Security audit of neptune-triton

Troels Henriksen (athas@sigkill.dk)

23rd of July, 2020

This document constitutes a security review of neptune-triton, a Futhark implementation of the Poseidon hash function. Neptune-triton is implemented in Futhark, a data-parallel purely functional programming language, with a Rust interface generated by genfut. This review discusses both general security properties and risks of Futhark as a language and compiler, as well as their concrete impact on neptune-triton specifically.

This review is based on neptune-triton 1.1.2 (revision 3b4b0c460abf3dc5ddf2ce23f91ed6c6e703b563, which uses Futhark 0.16.2 and genfut 0.3.0.

Issue severity levels

Every identified issue in this report is assigned a severity level from the following:

- **Critical** issues are immediate security issues and should be fixed immediately.
- **Major** issues will almost certainly cause problems to users, and should be fixed.
- **Medium** issues can potentially bring problems, but fixing them is not urgent.
- **Low** severity issues range from cosmetic to theoretical issues.

Some low-severity issues may not be practically solvable, as they are consequences of foundational design and implementation choices. They are kept in the report, as it is still worthwhile to be aware of them.

Conclusions up front

We have identified no issues of *critical* or *major* severity, 1 issue of *medium* severity, and 8 issues of *low* severity. Further, we have reached the following conclusions:

- The Futhark language by design is completely safe.
- Neptune-triton uses Futhark in a safe and well-defined manner.

- The Futhark compiler implementation is likely to generate only safe code, but the risk of code generation bugs is larger than for more widely used languages. However, it is very unlikely that compiler bugs would cause security issues for neptune-triton.
- It is very unlikely that the generated code contains memory errors that could be used to subvert system safety. Is is very unlikely that any malfunction of the generated code could affect anything outside the running process.
- The genfut-generated Rust uses the Futhark-generated C correctly, but fails to deallocate the Futhark context object appropriately, is missing some error checking, and does not guard users against all use-after-free errors. Due to the way it is used in neptune-triton, this currently has no negative consequences, but presents a risk of errors when modifying neptune-triton in the future.
- In the event of Futhark development ceasing, neptune-triton would need to migrate to another language, but not urgently.

The rest of the report will elaborate these conclusions, while also serving as a procedure for verifying future changes to neptune-triton.

Design of Futhark

Futhark is a parallel programming language designed and implemented at the University of Copenhagen since 2013. The implementation is open source and written in Haskell. Most of the implementation has been written by Troels Henriksen (author of this report). The rest has been written by students, open source contributors, and colleagues at the University of Copenhagen. The compiler is actively maintained by a team at the university, comprising Cosmin Oancea (associate professor), Troels Henriksen (postdoc) , Philip Munksgaard (PhD student), and Robert Schenk (PhD student).

Futhark is a side-effect free (purely functional) programming language. This means that programs in Futhark can perform only pure computation, and are not able to interact directly with the outside world. A Futhark program constitutes a range of *entry point functions*, which can be provided with arguments, and which then return values. The Futhark compiler transforms a Futhark program to a C program, which can be linked into a larger application, with Futhark entry points exposed as C functions. This allows Futhark code to be called by code written in any language that can call C functions.

As a purely functional language, a defective Futhark program can at worst crash in a controlled manner, produce wrong results, or go into an infinite loop. It cannot subvert the machine, modify files, or make network connections. Note that this diverges from other “nominally pure” languages such as Haskell, which *do* provides facilities for expressing arbitrary side effects (in Haskell’s case, the IO monad). While this makes Futhark code safe *by design*, preserving these properties in the generated code assumes a correct compiler. No compiler is completely bug-free, and we shall discuss to which degree we can trust the Futhark compiler to preserve these guarantees.

The main advantage of Futhark is that the compiler can generate efficient low-level parallel code from high-level Futhark programs. As of this writing, the compiler targets single-GPU execution via the standard OpenCL and CUDA low-level APIs. These APIs are not on their own safe, but depend on the compiler to generate code that only uses them in safe ways.

Design of neptune-triton

Neptune-triton uses Futhark to implement various cryptographic primitives for the Poseidon hash function, then exposes these primitives in the form of a Rust library.

