High-assurance zeroization

Santiago Arranz Olmos[1], Gilles Barthe[1,2], Ruben Gonzalez[1,6], Benjamin Grégoire[3], Vincent Laporte[4], Jean-Christophe Léchenet[3], Tiago Oliveira[1] and Peter Schwabe[1,5]

[1] MPI-SP, Bochum, Germany
[2] IMDEA Software Institute, Madrid, Spain
[3] Inria, Sophia-Antipolis, France
[4] Inria, Nancy, France
[5] Radboud University, Nijmegen, The Netherlands
[6] Neodyme AG, Munich, Germany

Abstract. In this paper we revisit the problem of erasing sensitive data from memory and registers during return from a cryptographic routine. While the problem and related attacker model is fairly easy to phrase, it turns out to be surprisingly hard to guarantee security in this model when implementing cryptography in common languages such as C/C++ or Rust. We revisit the issues surrounding zeroization and then present a principled solution in the sense that it guarantees that sensitive data is erased and it clearly defines when this happens. We implement our solution as extension to the formally verified Jasmin compiler and extend the correctness proof of the compiler to cover zeroization. We show that the approach seamlessly integrates with state-of-the-art protections against microarchitectural attacks by integrating zeroization into Libjade, a cryptographic library written in Jasmin with systematic protections against timing and Spectre-v1 attacks. We present benchmarks showing that in many cases the overhead of zeroization is barely measurable and that it stays below 2% except for highly optimized symmetric crypto routines on short inputs.

Keywords: Secret erasure, clear stack memory, defense in depth, high-assurance cryptography

1 Introduction

Essentially all cryptographic software uses memory to temporarily store sensitive data during execution. Usually this memory is allocated on the stack, which means that when a cryptographic routine terminates and returns control to the caller, the memory becomes "invalid", but the sensitive contents remain. It is often mandated that such sensitive data in memory is erased or *zeroized* once it is no longer used. While overwriting data with zeroes seems like an easy task, it turns out that there are multiple failure modes and that many popular open-source crypto libraries actually do not have a sound approach to memory zeroization and in some cases can be shown to leave content in stack memory that allows trivial key recovery. This failure to perform zeroization has been noticed for specific libraries before; in this paper, we systematize this observation by considering multiple libraries. The conclusion of our systematization is that unsound zeroization is a pervasive problem.

One might ask if this is an actual problem and what the reasons are to mandate zeroization – after all the memory is "invalid" after return. There are three reasons to consider an attacker, who obtains the stack contents after a cryptographic routine returns:

Certification. Certification of cryptographic software (or more generally "cryptographic modules") often requires zeroization of sensitive data. For example, the implementation guidance for FIPS 140-3 [oSTfCS20] states in Section 9.7.A:

> *"A cryptographic module shall provide methods to zeroize all plaintext secret and private cryptographic keys and CSPs within the module."*

Here, CSP stands for "critical security parameter". Similarly, the Common Criteria mandate in FCS_CKM_EXT.4(a) (*Cryptographic Key and Key Material Destruction*) that

> *"the TSF shall destroy cryptographic keys in accordance with a specified cryptographic key destruction method [selection: For volatile memory, the destruction shall be executed by a single direct overwrite [selection: consisting of a pseudo-random pattern using the TSF's RBG, consisting of a pseudo-random pattern using the host environment's RBG, consisting of zeroes] following by a read-verify[...]]",*

where TSF stands for the security functionality of the product under evaluation and RBG stands for "random bit generator".

Defense in depth. Cryptographic libraries are used by a plethora of applications and the application-level code is typically not under control of the library programmer. Bugs in the application that allow out-of-bound reads of memory may be used to obtain memory content from the whole address space of the application, including, of course, all stack content. The most prominent example for such a bug is HeartBleed [Syn14, CVE14]. While this bug was part of the OpenSSL library (and not some application), it was not a bug in any of the cryptographic components. Zeroization helps to limit the damage that an attacker can do when exploiting such a vulnerability.

Forward secrecy. Finally, and most importantly, a principled approach to zeroization is required to guarantee forward secrecy. Many modern cryptographic protocols are built in such a way that compromise of long-term keys does not allow an attacker to recover plaintext of messages sent in the past; see, e.g., [Don17, Per, Res18]. This is achieved by using only ephemeral keys to ensure confidentiality, which means that implementations need to ensure that those keys—and all information that allows recovery of these keys or plaintext encrypted under those keys—are erased from memory when they are no longer used. In other words, implementations need to ensure that keys are as ephemeral as specified in the protocol design.

For the remainder of this paper we assume that there *are* good reasons to mandate memory zeroization. Specifically, we consider the following attacker model induced by mandating zeroization.

Attacker model. Throughout this paper we consider an attacker who obtains leakage from a cryptographic computation. This leakage contains (possibly among other information) the content of all stack memory used by the cryptographic routine right after the routine returns control to the caller. We additionally include in the leakage the content of all registers and flags at the point of return from the cryptographic routine. We define a *cryptographic routine* to be an API function of a cryptographic library. This API function may call subroutines, but stack and register content is not leaked to the attacker every time such a subroutine returns. This reflects the idea that cryptographic computations need to "clean up" only when they return to a caller outside their control.

Note that this attacker model focuses on the core problem of zeroization and excludes several additional attack vectors that need to be addressed to prevent leakage of ephemeral

sensitive data when a device is compromised. The most obvious attack vector is leakage through microarchitectural side channels. We will return to this later in the paper and show that zeroization composes seamlessly with state-of-the art (speculative) timing-leakage protection. In addition, arguments to cryptographic routines like encryption keys or plaintext are commonly owned by the caller and so it is the caller's responsibility to zeroize those arguments once they are no longer used. This is out of scope of our approach, which is focused on protecting cryptographic libraries rather than application code. Finally, while stack (and register) zeroization is essential to prevent data from being written to disk as part of swapping or hibernation (aka "suspend to disk"), further measures are typically needed on system level to address this problem. For example, the `mlock` system call under Linux prevents memory regions from being swapped to disk and security-critical systems will probably want to disable hibernation altogether.

Contributions and organization of this paper. In Section 2 we first motivate why zeroization is not as easy as one might think by identifying 3 different failure modes (plus the approach of explicitly excluding leakage of architectural state from the attacker model). We investigate how zeroization is handled in 11 popular open-source cryptographic libraries written in C or Rust. None of those libraries actually addresses the issue of memory zeroization with a principled and sound approach. We show that, for example, routines in libsodium and OpenSSL allow trivial key-recovery in our attacker model, despite attempts to perform some stack zeroization.

We then, in Section 3, review possible solutions both on the caller's side and on the callee side and discuss why those solutions are not widely implemented and used.

The main contribution of this paper is presented in Section 4. We describe our principled approach to stack (and register) zeroization for the Jasmin framework for high-assurance cryptography [ABB+17, ABB+20]. The three main components of the Jasmin framework are its programming language, which can be used for writing high-speed implementations using the "assembly in the head" paradigm, its compiler, which comes with formal proofs of functional correctness and of preservation of side-channel protection (specifically preservation of constant-time), and its verification infrastructure, which can be used to prove functional correctness, constant-timeness, and reductionist security via an embedding to EasyCrypt. The Jasmin framework has been used for writing efficient and formally verified implementations of several key cryptographic primitives; in particular, it has been used to develop the Libjade library for post-quantum cryptography.

In this work, we use two main properties of Jasmin. First, Jasmin programs have a well-defined interface to outside callers (through `export` functions). Second, the Jasmin compiler can predict the stack usage of Jasmin programs at compile time. The latter is possible because, on the one hand, Jasmin programs are compiled as a whole, and on the other hand, for our applications, i.e. cryptographic primitives, programs do not use recursion. We use these two properties to leverage a compiler-based solution, which remains compatible with the global guarantees offered by Jasmin. In particular, we show that our modified compiler preserves correctness. In addition, we prove that zeroization integrates seamlessly with existing guarantees for constant-time and speculative constant-time [SBG+22].

