Let's DOIT: Using Intel's Extended HW/SW Contract for Secure Compilation of Crypto Code

Santiago Arranz-Olmos[1], Gilles Barthe[1,2], Benjamin Grégoire[3], Jan Jancar[4], Vincent Laporte[5], Tiago Oliveira[6] and Peter Schwabe[1,7]

[1] MPI-SP, Bochum, Germany
[2] IMDEA Software Institute, Madrid, Spain
[3] Inria, Sophia-Antipolis, France
[4] Masaryk University, Brno, Czechia
[5] Inria, Nancy, France
[6] SandboxAQ, Palo Alto, USA
[7] Radboud University, Nijmegen, The Netherlands

Abstract. It is a widely accepted standard practice to implement cryptographic software so that secret inputs do not influence the cycle count. Software following this paradigm is often referred to as "constant-time" software and typically involves following three rules: 1) never branch on a secret-dependent condition, 2) never access memory at a secret-dependent location, and 3) avoid variable-time arithmetic operations on secret data. The third rule requires knowledge about such variable-time arithmetic instructions, or vice versa, which operations are safe to use on secret inputs. For a long time, this knowledge was based on either documentation or microbenchmarks, but critically, there were never any guarantees for future microarchitectures. This changed with the introduction of the data-operand-independent-timing (DOIT) mode on Intel CPUs and, to some extent, the data-independent-timing (DIT) mode on Arm CPUs. Both Intel and Arm document a subset of their respective instruction sets that are intended to leak no information about their inputs through timing, even on future microarchitectures if the CPU is set to run in a dedicated DOIT (or DIT) mode.

In this paper, we present a principled solution that leverages DOIT to enable cryptographic software that is *future-proof constant-time*, in the sense that it ensures that only instructions from the DOIT subset are used to operate on secret data, even during speculative execution after a mispredicted branch or function return location. For this solution, we build on top of existing security type systems in the Jasmin framework for high-assurance cryptography.

We then use our solution to evaluate the extent to which existing cryptographic software built to be "constant-time" is already secure in this stricter paradigm implied by DOIT and what the performance impact is to move from constant-time to future-proof constant-time.

Keywords: data-operand-independent timing, Jasmin, high-assurance, constant-time code

1 Introduction

Since timing attacks against cryptographic software were first proposed by Kocher in 1996 [Koc96], it has become best practice to attempt to implement cryptography in such a way that the number of CPU cycles taken by computation is independent of any secret data. This programming paradigm is commonly referred to as *"constant-time programming,"* which is somewhat of a misnomer for various reasons. First, and most obviously, the number of CPU cycles taken by constant-time software is usually not *constant*, but rather independent of secret data. This becomes most obvious when encrypting messages of different length: the length is typically considered public and encryption of short messages is much faster than encryption of long messages. Second, at least traditionally, "constant-time" programming did not take into account that also *speculative* execution enables certain kinds of timing attacks. Such attacks are now known as "Spectre attacks" [KHF+19] and motivated the extension of constant-time formalizations to the stronger *speculative constant-time* notion [CDvG+20]. Third, it has recently been shown that even if the *cycle count* of a program is independent of secret inputs, the *timing* may not be [WPH+22; WPW+23]. The reason is that dynamic frequency scaling of modern CPU changes the duration of a single cycle and it turns out that this scaling depends on processed data. For the remainder of this paper we will consider this third issue out of scope, use the terms *timing* and *cycle count* synonymously and assume that the cycle count is not influenced by data-dependent frequency changes.

Despite these limitations, constant-time programming is widely accepted as a baseline defense against (software-visible) side-channel attacks and is a design goal of many cryptographic libraries [JFD+22, Sec. IV.B]. There are several dozen tools that aim at checking software for constant-time behavior (for an overview, see [Cen22]), and adherence of cryptographic software to certain constant-time programming rules is a required starting point in different recently proposed highly efficient protections against Spectre attacks [MNM+24; SBG+23; ABC+25].

However, there has always been an "elephant in the room" issue with the idea that carefully following a certain set of rules when writing cryptographic software can ensure secret-independence of timing: any set of rules has to rely on some assumptions about the hardware on which the software is running. Specifically, what is needed are assumptions about what inputs to which instructions *do not* influence the timing of the software. It is widely understood—in fact already pointed out by Kocher in [Koc96]—that branch conditions and addresses used in memory access have an influence on the timing of programs. Consequently, the term "constant-time" software is often used to refer to software that avoids secret-dependent branches and memory indexing. For example, many of the tools mentioned above check for exactly these two properties. On many microarchitectures this is not sufficient. As, again, already pointed out by Kocher in [Koc96], also certain arithmetic instructions *"such as multiplication and division"* may leak information about their inputs through the amount of cycles they take to execute. This was recently showcased in the KyberSlash vulnerability [BBB+24], where a division instruction with a secret-dependent dividend lead to private-key recovery. The problem is that the exact subset of instructions that is safe to use on secret inputs, depends on the microarchitecture.

For cryptographic software targeting only a well-defined set of known microarchitectures it is in principle possible to identify what instructions do not leak information about their inputs through timing, either by relying on detailed documentation of the microarchitecture or on microbenchmarking. Unfortunately, in many cases, cryptographic software is written without knowing what microarchitectures it will run on. Even worse, in many cases these microarchitectures may not even exist when the software is written.

This is why, for many years, implementers of cryptographic and other security-critical software have pointed out the need for *guarantees* by CPU manufacturers that a well-defined subset of the CPU instructions will take input-independent timing also on future microarchitectures. See, for example, the call for a new hardware-software contract by Ge, Yarom, and Heiser from 2018 [GYH18]. These calls were recently answered, with the introduction of *Data Independent Timing (DIT)* instructions for Arm [ARM20b], and *Data Operand Independent Timing (DOIT)* instructions for Intel [Int22b]. RISC-V implements a similar feature via the *Zkt* and *Zvkt* extensions that attest that their respective instruction subsets have data-independent timing. Both Arm's DIT and Intel's DOIT implementations require setting a flag in a machine-specific register to switch the CPU into the guaranteed "constant-time" mode for the respective instruction subsets.

One might think that now cryptographic software moves to systematically only using DIT or DOIT instructions on secret data, but unfortunately these new features in the hardware-software contract have, as far as we can tell, so far seen only very little application. The only exceptions we could find are the enabling of ARM's DIT in the Linux kernel starting with version 6.2.1 [Lin23] and the support for enabling Intel's DOIT in the `crypto/subtle` package from the Go programming language [Go24].

One reason might be that the introduction of DOIT by Intel was misunderstood and consequently met with rather harsh and public opposition by the Linux kernel developer community. For example, Biggers stated that *"this looks exactly like a CPU vulnerability"* [Big23b]. Another, more technical reason is that typical software-development toolchains do not have support for issuing only DIT or DOIT instructions when operating on secret data. In fact, mainstream programming languages and compilers do not even have a concept of secret data in the first place. Until now, the only way to build cryptographic software that only uses DIT or DOIT instructions, is to carefully handcraft all functions that operate on secrets entirely in assembly. This is in principle feasible, but becomes tedious when optimizing for multiple (micro)architectures. More importantly, it is error-prone, in particular when attempting to avoid non-DOIT operations on secret data also during speculative execution.

Contributions and Organization. In this paper we present a principled solution to implementing cryptographic software following the rules implied by Intel's DOIT instruction set. We integrate this solution with the Jasmin framework for high-assurance crypto [ABB$^+$17; ABB$^+$20] as follows:

- We incorporate knowledge about what instructions belong to the DOIT subset of instructions into the Jasmin compiler. We then use existing information-flow type systems in the Jasmin compiler to enforce that typable programs never use non-DOIT instructions on secret inputs, not even during speculative execution after mispredicted conditional branches or function return locations (Section 3).

- We investigate if existing cryptographic implementations written in Jasmin from Libjade [For23] are typable under these additional constraints and modify them where necessary to make them typable (Section 4).

- We present benchmarks of these implementations on different generations of Intel CPUs to understand the performance impact of running cryptographic software with and without the DOIT flag set and the impact of the modifications we had to make to restrict operations on secret data to the DOIT subset of instructions (Section 5). These benchmarks show that for highly optimized software, which typically makes heavy use of vector instructions, the performance impact is minimal. However, for non-vectorized reference implementations of cryptographic primitives involving rotations, the impact is substantial.

- Based on our investigation we give recommendations to several stakeholders: platform/ISA developers, software developers, kernel developers and compiler developers (Section 6).

- We conclude by discussing how a slightly more fine-grained definition of DOIT would make it easier and more efficient to implement cryptographic software, without significantly sacrificing freedom for future hardware optimizations (Section 7).

