secure compilers for general classes of security notions. Another thread of work shows that a concrete compiler, or a suitably modified variant, preserves a fixed notion of security. An example of the latter is COMPCERT-CT, which defines a variant of COMPCERT that preserves constant-time security [54]. Preservation of constant-time security has also been established for the JASMIN compiler [27], [28], [29]. Arranz Olmos et al. [30] prove in ROCQ preservation of speculative constant-time, which is a non-probabilistic and hence simpler security notion. We build upon their formalizations of the JASMIN passes, but for a different language featuring machine operations and a more complex, probabilistic notion of security.

Closest to our work is POMMES [55], a custom C-to-ARMv6-M assembly compiler. Unlike the preservation results above, POMMES provides no proof that compilation preserves security. Operating on the GIMPLE intermediate representation (IR) produced by `gcc`, it inherits the flattening of nested expressions to three-address code, yet assumes only that its C input is secure in the plain $d$-probing model. We show that this expression flattening preserves security *only* when the source is secure in the register-probing model *with glitches*, a strictly stronger assumption. POMMES also commits to a single, fixed leakage model that the authors acknowledge over-approximates practical leakage. It focuses on a model-specific register-allocation pass, incurring up to $3.5\times$ overhead over unoptimized but hardened C (`gcc -O0`). In contrast, our framework proves preservation for arbitrary leakage models and for several passes, and it applies transition-leakage hardening on top of optimized compilation, while having comparable performance to optimized C (cf. Table 4).

Closing the gap of integrating generic hardening against user-provided leakage models [12], [20] directly into a compilation pipeline remains an interesting research problem. Until then, automated post-compilation repair such as

ROSITA [56], [57] is a more practical alternative to harden compiled masked code.

**Formal verification of masked implementations.** There is a large body of work that develops formal verification tools for masked implementations [6], [13], [14], automated gadget verification [15], [16], [17], [18], masking-aware compilers [55], [58], and leakage models capturing concrete hardware behavior [12], [20]. At the hardware level, COCO [19] co-verifies masked software against gate-level CPU models. Gao et al. [59] reverse-engineer microarchitectural leakage features empirically to obtain practical leakage models. Bloem et al. [20] present an approach to provably complete leakage models for power side-channels, which employs a related concept to show that any leakage occurring at the gate level of processors can be simulated from leakage in a model. Any model satisfying this is then considered a contract, since implementations proven secure w.r.t. the model remain secure when executed on said processors. However, the work relates concrete realizations of contract and hardware, whereas we prove passes to preserve security for arbitrary input programs.

Many of these works operate with algorithms or source code, and assume that security is preserved by compilation. To our knowledge, ours is the first machine-checked proof that the main passes of a realistic compilation pipeline preserve the security notions these approaches rely on.

## 7. Conclusion

We present a language- and leakage-model-agnostic framework for provably secure compilation of probabilistic side-channel countermeasures. Its core is a *deterministic* leakage-preservation relation that implies preservation of the standard *probabilistic* security notions of masking ($d$-PS, $d$-NI, $d$-SNI, $d$-PINI, etc.) and of cryptographic constant-time, reducing each compiler pass to a deterministic trace-simulation obligation and avoiding probabilistic proofs altogether. We instantiate the framework in ROCQ on a model of the JASMIN core pipeline and several optimizing passes. We also tackle the challenge of securing implementations against architectural transition leakage by means of lightweight source annotations. Finally, we introduce LIBMASKING, an open-source JASMIN library of arbitrary-order masked gadgets that compiles securely to ARM and RISC-V and exposes C bindings.

Together, these results are a first step towards programming environments in which masked code is written or generated once, verified at the source level, optimized, and compiled to assembly while provably retaining its security. We identify two directions for further strengthening the security of masked software: mechanizing security proofs of the source JASMIN program with automated verification tools or EASYCRYPT (in addition to the functional correctness proof), and extending the formalization towards more realistic leakage models (e.g., including microarchitectural leakage).