In Section 5 we show that the overhead of our protections is very small. Specifically, we extend Spectre-v1 protected Libjade [For23] from [SBG+22] with our protections against architectural-state leakage and present benchmarks showing that the cost is, for many routines, barely measurable, and remains solidly below 2% for all primitives except for highly optimized symmetric crypto routines on very short inputs.

We conclude the paper with a discussion of directions for future work in Section 6.

Related work. Multiple earlier works consider zeroization of sensitive information from different angles. Already in 2005, Chow, Pfaff, Garfinkel, and Rosenblum showed that it is generally not safe to rely on the fact that data will eventually be overwritten [CPGR05]. Their experiments show that *"data can remain in memory for days or weeks, even persisting across reboots"*. They introduce the "data lifetime" cycle:

> *"The span from first write to last read is the ideal lifetime. The data must exist in the system at least this long. The span from first write to deallocation is the secure deallocation lifetime. The span from first write to the first write of the next allocation is the natural lifetime. Because programs often rely on reallocation and overwrite to eliminate sensitive data, the natural lifetime is the expected data lifetime in systems without secure deallocation."*

Our attacker model roughly corresponds to the secure deallocation lifetime. Several similar discussions are found in the literature. In particular, Percival [Per14] reports on the challenges and shortcomings of tricking a compiler into zeroing buffers and stacks. Chapman [Cha17] provides another thorough discussion of why systematic zeroization is hard and proposes a path forward mostly in the context of the Ada and SPARK programming languages. Parts of the paper are akin to our discussion in Section 2 and also come to similar conclusions. Unlike our work, [Cha17] emphasizes the task of identifying what data in a program is sensitive; the approach we present here does not require this analysis and hence we only briefly discuss this in Section 6. None of these works presents an implementation of a solution to systematic zeroization.

Yang, Johannesmeyer, Olesen, Lerner, and Levchenko [YJO+17] provide a detailed analysis of the issue of dead-store elimination and of some popular scrubbing techniques; their analysis can be seen as much more in-depth review of what we briefly recall in Section 2.1. In addition, they provide a scrubbing function `secure_memzero` that combines different scrubbing techniques, and evaluated the use of the function with GCC, LLVM, and Visual C. Last, they implement a scrubbing safe dead-store elimination option in LLVM.

Our work is most closely related to [SCA18]. In that work, Simon, Chisnall, and Anderson first investigate how mainstream compilers make it actively hard to achieve certain security-related properties including zeroization. They then propose three approaches to perform zeroization in the LLVM compilers—all approaches are implemented and publicly available from `https://github.com/lmrs2/zerostack`. The first approach, coined function based, applies zeroization to all sensitive functions. The second approach, coined stack-based, avoids zeroing the same stack area repeatedly. Their last approach, called call graph-based (**CGB**), is similar to what we have proposed for the Jasmin compiler. In particular, it does not support programs with unbounded recursion. As far as we know, their solution has unfortunately never been integrated into mainline LLVM. The main difference of our work is that we embed our solution into the Jasmin framework for high-assurance cryptography and are able to provide a formal proof of our approach. We briefly comment on the pros and cons of integrating zeroization in mainstream vs dedicated compilers in the conclusion.

Several works consider the challenges of memory scrubbing in the broader context of compiler-induced security bugs (CISB). These are discussed e.g. in [DPS15, XLD+23].

The problem of memory scrubbing has also been studied in the context of language-based security. Chong and Myers [CM05] propose a general language-based approach to let programmers express what data needs to be erased and a type system with multiple levels of confidentiality together with *declassification* and *erasure* operators. Hunt and Sands [HS08] study erasure as a noninterference property, and proposes a type system to check that erasure is performed, similar to [CM05]. Daniel, Bardin, and Rezk [DBR22] present a tool to analyze binaries, among other properties, for zeroization of sensitive data. These works do not consider the interactions between erasure and compilation.

Finally, there is a large body of work that formalizes the interactions between compilation and security. Many of these works focus on the theoretical underpinnings of these interactions, see e.g. [PAC19, ABC+21] for recent overviews. Other works focus on preservation or mitigation of specific properties, notably constant-time [CSJ+19, BGLP21].

Responsible disclosure. We informed the maintainers of libsodium and OpenSSL about our findings presented in Section 2.2. Both replied within a day acknowledging our findings and stating that they do not consider the findings a vulnerability, because neither library makes any claims about security in the attacker model we consider in this paper. Consequently they also did not request an embargo. OpenSSL considers the finding a *"real security problem (at least on some platforms/compilers/architectures)"* in the form of *"a missing security hardening"*, for which they would be *"likely to accept a patch"*.

Artifact. A software artifact that enables readers to reproduce the benchmarks we present in Section 5 and the key-recovery from libsodium's ChaCha20 implementation we discuss in Section 2.2 is available at `https://artifacts.formosa-crypto.org/data/clearstack.tar.bz2`.

2 Failure modes

One might think that overwriting used stack space and erasing values in registers before returning from a cryptographic routine should not be a difficult thing to do. Unfortunately this is not the case in many programming languages. In this section we look into different failure modes when attempting to protect cryptographic software in the attacker model we introduced in Section 1. We investigate approaches for stack zeroization taken by 9 popular open-source crypto libraries written in C/C++ (possibly with assembly), namely BearSSL 0.6 [Por23], GnuTLS 3.6.12 [Gnu23], libsodium 1.0.18 [Lib23], mbedtls 3.3 [Mbe23], NSS 3.91 [NSS23], OpenSSL 3.1.1 [Ope23], TinyDTLS 0.9 [Tin23], and WolfSSL 5.6.3 [Wol23]. We also consider two libraries written in Rust, namely ring 0.17 [Smi23] and Dalek 2.0 (RC3) [Dal23].

While many of these libraries make an attempt to zeroize sensitive data, none of them explicitly claims security in the attacker model we use in this paper. It should thus not surprise that none of the libraries is actually secure in this attacker model. In his reply to our disclosure e-mail, Denis, maintainer of libsodium, describes their approach as follows:

> *"Overwriting all secrets after use is not a goal of libsodium. It's silently done in a couple places where it's simple and cheap to do, but this is not a guarantee documented anywhere, and not something that's planned to be become one."*

In the subsequent subsections, we build up a hierarchy of failure modes and explain why languages like C/C++ or Rust and existing mainstream compilers make it close to impossible to not hit at least one of them.

2.0 Perform no zeroization

A common technique of implementers is to ignore or shift the problem of memory scrubbing. The BearSSL crypto library, for example, recommends to overwrite the stack with garbage data after BearSSL was used. This shifts the responsibility of memory scrubbing to the application developer employing BearSSL. Unfortunately, this is not at all apparent to users of the library. Besides, even if an application developer was aware of their responsibility to clear the stack, they would probably fail to do so. This is simply because on a source code level it is completely unclear how much stack space was used by BearSSL.

The popular *ring* implementation of cryptographic primitives employs the same approach, and does not wipe sensitive data from memory either. Even though it is written in the memory-safe language Rust, it can be employed by C/C++ programs. A memory-corruption in the C/C++ application using *ring* can therefore also leak sensitive data previously processed within the library. This shows that writing cryptography in a memory-safe language does not eliminate the need for memory scrubbing.

2.1 Zeroization falling prey to compiler optimizations

A straightforward way to overwrite memory in C/C++ programs is to overwrite the desired memory region in a loop or by calling the `memset` function. Both approaches are inherently flawed when it comes to zeroization of sensitive data. The reason is that compilers will commonly apply an optimization called dead-store elimination (DSE), i.e., removing stores of variables that will not be read anymore during their life time. Since zeroization occurs precisely once sensitive data is no longer needed, it is a textbook target for DSE.

From the libraries we investigated, TinyDTLS and NSS contain examples of attempts to zeroize sensitive data using `memset`. TinyDTLS calls `memset` directly to scrub memory from sensitive data, such as key material. NSS calls the macro `PORT_Memset`, which expands to `memset` on all platforms and, as demonstrated by [YJO+17], is eliminated by common compilers.