We remark that in this paper we focus on cryptographic software targeting Intel CPUs and thus the DOIT implementation, and leave Arm's DIT instruction set to future work. The reason is that support for Arm CPUs in the Jasmin framework is still immature and limited to the 32-bit ARMv7M instruction set; there simply are not that many implementations of cryptographic primitives targeting Arm available. Also, as we discuss in Section 2.3, the guarantees offered by Arm's DIT mode for future microarchitectures appear to be much weaker than for Intel's DOIT mode. However, it would be straight-forward to extend our DOIT extensions to the Jasmin framework to DIT.

Related Work. There is extensive literature on timing attacks and countermeasures. Most of these attacks exploit either secretly indexed memory access or secret branch conditions. A classic example for the former class of timing attacks are cache-timing attacks against AES [TTM$^+$02; Ber04; OST06], a cipher that was designed to be efficiently implemented using (secretly indexed) lookup tables. Software countermeasures employed alternative implementation approaches, most notably bitslicing [MN07; Kön08; KS09; PL18] and Hamburg's vector-permute-based approach [Ham09]. Prominent early examples for the latter class are attacks against scalar multiplication or exponentiation (e.g., [BB03; BT11]) and modular reduction (e.g., [Sch00]). A more recent example targets variable-time string comparison in an implementation of the Fujisaki-Okamoto transform [GJN20].

Attempting to systematically avoid these two major sources of timing leakage was popularized, among others, by the NaCl library [BLS12] and is today widely considered standard best practice for implementing cryptographic software [JFD$^+$22]. Unfortunately, as recently highlighted by Schneider, Lain, Puddu, Dutly, and Capkun [SLP$^+$24], modern optimizing compilers make it increasingly hard to achieve this goal when implementing cryptography in general-purpose high-level languages such as C, Rust, or Go.

Somewhat more relevant to the work we present in this paper are timing attacks exploiting variable-time arithmetic, such as `DIV` instructions on Intel CPUs, or certain multiplication instructions on PowerPC and various Arm CPUs. Examples include attacks against early-abort multiplications on ARMv7TDMI [GOP$^+$10], timing side-channels in floating point instructions [AKM$^+$15], and very recently the "KyberSlash" attack against the reference implementation of ML-KEM (Kyber) [BBB$^+$24].

The "Spectre" paper [KHF$^+$19] and subsequent works [KKS$^+$18; MR18; ZBC$^+$23] demonstrated that systematic protections against timing attacks need to avoid secretly indexed memory, secret branch conditions, and variable-time arithmetic on secrets also during *speculative* execution, e.g., after a mispredicted branch or function return location.

As we already mentioned earlier, there is very little work directly related to Intel's DOIT or Arm's DIT execution modes. There has been some debate about if the DOIT mode should be enabled by default in the Linux kernel [Big23b; Big23a], but enabling this mode is only one part of what needs to be done for systematic and future-proof constant-time execution. For the other part, namely systematically using only the DOIT (or DIT) subset of instructions on secret data, we are not aware of any prior works.

Also very relevant to this work is the Jasmin high-assurance framework and in particular its security type systems. We will discuss those in more detail in Section 2.5.

Artifact. We integrated support for the DOIT mode into the Jasmin compiler upstream, which supports (S)CT verification against Spectre-v1. However, we use a version that extends this support to Spectre-RSB [ABC$^+$25]. Alongside this paper we submit an artifact containing:

- a modified version of the Libjade library that is speculatively constant-time under DOIT;
- benchmarking scripts for the library; and
- scripts for scraping and working with the Intel DOIT and Arm DIT instruction lists.

We release all of our contributions under permissive open-source licenses compatible with the licenses of Jasmin and Libjade. The development is in `https://artifacts.formosa-`

crypto.org/data/letsdoit.tar.bz2.

2 Preliminaries

In this section, we define our threat model and give a brief overview of Intel's DOIT subset of instructions—and, for completeness, also Arm's DIT and RISC-V's Zkt.We then review the features of the Jasmin framework for high-assurance cryptography that are most relevant to this paper.

2.1 Threat Model

We protect against an attacker that observes the addresses of all memory accesses, the conditions of all control flow instructions, and, most relevant for this work, the arguments of non-DOIT instructions; this model aligns with previous work [ABB+16; SBG+22]. Regarding speculation, we assume that the operating system has the most recent microcode updates and sets the SSBD flag, thereby mitigating Spectre-v4 attacks. We consider that the attacker has complete control over the conditional branch predictor, the indirect branch predictor, and the return stack buffer, i.e., we combine the threat models described in [KHF+19; MR18; KKS+18]. We assume that the attacker observes the leakage described above, even under speculative execution.

2.2 Intel DOIT

Intel introduced the Data Operand Independent Timing (DOIT) feature for their CPUs in May 2022 [Int22b; Int22c]. On CPUs that enumerate the feature, i.e., the "Ice Lake" and later microarchitectures, DOIT mode can be enabled by setting bit zero in the model-specific register `IA32_UARCH_MISC_CTL`. In Linux, this flag can be set from user space using the `msr` driver, which offers an interface through `/dev/cpu/CPUNUM/msr`. When enabled, instructions within the DOIT subset will execute with timing independent of their data operands. Additionally, the data operands of these instructions will not affect the timing of other instructions (possibly outside of the DOIT subset). Note that instruction latency may still depend on other aspects, such as addresses, immediate values, and instruction encoding. Hence, in order to utilize DOIT securely, all of the usual constant-time programming guidelines need to be followed. Similarly, there are no guarantees of resistance to power, thermal or frequency-based side-channels given for the DOIT mode.

In addition to DOIT mode, Intel published a separate list of instructions—containing only vector instructions—that are intended to operate in DOIT mode, but on certain processors their run-time may depend on the value of the data operands under some values of the `MXCSR` register [Int23b; Int23a]. This list is titled the MXCSR Configuration Dependent Timing (MCDT). The guidance contains an explicit list of affected processors, as well as a comment that future processors are not expected to exhibit this timing variation. This points to the behavior being an unintended exception to the DOIT guarantees, i.e., a bug, on the affected processors. We verified that other than a documentation change to clarify behavior of `REP*` instructions[1] the DOIT list remains unchanged since its publication.

The DOIT mode is documented to control two other features: the data-dependent prefetcher (DDP) [Int22a] and the fast-store-forwarding prefetcher (FSFP) [Int22d] are disabled when the CPU is operating in DOIT mode. Note that the documentation explicitly mentions DOIT only in the case of the DDP; the fact that the FSFP is disabled as well was posted by an Intel employee to the linux-kernel mailing list [Big23a, Message from 2023-02-03 16:25].

[1]These `REP*` instructions' latency varies based on the loop count.

Note that CPUs that do not enumerate the feature behave as if DOIT mode is always enabled with regards to the instructions on the DOIT list. This, coupled with the explicit note that *"DOIT mode may have a performance impact, and Intel expects the performance impact of this mode may be significantly higher on future processors"* [Int22b] points to DOIT mode being more of a measure to safeguard possible *future* optimizations, rather than disabling current ones when it is turned on. This interpretation was confirmed by an Intel employee on the linux-kernel mailing list [Big23a, Message from 2023-02-01 18:09]. The performance impact of DOIT mode being very low on current CPUs was also shown in benchmarks by Linux users [Lar23].

2.3 Arm DIT

Arm introduced their own Data Independent Timing (DIT) feature in 2020 targeting Arm v8.4-A and beyond [ARM20b; ARM20a]. The feature operates in the same way as Intel DOIT. Similarly, Arm notes that it does not know of any CPU implementations prior to the introduction of the DIT feature that would have data dependent timing for the covered instructions.

In contrast to Intel DOIT, the list of covered instructions changed over time in case of Arm DIT. Furthermore, the documentation was at various points inconsistent in listing the instructions between the DIT page and individual instruction pages. We extracted the instructions from Arm's documentation in 3-month intervals since December 2020. The full list of changes is too large to include—we include it in our artifact—so we summarize some notable removals:

- 2021-12: removal of `RET`,
- 2023-03: removal of `SQCVTN`, `SQCVTUN`, `SQRSHRN`, `SQRSHRUN`, `UQCVTN`, `UQRSHRN`,
- 2023-09: removal of all `WHILE*` instructions,
- 2023-12: removal of `PEXT`,
- 2024-06: removal of `CNTP`, `PUNPKHI`, `PUNPKLO`,
- 2024-12: removal of all `RCW*` instructions.