## Acknowledgments

We thank Santiago Arranz Olmos and Vincent Laporte for their support with the ROCQ proof assistant and the JASMIN framework, and warmly thank Maximilian Orlt for his initial inputs that led to this project.

This work has been funded by the European Union through the ERC project 101077506 (AMAskZONE), under the European Union's research and innovation programs Horizon Europe (PRIVTOOLS/101124778) and Horizon 2020 (PSOTI/850990). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them. The project was co-funded by the Deutsche Forschungsgemeinschaft (DFG) within SFB 1119 CROSSING/236615297. It was supported by the German Federal Ministry of Research, Technology and Space (BMFTR), and the Hessian Ministry of Science and Research, Arts and Culture (HMWK) within the National Research Center for Applied Cybersecurity ATHENE. This work has received state aid managed by the National Research Agency under France 2030, bearing the reference 'ANR-22-PTCC-0001' (FORMASK PTCC P1).

## References

[1] D. Amiet, A. Curiger, L. Leuenberger, and P. Zbinden, "Defeating NewHope with a single trace," in *PQCrypto 2020*, 2020.

[2] S. Jendral, K. Ngo, R. Wang, and E. Dubrova, "Breaking SCA-protected CRYSTALS-Kyber with a single trace," in *HOST*, 2024.

[3] P. Ravi, S. S. Roy, A. Chattopadhyay, and S. Bhasin, "Generic side-channel attacks on CCA-secure lattice-based PKE and KEMs," *IACR TCHES*, 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8592

[4] European Union Agency for Cybersecurity, "Application of attack potential to smartcards," ENISA – EUCC State-of-the-Art Documents, Supporting Document Version 2, 2025. [Online]. Available: https://certification.enisa.europa.eu/publications/application-attack-potential-smartcards_en

[5] Y. Ishai, A. Sahai, and D. Wagner, "Private circuits: Securing hardware against probing attacks," in *CRYPTO 2003*, 2003.

[6] G. Barthe, S. Belaïd, F. Dupressoir, P.-A. Fouque, B. Grégoire, and P.-Y. Strub, "Verified proofs of higher-order masking," in *EUROCRYPT 2015, Part I*, 2015.

[7] M. Rivain and E. Prouff, "Provably secure higher-order masking of AES," in *CHES 2010*, 2010.

[8] J.-S. Coron, E. Prouff, M. Rivain, and T. Roche, "Higher-order side channel security and mask refreshing," in *FSE 2013*, 2014.

[9] J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F. Standaert, "On the cost of lazy engineering for masked software implementations," in *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS*, 2014. [Online]. Available: https://doi.org/10.1007/978-3-319-16763-3_5

[10] K. Papagiannopoulos and N. Veshchikov, "Mind the gap: Towards secure 1st-order masking in software," in *COSADE 2017*, 2017.

[11] B. Gigerl, R. Primas, and S. Mangard, "Secure and efficient software masking on superscalar pipelined processors," in *ASIACRYPT 2021, Part II*, 2021.

[12] G. Barthe, M. Gourjon, B. Grégoire, M. Orlt, C. Paglialonga, and L. Porth, "Masking in fine-grained leakage models: Construction, implementation and verification," *IACR TCHES*, 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8792

[13] G. Barthe, S. Belaïd, F. Dupressoir, P.-A. Fouque, B. Grégoire, P.-Y. Strub, and R. Zucchini, "Strong non-interference and type-directed higher-order masking," in *ACM CCS 2016*, 2016.

[14] G. Cassiers and F. Standaert, "Trivially and efficiently composing masked gadgets with probe isolating non-interference," *IEEE Trans. Inf. Forensics Secur.*, 2020.

[15] G. Barthe, S. Belaïd, G. Cassiers, P.-A. Fouque, B. Grégoire, and F.-X. Standaert, "maskVerif: Automated verification of higher-order masking in presence of physical defaults," in *ESORICS 2019, Part I*, 2019.

