Security evaluation of *dalek-cryptography* libraries

The *subtle*, *curve25519-dalek* and *bulletproofs* libraries

Ref.	**19-06-594-REP**
Version	**1.2**
Date	**August 6th, 2019**
Made for	**The TARI LABS**
Conducted by	**Quarkslab**

Quarkslab
SECURING EVERY BIT OF YOUR DATA

Quarkslab SAS
13 rue Saint Ambroise
75011 Paris, France

Contents

1 Project Information **1**

2 Executive Summary **2**
- 2.1 Context . . . 2
- 2.2 Methodology . . . 2
- 2.3 Chronology . . . 2
- 2.4 Report synthesis . . . 3
  - 2.4.1 Synthesis . . . 3
  - 2.4.2 Issues and recommendations . . . 3

3 Code overview **5**
- 3.1 Audited versions . . . 5
- 3.2 Dependencies . . . 5
  - 3.2.1 The *subtle* library . . . 5
  - 3.2.2 The *curve25519-dalek* library . . . 5
  - 3.2.3 The *bulletproofs* library . . . 6

4 The *subtle* library **7**
- 4.1 Introduction . . . 7
- 4.2 Code review and constant time validation . . . 7
  - 4.2.1 Code review . . . 7
  - 4.2.2 Constant time validation . . . 9
  - 4.2.3 Inlining optimizations . . . 11
- 4.3 Conclusion . . . 12

5 The *curve25519-dalek* library **14**
- 5.1 Code Review . . . 14
  - 5.1.1 *u64* back-end . . . 14
  - 5.1.2 *avx2* back-end . . . 15
  - 5.1.3 Curve Points . . . 16
  - 5.1.4 Serialization . . . 16
  - 5.1.5 Constant Time . . . 16
- 5.2 Issues . . . 16
  - 5.2.1 Overflow in *Scalar52* . . . 16
  - 5.2.2 Optimizations . . . 17
- 5.3 Conclusion . . . 18

6 The *bulletproofs* library **19**
- 6.1 Purpose . . . 19
- 6.2 Protocol . . . 19
  - 6.2.1 Build a proof . . . 19
  - 6.2.2 Verify a proof . . . 30
- 6.3 Code review . . . 31
  - 6.3.1 State machine . . . 31
  - 6.3.2 Message serialization . . . 32
  - 6.3.3 Constant Time . . . 32
- 6.4 Issues . . . 33
  - 6.4.1 *ProofShare* panic . . . 33
  - 6.4.2 Optimizations and recommendations . . . 33

6.5 Conclusion 35

7 The *x25519-dalek* and *ed25519-dalek* libraries **36**
7.1 The *x25519-dalek* library 36
7.1.1 Dependencies 36
7.1.2 Public key validation 36
7.2 The *ed25519-dalek* library 37
7.2.1 Dependencies 37
7.2.2 Notes on the usage 38

8 Conclusion **39**

Bibliography **40**

1. Project Information

Document History			
Version	**Date**	**Details**	**Authors**
1.2	06/08/2019	Update the *Issues and recommandations* section	Laurent Grémy & Nicolas Surbayrole
1.1	05/08/2019	Integrate feedback from the authors of the *dalek-cryptography* libraries	Laurent Grémy & Nicolas Surbayrole
1.0	24/07/2019	First version	Laurent Grémy & Nicolas Surbayrole

Quarkslab		
Contact	**Position**	**E-mail address**
Frédéric Raynal	Quarkslab CEO	`fraynal@quarkslab.com`
Matthieu Duez	Service Manager	`mduez@quarkslab.com`
Laurent Grémy	R&D Engineer	`lgremy@quarkslab.com`
Guillaume Heilles	R&D Engineer	`gheilles@quarkslab.com`
Nicolas Surbayrole	R&D Engineer	`nsurbayrole@quarkslab.com`

Monero Tari Labs		
Contact	**Position**	**E-mail address**
Cayle Sharrock	Head of Engineering	`caylemeister@tari.com`
Riccardo Spagni	Co-Founder	`the_pony@tari.com`

2. Executive Summary

2.1 Context

The Tari Labs mandated Quarkslab in order to investigate some of the dalek libraries. The Tari Labs has a project that implements the Tari protocol which relies on some of these libraries and especially the use of cryptographic primitives. In addition, the use of Bulletproofs [Bul] and its implementation by the authors of the *dalek* libraries will allow them to enable efficient confidential transactions on the blockchain in a near future.

2.2 Methodology

The audit has been divided into three stages with a fourth optional one. The stages were the following:

- **Stage 1**: focused on getting into the code base, the documentation and other materials available to identify the dependencies between the different projects under *dalek-cryptography*[1] and on understanding the main points of interests for the next stages of the audit.
- **Stage 2**: focused on assessing the low-level cryptographic operations performed in the libraries, mainly in *subtle* and *curve25519-dalek*.
- **Stage 3**: focused on assessing the implementation of *bulletproofs* and provide feedback on how to use it in the context of Tari.
- **Stage 4**: focused on assessing the implementation of *x25519-dalek* and *ed25519-dalek*.

2.3 Chronology

The audit was performed by two security engineers for a total of 30 man-days between the 4th of June and the 28th of June. Some details about the chronology are provided below:

- April 23rd, 2019: quote sent.
- June 4th, 2019: beginning of the audit.
- June 11th, 2019: internal meeting.
- June 21st, 2019: internal meeting.
- June 28th, 2019: end of the audit.
- July 24th, 2019: report sent.
- July 29th, 2019: final meeting.

[1] https://github.com/dalek-cryptography

2.4 Report synthesis

2.4.1 Synthesis

Among the *dalek-cryptography* projects, five projects were of main interest to the Tari Labs. All these projects are implemented in Rust.

- https://github.com/dalek-cryptography/subtle: traits and utilities for constant-time cryptographic implementations (BSD 3-Clause license).
- https://github.com/dalek-cryptography/curve25519-dalek: implementation of group operations of Ristretto and on Curve25519 (BSD-3-Clause license).
- https://github.com/dalek-cryptography/bulletproofs: implementation of Bulletproofs using Ristretto (MIT license).
- https://github.com/dalek-cryptography/x25519-dalek: X25519 elliptic curve Diffie-Hellman key exchange using *curve25519-dalek* (BSD 3-Clause license).
- https://github.com/dalek-cryptography/ed25519-dalek: Ed25519 signing and verification (BSD 3-Clause license).

In coordination with the Tari Labs, the Quarkslab's audit mainly focuses on the first three projects and only reviews the last two projects in a birds eye view way. We summarize in Figure 2.1 the dependencies between the projects and, in red, the audited projects.

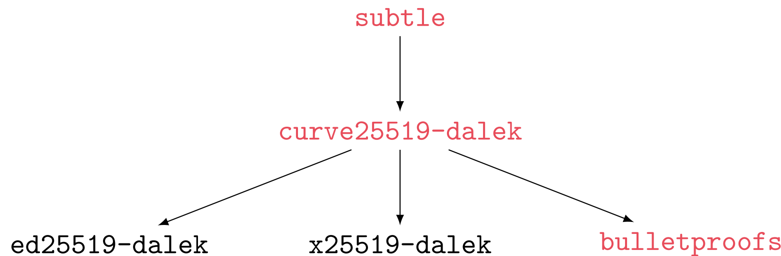

Fig. 2.1: Dependencies between the `dalek-cryptography` projects.

2.4.2 Issues and recommendations

The issue impact is set to *High* when a dependent library cannot fix the issue in an important feature of an audited library. The *Medium* impact is used to indicate important issues that can be mitigated by the dependent libraries. Others issues are marked as *Low*, *Option* or *Debug*.

The *subtle* library

ID	Vulnerability	Recommendation	Impact
`subtle-1`	New optimizations can be added in the Nightly version of Rust and can break the constant time property	Add some benchmarks and tests to verify the constant time property with a new Nightly Rust version	**Option**

The *curve25519-dalek* library

ID	Vulnerability	Recommendation	Impact
Curve-1	Overflow in *Scalar52* when importing a Scalar with the *from_bits* method	Assert in *Scalar52::to_bytes* that unused bits are null	**Medium**
Curve-2	The methods *multiscalar_mul* and *vartime_multiscalar_mul* on an *EdwardsPoint* will reject a customize iterator with imprecise but correct *size_hint*	Allow usage of a custom iterator and validate the size during the iteration on each iterator	**Option**
Curve-3	In *windows.rs*, an *assert_debug* method accepts erroneous values	Reduce the interval of accepted values	**Debug**

The *bulletproofs* library

ID	Vulnerability	Recommendation	Impact
BProof-1	A malicious party can send a crafted serialized message to the dealer that forces the dealer to panic	Do extra validation after deserializing *ProofShare*	**High**
BProof-2	The *size_hint* method of *AggregatedGensIter* iterator returns the original size of the iterator	Return the remaining size of the iterator	**Option**

The *x25519-dalek* library

ID	Vulnerability	Recommendation	Impact
X25519-1	Some public key may lead to compute the all-zero shared secret	Verify if a public key removes the contribution of the private key	**Option**

Optional dependency

ID	Vulnerability	Recommendation	Impact
Serde-1	The issue #151 of *rmp-serde* allows an attacker to allocate more than 2 GB of memory with a short serial. In a memory-constrained environment, this may slow down the system and lead to a crash of the application	Patch the library to verify the length before allocating memory, or perform small memory allocations	**Medium**
Serde-2	Custom implementation of *Serialize* and *Deserialize* do not work with all *serde* modules (e.g., *serde-json*)	List the *serde* modules that are compatible with the library or modify the implementation of the *Serialize* and *Deserialize* traits	**Option**

3. Code overview

3.1 Audited versions

We list below the audited versions of the five projects listed in Section 2.4.1:

- *subtle* version *2.1.0*, commit 34596d5701d7aa0bb88a76ec97cac8f51018988b ;
- *curve25519-dalek* version *1.2.1*, commit 45b316d26b64e8dc729b3463911a919cf31c6f4c;
- *bulletproofs* version *1.0.2*, commit 6a17ceb3bf3ce9b94cfa16a2a1a7311eef2dc6e7;
- *x25519-dalek* version *0.5.2*, commit 89c58a991bc958534bf28c646e1f0f88ce8333e6 and
- *ed25519-dalek* version *1.0.0-pre.1*, commit e527280d2e5781e070da269893cc514392312fdd.

We set the Rust version to nightly-2019-06-11. These libraries were compiled for the x86_64 architecture with avx2 support.

These versions were the last releases of each library at the beginning of the audit.

3.2 Dependencies

Note: Since the *x25519-dalek* and *ed25519-dalek* projects were not the main focus of the audit, we mention general information and we report on these projects in the appropriate section, i.e., Section 7.

In Figure 2.1, we draw a diagram that lists the dependencies between the different audited libraries. Note that in order to use the targeted version of the audit, we modified the file *Cargo.toml* of each library.