Such frequent changes of the guarantees provided the DIT mode might make it hard for developers to gain confidence that their implementations will not suddenly become "out of spec" and vulnerable to timing attacks in some future processor architecture. In fact, we remark that *removal* of instructions from the DIT subset defeats a large part of the purpose of these protections, namely the guarantee that software will remain secure also on future microarchitectures.

With Apple moving away from Intel to their own Arm-based architecture, it inherited Arm DIT mode [App20]. However, Apple documentation does not go into detail on which instructions are included (apart from linking to the Arm DIT documentation). In recent versions of Apple operating systems a new `timingsafe_enable_if_supported` API was added, which enables DIT mode if supported and limits speculative execution.

2.4 RISC-V Zkt and Zvkt

RISC-V defines two extensions that enforce data-independent timing for subsets of instructions: *Zkt* and *Zvkt* [RIS21; RIS23]. The former covers scalar instructions from several instruction sets: RVI, RVM, RVC, RVK, and RVB, while the latter covers vector instructions. Both require that listed instructions are implemented with data-independent timing. Similarly to Intel DOIT, they allow timing dependency on immediate values or instruction encoding. Unlike Intel DOIT and Arm DIT, which require to be enabled during runtime to have an effect, the Zkt and Zvkt extensions are hardware guarantees and do not require being enabled or disabled. The Zkt and Zvkt instruction lists have not changed since the extensions were ratified in November 2021 and September 2023, respectively. Notably, the Zkt extension includes access to the `seed` control and status register (CSR), which is used to extract randomness from a hardware entropy source. It requires that timing is independent of the extracted entropy. In the case of Intel DOIT, there is no such guarantee for the `RDRAND` instruction. It is unclear if such a guarantee holds for Arm's DIT as the `MRS` instruction is not on the DIT list, nor is the `RNDR` special purpose register.

2.5 The Jasmin Framework

Jasmin is a programming framework for high-assurance high-speed cryptography comprising a programming language, a formally verified compiler, a safety checker, a cryptographic constant-time checker, and speculative cryptographic constant-time checker.

Synthetically, developing a Jasmin program and verifying that it is constant-time involves the following steps: First, the developer writes the program in Jasmin and then annotates exported functions—i.e., the entry points of the program—to classify inputs and outputs as public or secret. Then, invoking the type checker leads to one of two scenarios: either the program is verified as constant-time, in which case the developer has succeeded, or the checker finds an information leak and raises an error with the location of the leak. After the type system accepts the program, invoking the compiler produces an assembly program that can be given to an off-the-shelf assembler such as GNU as. We describe the interface of the type checker in detail in Section Section 3.4.

The Language. The Jasmin language is low-level to allow a high degree of control over the generated assembly: most architecture-specific instructions are available to the programmer as primitive constructs in Jasmin, called *intrinsics*, such as `x = #ADD(y, z)`. It provides structured control flow (such as `if`, `while`, and function calls) and zero-cost abstractions (such as named `reg` variables, named `stack` variables, and arrays) to ease development, maintenance, and verification of highly optimized cryptographic routines.

Several standard cryptographic schemes have been implemented in Jasmin, see, for example, [ABB+20; ABB+19]. Additionally, the post-quantum key-encapsulation mechanism Kyber (now ML-KEM) has also been implemented and proven correct and secure [ABB+23; AOB+24]. Most of these developments have been integrated into Libjade [For23], an open-source cryptographic library written in Jasmin.

The Compiler. The Jasmin compiler is implemented and formally verified in the Coq proof assistant. It consists of standard compiler transformations such as dead-code elimination, unrolling of `for` loops, function inlining, instruction selection, stack allocation, register allocation, value propagation, and linearization of control flow into jumps. These passes gradually simplify the program from a structured source (Jasmin) to assembly code. Instruction selection replaces high-level constructs such as `x + y` by architecture-specific intrinsics such as `#ADD(x, y)`. Stack allocation lays out function frames: it arranges the offsets from the stack pointer for local variables and arrays, and attempts to share memory between variables that are not needed at the same time to optimize memory usage. Register allocation renames variables such that they coincide with architecture registers, e.g., `#ADD(x, y)` becomes `#ADD(RAX, RBX)`, and fails if this is not possible. The programmer must perform spilling manually at source level. Value propagation (called *propagate inline* in the compiler) is necessary to resolve compile-time constructs of the language, such as `reg bool le = (x <= y);` which improve the readability of the source, but require every use of `le` to be replaced by the appropriate flag combination `CF || ZF`. Finally, linearization replaces conditionals, loops, and function calls with jump, call, and return instructions, respectively.

$$\overset{\text{ASSIGN}}{\frac{\Gamma \vdash e : t \leq \Gamma(x)}{\Gamma \vdash x = e}} \qquad \overset{\text{COND}}{\frac{\Gamma \vdash e : \texttt{public} \qquad \Gamma \vdash c \qquad \Gamma \vdash c'}{\Gamma \vdash \texttt{if (}e\texttt{) \{}c\texttt{\} else \{}c'\texttt{\}}}} \qquad \overset{\text{INTRINSIC}}{\frac{\Gamma \vdash e_i : t_i \leq \Gamma(x) \text{ for each } i}{\Gamma \vdash x = \texttt{\#}I(e_1, \cdots, e_n)}}$$

Figure 1: Selected rules of the CT type system.

Most passes are proven directly in Coq, with few exceptions such as register allocation, which is implemented in OCaml and its output validated by a verified checker. The compiler targets two architectures: x86-64 and ARMv7-M. It also translates Jasmin programs into EasyCrypt [BDG+13] models to enable computer-verified proofs of functional correctness and connect to mechanized reductionist cryptographic security proofs in that proof assistant.

The CT and SCT Checkers. The Jasmin constant-time checker [SBG+22] verifies that branch conditions and memory accesses do not depend on secrets. That is, the inputs to entry point functions (called *exported* functions in Jasmin) are tagged as public or secret, and the checker disallows information flow from these values into branches or memory accesses. It is implemented as a type system; Figure 1 shows—simplified—selected rules for the CT checker. We write $\Gamma \vdash i$ to express that an instruction i, such as an assignment, an intrinsic, a conditional, or a loop, is well-typed under a context Γ (a context associates each variable with a type, either `public` or `secret`). The ASSIGN rule says that an assignment is well-typed if the type of variable on the left-hand side is at most that of the expression—e.g., an integer, a boolean, or another variable—on the right-hand side. In line with the literature on confidentiality, we consider that `secret` is greater than `public`. This means that we can assign a public value to a secret variable, since we enforce that secret variables do not leak. The COND rule states that a conditional is well-typed if its condition is public and each branch is well-typed. The rule for intrinsics is similar to the one for assignments, where the type of each argument of the instruction must be at most that of the assigned variable. Rules for memory accesses (not shown) enforce that the address expressions have a `public` type.

The speculative constant-time checker [SBG+23] is more complex, as it needs to keep track of both the *sequential* confidentiality of registers and arrays—in the same way as the CT checker—and also their *speculative* confidentiality. That is, contexts associate each variable with a pair of types. Under speculation, both public registers and arrays may receive secret values: for instance, a loop may misspeculate and continue executing past its bound, and thus a sequentially public load in the loop body may perform an out-of-bounds access and read from a secret region of memory. Therefore, the SCT type system for Spectre-v1 refines the rules for loads, stores, conditionals, and loops. The rule for loads sets the speculative type to `secret`, to account for reading out-of-bounds into a secret memory region. Analogously, the rule for store sets the speculative types of *every* array to `secret` if the stored value is secret, to account for writing out-of-bounds into a public memory region. The type system supports *selective speculative load hardening* (selSLH), a state-of-the-art mitigation against Spectre-v1. Essentially, the first component of the mitigation is to keep a *misspeculation flag* (MSF), a register whose value is `0x00..00` or `0xff..ff` depending on whether the branch predictor has misspeculated. Then, we use the MSF to mask values before leaking them. The SCT checker ensures that the programmer keeps the MSF correctly updated and masks all speculatively leaked values.

Recent work [ABC+25] further extends the checker to also protect against Spectre-RSB [KKS+18; MR18], by adapting the rules for calls and returns. These modifications correspond to a more powerful attacker—that controls branch and return address prediction—and were designed to be independent from the leakage model.