[16] D. Knichel, P. Sasdrich, and A. Moradi, "SILVER - statistical independence and leakage verification," in *ASIACRYPT 2020, Part I*, 2020.

[17] S. Belaïd, D. Mercadier, M. Rivain, and A. R. Taleb, "IronMask: Versatile verification of masking security," in *2022 IEEE Symposium on Security and Privacy*, 2022.

[18] M. Gourjon, M. Orlt, P. Pauls, and A. Treff, "eVer: Universal and automated verification of side-channel security for additive, inner product, polynomial and general code-based masking," *IACR Cryptol. ePrint Arch.*, 2026. [Online]. Available: https://eprint.iacr.org/2026/208

[19] B. Gigerl, V. Hadzic, R. Primas, S. Mangard, and R. Bloem, "Coco: Co-design and co-verification of masked software implementations on CPUs," in *USENIX Security 2021*, 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/gigerl

[20] R. Bloem, B. Gigerl, M. Gourjon, V. Hadzic, S. Mangard, and R. Primas, "Power contracts: Provably complete power leakage models for processors," in *ACM CCS 2022*, 2022.

[21] J. W. Bos, M. Gourjon, J. Renes, T. Schneider, and C. van Vredendaal, "Masking Kyber: First- and higher-order implementations," *IACR TCHES*, 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9064

[22] M. Abadi, "Protection in programming-language translations," in *ICALP 98*, 1998.

[23] M. Patrignani and D. Garg, "Secure compilation and hyperproperty preservation," in *CSF 2017 Computer Security Foundations Symposium*, 2017.

[24] C. Abate, R. Blanco, D. Garg, C. Hritcu, M. Patrignani, and J. Thibault, "Journey beyond full abstraction: Exploring robust property preservation for secure compilation," in *CSF 2019 Computer Security Foundations Symposium*, 2019.

[25] C. Abate, A. A. de Amorim, R. Blanco, A. N. Evans, G. Fachini, C. Hritcu, T. Laurent, B. C. Pierce, M. Stronati, and A. Tolmach, "When good components go bad: Formally secure compilation despite dynamic compromise," in *ACM CCS 2018*, 2018.

[26] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin, "Computer-aided security proofs for the working cryptographer," in *CRYPTO 2011*, 2011.

[27] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub, "Jasmin: High-assurance and high-speed cryptography," in *ACM CCS 2017*, 2017.

[28] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P.-Y. Strub, "The last mile: High-assurance and high-speed cryptographic implementations," in *2020 IEEE Symposium on Security and Privacy*, 2020.

[29] G. Barthe, B. Grégoire, V. Laporte, and S. Priya, "Structured leakage and applications to cryptographic constant-time and cost," in *ACM CCS 2021*, 2021.

[30] S. A. Olmos, G. Barthe, L. Blatter, B. Grégoire, and V. Laporte, "Preservation of speculative constant-time by compilation," *Proc. ACM Program. Lang.*, no. POPL, 2025.

[31] Signal. (2026) Commit `17e02e4` to `signalapp/noise-c`: "Use `mlkem-libjade`". [Online]. Available: https://github.com/signalapp/noise-c/commit/17e02e44c3

[32] S. A. Olmos, G. Barthe, L. Blatter, B. Grégoire, V. Laporte, and P. Torrini, "The jasmin compiler preserves cryptographic security," *CoRR*, vol. abs/2511.11292, 2025. [Online]. Available: https://arxiv.org/pdf/2511.11292v1

[33] S. Faust, V. Grosso, S. Merino Del Pozo, C. Paglialonga, and F.-X. Standaert, "Composable masking schemes in the presence of physical defaults & the robust probing model," *IACR TCHES*, 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/7270

[34] The Rocq Development Team. (2025) The Rocq prover. [Online]. Available: https://doi.org/10.5281/zenodo.17473943

[35] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, 2009.