The main Futhark code resides in [`poseidon.fut`]https://github.com/filecoin-project/neptune-triton/blob/3b4b0c460abf3dc5ddf2ce23f91ed6c6e703b563/library/poseidon.fut, which defines the following entry points:

- `init2`
- `init8`
- `hash8`
- `init11`
- `init2s`
- `init8s`
- `init11s`
- `mbatch_hash2`
- `mbatch_hash8`
- `mbatch_hash11`
- `mbatch_hash2s`
- `mbatch_hash8s`
- `mbatch_hash11s`
- `init_t8_64m`
- `build_tree8_64m`
- `simple8`

Auditing the algorithmic correctness of these is outside the scope of this report. We will assume that unit and integration tests have established the functional correctness of the code. In practice, neptune-triton has been used in the correct operation of a Filecoin testnet, so it is highly likely that the implementation is algorithmically correct.

The implementation further depends on the following two Futhark libraries:

- github.com/filecoin-project/fut-ff v0.1.9: an efficient implementation of finite fields. Maintained by same author as neptune-triton itself.
- github.com/athas/vector v0.6.0: an efficient implementation of small statically-sized arrays. Maintained by Troels Henriksen, who also

maintains the Futhark compiler.

The Futhark compiler is able to generate C code, while neptune-triton wishes to expose a Rust library. To bridge the gap, neptune-triton uses the tool genfut, which automatically generates Rust bindings to the C code generated for a Futhark program. In particular, genfut attempts to provide a safer Rust-style abstraction on top of the resource-unsafe C interface exposed by Futhark.

One crucial implementation detail for neptune-triton is that all generated code is *embedded* directly in the source tree (sometimes called "vendoring"). This means that compiling neptune-triton does not depend on running the Futhark compiler, nor genfut. Specifically, neptune-triton is structured into two parts:

- neptune-triton-generator contains the original Futhark code (`poseidon.fut`), as well as generator code for producing the actual neptune-triton library.
- neptune-triton, which is generated by neptune-triton-generator. Although this is generated code, committing it directly to the Git repository simplifies compilation by users. Specifically, it contains a.c, which is the C code generated by the Futhark compiler for `poseidon.fut`. This is the C source file that will eventually be turned into machine code by a C compiler, and linked into the final application.

The README for neptune-triton describes this design in detail, and documents how a user can re-generate the generated library. This addresses one common drawback of directly committing generated code, namely that there is no guarantee that the generated code corresponds exactly to what the generator produces. By letting the cautious user (or auditor) re-perform the generation, we gain assurance that the generated code has not been erroneously or maliciously modified.

Re-performing the code generation

This section describes our effort to re-generate the neptune-triton library, following the instructions.

The generated library contains a file futhark-version.txt that describes exactly which version of Futhark was used. As of this writing, this is Futhark 0.16.2.

The reproduction steps work, and as expected produce a new `a.c` file (based on timestamps) that is identical to the one already present.

A Continuous Integration job has been set up to mechanically verify that the reproducibility steps are correct. This provides confidence that the link beween the Futhark code and the generated C code will remain intact.

Safety of the Futhark-generated C code

This section will discuss potential risks and issues with the C code generated by the Futhark compiler. While no known compiler bugs exist that could subvert the safety guarantees of the Futhark language, it is still important to understand where they would possibly reside. Since C is an unsafe language, we depend on the correctness of the Futhark compiler when it comes to generating safe code. The Futhark compiler is not provably correct, and has had bugs in the past.

One important quality that significantly increases the safety of neptune-triton, is that we do not have to be on our guard against arbitrary or untrusted Futhark programs. We only have to ensure that for *one specific Futhark program* (`poseidon.fut`), the code that is generated is likely to be safe and correct. We do not have to be concerned with the risk that some other malicious Futhark program could subvert the compiler. The security risks are thus only related to malicious *input* to this one program.

General compiler bugs

Compared to heavily used industrial compilers with large development teams and hundreds of thousands of users, the Futhark compiler is relatively untested and likely much more buggy. However, it does have a reasonably rigorous and automated testing setup, comprising:

- Over 1400 integration test programs, comprising both negative and positive tests. For every commit, this suite is tested with every compiler backend.
- 50 benchmark programs, which are likewise tested with every compiler backend for every commit.
- A few library tests, which test specifically the C API interface that neptune-triton makes use of.