Other Compiler Infrastructures. Mainstream compilers, such as GCC and Clang, and also verified ones, such as CompCert, struggle with a significant correctness-security gap [DPS15]. That is, when compiling the high-level abstractions of the source language and striving for performance, the goal of these compilers is to preserve the functionality of the source program, not to respect its security guarantees. Thus, it is unsurprising that they sometimes introduce side-channel vulnerabilities [SLP+24; BBG+19]. CompCert, in a mildly modified version, stands out by preserving CT [BBG+19], although its use by cryptographic libraries is scarce. Furthermore, and particularly relevant for the present work, a critical challenge in mainstream languages such as C is the absence of a clear, fine-grained leakage model, which explains the lack of features analogous to the security types of Jasmin. For instance, the expression `x / 16` could be considered as leaking since it is a division, but it is equally valid to assume that the compiler will use a constant-time bit shift instead. As a result, it is challenging to define security properties in these languages rigorously. On the other hand, Jasmin enjoys a much narrower correctness-security gap by design, being a lower-level language that offers developers a high degree of control, and consequently has a straightforward leakage model, which allowed its compiler to be proven to preserve CT [BGL+21; SBG+22]. Lastly, several cryptographic schemes have been implemented in Jasmin, as shown in Section 4.

3 Extending the Jasmin Framework

In this section, we describe the modifications necessary to incorporate the DOIT leakage model into the framework. We assume that the DOIT mode is enabled via the `IA32_UARCH_MISC_CTL` flag, before running Jasmin-compiled code.

3.1 Extraction of DOIT Instructions

Intel does not provide a list of DOIT instructions in a machine-readable format. We thus scraped the list out of the documentation site, which contains a table of instruction mnemonic-opcode pairs. Since this list provides opcodes, one cannot assume that a given instruction is fully covered (i.e., that all of its opcodes are DOIT). As Jasmin outputs assembly language and not machine code, it can only use instructions that are fully covered in the DOIT list.

Our scraping and analysis found only three examples of uncovered opcodes:

- `MOV` with opcodes `0x8c` and `0x8e` that move values to and from the segment registers.
- `POP` with opcodes `0x0fa1`, `0x0fa9`, `0x1f`, `0x07`, and `0x17` that pop values from the FS and GS segment registers.
- `PUSH` with opcodes `0x0fa0`, `0x0fa8`, `0x1e`, `0x0e`, `0x06`, and `0x16` that push values from the segment registers.

As Jasmin does not use segment registers, the `MOV`, `POP` and `PUSH` instructions can be considered DOIT.

3.2 Changes to the Compiler

The main change to the compiler was to move the propagate-inline pass earlier in the compiler stack and update the corresponding end-to-end compiler proof. This was necessary to improve the precision of the CT and SCT checkers, since this pass transforms high-level conditions (e.g., `x <= y`) to architecture-specific flag combinations. The precision of the analysis increases as in the following example: let us consider a `CMP` instruction on public operands, then an `ADD` instruction on secret operands, and lastly some non-DOIT instruction

$$\frac{\Gamma \vdash e_i : t_i \leq \Gamma(x) \text{ for each } i \qquad \boldsymbol{I} \textbf{ not DOIT} \implies \textbf{every } \boldsymbol{t_i} \textbf{ is } \texttt{public}}{\Gamma \vdash x = \texttt{\#}I(e_1, \cdots, e_n)}\ \text{INTRINSIC}$$

Figure 2: New type system rule for intrinsics.

that depends on the zero flag. The zero flag is not set by the secret `ADD` instruction, only the carry and overflow flag are. To accept the non-DOIT instruction, the checkers need to understand which individual flags are set and read by each one of these instructions.

3.3 Changes to the Type System

To take DOIT instructions into consideration, for this work we introduce a new constraint (highlighted in **bold red** in Figure 2) that ensures that non-DOIT instructions execute only on public values. This extension carries seamlessly over to the speculative setting, i.e., it is straight-forward to (optionally) ensure that there is no data flow from secrets into arguments of non-DOIT instructions, even during speculative execution following a mispredicted branch or return location.

3.4 Usage

The Jasmin constant-time checker, accessed through the `jasmin-ct` binary, checks each exported function in a source file for constant-time. Developers can select between the default sequential constant-time mode and speculative constant-time mode using the `--sct` option. With the changes presented in this work, the `--doit` option enhances the leakage model by treating non-DOIT instructions as sources of leakage. Developers must annotate exported functions to denote the confidentiality of arguments and results, specifying whether they are public or secret using a straightforward syntax, e.g.,

```
fn foo(#secret reg u64 key, #public reg u64 nonce) -> #public reg u64 { ... }
```

Alternatively, developers can employ the `--infer` option to allow the checker to deduce the weakest type for each argument and return, bypassing the need for manual annotations. The responsibility remains with the developer: they must verify that inferred types align with their expectations, avoiding misclassification, such as typing a password as secret. These annotation procedures remain unaffected by our changes.

4 Application to Libjade

In this section we apply the Jasmin type system with DOIT extensions to the implementations of cryptographic primitives in Libjade. The starting point for this investigation is the Libjade version in the artifact of [ABC+25]; all code in this version of Libjade is already systematically avoiding data flow from secrets into branch conditions and memory addresses, even during speculative execution. It also already avoids arithmetic instructions that are known to leak information about their arguments through timing. For example, the Kyber implementation in Libjade was not affected by KyberSlash [BBB+24], because it implements divisions through multiplication instructions as described in [GM94].

4.1 Primitives and Implementations in Libjade

The version of Libjade we investigated contains a total of 44 implementations of cryptographic primitives (see Figure 3). Not all of these are completely independent; some of

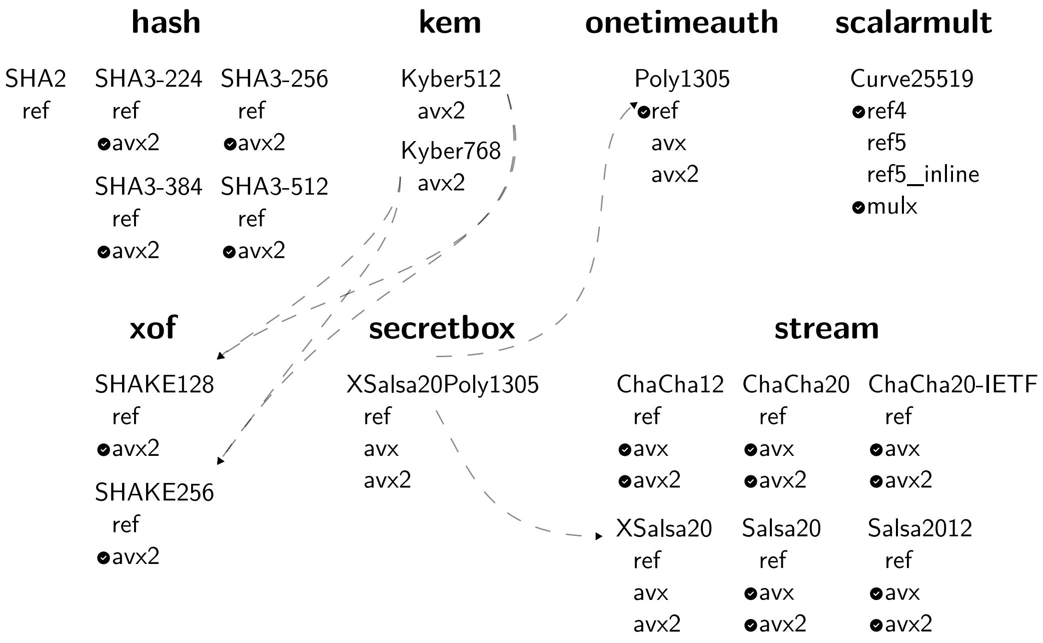

Figure 3: Overview of the implementations in Libjade. Arrows signify code sharing. Implementations that did not require changes to be DOIT are marked with ✓.

them share code. For example, multiple primitives from the FIPS-202 (SHA-3) standard may share code for the Keccak permutation.

Out of the 44 implementations available in this Libjade version, all targeting x86-64, 19 were already compliant with DOIT (i.e., they did not use instructions outside the DOIT list on secret data). Among these 19, 16 were vectorized implementations targeting AVX and AVX2. These included multiple instances of SHA-3, SHAKE, and ChaCha20. In addition, two non-vectorized implementations of Curve25519 were DOIT-compliant, as well as one Poly1305 implementation.

The remaining 25 implementations required changes. Of these, 17 were so-called reference implementations: they are optimized, often offering performance comparable to the best non-vectorized code publicly available, but they do not utilize AVX or AVX2 instructions or registers. Among the remaining 8 implementations, 6 AVX/AVX2 implementations were non-compliant with DOIT due to the use of non-vectorized code for some computations, meaning the DOIT violations were unrelated to the AVX code itself. The final two vectorized and Spectre-protected versions of Kyber (now ML-KEM) in this package also required fixes—not due to the use of divisions but because of other non-DOIT instructions and also challenges in verifying complex code protected against both Spectre-RSB and DOIT that we address next.