3.2.1 The *subtle* library

The *subtle* library does not rely on any external library other than the ones coming with a fresh Rust environment. The compilation has been done with the following command:

```
$ RUSTFLAGS="-C target_feature=+avx2" cargo build --release
```

3.2.2 The *curve25519-dalek* library

This library relies on[2]:

- *byteorder* version *^1.2.3*: Rust library for reading/writing numbers in big endian and little endian.
- *clear_on_drop* version *=0.2.3*: helpers for clearing sensitive data on the stack and heap.
- *digest* version *^0.8*: collection of cryptography-related traits.
- *rand_core* version *^0.3.0*: Rust library for random number generation.
- *subtle* version *^2*: forced to be version *2.1.0*.

[2] The version number follows the cargo caret requirements.

- *packed_simd ^0.3.0* (optional): portable packed SIMD Vectors for Rust standard library.
- *serde* version *^1.0* (optional): serialization framework for Rust.

Note that all these dependencies used 7 libraries as a back-end. We did not check if these projects have vulnerabilities or bugs that can affect the *curve25519-dalek* library, except obviously the *subtle* library.

The compilation has been done with the following command:

```
$ RUSTFLAGS="-C target_feature=+avx2" cargo build --release --no-default-features --
↪features "std avx2_backend"
```

3.2.3 The *bulletproofs* library

This library relies on:

- *byteorder* version *^1*: Rust library for reading/writing numbers in big endian and little endian.
- *clear_on_drop* version *^0.2*: helpers for clearing sensitive data on the stack and heap.
- *curve25519-dalek* version *^1.0.3*: forced to be version *1.2.1*.
- *digest* version *^0.8*: collection of cryptography-related traits.
- *failure* version *^0.1*: error management.
- *merlin* version *^1.1*: composable proof transcripts for public-coin arguments of knowledge.
- *rand* version *^0.6*: a Rust library for random number generation.
- *serde* version *^1*: serialization framework for Rust.
- *serde_derive* version *^1*: serialization framework for Rust.
- *sha3* version *^0.8*: SHA-3 (Keccak) hash function.
- *subtle* version *^2*: forced to be version *2.1.0*.

Note that all these dependencies used 29 libraries as a back-end. We did not check if these projects have vulnerabilities or bugs that can affect the *bulletproofs* library, except obviously the *subtle* and *curve25519-dalek* libraries. We, however, took a quick look at *merlin*, since it is a main component of the Bulletproofs implementation.

Note: To obtain the number of external librairies, we use the command `cargo tree --no-dev-dependencies`[1].

The compilation has been done with the following command:

```
$ RUSTFLAGS="-C target_feature=+avx2" cargo build --release --features "avx2_backend"
```

[1] https://github.com/sfackler/cargo-tree

4. The *subtle* library

4.1 Introduction

The *subtle* library contains several constant time arithmetical primitives, i.e., primitives for which computation time may depend on the types but never on the actual values or data being used.

The library documentation also includes the following warning:

```
This code is a low-level library, intended for specific use cases implementing
cryptographic protocols. It represents a best-effort attempt to protect against some
software side channels. Because side-channel resistance is not a property of software
alone, but of software together with hardware, any such effort is fundamentally
limited.

USE AT YOUR OWN RISK
```

The constant time property is thus only a best effort. We understand this property as follows:

- executed instructions do not depend on a secret (the values of a secret are not tested with a condition).
- a secret does not determine an address or an offset that will be used to retrieve data

This audit verifies that the code of the library and generated assembly with the frozen version of Rust do not have any trivial time leak. The audit does not validate that the library is immune to time leaks on a specific piece of hardware or outside the scope of the library.

4.2 Code review and constant time validation

The first part of the audit of the *subtle* code was to read and understand the code.

As the project does not include any benchmark to assess the constant time features, the second part was dedicated to test them.

4.2.1 Code review

The library defines a Boolean type *Choice* to implement constant time Boolean arithmetic and some trivial conditional operations:

- *EQ* on two integers.
- *EQ* on two arrays of integers with the same length.
- *OR* on two *Choice*.
- *AND* on two *Choice*.
- *XOR* on two *Choice*.
- *NOT* on a *Choice*.
- conditional selection and assignment of an integer depending on a *Choice*.
- conditional negation of an integer depending on a *Choice*.

The library also defines a *CtOption* type that attempts to represent the Rust Option type with

constant time and constant memory operations. This type uses a *Choice* to define if the value is accessible.

To implement this library with no variable time operation, the library has no dependency and limits the usage of *if* statements. The only present *if* statement is to assert that two arrays have the same length:

```
impl<T: ConstantTimeEq> ConstantTimeEq for [T] {
    #[inline]
    fn ct_eq(&self, _rhs: &[T]) -> Choice {
        let len = self.len();

        if len != _rhs.len() {
            return Choice::from(0);
        }
        ...
    }
}
```

In addition, all operations between two Boolean values don't use lazy boolean operators that skip the evaluation of the right operand if the result is already determined by the left operand.

Finally, the library tries to prevent compiler optimizations by inserting a *black_box*. The goal is to force the compiler to use *Choice* as an integer and to avoid the generation of Boolean-specific assembly instruction when using *Choice*.

```
/// This function is a best-effort attempt to prevent the compiler
/// from knowing anything about the value of the returned `u8`, other
/// than its type.
///
/// Uses inline asm when available, otherwise it's a no-op.
#[cfg(all(feature = "nightly", not(any(target_arch = "asmjs", target_arch = "wasm32
↪")))))]
fn black_box(input: u8) -> u8 {
    debug_assert!((input == 0u8) | (input == 1u8));

    // Pretend to access a register containing the input.  We "volatile" here
    // because some optimisers treat assembly templates without output operands
    // as "volatile" while others do not.
    unsafe { asm!("" :: "r"(&input) :: "volatile") }

    input
}
#[cfg(any(target_arch = "asmjs", target_arch = "wasm32", not(feature = "nightly")))]
#[inline(never)]
fn black_box(input: u8) -> u8 {
    debug_assert!((input == 0u8) | (input == 1u8));
    // We don't have access to inline assembly or test::black_box or ...
    //
    // Bailing out, hopefully the compiler doesn't use the fact that `input` is 0 or␣
↪1.
    input
}
```

However, the usage of this library must follow some guidelines to preserve the constant time features:

- The constant time property is only valid for values of the same type. The comparison

between two *u8* may have a different duration than the comparison between two *u128*.

- During the comparison of two slices with the same length, the comparison duration depends on the length. The length should not be a secret.
- When using the methods *map* and *and_then* of *CtOption*, the lambda function given in parameters must be executed in constant time.
- When using the *ConditionallyNegatable* trait, the *Neg* trait must be implemented in constant time.

An internal crate is present in the folder *fuzz* but doesn't work with the current version of *subtle* (the trait *subtle::ConditionallyAssignable* does not exist in the current version of *subtle*). The code can be easily adapted and used to fuzz the method *conditional_assign* with different integer sizes.

4.2.2 Constant time validation

The project contains no benchmark to validate the constant time property. In addition, the library recommends using the *nightly* version of Rust that has a release once a day with new improvements or features. The generated bytecode may change between two nightly versions with new optimizations that may break the constant time feature.

The issue #43 on Github tests the constant time functionality of some methods of *subtle* with *dudect*.

During the audit, we implemented our own tests to check if the compiled code is in constant time. The following code has been used to compute the average and the standard deviation of a method's duration:

```
#![feature(asm)]

use subtle::*;

#[inline]
fn gettime() -> u64 {
    let lo : u64;
    let hi : u64;
    unsafe {
        asm!("lfence;rdtsc;lfence" : "={eax}" (lo), "={edx}" (hi) : : "eax", "edx");
    }
    lo | (hi << 32)
}

trait Callable {
    fn call(&mut self);
}

fn time_test(desc : &str, foo: &mut dyn Callable, max: u64) {
    let n = 10000000;
    let mut v = 0;
    let mut variance = 0;
    for _ in 0..n {
        let mut time = max;
        // remove OS interrupt or big difference from the capture
        while time >= max {
```

(continues on next page)

(continued from previous page)

```
            time = gettime();
            foo.call();
            time = gettime() - time;
        }
        v += time;
        variance += time*time;
    }
    let average = (v as f64) / (n as f64);
    let deviation = (((variance as f64) / (n as f64)) - (average * average)).sqrt();

    println!("{} average: {}", desc, average);
    println!("{} standard deviation: {}", desc, deviation);
}

...
```

No constant deviation has been found between the call of a *subtle* method with two different parameters. Some little differences may occur but may be the consequence of OS interruptions, cache misses or the activity of another process during the test. This deviation was near zero and not constant between two executions. In addition, the previous code was also tested with the *mfence* memory barrier with a similar result.

As such approach is insufficient by itself and can only raise red flags in case of obvious time leaks, we analyzed the test binary to verify that the generated assembly code did not have any issue and checked for less obvious potential leaks.

For example, for the test of the *OR* operation between two *Choice*, we have the following generated code:

```
struct Vor {
    first: Choice,
    second: Choice,
    res: bool
}
impl Callable for Vor {
    fn call(&mut self) {
        self.res = bool::from(self.first | self.second);
    }
}
```

```
; <test_subtle::Vor as test_subtle::Callable>::call
push   rbx
mov    rbx,rdi
mov    al,BYTE PTR [rdi+0x1]
or     al,BYTE PTR [rdi]
movzx  edi,al
call   QWORD PTR [rip+0x3a88e] ; subtle::black_box
test   al,al
setne  BYTE PTR [rbx+0x2]
pop    rbx
ret

; subtle::black_box
sub    rsp,0x1
mov    BYTE PTR [rsp],dil
```

(continues on next page)

(continued from previous page)

```
mov     rax,rsp
mov     al,BYTE PTR [rsp]
add     rsp,0x1
ret
```

The Rust compiler does not optimize the code to a non-constant time assembly. However, the *black_box* method is still present and adds a constant overhead. The only method containing non-linear assembly code is the on which compares two arrays, with a loop iteration depending on the lengths of the arrays.

4.2.3 Inlining optimizations

Many methods of *subtle* are tagged with *#[inline]*. The assembly code of these methods will not be present on the compiled *subtle* library but in the dependent libraries when they need to use these methods. As a consequence, some optimizations that were not covered by the previous test can appear.