There are two weaknesses in the testing setup:

- The library tests are quite sparse and a recent addition. However, the actual compiler path and code generation is identical when doing integration tests and library tests, which ameliorates this weakness.
- The automatic testing of the GPU backends is done on a fairly small set of GPUs (GTX 780 Ti, K40, RTX 2080 Ti), all from one vendor (NVIDIA). While Futhark strives to generate portable GPU code, this is currently not automatically tested. At least one Futhark compiler developer (Troels Henriksen) does use an AMD GPU for manual testing and development.

Further, somewhat unusually among compilers, when compiling a program Futhark will always perform a full type check/validation of the intermediate representation after each of the approximately twenty distinct passes that make up the compiler. While this checking is not as complete as the one performed on the source language, it has proven very effective in avoiding the generation of invalid code, and instead crashing the compiler. Of the ten most recently fixed actual compiler bugs (#1053, #1043, #1026, #1025, #1007, #995, #992,

#971, #942, #941), all but #942 and #992 manifested as the internal type checking failing, and only #992 silently produced wrong results at run-time.

Provoking crashes

The Futhark-generated C code calls other C APIs, most notably OpenCL. While the code is careful to check return values for errors, in a few cases an API error will abort the entire process by calling the C `abort()` function. This is very undesirable for library code, and if an attacker can provoke these errors, they could cause the entire application to terminate (although not to silently malfunction). The generated code will currently `abort()` in the following cases.

Sources of abnormal termination during initialisation of the Futhark context (*severity: low*)

- When no OpenCL device can be found, or when a specific device has been requested and it cannot be found.
- When the Futhark program requires double-precision floating-point numbers, and the GPU does not support double-precision. This does not affect neptune-triton.
- If certain API operations such as `clGetPlatformInfo()` or `clCreateContext()` fail. It is unclear when this would happen, and is likely to only occur if the OpenCL driver itself is defective, or if the host system is completely out of memory.
- If the embedded OpenCL program cannot be compiled by the OpenCL driver.

While these issues are of course undesirable and should be fixed, they are rare, and difficult-to-impossible to provoke by an attacker, as they occur only during initialisation, without any potentially malicious input.

Sources of abnormal termination during execution of Futhark entry point functions (*severity: low*)

- A few safety checks that guard against possible compiler bugs, such as trying to allocate a negative amount of memory.

All other anomalous run-time situations (such as running out of GPU memory, or when OpenCL API calls return error codes for unexpected reasons) are gracefully handled by returning an error code from the C function that implements the entry point. This makes it unlikely that malicious input that causes the Futhark program to misbehave (e.g. by consuming all GPU memory) will cause the entire process to crash. However, the Futhark test suite does not rigorously exercise these error handling paths. For example, it is possible that error handling performs improper cleanup of memory resources, and if so, an attacker might exploit this to cause out-of-memory conditions. This can be circumvented by completely re-initialising the Futhark GPU context whenever an error condition is encountered. This is likely to be very rare anyway, so the performance impact is likely minimal (and in most cases, an error returned by Futhark may even be considered a fatal error at the application level).

Undefined behaviour in generated code (***severity: low***)

C is infamous for its notion of undefined behaviour, where expressions that break certain rules allow a C compiler to make unintuitive program transformations. Most of these rules are subtle, unexpected, and not diagnosed by the C compiler. Particularly pertinent examples include:

- Bit-shifting a 32-bit integer by more than 31 places.
- Signed integer arithmetic that produces overflow.
- Restrictions on pointer aliasing.

As Futhark generates significant amounts of C code, there is a risk of emitting code that runs afoul of these rules. While most undefined behaviour is harmless, it can in principle have quite chaotic and unpredictable effects, including security risks. Although the C code that is generated by the Futhark compiler is fully deterministic, neptune-triton does not (and likely *cannot*) pin down which C compiler should be used to compile the code, and the treatment of undefined behaviour can vary from one compiler version to the next. Similarly, the GPU kernels generated by Futhark are eventually compiled by the OpenCL implementation supplied by the GPU vendor, the version of which cannot reasonably be pinned.

The semantics of Futhark specify two's complement semantics for signed integers, with well-defined overflow. As a result, Futhark generates C code that uses *unsigned* integers, to avoid undefined overflow. The only signed integer arithmetic generated by Futhark is done as part of automatically generated address calculations, which are overflow-free.