4.2 Rotation Instructions in Chacha20 and SHA-3/SHAKE

A core operation in many symmetric cryptographic primitives is rotation. In Libjade, the primitives that make heavy use of rotations are ChaCha20, SHA-3, and SHAKE. The latter two furthermore serve as building blocks for ML-KEM. Under a strict interpretation of the DOIT list, applicable to the x86-64 implementations in this study, the rotate instruction—whether left or right—should not be used for cryptographic purposes. For instance, the ChaCha family of ciphers frequently employs this operation during its round function (over 32-bit words), as does the Keccak permutation (over 64-bit words). Such interpretation requires implementing the rotation operation using a different sequence of instructions to avoid the rotate instruction—for example, by combining left and right shifts with a logical bitwise OR.

The Jasmin programming language provides (almost) complete control over the generated assembly code. This control implies that the Jasmin developer determines which values reside in registers and memory during the algorithm's execution. Existing Jasmin implementations leverage this control to maximize performance: whenever advantageous, all registers are fully utilized. This is precisely the case with the current ChaCha and Salsa20 reference implementations, which require modifications to comply with DOIT. Replacing the rotation with a different sequence of instructions demands one free register to hold the intermediate value.

Consider the ChaCha20 reference implementation, initially performing at the same speed as the OpenSSL equivalent [ABB+20] and now protected against Spectre attacks with minimal overhead. On an Intel 11700K CPU, using standard benchmarking practices, the Spectre-protected version of this implementation takes 97222 CPU cycles to generate a 16KiB stream for encryption or decryption. Adjusting the implementation to free up one register (to accommodate the upcoming change in rotation computation) introduces some slowdown, with the execution time increasing to 103562 CPU cycles under the same conditions. Finally, replacing the 32-bit rotation with a sequence of shifts and a logical OR (to ensure DOIT compliance) has a substantial performance impact, increasing execution time to 161288 CPU cycles. There is little one can do to mitigate this performance impact, except using vector instructions, which we do in the AVX2 implementation.

In the case of the non-vectorized Keccak permutation, the patch was more straightforward as the Jasmin implementation already had one register available whenever we required a rotation. Hashing a 16KiB message using SHA3-256 took 138626 CPU cycles before replacing the rotate and 182626 CPU cycles afterward.

4.3 Endianess Conversion in SHA-256

The SHA-256 reference implementation also required patching. This implementation used two instructions not included in the DOIT list: `BSWAP` and—just as in ChaCha20 and Keccak—`ROR`, both operating on 32-bit words. Before any changes were made, and under the same benchmarking conditions as previously mentioned, hashing 16KiB of data with this implementation took 247110 CPU cycles. The `BSWAP` instruction reverses the byte order of 32-bit values, which were being copied from one memory location to another with byte-swapping in between. The first approach we took to remove the `BSWAP` instruction involved replacing it with a sequence of four loads followed by four writes to achieve the same result. This introduced a significant but not critical penalty, with the implementation taking 258088 CPU cycles for the same message length. The second approach replaced `BSWAP` with a sequence of bitwise operations (shifts, `AND`, `OR`), reducing the overhead slightly, with benchmarks yielding 256566 CPU cycles. The rotation instruction was addressed next, and just like for ChaCha20 and Keccak, its replacement caused a substantial overhead, with the implementation now requiring 342762 CPU cycles for 16KiB inputs.

4.4 Double-Precision Shift Instructions in Poly1305 and X25519

Double-precision shifts are not included in the DOIT list. These are used in the Poly1305 AVX and AVX2 implementations, as well as in one Curve25519 implementation. In the context of Poly1305 vectorized implementations, the specific instruction is `SHRD`, which allows a 64-bit variable to be shifted right while feeding the lower bits of a second variable into the higher bits of the resulting variable (instead of injecting zeros). This instruction is typically useful in multi-limb arithmetic and, in the case of Poly1305, is employed to prepare the vectorized state. Since it is not part of the main computation, replacing this instruction with a sequence of two shifts and a bitwise OR should not significantly impact performance. Intermediate benchmarks confirm this, with the number of cycles for 16KiB inputs increasing by just 34 cycles, from 8968 to 9002 CPU cycles for the AVX2 implementation.

In the 5-limb implementation of Curve25519, the patch is similar: the instruction used is a double-precision left shift, `SHRL`, and the strategy is the same —two shifts and a bitwise OR. Since this instruction is used multiple times during the multiplication routines, the performance impact is more significant in this case. Before patching, the implementation required 129850 CPU cycles to perform a scalar multiplication, whereas after patching, it required 141670 CPU cycles.

4.5 Vector-Extraction Instructions in Kyber

Regarding the Kyber implementations available in the Spectre-protected Libjade package, specifically the AVX2 implementations for Kyber512 and Kyber768, these used the `VMOVLPD` and `MOVHPD` instructions to extract data from a vectorized state. With respect to DOIT compliance, the patch was straightforward: we replaced these instructions with a sequence of moves and shifts. However, the implementations required further changes to ensure Spectre protections: the constant-time property with respect to DOIT must be done at later compilation stages, close to the assembly level, where the exact instructions used are already fixed by the compiler. This differs from previous work, where speculative constant-time properties are checked in earlier compilation stages, allowing the type checker access to more information about the program's structure. To address this—checking both the speculative and DOIT constant-time properties almost at the assembly level—we patched the implementation by introducing new protect directives on specific variables to ensure that both properties hold simultaneously. The overhead of this patch is negligible on the 11700K CPU, with all three operations (keypair generation, encapsulation, and decapsulation) taking roughly the same number of CPU cycles before and after the patch.

5 Benchmarks and Validation

In this section, we discuss the performance of Libjade implementations protected against Spectre attacks using only DOIT instructions. The benchmarks presented were obtained using three CPUs: Intel 8700K (Coffee Lake), Intel 11700K (Rocket Lake), and Intel 13900K (Raptor Lake).

We followed standard benchmarking practices, including disabling TurboBoost and HyperThreading and setting the clock speed to the base frequency. The CPU cycle measurements were obtained as follows: 1) recording the median value of 10000 runs; 2) repeating the experiment 11 times; and 3) selecting the median of the 11 experiments. The Kyber benchmarks use real `randombytes`, providing a performance assessment that reasonably reflects real-world executions.

Table 1 presents the CPU cycles for several AVX2 implementations, including Kyber768. The first column, "Impl.," identifies the implementation, and the second column, "Op.," is the operation (e.g., the length of the input).

For each CPU, the table includes six columns of cycle counts. The first column of this set, "Alt," reports the performance of alternative cryptographic library implementations. ChaCha20, Poly1305, and X25519 report the cycle counts for OpenSSL 3.4.0. XSalsa20Poly1305 corresponds to the fastest implementation available in libsodium 1.0.20 (it is not AVX2-optimized, hence the significant performance difference). SHA3-256 was taken from SUPERCOP 20250307 and corresponds to a Keccak Code Package implementation. For Kyber768, we refer to the implementation from mlkem-native (from the pq-code-package project). All code under "Alt" was compiled with Clang 18. The remaining five columns show cycle counts for different versions of the Jasmin code, with the plus sign in a column title indicating an accumulation of protections. The column "Plain" refers to the original Libjade implementations without any Spectre or DOIT protections. Column "+SSBD" refers to the plain implementation run with the SSBD CPU flag enabled, providing protection against Spectre v4 attacks. The "+v1" column adds protection against Spectre-v1 to the SSBD setup. The "+RSB" column adds protection against Spectre-RSB. Finally, the "+DOIT" column incorporates all protections, including DOIT. The first (leftmost) percentage column shows the overhead incurred from plain to "+RSB," while the second percentage column indicates the additional overhead introduced by this work, from "+RSB" to "+DOIT."

Table 1: The good. CPU cycle counts for a subset of the fastest implementations in Libjade, for Coffee, Rocket, and Raptor Lake CPUs.