2.2 Zeroization in API functions only

Libraries that perform zeroization and put effort into avoiding the DSE pitfall typically use volatile function pointers to `memset` (OpenSSL), declare volatile memory regions (WolfSSL), or employ memory barriers (Libsodium). See also the detailed discussion in [YJO+17]. A more unified approach for zeroization in C is in principle offered by the `memset_s` function, which is guaranteed to not be eliminated by the compiler. However, since `memset_s` is part of the optional C17 appendix K [Int17], it does not have to be supported by a standard-compliant compiler. At the time of writing, no mainstream compiler supports `memset_s`.

Unfortunately, employing a zeroization routine that does not fall prey to DSE is by itself not sufficient to erase all sensitive data. All remaining libraries we investigated (i.e., those not listed in Sections 2.0 and 2.1) applied zeroization to sensitive data only selectively; typically only to secret data in the stack frame of API functions. Sensitive data derived from this secret data and contained in stack frames further down the call stack, are not erased. An example of this scenario can be found in the ChaCha20 implementation of libsodium (and very similar in the ChaCha20 implementation of OpenSSL). The body of libsodium's ChaCha20 API function looks like this:

```
chacha_keysetup(&ctx, k);
chacha_ivsetup(&ctx, n, NULL);
memset(c, 0, clen);
chacha20_encrypt_bytes(&ctx, c, c, clen);
sodium_memzero(&ctx, sizeof ctx)
```

We can see that after the call to `chacha20_encrypt_bytes`, the local variable `ctx` containing the key is erased using the (carefully implemented) `sodium_memzero` routine. However, inside `chacha20_encrypt_bytes`, the entire context, including the encryption key, is copied onto the stack:

```
...
uint32_t j0, j1,  j2,  j3,  j4,  j5,  j6,  j7, \
         j8, j9, j10, j11, j12, j13, j14, j15;
```

```
...
j0  = ctx->input[0];
j1  = ctx->input[1];
...
j15 = ctx->input[15];
```

So even though the developer was actively zeroizing sensitive data in memory, the stack still contains a full copy of the secret key after libsodium returns control to the caller. To show that this is indeed an exploitable behavior, we developed a small example program with a memory-corruption bug that employs libsodium and leaks secret keys. [1]. The implementation of ChaCha20 in OpenSSL suffers from the exact same behavior.

The difficulties developers face by trying to identify all variables containing sensitive data is best described by the following commit message of Brian Smith, maintainer of the ring library [Smi23].

> *"Apart from that, by inspection, it is clear that there are many places in the code that don't call OPENSSL_cleanse where they "should". It would be difficult to find all the places where a call to OPENSSL_cleanse "should" be inserted. It is unlikely we'll ever get it right. Actually, it's basically impossible to get it right using this coding pattern."*

2.3 Zeroization on source level

Given the failure modes described in the previous subsections, one might think that zeroizing (potentially) sensitive data at the end of *all* functions, not just API functions, or zeroizing whenever a variable goes out of scope, might result in a sound solution. In Rust, there exist crates to implement this approach; most notably, the `zeroize` [Rus23b] and the `clear_on_drop` [Bar23] crates. The former one being used, e.g., by Dalek [Dal23].

Unfortunately, even this approach is insufficient when implemented on source (e.g., C, C++, or Rust) level. The reason is that not all data that is placed on the stack by the compiler is visible on source level. The most obvious example for data that is written on the stack and that is not visible on source level are callee-save registers that are spilled to the stack at the beginning of a function and restored to their original values before returning. More generally, the compiler is free to spill temporary variables on the stack as part of optimization and as a consequence it is largely out of the programmer's control what sensitive data is actually stored on the stack. For example, consider the `crypto_scalarmult` routine of the "`ref10`" implementation of X25519 [Ber06] included in the SUPERCOP benchmarking framework and shown in Listing 1. The source-visible variables declared at the beginning of the function account for a total of 328 bytes when compiled for AMD64 (7 variables of type `fe` account for a total of $7 \cdot 40 = 280$ bytes, the array `e` uses 32 bytes, and the 4 `int` variables take a total of 16 bytes). However, compiling this code with GCC 10.2 and flags `-O3 -fstack-usage` computes a (worst-case) stack usage of 464 bytes. Some of the extra 136 bytes are simply padding for alignment, but most are used for source-level invisible data stored on the stack. Note that the `-fstack-usage` flag computes worst-case stack usage on a per-function granularity – it does not take into account stack space used by the functions *called by* `crypto_scalarmult`. See also Section 3.

Like others before us [Per14, SCA18, YJO+17] we conclude that it is impossible to protect against the attacker we consider in this paper by zeroizing variables inside cryptographic routines on source level in commonly used languages such as C/C++ or Rust. It is thus not surprising that while many crypto libraries include some "best effort" stack zeroization, no library makes any claims about protecting against attackers who obtain residual stack data or content of registers after a crypto routine returns.

[1] https://github.com/rugo/chacha20-demo

Listing 1 Source code of `crypto_scalarmult` from the `ref10` implementation of X25519.

```
typedef crypto_int32 fe[10];

int crypto_scalarmult(unsigned char *q,
                      const unsigned char *n,
                      const unsigned char *p)
{
  unsigned char e[32];
  fe x1, x2, z2, x3, z3, tmp0, tmp1;
  int pos;
  unsigned int i, swap, b;

  for (i = 0;i < 32;++i) e[i] = n[i];
  e[0] &= 248;
  e[31] &= 127;
  e[31] |= 64;
  fe_frombytes(x1,p);
  fe_1(x2);
  fe_0(z2);
  fe_copy(x3,x1);
  fe_1(z3);

  swap = 0;
  for (pos = 254;pos >= 0;--pos) {
    b = e[pos / 8] >> (pos & 7);
    b &= 1;
    swap ^= b;
    fe_cswap(x2,x3,swap);
    fe_cswap(z2,z3,swap);
    swap = b;
    fe_sub(tmp0,x3,z3);
    fe_sub(tmp1,x2,z2);
    fe_add(x2,x2,z2);
    fe_add(z2,x3,z3);
    fe_mul(z3,tmp0,x2);
    fe_mul(z2,z2,tmp1);
    fe_sq(tmp0,tmp1);
    fe_sq(tmp1,x2);
    fe_add(x3,z3,z2);
    fe_sub(z2,z3,z2);
    fe_mul(x2,tmp1,tmp0);
    fe_sub(tmp1,tmp1,tmp0);
    fe_sq(z2,z2);
    fe_mul121666(z3,tmp1);
    fe_sq(x3,x3);
    fe_add(tmp0,tmp0,z3);
    fe_mul(z3,x1,z2);
    fe_mul(z2,tmp1,tmp0);
  }
  fe_cswap(x2,x3,swap);
  fe_cswap(z2,z3,swap);

  fe_invert(z2,z2);
  fe_mul(x2,x2,z2);
  fe_tobytes(q,x2);
  return 0;
}
```

3 Possible solutions

As it is impossible to offer a principled solution to zeroization *inside* the crypto routine *on source level*, we have two options: we either perform zeroization outside the crypto routine, i.e., on the caller side, or we work on lower than source level. We discuss these two options in the following.

3.1 Caller-side zeroization

Popular libraries such as BearSSL [Por23] recommend overwriting the stack after the call to API functions and offer routines to perform such zeroization on the caller side. The main limitation of this approach is that the caller has no information about how much stack space has been used by the crypto routine and thus needs to estimate stack usage. This is problematic, as a too low estimate could lead to secret data remaining on the stack and a too high estimate could violate stack size limits or even overwrite data in other segments, especially in an embedded setting. It might in principle be possible to compute the worst-case stack usage through per-function static analysis combined with call-graph analysis and forward that information from the build system. Both GCC and LLVM provide support for stack-usage analysis [GCC23, Rus23a], but we are not aware of any library using these features in their build system for zeroization. Furthermore, this approach does not take care of zeroing registers.