It is unlikely that the C code generated by the Futhark compiler (nor any large C program) is *completely* free of undefined behaviour in the strictest sense. However, the mere presence of undefined behaviour is not necessarily a security issue. In practice, security issues that arising from undefined behaviour are usually the result of the C compiler removing security checks.

Inspection of generated code

The generated `a.c` comprises approximately 110,000 lines of so-called "host code" (the code that runs on the CPU). This includes a few thousand lines of Futhark's runtime system, with the remainder being generated from neptune-triton itself. The `a.c` file also contains "device code" (that runs on the GPU), totaling approximately 340,000 lines. The device code is embedded as a string, but can be extracted into a separate file `kernels.cl` via the following program:

```c
#include "a.h"

int main() {
  struct futhark_context_config *cfg = futhark_context_config_new();
  futhark_context_config_dump_program_to(cfg, "kernels.cl");
  struct futhark_context *ctx = futhark_context_new(cfg);
}
```

The fact that 450,000 lines of C have been generated from the approximately 1100 lines of Futhark making up neptune-triton may seem excessive. This ballooning is partially due to Futhark's default compilation strategy being heavily centred on inlining and code duplication. However, also significant is that the generated C code is very "assembly-like", in that most statements are primitive operations that correspond to approximately one instruction. A representative excerpt follows:

```c
if (res_1342128) {
    res_1342129 = 1;
    res_1342130 = res_1342123;
} else {
    bool res_1342131 = ult64(unsign_arg_1342117, res_334688);
    bool res_1342132 = unsign_arg_1342117 == res_334688;
    bool x_1342133 = !res_1342132;
    bool x_1342134 = !res_1342131;
    bool y_1342135 = x_1342133 && x_1342134;
    bool res_1342136 = res_1342131 || y_1342135;

    res_1342129 = res_1342136;
    res_1342130 = res_1342131;
}
```

Thus, while the generated code is human-readable C in the strictest sense, it is much too voluminous and low-level for manual inspection. Ultimately, as with all compilers we must rely on the compiler being non-malicious (but see later section on *Supply chain risks*), but we can perform simple static analysis and inspection to establish rough confidence in the robustness and safety of the generated code.

Memory safety The generated code performs two kinds of dynamic memory allocations:

- Management of CPU memory with the usual `malloc()`/`free()`/`realloc()` operations.
- Management of GPU memory using OpenCL API operations such as `clCreateBuffer()` and `clEnqueueWriteBuffer()`.

We will discuss the security ramifications of each separately.

Safety of CPU memory For the generated entry points, all use of dynamic CPU memory is related to bookkeeping information, such as maintaining the free list in the GPU memory allocator, or maintaining reference counts. These allocations use sizes that are not derived from program input, and are indexed with values that are also not derived from program input. Further, this code is part of the embedded Futhark runtime system, which is the same for all compiled Futhark programs. Hence it is unlikely for malicious input to neptune-triton to result in misuse of CPU memory (e.g out-of-bounds accesses).

Safety of GPU memory (*severity: low*) GPU memory is (on most devices) physically distinct from CPU memory, and so invalid use of GPU memory cannot in general corrupt CPU memory.

The generated code ultimately allocates GPU memory through the OpenCL API function `clCreateBuffer()`, but mediated through a custom free-list based allocator. This free-list is in principle unbounded in size, and could theoretically be attacked by triggering behaviour that causes it to retain GPU memory unnecessarily, leading to erroneous out-of-memory situations. This vulnerability is unlikely for two reasons:

- If the generated code itself runs out of memory, it will automatically start freeing blocks from the free-list.
- For neptune-triton specifically, the sizes of memory allocations is not directly influenced by the input data except for its immediate size, and the sizes of allocations is the only dynamic input that affects the behaviour of the allocator.

To be completely safe, the C API function `futhark_context_clear_caches()` can be used to clear the free-list inbetween calls of Futhark entry points. Note that this will slow down future calls of entry points until the free-list has been sufficiently repopulated.