Impl.	Desc.	CPU	Alt.	Plain	+SSBD	+v1	+RSB	%	+DOIT	%
ChaCha20 -avx2	16KiB	8700K	19488	18965	18954	18908	18915	-0.26	18910	-0.03
		11700K	19256	19034	19060	19052	19056	0.12	19056	0.00
		13900K	19958	20254	20332	20328	20332	0.39	20336	0.02
Poly1305 -avx2	16KiB	8700K	8228	8393	8409	8415	8422	0.35	8419	-0.04
		11700K	8612	8910	8918	8968	8968	0.65	9002	0.38
		13900K	8556	8758	8832	8808	8932	1.99	8990	0.65
Poly1305 -ref	16KiB	8700K	18582	15976	15977	15997	15996	0.13	16000	0.03
		11700K	18226	17540	17544	17570	17572	0.18	17572	0.00
		13900K	16986	16144	16144	16160	16154	0.06	16154	0.00
XSalsa20 Poly1305 -avx2	16KiB	8700K	84827	31278	31267	31315	31310	0.10	31579	0.86
		11700K	82760	32512	32508	32538	32532	0.06	32742	0.65
		13900K	78916	32788	32850	32980	33012	0.68	33186	0.53
SHA3-256 -avx2	16KiB	8700K	162232	131523	141532	141540	141563	7.63	141575	0.01
		11700K	155112	140626	152708	155376	153416	9.10	153410	-0.00
		13900K	140112	140668	150856	152156	152154	8.17	152152	-0.00
X25519 -mulx	smult	8700K	115623	98293	99543	99742	99746	1.48	99743	-0.00
		11700K	116320	103238	104186	104480	104480	1.20	104466	-0.01
		13900K	103704	95414	96490	96592	96594	1.24	96590	-0.00
X25519 -ref4	smult	8700K	146625	124808	125180	125262	125173	0.29	125135	-0.03
		11700K	132118	121264	125928	126362	126364	4.21	126366	0.00
		13900K	111128	98908	103880	103800	103862	5.01	103874	0.01
Kyber768 -avx2	keypair	8700K	53403	46270	48160	48322	49122	6.16	49335	0.43
		11700K	48130	43056	45178	45394	46244	7.40	46686	0.96
		13900K	37496	38452	40548	40920	41756	8.59	42024	0.64
	enc	8700K	54700	56326	59035	59161	60537	7.48	60130	-0.67
		11700K	49616	55328	58252	58722	59504	7.55	59912	0.69
		13900K	37950	50780	53620	54024	54946	8.20	55242	0.54
	dec	8700K	68098	45829	47628	47880	49121	7.18	49085	-0.07
		11700K	60506	44878	47048	47336	48204	7.41	48444	0.50
		13900K	48514	46844	48928	49230	50164	7.09	50460	0.59

The performance of several implementations in Table 1 is not harmed by DOIT compliance. Specifically, the AVX2 versions of ChaCha20, SHA3, and X25519 exhibit a negligible DOIT overhead, with differences very close to zero. For instance, the difference between "+RSB" and "+DOIT" for the ChaCha20 AVX2 implementation processing a 16

Table 2: The not so good. CPU cycle counts for a subset of the reference implementations in Libjade, for Coffee, Rocket, and Raptor Lake CPUs.

Impl.	Desc.	CPU	Alt.	Plain	+SSBD	+v1	+RSB	%	+DOIT	%
ChaCha20 -ref	16KiB	8700K	95001	94297	94339	94549	94524	0.24	156554	65.62
		11700K	95914	94406	96934	97220	97222	2.98	161288	65.90
		13900K	83366	84116	85720	85816	85884	2.10	144730	68.52
XSalsa20 Poly1305 -ref	16KiB	8700K	139080	115317	115469	115343	115328	0.01	177807	54.18
		11700K	138928	113846	114992	114898	114904	0.93	176586	53.68
		13900K	122972	101910	102212	102172	102168	0.25	152438	49.20
SHA3-256 -ref	16KiB	8700K	154063	152273	156140	153593	153579	0.86	211147	37.48
		11700K	138574	131296	138620	138764	138626	5.58	182626	31.74
		13900K	110510	106200	102712	102878	102908	-3.10	144582	40.50
SHA256 -ref	16KiB	8700K	186329	270218	271725	269409	276094	2.17	396213	43.51
		11700K	170678	231720	252230	247452	247110	6.64	342762	38.71
		13900K	143142	202364	224224	224716	224352	10.87	303900	35.46
X25519 -ref5	smult	8700K	142636	136480	139287	138342	138433	1.43	147312	6.41
		11700K	124472	127476	129740	129800	129850	1.86	141670	9.10
		13900K	106474	103774	106328	106416	106450	2.58	118524	11.34

KiB stream is just four cycles, which falls within the range of unavoidable noise. In the case of Poly1305 AVX2, there was a slight change due to DOIT, but the overhead is negligible for the recent CPU generations under analysis, while it is non-existent for the older 8700K. The Poly1305 reference implementation is unaffected by DOIT compliance. Regarding the XSalsa20Poly1305 instantiation of `secretbox`, the overhead is more noticeable (between 0.53% and 0.86%) because non-vectorized code is used in these implementations (for computing the subkeys, which are small and do not justify the use of AVX2). As a result, they are affected by the rotate instruction discussed in the previous section.

For the AVX2 implementation of Kyber768, compliance with DOIT combined with Spectre-RSB protections yields good results, with reported overheads of less than 1%. This outcome is partly expected due to the non-critical nature of the changes made for DOIT. However, additional usage of the `protect` directive was necessary to ensure simultaneous protection against Spectre-RSB and DOIT. Interestingly, the machine equipped with the 8700K handled the proposed changes well. The performance even improved slightly (by 400 cycles). The results were consistent across the eleven runs, with the 25th and 75th percentiles for “+DOIT” being 60094 and 60140 cycles, respectively, compared to 60525 and 60557 cycles for the “+RSB” experiment.

Table 2 presents several less favorable results, clearly indicating that improvements are needed, particularly regarding the handling of rotation instructions. In this table, we also include the “alt” column for comparison with widely deployed implementations. ChaCha20, SHA3, and SHA256 are from OpenSSL 3.4.0; XSalsa20Poly1305 is from libsodium 1.0.20 (configured without assembly to access the non-vectorized implementation); and X25519 corresponds to the 5-limb implementation (amd64-51) taken from SUPERCOP 20250307. The overhead introduced by strictly adhering to DOIT for all implementations using rotations is substantial. The worst case is ChaCha20, where the overhead from “+RSB” to “+DOIT” exceeds 65% across all CPUs. In the case of XSalsa20Poly1305 (where the computations of Salsa20 are similar to ChaCha20 and rely heavily on rotations), the performance penalty is slightly mitigated because Poly1305 remains unaffected. The average overhead for the hash functions listed in the table is below 40%, which is still considerable. For the 5-limb implementation of Curve25519, the overhead remains within practical limits, staying below 7% on the oldest CPU generation. The information in the table provides a basis for analyzing the evolution of the code, and the costs associated with different types of protections. Interestingly, addressing and mitigating all Spectre attacks is, in several cases, significantly less costly than addressing instruction timing variations.

6 Discussion

Our work on enabling a DOIT-only implementation of Libjade as well as the inclusion of DOIT into the Jasmin (S)CT checker gave us insights which we now turn into recommendations for several stakeholders in the ecosystem. We consider four stakeholders: the platform/ISA developer, the crypto/software developer, the kernel developer, and the compiler/tool developer. Naturally, most of our recommendations go towards the platform/ISA developer, as they are most in control of shaping the guarantees.

Platform/ISA Developer

We consider the platform and ISA developer as a single entity, even though in the case of RISC-V they differ. We believe that they could improve usability of their constant-time instruction modes by using the following principles:

- Be as specific as possible with regards to the guarantees.
- Offer instruction sets and guarantees in machine-readable form.
- Consider instruction use-cases in cryptography and sensitive areas when deciding on the guarantees and evaluating their value vs. their preformance impact.
- Limit changes to the constant-time instruction sets and guarantees as much as possible. Version the changes and keep them publicly available.
- Be clear on which (micro-)architectures the guarantees apply to (and which version of them if they changed).
- Be clear in communicating the current and (expected) future performance impacts.

Crypto/Software Developer

The responsibility to actually use the DOIT/DIT/Zkt instruction subsets for sensitive applications falls on software developers. To avoid vulnerabilities like KyberSlash [BBB+24], we recommend the following:

- Move toward DOIT/DIT/Zkt implementations of cryptography and other critical software, particularly since many vector implementations of cryptographic primitives are almost inherently protected.
- Enable the DOIT/DIT mode when possible on the target platform.

Kernel Developer

Kernel developers are also key stakeholders, as runtime control for the DOIT/DIT modes has to be implemented in-kernel. We recommend the following:

- Offer a user-space API to enable the DOIT/DIT modes for sensitive applications that request it. The Apple DIT API described in Section 2.3 that also controls mitigations against other micro-architectural attacks may be a good choice.
- Use the DOIT/DIT modes for in-kernel cryptography and implement it using the DOIT/DIT/Zkt instruction subsets.

Compiler/Tool Developer

Finally, developers of compilers and tools for verifying constant-timeness (see [FDJ$^+$24] for an overview) can support the effort to move toward DOIT/DIT/Zkt implementations. We recommend that they

- Add options to limit or validate generated assembly against the DOIT/DIT/Zkt subsets of instructions.