An elegant approach in a similar spirit is to allocate a new memory segment prior to entering a cryptography library's code. This segment is then used as stack space and completely wiped once the library returns control to the program. An implementation of this approach in Rust is provided by the `Eraser` crate [Spr22]. However, the caller would again have to make a guess about how much stack space is used in the library. Additionally, allocating a new memory segment and moving the stack to that segment requires platform and operating-system dependent subroutines. These low-level subroutines need to be written in platform-specific assembly to directly manipulate the stack-pointer register.

3.2 Callee-side zeroization

If we want to implement zeroization on the callee side, i.e., "clean up before we return", we need to either implement the complete crypto routine, including zeroization, in assembly, or we require compiler support for zeroization of stack and registers. The first approach comes with the usual limitations of writing code in assembly: code is hard to maintain and error-prone, in particular when it comes to features like zeroization that are not covered by functional testing.

Consequently, the only remaining option for systematic zeroization is implementation in the compiler. Unsurprisingly, zeroization passes in mainstream compilers have been proposed before. GCC contributors discussed including a `clear_stack` function attribute for certain eligible functions and a `security_sensitive` attribute for variables [Gut16]. Similar work has been proposed for the LLVM compiler backend by Simon, Chisnall, and Anderson [SCA18]. Their work introduces a `__zero_on_return` function attribute that instructs the clang compiler to add a stack and register zeroization routine before a function's exit. In contrast to GCCs `clear_stack` proposal, this `__zero_on_return` approach also has an implementation employing the library's call graph. One thing these approaches have in common is that they still leave the developer in charge of deciding where zeroization happens by annotating what functions are "sensitive". More importantly though, these proposals for stack zeroization have unfortunately never been adopted in the mainline compilers and are thus not widely available to developers. Both GCC and clang however do support zeroing registers on function return with the `zero-call-used-reg`

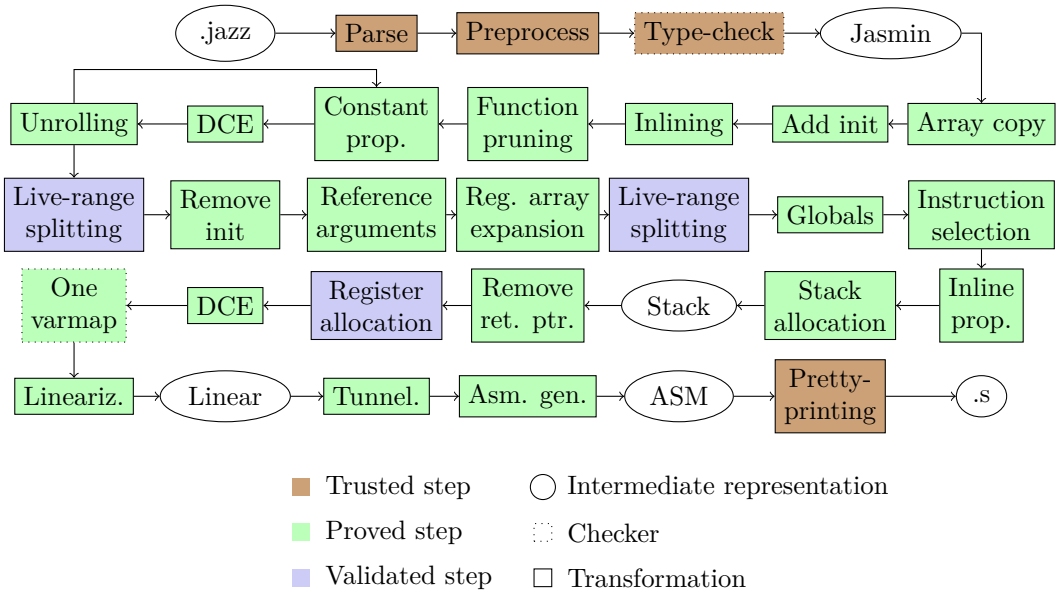


Figure 1: Program transformations in the Jasmin compiler

compiler option. This is possible as zeroing registers is a much simpler problem than zeroing the dynamically growing stack.

4 A principled solution in the compiler

In this section, we describe our principled approach for zeroing the stack. Our approach is integrated in the Jasmin compiler, and comes with formal guarantees of correctness and security. For completeness, we start by providing background on the Jasmin language.

4.1 Background on Jasmin

Jasmin [ABB+17, ABB+20] is a programming framework for developing efficient, high-assurance cryptography. The framework is built around the Jasmin programming language, which lets programmers write efficient and readable code using "assembly in the head". Informally, Jasmin is an assembly-like language with structured control flow; in other words, it provides explicit access to assembly instructions (except `GOTO`s) for the architecture the developer is targeting, and has the usual control flow constructs **if**, **while** and function calls, as well as unrolled loops (denoted by **for**). Jasmin also supports zero-cost abstractions that increase readability and are predictably translated to assembly. One example of zero-cost abstraction is variables. A variable in a Jasmin program is an n-bit word that has either the `reg` or `stack` storage class, determining its storage (variables are not spilled by the compiler). One key benefit of the Jasmin language is that it comes with a machine-checked semantics that specifies the behavior of Jasmin programs; the semantics is written in the Coq proof assistant.

Jasmin programs are compiled using the Jasmin compiler. Currently, the compiler targets the AMD64 (aka, x86-64) architecture and has experimental support for ARMv7-M. The compiler consists of around 30 passes, summarized in Figure 1. We now describe the most relevant passes for our purposes. The initial passes are Jasmin to Jasmin transformations. For instance, the Inlining pass removes inline functions by inserting their

bodies at their call sites. The Unrolling pass removes for loops by unrolling them. The Lowering pass replaces high-level assignments with specific assembly instructions, e.g. `x += y` with `ADD x, y` in x86-64. The Stack Allocation pass replaces stack variables by memory accesses relative to the stack pointer; it is in this pass that we can compute the amount of stack needed for the program. The Register Allocation pass assigns variables to architectural registers. The Linearization pass replaces structured control flow with `GOTO`s and labels, translating the program into a language called Linear. We insert our new stack and register zeroization passes after Linearization and before Tunneling, Assembly Generation and Pretty-printing.

To the exception of the parsing, preprocessing, type-checking and pretty-printing, each pass of the Jasmin compiler is certified in the Coq proof assistant. Following common practice in certified compilation, each pass is either implemented and verified in the Coq proof assistant, or it is implemented in an external language (OCaml) and its results are verified using a verifier which is implemented and verified in the Coq proof assistant. By combining the correctness results of each individual pass, one can prove that the Jasmin compiler is correct, i.e., it preserves the behavior of (safe) programs. This entails that functional correctness and reductionist security carry from source programs to the generated assembly.

In addition, the Jasmin framework provides a back-end to the EasyCrypt proof assistant [BGHZ11, BDG+14], which can be used for proving functional correctness and reductionist security. Last, the Jasmin framework provides guarantees about side-channel protections. In particular, there exist type systems for proving that Jasmin source programs are constant-time—the gold standard for protection against cache-based timing attacks in absence of speculation—or speculative constant-time—an enhancement of constant-time to protect against Spectre v1 attacks [KHF+19]. For details see [SBG+22].

The Jasmin framework has been used to develop efficient implementations of key cryptographic routines. In particular, Jasmin is the main medium used by the Libjade library, which provides formally verified implementations of post-quantum cryptographic algorithms [For23].

4.2 Design choices

Any sound approach to zeroization must make four design choices: when should it be applied, what precision should it achieve, how should it be implemented, and where in the compilation chain should it be performed? We comment on each design choice in turn.