That was the case with the *curve25519-dalek* library and the *EdwardPoints* type. The code can be simplified as follows when using the *avx2_backend* feature:

```
// backend/serial/u64/field.rs

#[derive(Copy, Clone)]
pub struct FieldElement51(pub (crate) [u64; 5]);

impl ConditionallySelectable for FieldElement51 {
  fn conditional_select(
      a: &FieldElement51,
      b: &FieldElement51,
      choice: Choice,
  ) -> FieldElement51 {
      FieldElement51([
          u64::conditional_select(&a.0[0], &b.0[0], choice),
          u64::conditional_select(&a.0[1], &b.0[1], choice),
          u64::conditional_select(&a.0[2], &b.0[2], choice),
          u64::conditional_select(&a.0[3], &b.0[3], choice),
          u64::conditional_select(&a.0[4], &b.0[4], choice),
      ])
  }
  ...
}

// edwards.rs

#[derive(Copy, Clone)]
pub struct EdwardsPoint {
    pub(crate) X: FieldElement51,
    pub(crate) Y: FieldElement51,
    pub(crate) Z: FieldElement51,
    pub(crate) T: FieldElement51,
}

impl ConditionallySelectable for EdwardsPoint {
    fn conditional_select(a: &EdwardsPoint, b: &EdwardsPoint, choice: Choice) ->
↪EdwardsPoint {
```

(continues on next page)

(continued from previous page)

```
        EdwardsPoint {
            X: FieldElement51::conditional_select(&a.X, &b.X, choice),
            Y: FieldElement51::conditional_select(&a.Y, &b.Y, choice),
            Z: FieldElement51::conditional_select(&a.Z, &b.Z, choice),
            T: FieldElement51::conditional_select(&a.T, &b.T, choice),
        }
    }
}
```

The generated assembly code uses *SIMD* and does not use the *xor* instruction but creates an equivalent logical operation.

```
; <<curve25519_dalek::edwards::EdwardsPoint as subtle::ConditionallySelectable>
↪::conditional_select>:
mov     rax,rdi
movzx   ecx,cl
neg     rcx
vmovq   xmm0,rcx
vpbroadcastq ymm0,xmm0
vpandn ymm1,ymm0,YMMWORD PTR [rsi]
vpand   ymm2,ymm0,YMMWORD PTR [rdx]
vpandn ymm3,ymm0,YMMWORD PTR [rsi+0x20]
vpor    ymm1,ymm2,ymm1
vpand   ymm2,ymm0,YMMWORD PTR [rdx+0x20]
vpor    ymm2,ymm2,ymm3
vpandn ymm3,ymm0,YMMWORD PTR [rsi+0x40]
vpand   ymm4,ymm0,YMMWORD PTR [rdx+0x40]
vpandn ymm5,ymm0,YMMWORD PTR [rsi+0x60]
vpand   ymm6,ymm0,YMMWORD PTR [rdx+0x60]
vpor    ymm3,ymm4,ymm3
vpor    ymm4,ymm6,ymm5
vpandn ymm5,ymm0,YMMWORD PTR [rsi+0x80]
vpand   ymm0,ymm0,YMMWORD PTR [rdx+0x80]
vpor    ymm0,ymm0,ymm5
vmovdqu YMMWORD PTR [rdi],ymm1
vmovdqu YMMWORD PTR [rdi+0x20],ymm2
vmovdqu YMMWORD PTR [rdi+0x40],ymm3
vmovdqu YMMWORD PTR [rdi+0x60],ymm4
vmovdqu YMMWORD PTR [rdi+0x80],ymm0
vzeroupper
ret
```

Since the compiler does not determine that *Choice* can only get the value *0* or *1*, the generated code is still valid for any *u8*. Here, the inlined generated code is faster than a version with multiple inlined calls, without breaking the linear structure. However, we cannot be sure that the inlined code will always be linear and will keep the constant time property.

4.3 Conclusion

No issue was found in the *subtle* library. However, the constant time property for the *map* and *and_then* methods depends on the lambda method given as parameter. The usage of this library must follow some guidelines to preserve the constant time property in the dependent libraries and binaries.

During the audit, the library was compiled with a frozen version of nightly Rust (see Section 3.1). We validated that the generated code was linear or with branches and conditions that do not depend on sensitive value. However, a lot of methods can be inlined in the dependent libraries or binaries. The inlined analyzed code was much optimized without breaking the linearity. We could not verify that the inlined code will always keep this behavior in any context.

The lack of benchmarks in this library may lead to misunderstand the meaning of the *best effort* constant time given by *subtle*. We recommend implementing some benchmarks about this library to easily validate the properties on a given platform and to verify that new optimizations on nightly Rust do not break the property.

5. The *curve25519-dalek* library

The *curve25519-dalek* library allows performing group operations on the Curve25519 [Ber] using:

- the Montgomery form and
- the (twisted) Edwards form.

The Curve25519, an Edwards curve, is selected for many reasons: one of which is the efficiency of the operations needed to perform usual cryptographic protocols using group operations, such as Diffie-Hellman key exchange and signature algorithm. However, the order of the group of points on this curve is not prime[1], which may lead to deal with small order points. These points may lead to compute predictable values, see Section 7.1.2, signature malleability and other issues[2].

A possible mitigation of these issues is to perform checks at different levels when a protocol is run, see for example Section 8 of [EdDSA] or Section 7 of [ECS]. However, these checks are not always easy to perform inside more complex protocols, not always implemented or impossible to use. Based on [Ham] which deals with a prime-order group which uses Curve448 internally, the authors of Ristretto, see [Ris] and [TRG], propose to define a prime-order group whose group elements are represented through points on the Curve25519 using the Edwards form.

Since the API exposed in *curve25519-dalek* and used in *bulletproofs* only uses the Ristretto operations, and then the Edwards form, we have mainly audited parts of these implementations that are modules:

- Edwards.
- Ristretto.
- Scalar.

5.1 Code Review

The Rust language is designed to avoid misuses of pointers and buffer overflows. However, the integer overflow or underflow is not validated on each operation to avoid a substantial overhead. As the library *curve25519-dalek* manipulates big integers, this issue can have an important impact on the correctness of the operations.

The *curve25519-dalek* library uses different back-ends in order to reduce the computation time without breaking the correctness. The audit was focused on the *u64* back-end and the *avx2* extension.

5.1.1 *u64* back-end

The *u64* back-end is located in src/backend/serial/u64. It defines two structures:

- *FieldElement51* with arithmetic operations modulo $2^{255} - 19$;
- *Scalar52* with arithmetic modulo $L = 2^{252} + 27742317777372353535851937790883648493$.

[1] It is equal to $8 \cdot L = 8 \cdot (2^{252} + 27742317777372353535851937790883648493)$.

[2] For example, https://web.getmonero.org/2017/05/17/disclosure-of-a-major-bug-in-cryptonote-based-currencies.html.

Some standard constants for elliptic curves are also defined. The two structures decompose integers of 256 bits in 5 integers stored in *u64* variables.

FieldElement51

In *FieldElement51*, the initial integer is split in 5 integers of 51 bits (The Most Significant Bit of the big integers is always 0). The library attempts to decrease the latency of every operation by allowing a non-canonical representation of integers in a given limit b, where $x < 2^{51+b}$, x representing a 51 bit integer in the canonical representation.

The accepted limit of b for each method is mentioned as comments in the code and allows us to validate its correctness. The subtraction does not overflow if $b < 4$ and the multiplication with $b < 3.365$. Both of these operations call the *reduce* operation at the end to return a result where $b < 0.000000001$.

However, the addition does not include any reduction operation. This allows the values to overflow if additions are chained. The *FieldElement51* structure is not public but is used in *EdwardsPoint*. We did not find during the audit a suitable operation on a *EdwardsPoint* that can lead to an overflow of a *FieldElement51*.

Scalar52

For *Scalar52*, the integer is split in 4 integers of 52 bits and one of 48 bits. Some arithmetical operations on *Scalar52* are only valid with a canonical integer representation modulo L. To avoid underflows in the result, the subtraction adds L to the result if it's negative to prevent underflow. The addition performs a subtraction by L on the result to keep the sum of two canonical integers in a canonical form. For these additions and subtractions, if a number in a non-canonical form is passed, the result may overflow the integer representation (4 integers of 52 bits and 1 of 48 bits) and the method *to_bytes* will fail to reconstruct a valid number.

Some multiplications and reductions are present in this object. The code review performed on these methods has revealed no issue.

5.1.2 *avx2* back-end

The *avx2* back-end is used to enable the support of *SIMD* instructions. This back-end will be used to compute multi-scalar multiplications.

The *avx2* back-end is used to create a *FieldElement2625x4*. It is built with four *FieldElement51* coordinates that are present in an *EdwardsPoint*. The initial value is split into 40 integers of 32 bits, each with 25 or 26 bits of the value. So, the value can be stored in a non-canonical form but limited by b. All operations have the pre- and post-conditions on b that are well documented to limit overflow. The back-end also defines two structures *ExtendedPoint* and *CachedPoint* that uses *FieldElement2625x4*.

During the audit, we reviewed the conditions on b and validated that overflow may occur if these conditions were not fulfilled. We validated that all methods of *ExtendedPoint* and *CachedPoint* correctly validated pre- and post-conditions of *FieldElement2625x4*. For *ExtendedPoint*, the code will always keep $b < 0.007$ outside of the method. For *CachedPoint*, the post-conditions indicate that the maximum value of b is 1. The *CachedPoint* includes a warning about repeatedly negating a point. The precondition of the negation is not valid for the second negation but the actual implementation returns the same representation after two negations (the test case *test_multiple_neg* confirms this behavior).

5.1.3 Curve Points

The library uses the previous back-end in the structures *EdwardsPoint* and *RistrettoPoint* that represent respectively a valid point on the curve in the Edwards form and a point in the Ristretto group. *CompressedEdwardsY* and *CompressedRistretto* are the compressed representations of these points in a slice of 32 bytes. The *decompress* and *compress* methods allow switching between the two representations. The *decompress* method returns an *Option* result that can be *None* if the point is not valid. For the *RistrettoPoint*, this is in accordance with the specification of the decoding algorithm of Ristretto.

The *RistrettoPoint* uses *EdwardsPoint* that uses the *FieldElement51* structure to represent the coordinates of the point. In the case of a multi-scalar multiplication, the avx2 back-end is used.

5.1.4 Serialization

The previous curve points and the *Scalar* type implement the *Serialize* and *Deserialize* traits of *serde* to provide a way to serialize the structures. For *CompressedEdwardsY* and *CompressedRistretto*, the serialization uses 32 bytes for a point. To avoid creating a non-valid point during the decompression, the representation of *EdwardsPoint* and *RistrettoPoint* are the same as their compressed forms. The *decompress* method is called to deserialize and returns an error if the serialization is not a valid point.

For *Scalar*, the decompression validates that the number is in a canonical form. The overflow in the *Scalar52* cannot occur when deserializing a value.

To test the implementation, we tried to use *serde-json* but it fails to deserialize a valid serialization. The expected format to deserialize is a string when the serializer gives an array of integers. The Tari Labs informed us that they use *rmp-serde*. We fuzzed the implementation with honggfuzz-rs and did not find any issue in the library. However, if *rmp-serde* is used, we recommend patching the issue #151 to avoid that a malformed serial of five bytes allocates more than 2 GB of memory.

5.1.5 Constant Time

The library has some methods in variable time. These methods can be easily identified by their name that almost always begins with *vartime*. The code is linear (and may be expected to be constant time) in other methods, especially in the implementation of arithmetical operations. However, even without the *#[inline]* keyword, some methods are inlined inside the crate and we did not validate that all the generated assembly is still linear.