7 Conclusion

In this paper we present a principled approach to developing cryptographic software in the constant-time regime as implied by Intel's DOIT subset of instructions. The big advantage to limiting all operations on secret data to the DOIT subset is that the software is, for the first time, guaranteed to run in (speculative) "constant-time" also on future microarchitectures.

The proposed approach critically relies on already existing information-flow type systems in the Jasmin compiler. We believe that it would be possible to offer similar DOIT support also in general-purpose mainstream languages and compilers, if those had similar leakage models and type systems in place. The idea of such an information-flow type system in Rust and LLVM is discussed in [Hos20], but as far as we know, this idea has not been implemented, yet.

We also show that not all existing "constant-time" cryptographic software is already in the DOIT regime—there are some instructions that are not leaking information about their operands on existing microarchitectures, but that are not guaranteed to maintain this non-leaking behavior also on future microarchitectures. The most notable example are rotate instructions, which are widely used in non-vectorized implementations of ARX ciphers such as ChaCha20 and SHA-2 and in implementations of Keccak.

There are two conclusions one can draw from the large overhead introduced by removing those rotation instructions. One option is to conclude that performance-oriented implementations will anyway rely on vector instructions and environments that do not use those vectorized implementations clearly do not care about performance anyway.

However, it is also interesting to observe that in all uses of rotate instructions we found, the rotation distance is public (in fact, it is a compile-time constant); what is secret is the value that is rotated. We believe that here it could be interesting for Intel to investigate a slightly more fine-grained approach to DOIT guarantees in the following sense: Instead of stating for each *instruction* if it is in the DOIT subset or not, one could state for each instruction, which of its *inputs* is guaranteed to not leak through timing on future microarchitectures when running in DOIT mode. Such a per-input instead of per-instruction granularity would allow to express, for example, that rotate instructions may leak the rotation distance, but not the value that is rotated. This would make it possible for Intel to introduce hardware optimizations for rotate instructions that, e.g., accelerate short rotation distances and still allow implementations of ChaCha20 or Keccak to use those instructions in a future-proof way. Should Intel opt to extend their DOIT guarantees in this way, we would require only small changes to support this in our implementation of DOIT in the Jasmin framework. Even though comparing ISAs is complex and out-of-scope for this paper, we remark that ARM's DIT and RISC-V's Zkt include rotations, which are constant-time w.r.t. the data, but not the immediate rotation amount. This means that an important source of overhead would not be present in those architectures, though other, different sources may still exist.

Acknowledgements

J. Jancar was supported by RedHat Research Czech and the Cyber-security Excellence Hub in Estonia and South Moravia (CHESS, 101087529). This research was supported by the Deutsche Forschungsgemeinschaft (DFG, German research Foundation) as part of the Excellence Strategy of the German Federal and State Governments – EXC 2092 CASA - 390781972; the German Federal Ministry of Education and Research (BMBF) in the course of the 6GEM research hub under grant number 16KISK038; and the Agence Nationale de la Recherche (ANR, French National Research Agency) as part of the France 2030 programme – ANR-22-PECY-0006.

References

[ABB+16] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016: 25th USENIX Security Symposium*, pages 53–70, Austin, TX, USA. USENIX Association, August 2016.

[ABB+17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: high-assurance and high-speed cryptography. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 1807–1823, Dallas, TX, USA. ACM Press, October 2017. DOI: 10.1145/3133956.3134078.

[ABB+19] José Bacelar Almeida, Cecile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. Machine-checked proofs for cryptographic standards: indifferentiability of sponge and secure high-assurance implementations of SHA-3. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 1607–1622, London, UK. ACM Press, November 2019. DOI: 10.1145/3319535.3363211.

[ABB+20] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: high-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy*, pages 965–982, San Francisco, CA, USA. IEEE Computer Society Press, May 2020. DOI: 10.1109/SP40000.2020.00028.

[ABB+23] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, Antoine Séré, and Pierre-Yves Strub. Formally verifying Kyber episode IV: implementation correctness. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(3):164–193, 2023. DOI: 10.46586/tches.v2023.i3.164-193.

[ABC+25] Santiago Arranz Olmos, Gilles Barthe, Chitchanok Chuengsatiansup, Benjamin Gregoire, Vincent Laporte, Tiago Oliveira, Peter Schwabe, Yuval Yarom, and Zhiyuan Zhang. Protecting cryptographic code against spectre-rsb: (and, in fact, all known spectre variants). In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, pages 933–948, Rotterdam, Netherlands. Association for Computing Machinery, 2025. ISBN: 9798400710797. DOI: 10.1145/3676641.3716015.

[AKM+15] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy*, pages 623–639, San Jose, CA, USA. IEEE Computer Society Press, May 2015. DOI: 10.1109/SP.2015.44.

[AOB+24] José Bacelar Almeida, Santiago Arranz Olmos, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Cameron Low, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, and Pierre-Yves Strub. Formally verifying kyber - episode V: machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology – CRYPTO 2024, Part II*, volume 14921 of *Lecture Notes in Computer Science*, pages 384–421, Santa Barbara, CA, USA. Springer, Cham, Switzerland, August 2024. DOI: 10.1007/978-3-031-68379-4_12.

[App20] Apple. Enable DIT for constant-time cryptographic operations, 2020. URL: https://developer.apple.com/documentation/xcode/writing-arm64-code-for-apple-platforms#Enable-DIT-for-constant-time-cryptographic-operations (visited on 01/09/2025).

[ARM20a] ARM. DIT, Data Independent Timing, 2020. URL: https://developer.arm.com/documentation/ddi0601/2024-12/AArch64-Registers/DIT--Data-Independent-Timing?lang=en (visited on 01/09/2025).

[ARM20b] ARM. How is instruction timing affected by the FEAT_DIT architectural feature?, 2020. URL: https://developer.arm.com/documentation/ka005181/latest (visited on 01/09/2025).

[BB03] David Brumley and Dan Boneh. Remote timing attacks are practical. In *USENIX Security 2003: 12th USENIX Security Symposium*, Washington, DC, USA. USENIX Association, August 2003.

[BBB+24] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales Paiva, Prasanna Ravi, and Goutam Tamvada. KyberSlash: exploiting secret-dependent division timings in Kyber implementations. Cryptology ePrint Archive, Paper 2024/1049, 2024. URL: https://eprint.iacr.org/2024/1049 (visited on 04/14/2025).

[BBG+19] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. DOI: 10.1145/3371075.

[BDG+13] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In Alessandro Aldini, Javier López, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2013. DOI: 10.1007/978-3-319-10082-1_6.

[Ber04] Daniel J. Bernstein. Cache-timing attacks on AES, 2004. URL: http://cr.yp.to/papers.html#cachetiming (visited on 01/15/2025).

[BGL$^+$21] Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. Structured leakage and applications to cryptographic constant-time and cost. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 462–476, Virtual Event, Republic of Korea. ACM Press, November 2021. DOI: 10.1145/3460120.3484761.

[Big23a] Eric Biggers. [PATCH] x86: enable Data Operand Independent Timing Mode. Posting to the Linux kernel mailing list, 2023. URL: https://lore.kernel.org/lkml/20230125012801.362496-1-ebiggers@kernel.org/t/ (visited on 01/09/2025).

[Big23b] Eric Biggers. Should linux set the new constant-time mode cpu flags? Posting to the Linux kernel mailing list, 2023. URL: https://lore.kernel.org/lkml/YwgCrqutxmX0W72r@gmail.com/T/ (visited on 01/15/2025).

[BLS12] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology - LATINCRYPT 2012: 2nd International Conference on Cryptology and Information Security in Latin America*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176, Santiago, Chile. Springer Berlin Heidelberg, Germany, October 2012. DOI: 10.1007/978-3-642-33481-8_9.

[BT11] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In Vijay Atluri and Claudia Díaz, editors, *ESORICS 2011: 16th European Symposium on Research in Computer Security*, volume 6879 of *Lecture Notes in Computer Science*, pages 355–371, Leuven, Belgium. Springer Berlin Heidelberg, Germany, September 2011. DOI: 10.1007/978-3-642-23822-2_20.

[CDvG$^+$20] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new spectre era. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 913–926. ACM, 2020. DOI: 10.1145/3385412.3385970.

[Cen22] Centre for Research on Cryptography and Security. Constant-timeness verification tools, 2022. URL: https://crocs-muni.github.io/ct-tools/ (visited on 01/06/2025).

[DPS15] Vijay D'Silva, Mathias Payer, and Dawn Xiaodong Song. The correctness-security gap in compiler optimization. In *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*, pages 73–87. IEEE Computer Society, 2015. DOI: 10.1109/SPW.2015.33.