When to zeroize? The first question is when should zeroization be applied. In order to achieve the best trade-offs between security and performance, the ideal compromise is to apply zeroization at the interface between cryptographic routines and application-level code. Conveniently, Jasmin offers a clean interface between cryptographic code and the caller. Specifically, Jasmin programs contain **export** functions and internal functions. Internal functions cannot be called from the outside. On the contrary, **export** functions are the entry point of the programs, and the exit of the **export** function is the point where we exit the Jasmin world and return to the outside world. This is a natural place to introduce the zeroization code. However, this does not mean that we do not clear the memory used by the internal functions! At the end of the **export** function, the code injected by the compiler zeroizes the entire stack used during the call of that function, that is, not only the stack frame of the **export** function, but also the stack space used by the internal functions it transitively calls.

What precisely to zeroize? Any approach to zeroization must decide between precision and simplicity. On the one hand, minimizing the number of memory writes could potentially

minimize overhead. On the other hand, a complex approach may be more difficult to implement, even more so because the Jasmin compiler is verified, without necessarily bringing any significant performance improvement over a naive approach.

We choose to implement a simple and systematic approach, where the injected code blindly clears all used stack space and auxiliary registers, with the hope that the zeroization will be cheap anyway. This hope was also motivated by the fact that stack allocation in the Jasmin carefully minimizes overall stack usage by re-using space when stack arrays go out of scope. The hope was confirmed by our benchmarks, see Section 5.

How to implement zeroization? The natural implementation of stack zeroization is using a loop, repeatedly clearing the stack one chunk at a time. But since the bound on the stack size is known at compile time, we can also unroll this loop and generate sequential code performing zeroization. The first approach has the advantage of keeping the code size overhead small, the second one should typically be faster. We can also imagine mixing both approaches, i.e., use a partially unrolled loop. However, we choose to implement only these two strategies, that we call *loop* and *unrolled*, respectively.

Another design choice is the clearing instruction that is used at each step. We simply use `MOV` and `VMOV` to write zeros. As for what size to clear at each step, here again we choose the simple approach. All the instructions introduced clear the same number of bits. This implies that we can clear only a size that is a multiple of this number. If the stack size is not a multiple, we round up and clear the next largest multiple of stack bytes. This means that enabling the zeroization has an impact on the analysis mentioned in the previous paragraph. Details are given in the **Analysis** paragraph in Section 4.3. The user can select the size cleared at each step, either 8, 16, 32, 64, 128 or 256 bits. If this is less than or equal to 64 bits, we use `MOV`. Otherwise, we use `VMOV`. There is also a default value which is the stack alignment of the **export** function.

Where in the compiler should zeroization be performed? The insertion of the zeroization pass in the compiler chain involves several considerations. First, zeroization can only be performed once the compiler has fixed the stack space used by the **export** function and all functions further down the call stack. In the Jasmin compiler, this done during Stack Allocation, so we must insert the Zeroization pass after Stack Allocation. We choose to perform Zeroization after Linearization, although it could go at any point after Register Allocation.

4.3 Implementation overview

Implementing zeroization in the compiler comprises two main tasks: computing what to clear (which regions of memory need zeroizing) and generating the corresponding assembly instructions.

Analysis. During the Stack Allocation pass, the compiler computes two pieces of information for each function (whether it is **export** or internal): the size of the stack frame used and the minimal stack alignment required. For the frame size, the compiler determines a frame layout, trying to share stack memory for local variables (meaning that if a stack variable is dead, then it tries to reuse that stack region), and then it deduces the amount of stack needed by the function. The stack alignment is deduced from the writes to the stack. It is the minimum alignment required so that all writes to the stack are aligned.

The compiler does this for each function, and then propagates the information to account for function calls, from the callees to the callers, summing the stack frame sizes and taking the maximal alignment. Each function must indeed take into account the stack used by the functions it calls, directly or transitively. Since Jasmin does not support

recursive functions, the call graph is acyclic and this propagation is not difficult. The result is an upper bound on the amount of stack used and an alignment for each function, taking function calls into account. It is an upper rather than an exact bound because, for instance, the function might call some other function conditionally with respect to a runtime value, so the analysis must be conservative. This bound is tight in the sense that there exists an execution that uses this amount of stack. The same reasoning with respect to conditional calls applies to the alignment.

When stack zeroization is applied, the analysis is amended. Three changes are made. First, if the size to be cleared at each step is greater than the alignment of the **export** function, the alignment is increased so that both match. This is needed so that the writes introduced by the zeroization are aligned. Second, the frame size of the **export** function is rounded up, so that it is a multiple of the alignment, meaning that some padding is inserted in the frame. To explain that point, we must give some details about the code generated by the Jasmin compiler for **export** functions. To allocate the frame of the **export** function on the stack, the stack pointer is first decreased by the size of the stack frame, then aligned on the alignment of the function. In the clearing code, we start by aligning the stack pointer. Roughly speaking, we need the decrease and the alignment to commute. One way to achieve that is having the frame size be a multiple of the alignment. Third, the bound on the stack size is rounded up, so that it is a multiple of the size of the clear step. This is to take into account that, as mentioned in Section 4.2, we can clear only a size that is a multiple of the size of the clear step.

Transformation. The compiler injects zeroization code at the end of **export** functions. At that point, the stack pointer is already restored to its initial value. For that reason, the zeroization code has to redo the alignment performed at the entry of **export** functions. It copies the stack pointer to another register and computes the alignment. Then it introduces a loop or a sequential code, depending whether the strategy is loop or unrolled.

Interestingly, some modification to the compiler was required for implementing the transformation. In particular, the formal development was modified to support compilation passes which introduce new labels in the Linear.

Illustrative example. Let us consider the schematic Jasmin program shown in Listing 2, on the left and let us first illustrate the analysis on it. Function **f** uses 1 byte of stack, but actually it also has to save the return address on the stack, which requires 8 bytes, so 9 bytes in total. The alignment is 64 bits, because of the 64-bit return address. For alignment reasons, the frame size of internal functions are rounded up to a multiple of the alignment, so the final frame size is 16 bytes. Function **main** uses 4 bytes of stack for itself, and calls **f**. This results in 20 bytes being used by **main**. As for the alignment, **main** inherits the alignment of **f**, 64 bits.

Let us assume that the user applies the stack zeroization feature with the default clear step size, 64 bits, corresponding to the alignment of **main**. The alignment of **main** is unchanged, but the frame size is rounded up and becomes 8 bytes. **f** still uses 16 bytes of stack. The stack size used by main is thus 24 bytes. This is already a multiple of the clear step size, 64 bits, so it does not need to be rounded up.

The resulting assembly for strategies loop and unrolled is shown in Figure 2, in the middle and on the right, respectively. We can observe that in both cases, the clearing code start with aligned the stack pointer, mimicking the initial alignment at the start of **main**. In the loop strategy, a counter is set up before entering the loop. The counter is decreasing, the loop exits when it reaches 0. Written this way, it is enough to check the flags set by `addq`, there is no need to call a comparison operator. In the unrolled case, the code consists in as many clearing instructions as needed. In this example, we need 3.

Listing 2 A schematic Jasmin program **(a)**, and the assembly produced with a clear step of 64 bits, for strategies loop **(b)** and unrolled **(c)**.

(a) Jasmin program

```
1  fn f () ⟶ reg u64 {
2    stack u8 s;
3    ...
4    return r;
5  }
6
7  export fn main () ⟶ reg u64 {
8    stack u32 s;
9    ...
10   r = f ();
11   return r;
12 }
```

(b) Code produced with loop strategy

```
main:
  movq %rsp, %rsi        // Save the SP.
  leaq -8(%rsp), %rsp    // Allocate a word.
  andq $-8, %rsp         // Align.
  ...
  ...                    // Code.
  ...
  movq %rsi, %rsp        // Restore SP.
  andq $-8, %rsp         // Align.
  subq $16, %rsp         // Point at max stack.
  movq $16, %rdi         // Set up counter.
zloop:
  subq $8, %rdi
  movq $0, (%rsp,%rdi)   // Zeroize.
  jne zloop
  movq %rsi, %rsp        // Restore SP.
  ret
```

(c) Code produced with unroll strategy

```
main:
  movq %rsp, %rsi        // Save the SP.
  leaq -8(%rsp), %rsp    // Allocate a word.
  andq $-8, %rsp         // Align.
  ...
  ...                    // Code.
  ...
  movq %rsi, %rsp        // Restore SP.
  andq $-8, %rsp         // Align.
  subq $16, %rsp         // Point at max stack.
  movq $0, 8(%rsp)       // Zeroize
  movq $0, (%rsp)        // Zeroize
  movq %rsi, %rsp        // Restore SP.
  ret
```

4.4 Register and flag zeroization

The compiler also offers to developers the possibility to zeroize registers, flags or XMM registers, or a combination of these. No analysis is needed in this case, since the targets of zeroization are known.