5.2 Issues

The following issues were found in *curve25519-dalek*. The only issue that adds a real risk is the overflow in *Scalar52*.

5.2.1 Overflow in *Scalar52*

The *from_bits* method in *Scalar* allows creating a *Scalar52* in a non-canonical form. However, the additions and subtractions in *Scalar52* are only valid with a canonical form. The overflow can be triggered by:

- creating a *Scalar* with *from_bits* that was greater than L and

- calling some addition, subtraction or negation.

During the addition, the value will be unpacked to a *Scalar52*. The overflow will occur in the largest *u64* when its value becomes larger than 2^{48}. This overflow is not detected by Rust because the value does not go around the *u64* limit. The *Scalar::pack* method and its call to *Scalar52::to_bytes* will not verify that the 16 highest bits of the most significant *u64* are all null.

During the first part of the subtraction, the *wrapping_sub* method is used to allow the underflow behavior. If the highest bit is set at the end, the method adds L to the result to return a positive value. However, if the value is lower than -L, the result will be still negative and the *Scalar52::to_bytes* will fail to detect the invalid state.

The following test triggers this issue with the addition and the negation:

```
#[test]
fn test_scalar_overflow() {
    let tested =  [
            0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
            0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
            0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
            0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x7f,
        ];
    let a = Scalar::from_bytes_mod_order(tested);
    let b = Scalar::from_bits(tested);

    assert_eq!(a, b.reduce());

    let a_res = a + a + a;
    let b_res = b + b + b;

    assert_eq!(a_res.reduce(), b_res.reduce()); // fail

    let neg_a = -a;
    let neg_b = -b;

    assert_eq!(neg_a.reduce(), neg_b.reduce()); // also fail
}
```

As the *from_bits* method is mandatory to implement *x25519*, we recommend adding a few checks to validate that the *Scalar* created with this method cannot be added or subtracted without being reduced. An assert can be added in *Scalar52::to_bytes* to verify that the 16 highest bits of the most significant *u64* are null.

5.2.2 Optimizations

The following issues may be corrected as improvements. However, their correction is not required for a secure usage of the library.

size_hint in *multiscalar_mul* and *vartime_multiscalar_mul*

The methods *EdwardsPoint::multiscalar_mul* and *EdwardsPoint::vartime_multiscalar_mul* use size_hint to determine the length of their parameters. The documentation of the *size_hint* method allows returning a range of values. However, the previous multiplication methods verify that the result is a range with a single value. As the standard library returns the exact size, we

recommend specifying in the documentation that the *size_hint* method should return the exact size if a custom iterator is used. The size can also be validated in the *Strauss* and *Pippenger* algorithms, after the first *map* and *collect* calls on each iterator.

debug_assert too permissive in *window.rs*

In the file *window.rs*, the following code is used to access an index of *NafLookupTable8*:

```
#[derive(Copy, Clone)]
pub(crate) struct NafLookupTable8<T>(pub(crate) [T; 64]);

impl<T: Copy> NafLookupTable8<T> {
    pub fn select(&self, x: usize) -> T {
        debug_assert_eq!(x & 1, 1);
        debug_assert!(x < 256); // need to be < 128

        self.0[x / 2]
    }
}
```

The debug assert is too permissive and accepts values between 129 and 256 that are invalid offsets. As Rust validates the index, the code will panic in debug or release mode.

5.3 Conclusion

The internal representation of integers can allow some overflows that will not be detected by Rust in debug mode. During the audit, we found only one method that triggered an overflow through the public API of the library. The library follows the guideline of *subtle*. Moreover, except for methods marked as *variable time*, the code is linear and may be expected to be constant-time, but we did not write specific tests to check this behaviour during this audit. The point and scalar are always serialized in a compressed form, and the deserialization process properly validates points and canonical forms when using scalars.

6. The *bulletproofs* library

6.1 Purpose

Bulletproofs refer to non-interactive zero-knowledge (NIZK) proof protocol [Bul]. A bulletproof is mainly used to allow a Prover to convince a Verifier that a given secret value lies in a given range. In addition to an individual proof for each committed value, the protocol is also designed to deal with aggregated rangeproofs, which are smaller than individual proofs concatenated, and multi-party computation (MPC) to aggregate proofs of multiple parties.

The MPC feature of Bulletproofs is one of the main interests in the context of the Tari Labs.

6.2 Protocol

Note: The documentation of the bulletproofs library provides extensive notes about the protocol and the computations performed during the protocol. Instead of rewriting all the work in this report, we will use the notation introduced in the documentation and refer to the corresponding parts of the documentation.

6.2.1 Build a proof

This part refers to the "Party and Dealer's algorithm" section of https://doc-internal.dalek.rs/bulletproofs/range_proof/index.html. When the source of a function is not documented here, functions referring to a party can be found in src/range_proof/party.rs, and src/range_proof/dealer.rs for the dealer. Note that the pieces of code have sometimes been modified for the needs of the presentation and are placed under the license MIT License Copyright (c) 2018 Chain, Inc.

Setup

The MPC protocol specifies how the parties involved in the construction of the proof must interact with each other. First of all, the protocol describes a dealer (i.e., the Prover) which will be responsible for building the aggregated proof from the inputs of the parties[1]. Note that the security assumption of such an MPC protocol is secure against an *honest-but-curious adversary*, also known as passive corruption security or semi-honest security. Such an adversary can be described as not deviating from the protocol and attempting to obtain as much information as he can from the transcript of the proof (that is all the data that a Verifier will need to be convinced of the proof). Since such an assumption is often not verified in practice, the developers of *bulletproofs* have integrated some mitigations against attacks coming from a malicious adversary. We will discuss some of them during the description of the protocol. We consider a model in which the communications between the dealer and a party cannot be crafted by another party, which means that the communications are protected to ensure message integrity: confidentiality is less important, since a cheating dealer would have all the contributions of the parties and must not be able to recover the secrets. In addition, we consider that the dealer cannot require more data from a party without restarting the protocol from scratch.

[1] The dealer may be one of the parties.

The Bulletproofs paper states in Section 4.5 that two MPC protocols are available. The developers of the *bulletproofs* library chose to implement one of them: the one with a constant number of rounds, independent of the number of parties. Note that the number m of parties must be a power of 2 in the implementation. However the authors of the paper indicate that "the protocol could be easily adapted for other values of m". To begin the protocol, each participant (i.e., the parties and the dealer) must agree on:

- the number of participants m and
- the range $[0, 2^n)$ in which the secret values must lie, where n must be equal to one the following values: 8, 16, 32 or 64.

In addition, each party has a secret value, which will be named $v_{(j)}$, and a secret random scalar, which will be named $\tilde{v}_{(j)}$, a blinding value. With such setup, the protocol can begin, following the scheme summarized in Fig. 6.1. The random scalar is chosen using any cryptographically secure pseudorandom number generator, which must implement the functions of the traits *rand_core::CryptoRng* and *rand_core::RngCore*.

```
let mut rng = rand::thread_rng(); let v0_blinding = Scalar::random(&mut rng);
```

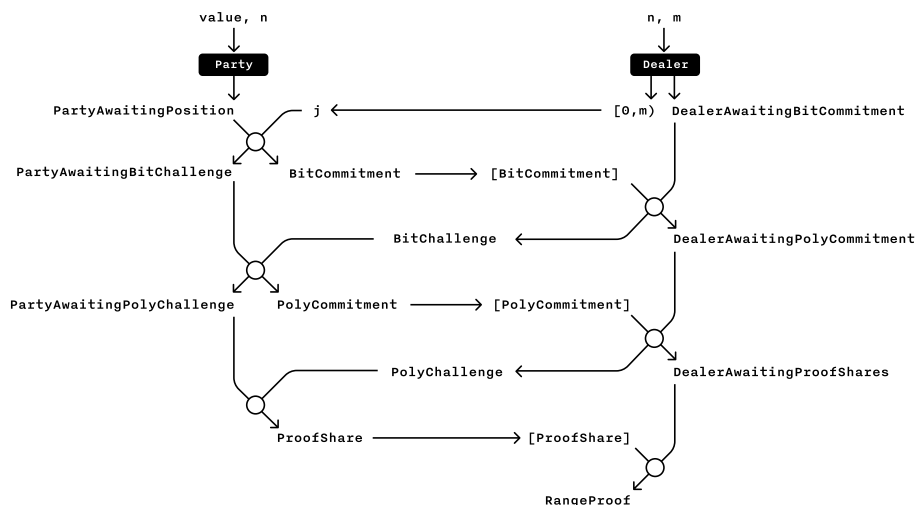

Fig. 6.1: Exchanges between the dealer and a party (Copyright (c) 2018 Chain, Inc., https://doc-internal.dalek.rs/bulletproofs/range_proof_mpc/index.html).

An example of code to run the protocol on a single machine with a fake exchange can be found in the detect_dishonest_party_during_aggregation function of the file src/range_proof/mod.rs. We quote it here with fewer parties than in the original test and without the dishonesty feature.

```
// Setup
let m = 2; let n = 32;

let pc_gens = PedersenGens::default(); let bp_gens = BulletproofGens::new(n, m);

let mut rng = rand::thread_rng();
let mut transcript = Transcript::new(b"AggregatedRangeProofTest");

let v0 = rng.gen::<u32>() as u64; let v0_blinding = Scalar::random(&mut rng);
```

(continues on next page)

(continued from previous page)

```
let party0 = Party::new(&bp_gens, &pc_gens, v0, v0_blinding, n).unwrap();

let v1 = rng.gen::<u32>() as u64; let v1_blinding = Scalar::random(&mut rng);
let party1 = Party::new(&bp_gens, &pc_gens, v1, v1_blinding, n).unwrap();

let dealer = dealer::new(&bp_gens, &pc_gens, &mut transcript, n, m).unwrap();

// Position
let (party0, bit_com0) = party0.assign_position(0).unwrap();
let (party1, bit_com1) = party1.assign_position(1).unwrap();

// Generation of the y and z challenges
let bit_commitments = vec![bit_com0, bit_com1];
let (dealer, bit_challenge) = dealer
    .receive_bit_commitments(bit_commitments).unwrap();

// Evaluation with the y and z challenges
let (party0, poly_com0) = party0.apply_challenge(&bit_challenge);
let (party1, poly_com1) = party1.apply_challenge(&bit_challenge);

// Generation of the x challenge
let poly_commitments = vec![poly_com0, poly_com1];
let (dealer, poly_challenge) = dealer
    .receive_poly_commitments(poly_commitments).unwrap();

// Evaluation with the x challenge
let share0 = party0.apply_challenge(&poly_challenge).unwrap();
let share1 = party1.apply_challenge(&poly_challenge).unwrap();

// Aggregation of the proofs
dealer.receive_shares(&[share0, share1]);
```

Each party and the dealer create two generators B and $\widetilde{B}$. These two generators must be generated in the same way for the parties and the dealer, it is sufficient that the discrete logarithm of $\widetilde{B}$ in basis B is unknown. Since Ristretto is a group of prime order, each element of the group is a generator of the group (or is zero), as defined in src/generators.rs.