[FDJ$^+$24] Marcel Fourné, Daniel De Almeida Braga, Jan Jancar, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "These results must be false": A usability evaluation of constant-time analysis tools. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024: 33rd USENIX Security Symposium*, Philadelphia, PA, USA. USENIX Association, August 2024.

[For23] Formosa Crypto Team. Libjade, 2023. URL: https://github.com/formosa-crypto/libjade (visited on 07/15/2023).

[GJN20] Qian Guo, Thomas Johansson, and Alexander Nilsson. A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 359–386, Santa Barbara, CA, USA. Springer, Cham, Switzerland, August 2020. DOI: 10.1007/978-3-030-56880-1_13.

[GM94] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 61–72. ACM, 1994. URL: https://gmplib.org/~tege/divcnst-pldi94.pdf (visited on 04/14/2025).

[Go24] Go, 2024. URL: https://github.com/golang/go/commit/bc1da38 (visited on 04/14/2025).

[GOP+10] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. Side-channel analysis of cryptographic software via early-terminating multiplications. In Donghoon Lee and Seokhie Hong, editors, *ICISC 09: 12th International Conference on Information Security and Cryptology*, volume 5984 of *Lecture Notes in Computer Science*, pages 176–192, Seoul, Korea. Springer Berlin Heidelberg, Germany, December 2010. DOI: 10.1007/978-3-642-14423-3_13.

[GYH18] Qian Ge, Yuval Yarom, and Gernot Heiser. No security without time protection: we need a new hardware-software contract. In *Proceedings of the 9th Asia-Pacific Workshop on Systems, APSys 2018, Jeju Island, Republic of Korea, August 27-28, 2018*, 1:1–1:9. ACM, 2018. DOI: 10.1145/3265723.3265724.

[Ham09] Mike Hamburg. Accelerating AES with vector permute instructions. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems – CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 18–32, Lausanne, Switzerland. Springer Berlin Heidelberg, Germany, September 2009. DOI: 10.1007/978-3-642-04138-9_2.

[Hos20] Diane Hosfelt. Added secret types. Rust RFC pull request #2859, 2020. URL: https://github.com/rust-lang/rfcs/pull/2859 (visited on 04/14/2025).

[Int22a] Intel. Data Dependent Prefetcher, 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/data-dependent-prefetcher.html (visited on 01/09/2025).

[Int22b] Intel. Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance, 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html (visited on 01/09/2025).

[Int22c] Intel. Data Operand Independent Timing Instructions, 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/data-operand-independent-timing-instructions.html (visited on 01/09/2025).

[Int22d] Intel. Fast Store Forwarding Predictor, 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/fast-store-forwarding-predictor.html (visited on 01/09/2025).

[Int23a] Intel. MCDT Data Operand Independent Timing Instructions, 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/mcdt-data-operand-independent-timing-instructions.html (visited on 01/09/2025).

[Int23b] Intel. MXCSR Configuration Dependent Timing, 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/mxcsr-configuration-dependent-timing.html (visited on 01/09/2025).

[JFD+22] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "They're not that hard to mitigate": what cryptographic library developers think about timing attacks. In *2022 IEEE Symposium on Security and Privacy*, pages 632–649, San Francisco, CA, USA. IEEE Computer Society Press, May 2022. DOI: 10.1109/SP46214.2022.9833713.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy*, pages 1–19, San Francisco, CA, USA. IEEE Computer Society Press, May 2019. DOI: 10.1109/SP.2019.00002.

[KKS+18] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *USENIX WOOT*, 2018. URL: https://www.usenix.org/conference/woot18/presentation/koruyeh (visited on 04/14/2025).

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113, Santa Barbara, CA, USA. Springer Berlin Heidelberg, Germany, August 1996. DOI: 10.1007/3-540-68697-5_9.

[Kön08] Robert Könighofer. A fast and cache-timing resistant implementation of the AES. In Tal Malkin, editor, *Topics in Cryptology – CT-RSA 2008*, volume 4964 of *Lecture Notes in Computer Science*, pages 187–202, San Francisco, CA, USA. Springer Berlin Heidelberg, Germany, April 2008. DOI: 10.1007/978-3-540-79263-5_12.

[KS09] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems – CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17, Lausanne, Switzerland. Springer Berlin Heidelberg, Germany, September 2009. DOI: 10.1007/978-3-642-04138-9_1.

[Lar23] Michael Larabel. Linux developers evaluating new "DOITM" security mitigation for latest Intel CPUs, 2023. URL: https://www.phoronix.com/review/intel-doitm-linux/2 (visited on 01/09/2024).

[Lin23] Linux, 2023. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/arch/arm64/kernel/entry.S?h=v6.2.1#n200 (visited on 04/14/2025).

[MN07] Mitsuru Matsui and Junko Nakajima. On the power of bitslice implementation on intel core2 processor. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 121–134, Vienna, Austria. Springer Berlin Heidelberg, Germany, September 2007. DOI: 10.1007/978-3-540-74735-2_9.

[MNM+24] Nicholas Mosier, Hamed Nemati, John C. Mitchell, and Caroline Trippel. Serberus: protecting cryptographic code from spectres at compile-time. In *2024 IEEE Symposium on Security and Privacy*, pages 4200–4219, San Francisco, CA, USA. IEEE Computer Society Press, May 2024. DOI: 10.1109/SP54263.2024.00048.

[MR18] Giorgi Maisuradze and Christian Rossow. ret2spec: speculative execution using return stack buffers. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 2109–2122, Toronto, ON, Canada. ACM Press, October 2018. DOI: 10.1145/3243734.3243761.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20, San Jose, CA, USA. Springer Berlin Heidelberg, Germany, February 2006. DOI: 10.1007/11605805_1.

[PL18] Jin Hyung Park and Dong Hoon Lee. FACE: fast AES CTR mode encryption techniques based on the reuse of repetitive data. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):469–499, 2018. ISSN: 2569-2925. DOI: 10.13154/tches.v2018.i3.469-499.

[RIS21] RISC-V. Zkt extension :: RISC-V ISA manual, 2021. URL: https://riscv-software-src.github.io/riscv-unified-db/manual/html/isa/isa_20240411/exts/Zkt.html (visited on 04/14/2025).

[RIS23] RISC-V. Zkvt extension :: RISC-V ISA manual, 2023. URL: https://riscv-software-src.github.io/riscv-unified-db/manual/html/isa/isa_20240411/exts/Zvkt.html (visited on 04/14/2025).

[SBG+22] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. Enforcing fine-grained constant-time policies. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 83–96, Los Angeles, CA, USA. ACM Press, November 2022. DOI: 10.1145/3548606.3560689.

[SBG+23] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. Typing high-speed cryptography against spectre v1. In *2023 IEEE Symposium on Security and Privacy*, pages 1094–1111, San Francisco, CA, USA. IEEE Computer Society Press, May 2023. DOI: 10.1109/SP46215.2023.10179418.

[Sch00] Werner Schindler. A timing attack against RSA with the Chinese remainder theorem. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 109–124, Worcester, Massachusetts, USA. Springer Berlin Heidelberg, Germany, August 2000. DOI: 10.1007/3-540-44499-8_8.

[SLP+24] Moritz Schneider, Daniele Lain, Ivan Puddu, Nicolas Dutly, and Srdjan Capkun. Breaking Bad: how compilers break constant-time implementations. arXiv report 2410.13489, 2024. URL: https://arxiv.org/abs/2410.13489 (visited on 04/14/2025).

[TTM+02] Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Minematsu, and Hiroshi Miyauchi. Cryptanalysis of block ciphers implemented on computers with cache. In *Proceedings of the International Symposium on Information Theory and Its Applications, ISITA 2002*, pages 803–806, 2002.

[WPH+22] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. Hertzbleed: turning power side-channel attacks into remote timing attacks on x86. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022: 31st USENIX Security Symposium*, pages 679–697, Boston, MA, USA. USENIX Association, August 2022.

[WPW+23] Yingchen Wang, Riccardo Paccagnella, Alan Wandke, Zhao Gang, Grant Garrett-Grossman, Christopher W. Fletcher, David Kohlbrenner, and Hovav Shacham. DVFS frequently leaks secrets: hertzbleed attacks beyond SIKE, cryptography, and CPU-only data. In *2023 IEEE Symposium on Security and Privacy*, pages 2306–2320, San Francisco, CA, USA. IEEE Computer Society Press, May 2023. DOI: 10.1109/SP46215.2023.10179326.

[ZBC+23] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. Ultimate SLH: taking speculative load hardening to the next level. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023: 32nd USENIX Security Symposium*, pages 7125–7142, Anaheim, CA, USA. USENIX Association, August 2023.