4.5 Correctness and security

We have proved that the Jasmin compiler with the zeroization pass achieves correctness and security. Both statements are stated as end-to-end results, i.e., as relations between source code and assembly code. The (simplified) statements of correctness and security are displayed in Figure 2.

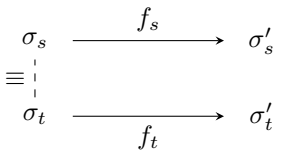
The correctness result is a classic simulation, stating that if an **export** function f_s from the source program transforms state σ_s into state σ'_s, and its compilation f_t in the compiled program transforms σ_t into σ'_t, and if σ_s is equivalent to σ_t, then σ'_s is equivalent to σ'_t, where $\equiv$ is a relation between source states and target states. End-to-end correctness of the compiler follows from correctness of each individual pass. The correctness statement of other passes could be reused.

The security result states that under the assumptions above, all memory locations in σ'_t whose addresses are undefined in σ_s either retain their value from σ_t or contain zeros (see Figure 2). The compiler only writes stack data to memory addresses that are invalid in σ_s, so it follows that everything an adversary learns about the stack from σ'_t was present in σ_t already. In fact, our security statement considers a more refined attacker model than previously discussed: when an **export** function returns, the attacker acquires the memory contents between sp and $sp - n$, together with registers and flags, where sp is the stack pointer before the function executed and n the amount of extra stack memory needed by the compiler. Our transformation ensures that all the memory given to the adversary is either valid and originates from the client code, or is equal to 0. Note that in this model the attacker learns the amount of used stack, but this is public since it's statically determined from the program.

The proof of security is done for the zeroization pass. Then, one proves that the security property is preserved by all subsequent passes. However, in order to establish our end-to-end statement, it has also been necessary to prove that passes before zeroization do not introduce arbitrary writes. For instance, we have proved that stack allocation respects stack usage predicted by the compiler, i.e. it does not write outside of the region of the stack that the compiler predicted to use. The zeroization property for the entire compiler is as follows: If an **export** function f_s from the source program transforms state σ_s into state σ'_s, and its compilation f_t in the compiled program transforms σ_t into σ'_t, and if σ_s is equivalent to σ_t, then memory locations in σ'_t whose addresses are undefined in σ_s either retain their value from σ_t or contain zeros.

4.6 Combining leakage models

As stated in the introduction, our threat model so far is limited to an attacker that can observe the contents of the registers and flags upon return of the cryptographic routine, as well as the contents of the stack used by the cryptographic routine. This threat model is orthogonal to threat models for micro-architectural attacks, including cache attacks, or speculative attacks like Spectre. In this section, we sketch how our countermeasure can soundly be combined with existing approaches to protect against this type of side-channel attacks. For completeness, we briefly review the prominent models for cache-based timing attacks, without and with speculative execution.



- Correctness: $\sigma'_s \equiv \sigma'_t$
- Security: $\neg\,\mathsf{valid}(\sigma_s, p) \implies \sigma'_t\,[p] = \sigma_t\,[p] \vee \sigma'_t\,[p] = 0$

where $\equiv$ is a relation between source and assembly states and $\neg\,\mathsf{valid}(\sigma_s, p)$ says that p is not a valid address w.r.t. source state σ_s.

Figure 2: Correctness and security of zeroization.

The constant-time policy. In absence of speculative execution, the baseline for side-channel protection is (cryptographic) constant-time. Informally, the constant-time policy considers a leakage model where program execution leaks all control-flow decisions and all addresses (not values) of memory accesses. The constant-time property states that an attacker cannot learn any secret information from leakage. Formally, the property is stated relative to a notion of indistinguishability between states, where two states are indistinguishable if they coincide on the fragment of the memory that can be accessed directly by the attacker. Then, we say that a program is constant-time if leakage cannot separate between any two executions starting from indistinguishable states.

We can now combine the two threat models, and consider an attacker that can observe both the constant-time leakage and the contents of the stack, registers and flags after program execution does not learn anything about secrets. For lack of a classic name, we call this model stack constant-time. It is not too hard to observe that zeroization transforms a constant-time program into a stack constant-time program. Informally, this is a consequence of the correctness of stack zeroing, and of the following two observations. First, the part of the stack that is zeroed by zeroization is determined statically and independently of the values held in memory. Second, zeroing the same part of the stack preserves state indistinguishability.

The speculative constant-time policy. The baseline for side-channel protection in presence of speculative execution is speculative constant-time. The leakage model is similar to the one of constant-time. However, the attacker can now actively influence all control-flow decisions, and the addresses of unsafe memory reads and writes carried during misspeculation. More formally, the operational semantics of programs is extended with a set of directives that are controlled by the attacker and determine the control-flow of the program. Then, a program is speculative constant-time if, for every choice of the attacker at control-flow points, leakage cannot separate between any two speculative executions starting from indistinguishable states.

Similar to the previous case, we can define a notion of stack speculative constant-time. However, in this case one cannot prove that our zeroization procedure using loops transforms a speculative constant-time program into a stack speculative constant-time program. This is because the transformation does not prevent early abort attacks, and so an attacker with control over the branch predictor can force the whole zeroization to be skipped [SBB+22]. To avoid this attack, we offer a compiler flag to add a fence at the end of the zeroization loop. Note that the attack does not work on the variant of zeroization using an unrolled loop. So, in both cases, we can prove that zeroization transforms a speculative constant-time program into a stack speculative constant-time program.

Integration with the Jasmin compiler. Note that in contrast with the correctness and security results of the previous section, the claims of this paragraph are not machine-checked in the proof assistant, and they are not limited to the zeroization pass. Formalizing the results in the Coq proof assistant would be possible with reasonable effort. However, there are some main obstacles to extend them to end-to-end results. We discuss these obstacles below.

In the case of constant-time, the end-to-end result would state that the Jasmin compiler with zeroization transforms a constant-time source program into a stack constant-time program. A prerequisite for proving this end-to-end result would be to prove that the Jasmin compiler preserves constant-time. Indeed, such a preservation result exists. Unfortunately, it has been proved for a previous version of the Jasmin compiler [BGLP21] and the proof has not yet been merged into the latest state in the `main` branch.

In the case of speculative constant-time, there is currently no formal guarantee that the compiler preserves speculative constant-time (for Spectre v1). This remains an exciting direction for future research.

5 Benchmarks and validation

This section evaluates the cost of zeroization. As a starting point, we use the artifact from [SBG+22], which protects cryptographic implementations from Libjade [For23] against Spectre v1. We produced the data in this section with a machine with Linux Debian 5.10.0-21-amd64, GCC 10.2.1, and equipped with an Intel Core i7-10700K (Comet Lake) with hyperthreading and TurboBoost disabled.

Table 1 reports the cycle counts for six cryptographic primitives and nine implementations. For implementations with variable-input-length, such as ChaCha20 and Poly1305, we include the measurements for 128, 1024, and 16384 bytes. Each reported value in Table 1 corresponds to a median of 10000 executions. Given that the overhead of zeroization from our approach is small (from an absolute perspective), we repeated the experiment a total of eleven times, and included in the table the median of these (which can be interpreted as an approximation of the median of 110000 executions).