Note: Using the default paramaters, these generators are always the same for each Bulletproofs protocol instantiation. The generator $\widetilde{B}$ is generated in a *nothing-up-my-sleeve* way using the SHA3-512 function of B expressed in bytes and converted into a Ristretto point, as shown in the following piece of code. However, since these two values are public, they can be set to any other values by using the standard API: in such a case, we recommand to carefully select B and $\widetilde{B}$ with respect to the properties listed previously.

```
#[derive(Copy, Clone)]
pub struct PedersenGens {
    /// Base for the committed value
    pub B: RistrettoPoint,
    /// Base for the blinding factor
    pub B_blinding: RistrettoPoint,
}

impl Default for PedersenGens {
```

(continues on next page)

(continued from previous page)

```
    fn default() -> Self {
        PedersenGens {
            B: RISTRETTO_BASEPOINT_POINT,
            B_blinding: RistrettoPoint::hash_from_bytes::<Sha3_512>(
                RISTRETTO_BASEPOINT_COMPRESSED.as_bytes(),
            ),
        }
    }
}
```

The $n \cdot m$ Bulletproofs generators are sampled in a reproducible way, in order for the parties to generate the same Bulletproof generators. The *BulletproofGens* structure is a vector of vectors of *RistrettoPoint*, generated with the SHAKE256 sponge construction used in *GeneratorsChain*.

```
pub fn new(gens_capacity: usize, party_capacity: usize) -> Self {
    use byteorder::{ByteOrder, LittleEndian};

    BulletproofGens {
        gens_capacity,
        party_capacity,
        G_vec: (0..party_capacity)
            .map(|i| {
                let party_index = i as u32;
                let mut label = [b'G', 0, 0, 0, 0];
                LittleEndian::write_u32(&mut label[1..5], party_index);

                GeneratorsChain::new(&label)
                    .take(gens_capacity)
                    .collect::<Vec<_>>()
            })
            .collect(),
        H_vec: (0..party_capacity)
            .map(|i| {
                let party_index = i as u32;
                let mut label = [b'H', 0, 0, 0, 0];
                LittleEndian::write_u32(&mut label[1..5], party_index);

                GeneratorsChain::new(&label)
                    .take(gens_capacity)
                    .collect::<Vec<_>>()
            })
            .collect(),
    }
}
```

Note: The code to generate *G_vec* and *H_vec* is highly similar and appears to be duplicated code.

Position

During the first exchange between the dealer and a party, the dealer will send a number j in the interval $[0, m)$. A cheating dealer may want to assign for some party the same value j, or a party may not want to follow the index assigned by the dealer. In such a case, the building

proof will be rejected[2], but we do not know if it is possible for an honest dealer or Verifier to detect which party does not follow the protocol.

In addition, we cannot be certain in such a case if someone having access to the transcript may extract information about the secret value of a party.

Once the parties have received their own position j, each party must compute the values $V_{(j)}$, $A_{(j)}$ and $S_{(j)}$. As in the setup step, the *thread_rng* function is used to randomly generate the blinding factors $\widetilde{a}_{(j)}$ and $\widetilde{s}_{(j)}$ used to compute the Pedersen commitments of $a_{(j)}$ and $s_{(j)}$.

```
let a_blinding = Scalar::random(&mut rng);
...
let s_blinding = Scalar::random(&mut rng);
```

Since the computation of $A_{(j)}$ uses the secret $v_{(j)}$, the implementation tries to be constant time (it looks like statically, be we cannot ensure that it keeps the same behavior dynamically). It uses the fact that the contribution of the generator at index i of $\mathbf{G}_{(j)}$ and the generator at index i of $\mathbf{H}_{(j)}$ cannot appear at the same time.

```
let mut i = 0;
for (G_i, H_i) in bp_share.G(self.n).zip(bp_share.H(self.n)) {
    // If v_i = 0, we add a_L[i] * G[i] + a_R[i] * H[i] = - H[i]
    // If v_i = 1, we add a_L[i] * G[i] + a_R[i] * H[i] =   G[i]
    let v_i = Choice::from(((self.v >> i) & 1) as u8);
    let mut point = -H_i;
    point.conditional_assign(G_i, v_i);
    A += point;
    i += 1;
}
```

The computation of $S_{(j)}$ needs to generate $2n$ random scalars split into two vectors of n scalars $\mathbf{s}_{L,(j)}$ and $\mathbf{s}_{R,(j)}$.

```
let s_L: Vec<Scalar> = (0..self.n).map(|_| Scalar::random(&mut rng)).collect();
let s_R: Vec<Scalar> = (0..self.n).map(|_| Scalar::random(&mut rng)).collect();
```

Then, the *multiscalar_mul* operation is applied to:

- a vector $\mathbf{v}_0$ (notation introduced in this document) of $2n+1$ scalars which aggregates $\widetilde{s}_{(j)}$, all the elements of $\mathbf{s}_{L,(j)}$ and $\mathbf{s}_{R,(j)}$ and
- a vector $\mathbf{v}_1$ (notation introduced in this document) of $2n+1$ *RistrettoPoint* which aggregates $\widetilde{B}$, all the elements of $\mathbf{G}_{(j)}$ and $\mathbf{H}_{(j)}$.

```
let v0 = iter::once(&s_blinding).chain(s_L.iter()).chain(s_R.iter());
let v1 = iter::once(&self.pc_gens.B_blinding).chain(bp_share.G(self.n))
              .chain(bp_share.H(self.n));

// Computation of S_j
let S = RistrettoPoint::multiscalar_mul(v0, v1);
```

The *multiscalar_mul* function must be constant time, but we did not have enough time to verify it. This function uses as a back-end the *EdwardPoint::multiscalar_mul* which uses itself *scalar_mul::straus::Straus::multiscalar_mul*. The last function is documented as being "Constant-time Straus using a fixed window of size 4".

[2] We have tested 2^{10} times a modification of the *detect_dishonest_party_during_aggregation* function to test different bad apportionment and the proof was always rejected.

A *BitCommitment* object is composed of $V_{(j)}$, $A_{(j)}$ and $S_{(j)}$.

If a party lies at least on one of these values, the proof must not be verified.

Generation of the y and z challenges

Once all the parties have sent their *BitCommitment*, the dealer must put the *BitCommitment* in order for a vector to be able to fully execute the receive_bit_commitments function.

At this point of the protocol, it is useful to describe what a *Transcript* is. The *Transcript* structure is defined inside the merlin crate. However, the implementation of the dealer inside *bulletproofs* uses almost only wrapper functions defined in src/transcript.rs except for the *clone* function, and mainly two functions which are *append_point* and *challenge_scalar*. In addition to writing the point in the transcript with *append_point*, the function modifies the internal state of a hash function used as a pseudo-random function. The *challenge_scalar* function uses the *challenge_bytes* function of *merlin::transcript::Transcript* itself using the *prf* function of *merlin::strobe::Strobe128*. Auditing this scalar generation was out of the scope of the audit, but it uses state-of-the-art algorithms, as the Keccak one which has given the standardized SHA-3 hash function[3].

To generate the y and z challenges, the dealer adds:

- each individual $V_{(j)}$;
- the sum $A = \sum_{j=0}^{m-1} A_{(j)}$ and
- the sum $S = \sum_{j=0}^{m-1} S_{(j)}$.

Even if the section 4.4 of [Bul] does not explicitly use the $V_{(j)}$, we don't have any objection to use them as random sources of entropy. Then, instead of generating $y = \mathsf{H}(A, S)$, we have $y = \mathsf{H}(\{V_{(j)}\}_{j\in[0,m)}, A, S)$. In addition, even if y is not added to the transcript[4], since extracting y from the construction of the hash function will modify the internal state of the sponge construction of the hash function, we have $z = \mathsf{H}(\{V_{(j)}\}_{j\in[0,m)}, A, S, y)$.

All the following steps are implemented in the receive_bit_commitments function as

```
// Commit each V_j individually
for vc in bit_commitments.iter() {
    self.transcript.append_point(b"V", &vc.V_j);
}

// Commit aggregated A_j, S_j
let A: RistrettoPoint = bit_commitments.iter().map(|vc| vc.A_j).sum();
self.transcript.append_point(b"A", &A.compress());

let S: RistrettoPoint = bit_commitments.iter().map(|vc| vc.S_j).sum();
self.transcript.append_point(b"S", &S.compress());

let y = self.transcript.challenge_scalar(b"y");
let z = self.transcript.challenge_scalar(b"z");
let bit_challenge = BitChallenge { y, z };
```

[3] More details can be found at https://keccak.team/.

[4] It is useless, since regenerating y (and z) can be done by anybody with A, S and $V_{(j)}$.

Evaluation with the y and z challenges

The dealer now sends the same *BitChallenge* to each of the parties. A dishonest dealer may want to:

- send different pairs of challenges to different parties;
- send specific values of y and z.

Sending different challenges will lead to a rejected proof. If we did not have time to check if this may however leak information about the secret, we do not see any trivial scenario which can exploit such an attack. Choosing specific y and z seems to be more dangerous, as we show in the following configuration.

If $y = 0$:

- $\mathbf{r}_{1,(j)} = (\mathbf{s}_{R,(j)}[0], 0, 0, \ldots, 0)$.[5]

If $z = 0$:

- $\mathbf{l}_{0,(j)} = \mathbf{a}_{L,(j)}$.
- $\mathbf{r}_{0,(j)} = \mathbf{y}^n_{(j)} \circ \mathbf{a}_{R,(j)}$ which becomes $\mathbf{r}_{0,(j)} = (\mathbf{a}_{R,(j)}[0], 0, 0, \ldots, 0)$ if $y = 0$.

These values are not directly exposed since:

- they are used to compute $t_{0,(j)}$, $t_{1,(j)}$ and $t_{2,(j)}$ and
- the values $t_{1,(j)}$ and $t_{2,(j)}$ are hidden in the T_1 and T_2 Pedersen commitments.

One or both of these configurations lead to commit values that only use parts of the secrets. In addition, it is highly improbable to have zero as a random value[6]. We may then suppose that the dealer is a cheating dealer and stop the protocol of the MPC computation. However, and according to Issue #87,

> T_1 and T_2 [...] are blinded independently from any dealer messages and therefore can't leak information.

It is therefore safe to continue the protocol, even with one or both challenges equal to zero.