[36] A. Battistello, J.-S. Coron, E. Prouff, and R. Zeitoun, "Horizontal side-channel attacks and countermeasures on the ISW masking scheme," in *CHES 2016*, 2016.

[37] G. Cassiers and F.-X. Standaert, "Provably secure hardware masking in the transition- and glitch-robust probing model: Better safe than sorry," *IACR TCHES*, 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8790

[38] S. Belaïd, G. Cassiers, C. Mutschler, M. Rivain, T. Roche, F.-X. Standaert, and A. R. Taleb, "SoK: A methodology to achieve provable side-channel security in real-world implementations," *CiC*, 2025.

[39] Jasmin. (2025) Commit `a4a0681` to `jasmin-lang/jasmin`: "Regalloc: use 'inplace' annotation as hint". [Online]. Available: https://github.com/jasmin-lang/jasmin/commit/a4a0681c78

[40] S. Belaïd, J.-S. Coron, E. Prouff, M. Rivain, and A. R. Taleb, "Random probing security: Verification, composition, expansion and new constructions," in *CRYPTO 2020, Part I*, 2020.

[41] S. Berndt, T. Eisenbarth, S. Faust, M. Gourjon, M. Orlt, and O. Seker, "Combined fault and leakage resilience: Composability, constructions and compiler," in *CRYPTO 2023, Part III*, 2023.

[42] A. Moss, E. Oswald, D. Page, and M. Tunstall, "Compiler assisted masking," in *CHES 2012*, 2012.

[43] D. Mercadier and P. Dagand, "Usuba: high-throughput and constant-time ciphers, by construction," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2019.

[44] S. Belaïd, P.-É. Dagand, D. Mercadier, M. Rivain, and R. Wintersdorff, "Tornado: Automatic generation of probing-secure masked bitsliced implementations," in *EUROCRYPT 2020, Part III*, 2020.

[45] S. Belaïd, D. Goudarzi, and M. Rivain, "Tight private circuits: Achieving probing security with the least refreshing," in *ASIACRYPT 2018, Part II*, 2018.

[46] D. Goudarzi, A. Journault, M. Rivain, and F.-X. Standaert, "Secure multiplication for bitslice higher-order masking: Optimisation and comparison," in *COSADE 2018*, 2018.

[47] D. Goudarzi and M. Rivain, "How fast can higher-order masking be in software?" in *EUROCRYPT 2017, Part I*, 2017.

[48] L. Simon, D. Chisnall, and R. J. Anderson, "What you get is what you C: Controlling side effects in mainstream C compilers," in *2018 IEEE European Symposium on Security and Privacy*, 2018.

[49] V. D'Silva, M. Payer, and D. X. Song, "The correctness-security gap in compiler optimization," in *IEEE Symposium on Security and Privacy Workshops, SPW*, 2015.

[50] J. Xu, K. Lu, Z. Du, Z. Ding, L. Li, Q. Wu, M. Payer, and B. Mao, "Silent bugs matter: A study of compiler-introduced security bugs," in *USENIX Security*, 2023.

[51] G. Barthe, B. Grégoire, and V. Laporte, "Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time"," in *CSF 2018 Computer Security Foundations Symposium*, 2018.

[52] F. Besson, A. Dang, and T. P. Jensen, "Securing compilation against memory probing," in *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2018*, 2018.

[53] S. T. Vu, A. Cohen, A. de Grandmaison, C. Guillon, and K. Heydemann, "Reconciling optimization with secure compilation," *Proc. ACM Program. Lang.*, no. OOPSLA, 2021.

[54] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu, "Formal verification of a constant-time preserving C compiler," *Proc. ACM Program. Lang.*, no. POPL, 2020.

[55] J. Zeitschner and A. Moradi, "PoMMES: Prevention of micro-architectural leakages in masked embedded software," *IACR TCHES*, 2024.