The fourth column from Table 1, **Baseline**, corresponds to the results of our experiments without performing any zeroization. The fifth column, **Loop**, presents the cycle counts when we perform a complete stack zeroization using a loop with a fence instruction after it, to prevent early-abort speculative execution attacks (`-stack-zeroization loopSCT`). In addition to the stack cleaning, for reference implementations (ref in the table) we clear all 64-bit registers that might leak, and for vectorized implementations, we also set the 256-bit registers to zero. The overhead of the **Loop** setup (compared to the Baseline) is shown in the next column. The highest overhead in this column is 32.56%, corresponding to the vectorized version of Poly1305 when operating on 128 bytes of input data: the cost of zeroization is fixed, and for this particular case, the average expected overhead is just 56 CPU cycles. This cost is quickly amortized for larger inputs and converges to overheads close to zero, demonstrated by the 1.09% overhead for the same implementation and an input length of 16 KiB.

The CPU cycles reported in column **Unroll** correspond to a straight-line code zeroization (option `-stack-zeroization unrolled`). The register zeroing is performed in the same way as in **Loop**. Generally, the overhead of zeroing the stack using the straight-line code is lower when compared to zeroization using a loop. Both overhead columns show a negative value for the scalar multiplication of X25519, -0.02% and -0.04%, for the loop and unrolled variants, respectively. We suspect this result to be due to different code alignment resulting from zeroization and will continue to investigate the matter. The computation overhead for zeroing the state in all Kyber's operations is small and below 2% for all reported measurements except one, Kyber512 decapsulation. Kyber768 avx2

Table 1: Benchmark results on an Intel Core i7-10700K (Comet Lake) CPU

Primitive	**Impl.**	**Op.**	**Baseline**	**Loop**	**overh.** [%]	**Unroll**	**overh.** [%]
ChaCha20	avx2	128 B	372	458	23.12	398	6.99
	ref	128 B	796	840	5.53	810	1.76
	avx2	1 KiB	1254	1304	3.99	1256	0.16
	ref	1 KiB	5996	6038	0.70	6008	0.20
	avx2	16 KiB	19076	19104	0.15	19144	0.36
	ref	16 KiB	94618	94646	0.03	94660	0.04
Poly1305	avx2	128 B	172	228	32.56	184	6.98
	ref	128 B	172	212	23.26	176	2.33
	avx2	1 KiB	684	744	8.77	702	2.63
	ref	1 KiB	1044	1080	3.45	1052	0.77
	avx2	16 KiB	8422	8514	1.09	8468	0.55
	ref	16 KiB	15932	15964	0.20	15970	0.24
secretbox	avx2	128 B	1244	1342	7.88	1282	3.05
	ref	128 B	1702	1760	3.41	1710	0.47
	avx2	1 KiB	3110	3216	3.41	3158	1.54
	ref	1 KiB	8006	8052	0.57	8028	0.27
	avx2	16 KiB	31342	31434	0.29	31450	0.34
	ref	16 KiB	115402	115426	0.02	115496	0.08
X25519	mulx	smult	98334	98432	0.10	98304	-0.03
Kyber512	avx2	keypair	25884	26256	1.44	26282	1.54
	avx2	enc	35416	35860	1.25	36024	1.72
	avx2	dec	27984	28886	3.22	28470	1.74
Kyber768	avx2	keypair	43096	43402	0.71	43352	0.59
	avx2	enc	55134	55490	0.65	55268	0.24
	avx2	dec	44294	44938	1.45	44756	1.04

implementation uses 15392, 18432, and 19552 bytes of the stack in keypair, enc, and dec, respectively, and if we consider that cost of clearing the stack lies between 400 and 600 CPU cycles, on average, roughly 32 bytes of stack are cleared each CPU cycle.

In the context of code size, the assembly file produced by the Jasmin compiler for Kyber768 avx2 has 33761 lines for the **Baseline** version. For the **Loop** and **Unroll** variants, this increases to 33869 and 35519, respectively. In **Unroll**, one instruction is issued for every 32 stack bytes: in the particular case of dec, which uses 19552, this corresponds to 611 instructions.

Overall, Table 1 shows that the overhead is very small and in many cases barely measurable for both variants, so in scenarios where code size is a concern and stack space usage is intensive, using the **Loop** option is recommended; otherwise, the **Unroll** option performs better on average.

Validation. To validate the effectiveness of stack zeroization code, we call each function from the Libjade API in a C wrapper that reads the region of memory that the function used as stack. More specifically, we implement a test case for each library function, where we first fill the memory region that the function will use as stack with the output of a PRF. Then, we call the Libjade function and subsequently assert that the memory region we filled with the PRF was zeroized (in our current implementation this means overwritten with zeros). On the other hand, for register zeroization we inspected the assembly output of the compiler and ensured each register was being overwritten with zeros.

6 Discussion and future work

The solution to zeroization we present in this paper offers many desirable properties: it is principled in the sense that it is guaranteed *that* sensitive data on the stack and in registers is cleared and it is well defined *when* this happens. As we showed in Section 5, the cost for zeroization is typically very small. Integration into the Jasmin compiler releases library developers from the error-prone task of taking care of zeroization and ensures that zeroization takes place on the callee side, where the responsibility naturally resides.

There are also some drawbacks to our solution, most obviously that it requires writing cryptography in the Jasmin programming language. This may not be an option for various reasons, most notably that, so far, the Jasmin compiler only targets the AMD64 (and, experimentally, ARMv7-M) architectures. However, for projects that *can* integrate assembly code generated by the Jasmin compiler, switching to Jasmin implementations comes with additional benefits like certified compilation, timing-attack and Spectre-v1 protection [SBG+22], and an interface to EasyCrypt [BGHZ11] proofs of functional correctness and reductionist proofs of security. One obvious line of future work is to extend Jasmin's set of supported target architectures, including the zeroization support we presented here.

Furthermore, recall that our proposed approach erases *all* data and does so only when returning from an **export** function. One can imagine investigating more fine-grained approaches in two different senses: First, as Jasmin features a information-flow type system, one could decide to erase secret data only. While the information about secrecy is in principle available, it is not trivial to decide at the end of an **export** function for each address in the used stack space, if the last write contained secret data or not. Not only would this approach add considerable complexity and in most cases offer only small benefits, it would also create a performance conflict with Spectre v1 protection: For the selective SLH countermeasures implemented in [SBG+22], it is beneficial to declare data as "secret" whenever this data is not used as address or branch condition, even if it not secret from a cryptographic point of view. This approach of declaring as much data as possible as "secret" would, for most cryptographic routines, mean that anyway almost all stack data needs to be cleared. Distinguishing "truly secret" and "additional secret" data in the type system would probably be possible, but again add complexity in both the compiler and Jasmin implementations. Second, one could decide to implement more frequent zeroization, for example at the end of every (also Jasmin-internal) function or even whenever a variable goes out of scope. Again, given on our benchmarks of fast functions on short inputs, we expect such an approach to be quite expensive in terms of performance.

The Jasmin compiler is an ideal context for developing security-aware compilation techniques, for three reasons. First, Jasmin is primarily targeted to cryptographic software, which mandates the use of these transformations. Second, the Jasmin compiler is arguably simpler than a mainstream compiler, making the implementation and integration of these transformations much simpler. Last, the overarching goal of the Jasmin/EasyCrypt approach to high-assurance cryptography is to derive assembly-level guarantees against implementation adversaries. As noted by Percival, proving properties such as forward secrecy at assembly-level mandates the use of a formally verified zeroing compiler, so there is a very strong motivation to adopt and maintain zeroization in the Jasmin compiler. Nevertheless, we express our hope that eventually also mainstream compilers will offer a clean approach to memory zeroization, for example, by adopting the solution proposed in [SCA18] for LLVM.

Acknowledgements

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; the European Commission through the ERC Starting Grant 805031 (EPOQUE); 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+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, October 31 – November 2, 2017. ACM Press.