The computation performed by a party begins by computing each coefficient of the degree one polynomial stacked into $\mathbf{l}_{(j)}(x)$ and $\mathbf{r}_{(j)}(x)$.

```
let zz = vc.z * vc.z;
let mut exp_y = offset_y;
let mut exp_2 = Scalar::one();
for i in 0..n {
    let a_L_i = Scalar::from((self.v >> i) & 1);
    let a_R_i = a_L_i - Scalar::one();

    l_poly.0[i] = a_L_i - vc.z;
    l_poly.1[i] = self.s_L[i];
    r_poly.0[i] = exp_y * (a_R_i + vc.z) + zz * offset_z * exp_2;
    r_poly.1[i] = exp_y * self.s_R[i];

    exp_y *= vc.y; // y^i -> y^(i+1)
    exp_2 = exp_2 + exp_2; // 2^i -> 2^(i+1)
}
```

[5] The notation $\mathbf{a}[0]$ is for the first element of the vector $\mathbf{a}$.
[6] It appears only $1/(2^{252} + 27742317777372353535851937790883648493)$ time from a uniform distribution.

Then, $t_{(j)}(x)$ is computed by the function *inner_product* which implements the described Karatsubas method to perform it. Once the coefficient of degree one, $t_{1,(j)}$, and the coefficient of degree two, $t_{2,(j)}$, are computed, the party has to generate two blinding factors to compute the Pedersen commitments of these two coefficients. The two commitments $T_{1,(j)}$ and $T_{2,(j)}$ are packed into *PolyCommitment*.

```
let t_poly = l_poly.inner_product(&r_poly);

let t_1_blinding = Scalar::random(&mut rng);
let t_2_blinding = Scalar::random(&mut rng);
let T_1 = self.pc_gens.commit(t_poly.1, t_1_blinding);
let T_2 = self.pc_gens.commit(t_poly.2, t_2_blinding);

let poly_commitment = PolyCommitment {
    T_1_j: T_1,
    T_2_j: T_2,
};
```

Generation of the x challenge

As for the aggregation of the contributions of the parties to compute S and A to compute the y and z challenges, the dealer aggregates the elements of the *PolyCommitment* in the order of their index. Then, the dealer computes T_1 (respectively T_2) as the sum of all the contributions of all the $T_{1,(j)}$ (respectively $T_{2,(j)}$).

```
let T_1: RistrettoPoint = poly_commitments.iter().map(|pc| pc.T_1_j).sum();
let T_2: RistrettoPoint = poly_commitments.iter().map(|pc| pc.T_2_j).sum();
```

These two values are added to the transcript. A cheating party may commit some crafted data in order to get information about the secret values from the transcript, but we do not know if it is possible. From the transcript, the dealer gets a random scalar x in the same way as we got y and z.

```
self.transcript.append_point(b"T_1", &T_1.compress());
self.transcript.append_point(b"T_2", &T_2.compress());

let x = self.transcript.challenge_scalar(b"x");
let poly_challenge = PolyChallenge { x };
```

Evaluation with the x challenge

With the x challenge, each party must evaluate the vectors of polynomials $\mathbf{l}_{(j)}$, $\mathbf{l}_{(j)}$ and the polynomial $t_{(j)}$. If x is equal to zero, the evaluation will result in removing the contribution of the blinding factors in the rest of the computations. This is why the authors of the library raise an error in such a case.

```
if pc.x == Scalar::zero() {
    return Err(MPCError::Maliciousdealer);
}
```

First of all, the party computes a blinding factor $\widetilde{t}_{(j)}(x) = z^2 z_{(j)}\widetilde{v}_{(j)} + x\widetilde{t}_{1,(j)} + x^2\widetilde{t}_{2,(j)}$. Note that in the documentation, the factor $z_{(j)}$ is forgotten.

```
let t_blinding_poly = util::Poly2(
    self.z * self.z * self.offset_z * self.v_blinding,
    self.t_1_blinding,
    self.t_2_blinding,
);
```

Then, the (vectors of) polynomials $t_{(j)}(x)$, $\widetilde{t}_{(j)}(x)$, $\mathbf{l}_{(j)}(x)$ and $\mathbf{l}_{(j)}(x)$ are evaluated with the x value sent by the dealer. With $\widetilde{a}_{(j)}$ and $\widetilde{s}_{(j)}$ randomly selected during the beginning of the protocol by each party, the last polynomial $\widetilde{e}_{(j)}(x) = \widetilde{a}_{(j)} + x\widetilde{s}_{(j)}$ is evaluated.

```
let t_x = self.t_poly.eval(pc.x);
let t_x_blinding = t_blinding_poly.eval(pc.x);
let e_blinding = self.a_blinding + self.s_blinding * &pc.x;
let l_vec = self.l_poly.eval(pc.x);
let r_vec = self.r_poly.eval(pc.x);
```

A *ProofShare*, the aggregation of all these evaluations, is then sent to the dealer. It is the last communication between the parties and the dealer.

Aggregation of the proofs

Setup

As usual, the dealer must aggregate the *ProofShare* of the parties in the order of their index.

First of all, the dealer computes $t(x)$, $\widetilde{t}(x)$ and $\widetilde{e}$ by adding the contribution of each party. These three values are added to the transcript, which allows in particular to obtain a challenge scalar w, used only by the dealer to create $Q = w \cdot B$ which will be used for the inner product proof.

```
let t_x: Scalar = proof_shares.iter().map(|ps| ps.t_x).sum();
let t_x_blinding: Scalar = proof_shares.iter().map(|ps| ps.t_x_blinding).sum();
let e_blinding: Scalar = proof_shares.iter().map(|ps| ps.e_blinding).sum();

self.transcript.append_scalar(b"t_x", &t_x);
self.transcript
    .append_scalar(b"t_x_blinding", &t_x_blinding);
self.transcript.append_scalar(b"e_blinding", &e_blinding);

// Get a challenge value to combine statements for the IPP
let w = self.transcript.challenge_scalar(b"w");
let Q = w * self.pc_gens.B;
```

The last step of the setup is the definition of an $n \cdot m$ vector $\mathbf{H}'$. An entry i of $\mathbf{H}'$ is equal to $y^{-i} \cdot \mathbf{H}_i$. The concatenation of the $\mathbf{l}_{(j)}(x)$ (respectively $\mathbf{r}_{(j)}(x)$) contribution is computed as follows:

```
let Hprime_factors: Vec<Scalar> = util::exp_iter(self.bit_challenge.y.invert())
    .take(self.n * self.m)
    .collect();
let l_vec: Vec<Scalar> = proof_shares
    .iter()
    .flat_map(|ps| ps.l_vec.clone().into_iter())
    .collect();
```

Inner product proof

The part of the documentation for this step can be found at https://doc.dalek.rs/bulletproofs/inner_product_proof/index.html, the code can be found at src/inner_product_proof.rs. To be performed, the computation needs the following input:

- the transcript maintained by the dealer;
- Q;
- $\mathbf{H}'$;
- $\mathbf{G}$;
- $\mathbf{H}$;
- $\mathbf{l}$ and
- $\mathbf{r}$.

Note that in the documentation, the $\mathbf{H}'$ vector is not used since its use is only for performance reasons and that the vectors $\mathbf{l}$ and $\mathbf{r}$ are named $\mathbf{a}$ and $\mathbf{b}$. For clarity, we will follow the notation used in the documentation and implementation of the inner product proof.

Note: For this section, n is the size of the vectors, i.e., $n \cdot m$ in the previous notation. This is to match the documentation and the code.

The algorithm must be run in $\log_2(n)$ rounds. Since there is a check that n is a power of two, the use of the function *next_power_of_two* is unneeded.

```
assert!(n.is_power_of_two());
let lg_n = n.next_power_of_two().trailing_zeros() as usize; ///
n.trailing_zeros()
```

At each step j, the length of the vectors for the computation is divided by two. The vectors are then split into two equal parts.

```
n = n / 2;
let (a_L, a_R) = a.split_at_mut(n);
```

The computation of L_j is performed by the *vartime_multiscalar_mul* method with two vectors that pack:

- for $\mathbf{v}_0$, the lowest half part (left part) of $\mathbf{a}$, the highest half part (right part) of $\mathbf{b}$ and the inner product c_L between the lowest half (left part) of $\mathbf{a}$ and the highest half (right part) of $\mathbf{b}$ and
- for $\mathbf{v}_1$, the lowest half part (left part) of $\mathbf{G}$, the highest half part (right part) of $\mathbf{b}$ and the point Q.

```
let c_L = inner_product(&a_L, &b_R);
let v0 = a_L.iter().chain(b_R.iter()).chain(iter::once(&c_L));
let v1 = G_R.iter().chain(H_L.iter()).chain(iter::once(Q));
let L = RistrettoPoint::vartime_multiscalar_mul(v0, v1).compress();
```

The computation of R_j looks highly similar to the computation of L_j.

```
let c_L = inner_product(&a_R, &b_L);
let v0 = a_R.iter().chain(b_L.iter()).chain(iter::once(&c_R));
let v1 = G_L.iter().chain(H_R.iter()).chain(iter::once(Q));
let R = RistrettoPoint::vartime_multiscalar_mul(v0, v1).compress
```

The two *CompressedRistretto* are added to the transcript, which allows the dealer to get a random scalar u. This scalar is used to compute the coordinates of the vectors $\mathbf{a}$, $\mathbf{b}$, $\mathbf{H}$ and $\mathbf{G}$ for the next round of computations.

For i in $[0, n)$:

- $\mathbf{a}[i] \leftarrow u \cdot \mathbf{a}_{\text{lo}}[i] + u^{-1} \cdot \mathbf{a}_{\text{hi}}[i]$
- $\mathbf{b}[i] \leftarrow u^{-1} \cdot \mathbf{b}_{\text{lo}}[i] + u \cdot \mathbf{b}_{\text{hi}}[i]$
- $\mathbf{G}[i] \leftarrow u^{-1} \cdot \mathbf{G}_{\text{lo}}[i] + u \cdot \mathbf{G}_{\text{hi}}[i]$
- $\mathbf{H}[i] \leftarrow u \cdot \mathbf{H}_{\text{lo}}[i] + u^{-1} \cdot \mathbf{H}_{\text{hi}}[i]$

```
transcript.append_point(b"L", &L);
transcript.append_point(b"R", &R);

let u = transcript.challenge_scalar(b"u");
let u_inv = u.invert();

for i in 0..n {
    a_L[i] = a_L[i] * u + u_inv * a_R[i];
    b_L[i] = b_L[i] * u_inv + u * b_R[i];
    G_L[i] = RistrettoPoint::vartime_multiscalar_mul(&[u_inv, u], &[G_L[i], G_R[i]]);
    H_L[i] = RistrettoPoint::vartime_multiscalar_mul(&[u, u_inv], &[H_L[i], H_R[i]]);
}

a = a_L; b = b_L; G = G_L; H = H_L;
```

At the end of the last round, the proof is constructed by aggregating:

- all the intermediate values of L_j;
- all the intermediate values of R_j;
- the value of $\mathbf{a}_0$ and
- the value of $\mathbf{b}_0$.