[56] M. A. Shelton, N. Samwel, L. Batina, F. Regazzoni, M. Wagner, and Y. Yarom, "Rosita: Towards automatic elimination of power-analysis leakage in ciphers," in *NDSS 2021*, 2021.

[57] M. A. Shelton, L. Chmielewski, N. Samwel, M. Wagner, L. Batina, and Y. Yarom, "Rosita++: Automatic higher-order leakage elimination from cryptographic code," in *ACM CCS 2021*, 2021.

[58] A. Abromeit, F. Bache, L. A. Becker, M. Gourjon, T. Güneysu, S. Jorn, A. Moradi, M. Orlt, and F. Schellenberg, "Automated masking of software implementations on industrial microcontrollers," in *Design, Automation & Test in Europe Conference & Exhibition, DATE*, 2021.

[59] S. Gao, E. Oswald, and D. Page, "Towards micro-architectural leakage simulators: Reverse engineering micro-architectural leakage features is practical," in *EUROCRYPT 2022, Part III*, 2022.

## Appendix A. Leakage Preservation Counterexample in C

We give a concrete probing-security violation in the naïve C implementation of the AND gadget compiled with `gcc -O1` for $d = 1$. Listing 3 shows the compiled ARM assembly. The ISW AND gadget for $d = 1$ computes

$$c_0 = a_0 b_0 + r,$$
$$c_1 = a_0 b_1 + a_1 b_0 + a_1 b_1 + r,$$

where $r$ is the randomness, $c_0, c_1$ are the output shares, and $a_0, a_1, b_0, b_1$ are the input shares. Security requires that no single intermediate value reveals more information than is obtained by learning one share of each input. However, `gcc` reorders the computation so that the attacker can probe the intermediate in register `r3` on line 20 of the listing, thereby learning $a_0 b_1 + a_1 b_0$. This breaks $d$-SNI and is in contrast to the C source code, where the intermediate $a_0 b_1 + a_1 b_0$ only ever shows up masked by the randomness $r$.

```
1  push    {r4, r5, lr}
2  ldr     r4, [r1, #0] ; r4 = a[0]
3  ldr     r5, [r2, #0] ; r5 = b[0]
4  and.w   ip, r5, r4   ; ip = a[0] & b[0]
5  str.w   ip, [r0]     ; c[0] = a[0] & b[0]
6  ldr     r4, [r1, #4] ; r4 = a[1]
7  ldr     r5, [r2, #4] ; r5 = b[1]
8  and.w   lr, r5, r4   ; lr = a[1] & b[1]
9  str.w   lr, [r0, #4] ; c[1] = a[1] & b[1]
10 ldr     r4, [r3, #0] ; r4 = r
11 eor.w   r3, r4, ip   ; r3 = r + a[0] & b[0]
12 str     r3, [r0, #0] ; c[0] = r + a[0] & b[0]
13 ldr     r3, [r1, #0] ; r3 = a[0]
14 ldr     r1, [r1, #4] ; r1 = a[1]
15 ldr     r5, [r2, #4] ; r5 = b[1]
16 ands    r3, r5       ; r3 = a[0] & b[1]
17 ldr     r2, [r2, #0] ; r2 = b[0]
18 ands    r2, r1       ; r2 = a[1] & b[0]
19 ; The following line contains the violation:
20 eors    r3, r2       ; r3 = a[0] & b[1]
21                      ;    + a[1] & b[0]
22 eor.w   r3, r3, lr   ; r3 = a[0] & b[1]
23                      ;    + a[1] & b[0]
24                      ;    + a[1] & b[1]
25 eors    r3, r4       ; r3 = ... + r
26 str     r3, [r0, #4] ; c[1] = ...
27 pop     {r4, r5, pc}
```

Listing 3. ARM assembly of AND gadget compiled with `gcc -O1` with a $d$-SNI violation on line 20.