[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, May 18–21, 2020. IEEE Computer Society Press.

[ABC+21] Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Catalin Hritcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. An extended account of trace-relating compiler correctness and secure compilation. *ACM Trans. Program. Lang. Syst.*, 43(4):14:1–14:48, 2021.

[Bar23] Cesar Eduardo Barros. Clear On Drop Source Code, 2023. https://github.com/cesarb/clear_on_drop (accessed 2023-07-15).

[BDG+14] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A Tutorial. In Alessandro Aldini, Javier Lopez, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer International Publishing, 2014.

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006: 9th International Conference on Theory and Practice of Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228, New York, NY, USA, April 24–26, 2006. Springer, Heidelberg, Germany.

[BGHZ11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90, Santa Barbara, CA, USA, August 14–18, 2011. Springer, Heidelberg, Germany.

[BGLP21] 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, November 15–19, 2021. ACM Press.

[Cha17] Roderick Chapman. Sanitizing sensitive data: How to get it right (or at least less wrong...). In Johann Blieberger and Markus Bader, editors, *Reliable Software Technologies – Ada-Europe 2017*, volume 10300 of *Lecture Notes in Computer Science*, pages 37–52. Springer International Publishing, 2017.

[CM05] S. Chong and A.C. Myers. Language-based information erasure. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 241–254, 2005. https://people.seas.harvard.edu/~chong/pubs/csfw05_erasure.pdf.

[CPGR05] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In Patrick D. McDaniel, editor, *USENIX Security 2005: 14th USENIX Security Symposium*, Baltimore, MD, USA, July 31 – August 5, 2005. USENIX Association.

[CSJ+19] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. Fact: a DSL for timing-sensitive computation. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 174–189. ACM, 2019.

[CVE14] CVE-2014-0160. Available from MITRE, CVE-ID CVE-2014-0160., 2014. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160 (accessed 2023-07-15).

[Dal23] Dalek Authors. Dalek x25519 Source Code, 2023. https://github.com/dalek-cryptography/x25519-dalek/blob/2.0.0-rc.3/src/x25519.rs#L73 (accessed 2023-07-15).

[DBR22] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure. *ACM Transactions on Privacy and Security*, 26(2), 2022. https://binsec.github.io/assets/publications/papers/2022-tops.pdf.

[Don17] Jason A. Donenfeld. WireGuard: Next generation kernel network tunnel. In *ISOC Network and Distributed System Security Symposium – NDSS 2017*, San Diego, CA, USA, February 26 – March 1, 2017. The Internet Society.

[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.

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

[GCC23] GCC Authors. The GCC Documentation – Static Stack Usage Analysis, 2023. https://gcc.gnu.org/onlinedocs/gnat_ugn/Static-Stack-Usage-Analysis.html (accessed 2023-07-13).

[Gnu23] GnuTLS Authors. GnuTLS Source Code, 2023. https://github.com/gnutls/gnutls/blob/gnutls_3_6_12/lib/safe-memfuncs.c (accessed 2023-07-15).

[Gut16] Daniel Gutson. Proposal: Zero the local stack on function exit, 2016. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=69976 (accessed 2023-07-23).

[HS08] Sebastian Hunt and David Sands. Just forget it – the semantics and enforcement of information erasure. In Sophia Drossopoulou, editor, *Programming Languages and Systems*, volume 4960 of *Lecture Notes in Computer Science*, pages 239–253. Springer, 2008.

[Int17] International Organization for Standardization. The C17 Programming Language Standard, 2017. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2310.pdf.

[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, May 19–23, 2019. IEEE Computer Society Press.

[Lib23] Libsodium Authors. Libsodium Source Code, 2023. https://github.com/jedisct1/libsodium/blob/master/src/libsodium/sodium/utils.c#L125 (accessed 2023-07-15).

[Mbe23] MbedTLS Authors. MbedTLS Documentation, 2023. https://github.com/Mbed-TLS/mbedtls/blob/v3.3.0/library/platform_util.c#L65 (accessed 2023-07-15).

[NSS23] NSS Authors. NSS Source Code, 2023. https://hg.mozilla.org/projects/nss/file/b7888f994479307ea70bfbd5c2b1bb4cd6c36a55/lib/softoken/pkcs11c.c#l738 (accessed 2023-15-07).

[Ope23] OpenSSL Authors. OpenSSL Source Code, 2023. https://github.com/openssl/openssl/blob/0e9725bcb90770d967351b977407b174bbd91869/crypto/mem_clr.c (accessed 2023-07-15).

[oSTfCS20] National Institute of Standards, Technology, and Canadian Centre for Cyber Security. Implementation guidance for fips 140-3 and the cryptographic module validation program, 2020. last updated on March 17, 2023. https://csrc.nist.gov/CSRC/media/Projects/cryptographic-module-validation-program/documents/fips%20140-3/FIPS%20140-3%20IG.pdf (accessed 2023-07-15).

[PAC19] Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Comput. Surv.*, 51(6):125:1–125:36, 2019.

[Per] Trevor Perrin. Noise protocol framework. https://noiseprotocol.org/noise.pdf (Revision 34 vom 2018-07-11).

[Per14] Colin Percival. Zeroing buffers is insufficient. Post on on the Daemonic Dispatches blog, 2014. https://www.daemonology.net/blog/2014-09-06-zeroing-buffers-is-insufficient.html (accessed 2023-07-16).

[Por23] Thomas Pornin. BearSSL API Overview – Memory Wiping, 2023. https://www.bearssl.org/api1.html#memory-wiping (accessed 2023-07-13).

[Res18] Eric Rescorla. The transport layer security (TLS) protocol version 1.3. IETF RFC 8446, 2018. https://rfc-editor.org/rfc/rfc8446.txt.

[Rus23a] Rust Authors. Rust's emit-stack-sizes, 2023. https://doc.rust-lang.org/unstable-book/compiler-flags/emit-stack-sizes.html (accessed 2023-07-13).

[Rus23b] RustCrypto Project Developers. Zeroize Source Code, 2023. https://github.com/RustCrypto/utils/tree/master/zeroize (accessed 2023-07-15).

[SBB+22] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. Spectre declassified: Reading from the right place at the wrong time. Cryptology ePrint Archive, Report 2022/426, 2022. https://eprint.iacr.org/2022/426.

[SBG+22] 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. Cryptology ePrint Archive, Report 2022/1270, 2022. https://eprint.iacr.org/2022/1270.

[SCA18] Laurent Simon, David Chisnall, and Ross Anderson. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 1–15, 2018. https://www.cl.cam.ac.uk/~rja14/Papers/whatyouc.pdf.

[Smi23] Brian Smith. ring Source Code, 2023. https://github.com/briansmith/ring/commit/b76f52c03a667c2ac6793ff566b55ad060421759 (accessed 2023-07-15).

[Spr22] Amber Sprenkels. Eraser, 2022. https://github.com/dsprenkels/eraser (accessed 2023-07-15).

[Syn14] Synopsys Inc. The Heartbleed Bug, 2014. https://heartbleed.com/ (accessed 2023-15-07).

[Tin23] TinyDTLS Authors. TinyDTLS Source Code, 2023. https://github.com/eclipse/tinydtls/blob/004aba8f7a1f7b70eb1a43dfa9fc4be644daa4ca/crypto.c#L254 (accessed 2023-07-15).

[Wol23] WolfSSL Authors. WolfSSL Source Code, 2023. https://github.com/wolfSSL/wolfssl/blob/e2424e67444a360eab615b53fd5649ff355ad68b/wolfcrypt/src/misc.c#L349 (accessed 2023-07-15).

[XLD+23] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. Silent bugs matter: A study of compiler-introduced security bugs. In *Proceedings of USENIX Security Symposium, 2023*. USENIX, 2023.

[YJO+17] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. Dead store elimination (still) considered harmful. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017: 26th USENIX Security Symposium*, pages 1025–1040, Vancouver, BC, Canada, August 16–18, 2017. USENIX Association.