Range proof

The proof sent by the dealer to the Verifier is then composed of the inner product proof, and the scalars:

- $t(x)$
- $\tilde{t}(x)$
- $\tilde{e}$

The proof is also composed of the *CompressedRistretto*:

- A
- S

- T_1
- T_2

6.2.2 Verify a proof

An example to verify a proof can be found in src/range_proof/mod.rs in the *verify_multiple* function. This method can be used by the dealer to verify that the proof is well formed before sending it to the Verifier.

To verify a proof, the Verifier needs as inputs:

- the transcript provided by the dealer;
- the m commitments $V_{(j)}$ and
- the n parameter.

Note: Even if the $V_{(j)}$ values are added (one by one) to the transcript, they are always overwritten by each other. This leads to the fact that we cannot keep track of all the values of the $V_{(j)}$.

In order for the Verifier to find the same pseudo-random values (x, y and z) as those obtained by the dealer, the Verifier must create a transcript exactly in the same way as the one followed by the dealer.

To generate x, the Verifier adds to the transcript each $V_{(j)}$ and takes the A and S values from the transcript provided by the dealer. Before continuing the verification of the proof, the Verifier checks if A and S are not the identity point. The Verifier can then extract the two challenges y and z, which must be the same as the one obtained by an honest dealer.

```
for V in value_commitments.iter() {
    transcript.append_point(b"V", V);
}

transcript.validate_and_append_point(b"A", &self.A)?;
transcript.validate_and_append_point(b"S", &self.S)?;

let y = transcript.challenge_scalar(b"y");
let z = transcript.challenge_scalar(b"z");
```

The same mechanism is used to extract x, by adding the Ristretto points T_1 and T_2 from the transcript of the dealer to the transcript of the Verifier. Same operations on w, with $t(x)$, $\widetilde{t}(x)$ and $\widetilde{e}(x)$.

Then, the Verifier needs to generate the scalars u_1 to u_k and s_0 to s_{n-1}. This is achieved with the *verification_scalars* method implemented in *src/inner_product_proof.rs*. Since the Verifier needs to have the square of all the u_i and their inverse, the function returns $\{u_1^2, u_2^2, \ldots, u_k^2, u_1^{-2}, u_2^{-2}, \ldots, u_k^{-2}, s_0, s_1, \ldots, s_{n-1}\}$.

The Verifier also needs a random scalar c, which comes from another source of entropy than the one of the transcript, and the scalars a and b of the inner product proof.

```
let c = Scalar::random(&mut rng);
let a = self.ipp_proof.a;
let b = self.ipp_proof.b;
```

Once all these steps are performed, the Verifier must compute the right-hand side of the large equation given in the *Verifier's algorithm <https://doc-internal.dalek.rs/bulletproofs/range_proof/index.html#verifiers-algorithm>* section of the documentation. This large computation is computed thanks to the *RistrettoPoint::optional_multiscalar_mul* method with two large vectors that aggregate the scalars and the *RistrettoPoint* that are used. This *mega_check* does not follow the same order as the one from the addition chain. We present in the following piece of code the sequence given in the documentation.

```
let mega_check = RistrettoPoint::optional_multiscalar_mul(
    iter::once(Scalar::one()) // 1
        .chain(iter::once(x)) // x
        .chain(value_commitment_scalars), // cz^(i+2)
        .chain(iter::once(c * x)) // c * x
        .chain(iter::once(c * x * x)) // c * x^2
        .chain(iter::once(basepoint_scalar)) // w * (self.t_x - a * b) + c * (delta(n,
↪ m, &y, &z) - self.t_x)
        .chain(iter::once(-self.e_blinding - c * self.t_x_blinding)) // -tilde(e) - c␣
↪* tilde(t)(x)
        .chain(g) // -z - a * s_i
        .chain(h) // z + y^i * z^2 * ? - b * s_i^(-1)
        .chain(x_sq.iter().cloned()) // u_i^2
        .chain(x_inv_sq.iter().cloned()) // u_i^(-2)
    iter::once(self.A.decompress()) // A
        .chain(iter::once(self.S.decompress())) // S
        .chain(value_commitments.iter().map(|V| V.decompress()), // V_j
        .chain(iter::once(self.T_1.decompress())) // T_1
        .chain(iter::once(self.T_2.decompress())) // T_2
        .chain(iter::once(Some(pc_gens.B))) // B
        .chain(iter::once(Some(pc_gens.B_blinding))) // tilde(B)
        .chain(bp_gens.G(n, m).map(|&x| Some(x))) // G
        .chain(bp_gens.H(n, m).map(|&x| Some(x))) // H
        .chain(self.ipp_proof.L_vec.iter().map(|L| L.decompress())) // L
        .chain(self.ipp_proof.R_vec.iter().map(|R| R.decompress())) // R
)
```

6.3 Code review

In addition to the protocol implementation review, we also audited the code to validate the protections implemented in the library to prevent an attack on the protocol.

6.3.1 State machine

Each state of the party and the dealer is implemented in a separate structure:

- DealerAwaitingBitCommitments
- DealerAwaitingPolyCommitments
- DealerAwaitingProofShares

- PartyAwaitingPosition
- PartyAwaitingBitChallenge
- PartyAwaitingPolyChallenge

To assert that the state machine is correctly followed and that a used state cannot be reused, the method that computes the next step takes the ownership on the current state. In Rust, when a method takes the ownership of an object, this object cannot be reused afterwards. The only exception is if the state implements the traits *Clone* or *Copy*, but this is not the case here. In addition, we tried to implement these traits in a dependent library but the rust error E0117 prevents to implement a foreign trait on a foreign structure without using a local structure. Finally, the internal value of each state is private, which disallows a legit dependent library to use the state outside of the state machine.

The library cannot prevent an attacker to run a new proof computation as a party. This attack must be prevented by the dependent libraries. In the same way, a dependent library must not try to bypass the previous behavior (e.g., with the help of *unsafe* statements).

6.3.2 Message serialization

The library implements the *Serialize* and *Deserialize* traits on the messages exchanged between the parties and the dealer in src/range_proof/messages.rs and on the final proof in srcrange_proof/mod.rs.

Messages and proofs contain some structures of *curve25519-dalek* that we already fuzzed in the previous part (see Section 5.1.4). For the *bulletproofs* library, we fuzzed each message with the next operation. The proof was fuzzed with the *verify_multiple* method that was used to validate the proof. The goal was to assert that an attacker that sends a malicious package will only get a false proof or an error.

During the fuzzing of *ProofShare*, the library panicked in some cases when the length of *l_vec* and *r_vec* were not the expected ones. As opposed to the proof, the serializer is not customized and does not check the length of vectors. All vectors of all parties are concatenated before computing the inner product. In the beginning of InnerProductProof::create, the length is validated with *assert_eq* and causes a panic. In addition, two colluding parties can change the length of their vectors in such a way that the agglomerated vector has the right length. We recommend validating the length of each party's vector in the dealer and return an error if one of them has not the expected length.

6.3.3 Constant Time

At some points, the library uses variable time methods with branches. However, we did not find any usage of these methods with the secrets v and *v_blinding*. These secrets use the constant time methods of *subtle* and *curve25519-dalek* and no *if* statement is used on these values. We did not analyze all generated assembly code to assert that there is no optimization branch. As a consequence, the constant time property is only based on code linearity.

As the computations of *bulletproofs* include some variable time computation on public values, and the duration of the computation is too long to avoid OS interruptions, we cannot conclude about real constant time in the *bulletproofs* library. However, if the protocol was strictly implemented, a remote attacker who wants to use a timing attack to leak a secret value (i.e. v or $v_blinding$) can only use the timing on the message. We conclude that a timing attack on *bulletproofs* for a remote attacker (which can be another party or the dealer) has a low risk to

success if the protocol is followed and not replayed.

6.4 Issues

In *bulletproofs*, the only real issue was in the *ProofShare* serial which was not completely checked and can force a legit dealer to panic.

6.4.1 *ProofShare* panic

A malicious party can send a *ProofShare* that will make the dealer panic. The consequence of this panic will depend on the dependent libraries. This issue was only valid if a party cannot be trusted and can send arbitrary serialized messages.

The serializer used for this purpose has no impact on this vulnerability. If we use *rmp-serde*, the following valid serial of *ProofShare* can be used to trigger this issue :

```
let proofshare_raw = [149, 196, 32, 0, 0, 0, 0, 0,
                      0, 0, 0, 0, 0, 0, 0, 0,
                      0, 0, 0, 0, 0, 0, 0, 0,
                      0, 0, 0, 0, 0, 0, 0, 0,
                      0, 0, 0, 196, 32, 0, 0, 0,
                      0, 0, 0, 0, 0, 0, 0, 0,
                      0, 0, 0, 0, 0, 0, 0, 0,
                      0, 0, 0, 0, 0, 0, 0, 0,
                      0, 0, 0, 0, 0, 196, 32, 0,
                      0, 0, 0, 0, 0, 0, 0, 0,
                      0, 0, 0, 0, 0, 0, 0, 0,
                      0, 0, 0, 0, 0, 0, 0, 0,
                      0, 0, 0, 0, 0, 0, 0, 144,
                      144];
```

To avoid this problem, the length of each *ProofShare* vector must be independently validated with the length of **G** in DealerAwaitingProofShares::assemble_shares and ProofShare::audit_share before using the provided values.

6.4.2 Optimizations and recommendations

size_hint on *AggregatedGensIter*

The *size_hint* method of *AggregatedGensIter* does not respect its definition in the documentation of the trait Iterator. This method must return the remaining length of the iterator whereas the current implementation returns the original length.

```
#[test]
fn custom_test() {
    let gens = BulletproofGens::new(64, 8);
    let mut G_iter = gens.G(64, 8);
    assert_eq!(G_iter.size_hint().0, 512);

    G_iter.next().unwrap();
    assert_eq!(G_iter.size_hint().0, 511); // fail
}
```

The implementation should be corrected by:

```
fn size_hint(&self) -> (usize, Option<usize>) {
    let size = self.n * (self.m - self.party_idx) - self.gen_idx;
    (size, Some(size))
}
```

Reusing computations

In the *PartyAwaitingBitChallenge::apply_challenge* implementation, it is also possible to reuse a part of the computations. Indeed, since $z_{(j)}$ does not change during the loop, it is possible to precompute $z^2 \cdot z_{(j)}$.

```
- let zz = vc.z * vc.z;
+ let offset_zz = vc.z * vc.z * offset_z;
  let mut exp_y = offset_y;
  let mut exp_2 = Scalar::one();
  for i in 0..n {
      let a_L_i = Scalar::from((self.v >> i) & 1);
      let a_R_i = a_L_i - Scalar::one();

      l_poly.0[i] = a_L_i - vc.z;
      l_poly.1[i] = self.s_L[i];
-     r_poly.0[i] = exp_y * (a_R_i + vc.z) + zz * offset_z * exp_2;
+     r_poly.0[i] = exp_y * (a_R_i + vc.z) + offset_zz * exp_2;
      r_poly.1[i] = exp_y * self.s_R[i];

      exp_y *= vc.y; // y^i -> y^(i+1)
      exp_2 = exp_2 + exp_2; // 2^i -> 2^(i+1)
  }
```

This can also be used to create a *PartyAwaitingPolyChallenge*, since its method uses only the result of $z^2 \cdot z_{(j)}$.