## Appendix B. Generative AI Usage

This paper was edited for grammar using Grammarly and ChatGPT. All lemmas, definitions, and proof approaches were developed by the authors without AI assistance. AI tools assisted novice ROCQ users in drafting individual proof scripts; in this mode, the agents had no ability to modify code directly.

In one dedicated experiment, we paired the Model Context Protocol (MCP) server ROCQ-MCP[5] with Claude Opus 4.6 to close the lemma `non_flattening_step` of the flattening pass. The proof structure of this lemma mirrors that of `live_step` from dead-code elimination, which we had previously proved manually. Given this analogy, access to a copy of our ROCQ formalization, and a high-level proof sketch, the agent closed the goal within roughly 30 minutes, consuming 15–20M tokens.

All AI-generated content was thoroughly reviewed by the authors, and the authors retain full responsibility for all AI-generated code. We verified that generated ROCQ proofs do not modify definitions or introduce axioms, so that AI assistance cannot compromise the soundness of our results.

## Appendix C. Open Science

We release all artifacts of this work at https://doi.org/10.5281/zenodo.22827961. The repository contains our ROCQ proofs, LIBMASKING sources (including the gadgets, KECCAK implementations, EASYCRYPT proofs, SCVERIF leakage models, and test environment), and the benchmarking environment along with detailed instructions to verify our proofs and reproduce our results.

5. https://github.com/LLM4Rocq/rocq-mcp

## Appendix D.
## Ethics Considerations

This work presents a framework for secure compilation of side-channel countermeasures and a library of masked gadgets. Our framework involves no human subjects, no collection or processing of user data, and no analysis of real-world vulnerabilities or deployed systems. Our results of secure and efficient compilation of masking greatly improve the utility of masking for the security community. We do not modify the underlying cryptographic primitives or masking algorithms, thereby maintaining the original security guarantees of the masked algorithms. As such, we believe that our work overall benefits the research community.

While we do not immediately see potential for misuse due to our contributions, there may be risks tied to masking or the application of side-channel countermeasures in general. Since masking is used on low-power platforms, security breaches harm users who rely on the security of underlying masking schemes, as well as the security of deployed implementations. However, we argue that our work provides a valuable step in mitigating harm, as we can rule out certain security breaches that could have occurred due to insecure compilation. Nonetheless, we strongly advise practitioners and researchers to carefully check deployed cryptographic algorithms and verify side-channel mitigation, especially in scenarios with large side-channel attack surfaces.

The authors attest that we have read the ethics considerations as per the conference call for papers. We believe that our research was conducted ethically and that our results are designed to benefit the security research community.

## Appendix E.
## Meta-Review

The following meta-review was prepared by the program committee for the 2027 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

### Summary

This paper develops a framework for proving that compiler transformations preserve probabilistic side-channel security for masked cryptographic programs. It introduces a deterministic leakage-preservation relation, mechanizes key results in ROCQ for a JASMIN-like compiler pipeline, and provides the LIBMASKING library together with validation and performance results.

### Scientific Contributions

- Provides a Valuable Step Forward in an Established Field
- Addresses a Long-Known Issue
- Creates a New Tool to Enable Future Science

### Reasons for Acceptance

1) The paper provides a valuable step forward in an established field. Its deterministic leakage-preservation relation reduces preservation of probabilistic masking notions to deterministic simulation arguments for individual compiler transformations. The main benefit is that each compiler pass can be handled without introducing a separate probabilistic proof.
2) The paper addresses a long-known issue. Compiler optimizations can invalidate masking countermeasures, and the paper gives a principled way to reason about this problem. The result on expression flattening and the need for glitch-aware leakage are particularly insightful. The source-level mechanisms for transition leakage also help connect the formal framework to compiled code.
3) The paper creates a new tool to enable future science. The ROCQ development and the LIBMASKING JASMIN library are a substantial effort. The released artifact, SCVERIF validation of generated binaries, and performance evaluation make the work easier to reproduce and build on.