Usage of *rmp-serde*

During the audit, we used *rmp-serde* as the serializer to fuzz some structures. We have locally patched the issue #151 to fuzz the code efficiently. This piece of code allocates the buffer by pages of 4096 bytes, and can return the *UnexpectedEof* error of the Read.read_exact method without allocating the full buffer length.

```
diff --git a/rmp-serde/src/decode.rs b/rmp-serde/src/decode.rs
index ab20c78..6310e9f 100644
--- a/rmp-serde/src/decode.rs
+++ b/rmp-serde/src/decode.rs
@@ -570,8 +570,14 @@ impl<R: Read> ReadReader<R> {
 impl<'de, R: Read> ReadSlice<'de> for ReadReader<R> {
     #[inline]
     fn read_slice<'a>(&'a mut self, len: usize) -> Result<Reference<'de, 'a, [u8]>,␣
↪io::Error> {
+        let mut l = 4096;
+        while l < len {
+            self.buf.resize(l, 0u8);
+            self.rd.read_exact(&mut self.buf[(l-4096)..])?;
+            l += 4096;
```

(continues on next page)

(continued from previous page)

```
+        }
         self.buf.resize(len, 0u8);
-        self.rd.read_exact(&mut self.buf[..])?;
+        self.rd.read_exact(&mut self.buf[(l-4096)..])?;

         Ok(Reference::Copied(&self.buf[..]))
     }
```

We recommend patching this issue to avoid a too big memory allocation, and possibly a crash in memory-constrained environments.

6.5 Conclusion

We found no deviation from the original article introducing the protocol. The extensive notes and documentation helped a lot to understand the choices made by the authors of the implementation. The usage of this library includes some protections regarding the protocol. However, dependent libraries must implement some protections against the protocol replay and handle *bulletproofs* errors.

7. The *x25519-dalek* and *ed25519-dalek* libraries

7.1 The *x25519-dalek* library

7.1.1 Dependencies

This library relies on:

- *clear_on_drop* version *^0.2*: helpers for clearing sensitive data on the stack and heap.
- *curve25519-dalek* version *^1*: forced to be version *1.2.1*.
- *rand_core* version *^0.3*: Rust library for random number generation.

Note that all these dependencies used 8 libraries as a back-end if compiled with:

```
$ RUSTFLAGS="-C target_feature=+avx2" cargo build --release
```

7.1.2 Public key validation

We report here a point of interest which is still a debate between cryptographers. This debate can be summarized in the blog post [Aum]. Indeed, some values of a public key may blind the contribution of a private one, leading to compute a shared secret equal to zero. These public keys are well-known[1]. Following [GVY], the *libsodium* library with the *has_small_order*[2] function rejects a public key composed of these small-order points.

Alternatively or in addition to this test, and as specified in the RFC [ECS], it is also recommended to check if the common secret computed by the function `diffie_hellman` is the all-zero value: this check must be performed with a constant-time implementation, as provided by the *subtle* library.

The following piece of code lists the seven public keys leading to a zero secret key.

```
// Cargo.toml piece of code
// [package]
// name = "test-x25519"
// version = "1.0.0"
// edition = "2018"
// [dependencies]
// curve25519-dalek = "1.2.1"
// x25519-dalek = "0.5.2"
// rand_os = "0.1"

use rand_os::OsRng;

use x25519_dalek::{PublicKey, EphemeralSecret};

fn main() {
    let bob_public = [
        PublicKey::from([0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                         0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

(continues on next page)

[1] https://cr.yp.to/ecdh.html#validate

[2] https://github.com/jedisct1/libsodium/blob/61992a838df3db8a27443e089656fb1ac0bc1608/src/libsodium/crypto_scalarmult/curve25519/ref10/x25519_ref10.c#L17

(continued from previous page)

```
                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ]),
        PublicKey::from([0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ]),
        PublicKey::from([0xe0, 0xeb, 0x7a, 0x7c, 0x3b, 0x41, 0xb8, 0xae,
                         0x16, 0x56, 0xe3, 0xfa, 0xf1, 0x9f, 0xc4, 0x6a,
                         0xda, 0x09, 0x8d, 0xeb, 0x9c, 0x32, 0xb1, 0xfd,
                         0x86, 0x62, 0x05, 0x16, 0x5f, 0x49, 0xb8, 0x00, ]),
        PublicKey::from([0x5f, 0x9c, 0x95, 0xbc, 0xa3, 0x50, 0x8c, 0x24,
                         0xb1, 0xd0, 0xb1, 0x55, 0x9c, 0x83, 0xef, 0x5b,
                         0x04, 0x44, 0x5c, 0xc4, 0x58, 0x1c, 0x8e, 0x86,
                         0xd8, 0x22, 0x4e, 0xdd, 0xd0, 0x9f, 0x11, 0x57, ]),
        PublicKey::from([0xec, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
                         0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
                         0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
                         0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x7f, ]),
        PublicKey::from([0xed, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
                         0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
                         0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
                         0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x7f, ]),
        PublicKey::from([0xee, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
                         0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
                         0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
                         0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x7f, ]),
        ];

    for x in 0..bob_public.len() {
        let mut csprng = OsRng::new().unwrap();
        let alice_secret = EphemeralSecret::new(&mut csprng);

        let alice_shared_secret = alice_secret.diffie_hellman(&bob_public[x]);

        assert_eq!(alice_shared_secret.as_bytes(), &[0; 32]);
    }
}
```

Depending on the taxonomy and the purpose of the library, this non-validation may not be an issue.

7.2 The *ed25519-dalek* library

7.2.1 Dependencies

This library relies on:

- *clear_on_drop* version *^0.2*: helpers for clearing sensitive data on the stack and heap.
- *curve25519-dalek* version *^1*: forced to be version *1.2.1*.
- *failure* version *^0.1.1*: error management.
- *rand* version *^0.6*: a Rust library for random number generation.
- *sha2* version *^0.8*: SHA-2 hash functions.

- *serde* version *^1.0* (optional): serialization framework for Rust.

Note that all these dependencies use as a back-end 23 libraries if compiled with:

```
$ RUSTFLAGS="-C target_feature=+avx2" cargo build --release --no-default-features --
↪features "std avx2_backend"
```

7.2.2 Notes on the usage

We highlight here a point that is acknowledged in the documentation of the project, in the section A Note on Signature Malleability.

> We could eliminate the malleability property by multiplying by the curve cofactor, however, this would cause our implementation not to match the behavior of every other implementation in existence. As of this writing, RFC 8032 [EdDSA], "Edwards-Curve Digital Signature Algorithm (EdDSA)", advises that the stronger check should be done. While we agree that the stronger check should be done, it is our opinion that one shouldn't get to change the definition of "ed25519 verification" a decade after the fact, breaking compatibility with every other implementation.
>
> In short, if malleable signatures are bad for your protocol, don't use them. Consider using a curve25519-based Verifiable Random Function (VRF), such as Trevor Perrin's VXEdDSA [Per], instead. We plan[3] to eventually support VXEdDSA in curve25519-dalek.

A discussion about the compatibility of the library with *libsoduim* is open inside the project[4]. In order to mitigate issues coming from the malleability of the signature scheme, it is possible for the developers using this library to use a check similar to the one performed in *libsodium*[5].

[3] https://github.com/dalek-cryptography/curve25519-dalek/issues/9
[4] https://github.com/dalek-cryptography/ed25519-dalek/issues/20
[5] See for example https://github.com/jedisct1/libsodium/commit/4099618de2cce5099ac2ec5ce8f2d80f4585606e.

8. Conclusion

This report summarizes the audit on the *subtle*, *curve25519-dalek* and *bulletproofs* libraries, as well as a more marginal audit of the *x25519-dalek* and *ed25519-dalek* libraries. The choice of Rust as a language probably avoided most common problems that may have been found in an implementation with a language such as C. The different projects as well as their code are thoroughly documented. This allows any interested party to validate the implementation choices. However, some issues have been found. Some recommendations have also been formulated to improve the library and its usage by the dependent libraries. It is important to note that some tests are non-deterministic (using randomness seeded at runtime) and may reduce their reproducibility. However this also makes it possible to cover a larger variety of inputs across time. We recommend at least to output the seed used during a failed test to enable easier debugging.

These libraries include some mitigations against side-channel attacks, such as constant time operations with linear code and constant memory access. These mitigations are only provided on a best-effort basis and are limited by their scope: hardware issues or updates on the nightly version of the Rust compiler may compromise them (such as unintended new optimizations or features that lead to a code no longer being constant time after compilation). However, the libraries do their best to avoid such mitigation breaks by using some features of the compiler. We recommend nonetheless to add additional tests to check for unexpected regressions of mitigations between unstable versions and to periodically check the desired properties on samples of the compiled assembly. We also recommend to use a stable version of the Rust compiler when the required features become stable.

9. Bibliography

[Aum] J.-P. Aumasson, Should Curve25519 keys be validated?, April 25, 2017. https://research.kudelskisecurity.com/2017/04/25/should-ecdh-keys-be-validated/

[Ber] D. J. Bernstein, Curve25519: New Diffie-Hellman Speed Records, PKC 2006, pages 207–228. https://cr.yp.to/ecdh/curve25519-20060209.pdf

[Bul] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille and G. Maxwell, Bulletproofs: Short Proofs for Confidential Transactions and More, IEEE Symposium on Security and Privacy 2018, https://eprint.iacr.org/2017/1066 (version of the 1st of July, 2018) and https://crypto.stanford.edu/bulletproofs/

[ECS] A. Langley, M. Hamburg and S. Turner, Elliptic Curves for Security, IETF RCF 7748, January 2016. https://tools.ietf.org/html/rfc7748

[EdDSA] S. Josefsson and I. Liusvaara, Edwards-Curve Digital Signature Algorithm (EdDSA), IETF RFC 8032, January 2017. https://tools.ietf.org/html/rfc8032

[GVY] D. Genkin, L. Valenta and Yuval Yarom, May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519, Conference on Computer and Communications Security, pages 845–858. https://eprint.iacr.org/2017/806

[Ham] M. Hamburg, Decaf: Eliminating Cofactors Through Point Compression, CRYPTO 2015, pages 705–723. https://eprint.iacr.org/2015/673

[Per] T. Perrin, The XEdDSA and VXEdDSA Signature Schemes, October 20, 2016. https://signal.org/docs/specifications/xeddsa

[Ris] H. de Valence, J. Grigg, G. Tankersley, F. Valsorda and I. Lovecruft, The ristretto255 Group, May 8, 2019. https://datatracker.ietf.org/doc/draft-hdevalence-cfrg-ristretto

[TRG] H. de Valence, I. Lovecruft and T. Arcieri, The Ristretto Group. https://ristretto.group/