In the generated code, GPU memory is read and written in two different ways:

- From the CPU, through the OpenCL API functions `clEnqueueReadBuffer()` and `clEnqueueWriteBuffer()`. These are implicitly bounds-checked by the OpenCL API, and so cannot be used to corrupt GPU memory.
- From device code through ordinary C pointer reads and writes. These are as unchecked as any other C pointer operation, and we therefore depend on the Futhark compiler only generating correct accesses, and guarding any risky accesses with explicit bounds checks. Since GPU memory is separate from CPU memory (and modern GPUs have memory protection), it is very unlikely that memory errors on the GPU could be used to compromise the system. Further, the explicit array indexing in `poseidon.fut` is not based on the values of potentially malicious input.

Call graph analysis GNU cflow is a tool that determines the static call graph for a C program. Using the command `cflow -T a.c` we can produce an ASCII art tree that, for each (CPU) function in the generated code, shows us which other functions are called. As the generated code makes no use of function pointers, this static call graph is reliable. From the call graph we can conclude the following:

- File system functions (`fopen()` and similar) are used only in the context initialisation functions. By inspection, we determine that dynamically, these file operations are only performed in non-default configurations (e.g. the use of `futhark_context_config_dump_program_to()` as above).
- Network functions such as `accept()` are never used.
- Dangerous functions such as `system()` are never used.
- There are many calls to OpenCL API functions in the generated entry points, but none of these are documented in the OpenCL spec to have any file system/network interactions. It is outside the scope of this report to investigate whether any particular implementation of OpenCL correctly implements this property. It is likely that some OpenCL implementations perform some filesystem-based caching of compiled GPU code, but this should have no security ramifications.

Static call graph analysis is not robust against actively malicious code, as it is not hard to hide calls to functions, but it shows that the generated code does appear to implement the design goal of not having any effect outside the current process's memory space.

Futhark language issues

This section covers pitfalls and infelicities in Futhark as a language, and to which degree neptune-triton is affected.

Undefined behaviour

For performance reasons, some Futhark expressions are directly translated to underlying C expressions. In particular, bit shifts are translated directly to C bit shifts, with their sometimes unexpected semantics. In at least one case (#1056) this has confused a user who shifted a 32-bit number by 32 places, which has no effects on x86.

The rest of this section investigates all instances of bit-shifting in neptune-triton.

`poseidon.fut` itself contains one bit shift of a 32-bit integer in the `mk_arity_tag` function, where a literal `1` is shifted left by `arity` bits, where `arity` is ultimately defined in the `Params` module passed to the `make_hasher` parametric module. By inspection, we find that `arity` is in practice always 2, 4, 8, or 11.

The github.com/filecoin-project/fut-ff library contains two occurrences of *bit rotation*, implemented by shifting left by 1 bit and right by the number of bits in the type minus one.

We conclude that neptune-poseidon does not exhibit any undefined behaviour at the Futhark level.

Nondeterministic behaviour

While Futhark is semantically a deterministic programming language, efficiency concerns forces it to have some constructs that must be used in a certain way, with the risk of nondeterministic behaviour if they are misused. In most cases, the compiler cannot detect misuse. The potentially problematic constructs and their requirements are:

- `reduce op x` and `scan op x`: requires that `op` is an associative operator, and that `x` is a neutral element for `op`.
- `reduce_comm op x`: requires that `op` is an associative *and* commutative operator, and that `x` is a neutral element for `op`.
- `scatter dest is xs`: requires that whenever `is` contains in-bounds duplicates, the corresponding values in `xs` must also be identical.
- `reduce_by_index op x is xs`: requires that `op` is an associative *and* commutative operator with `x` as its neutral element.

By inspection, we find that `poseidon.fut` (including its dependencies) uses these constructs only in one definition contained in the `make_hasher` parametric module (linewrapped to fit within margins):

```
let scalar_product (a: elements) (b: elements) =
  Field.(reduce_comm (+) zero (map2 (*) a b))
```

This is a reduction with the operator `+` and neutral element `zero` from the `Field` module. `Field` is a *module parameter*, so we must ensure that any instantiation of `make_hasher` provides a `Field` module where `+` is associative with `zero` as its neutral element. By inspection, we find that `make_hasher` is only ever applied to the field module `bls12_381`, which is an application of the parametric module `big_field` from the github.com/filecoin-project/fut-ff library. There we find the following definitions (linewrapped to fit within margins):

```
let zero: t = LVec.replicate M.zero

let (a: t) + (b: t) =
  if p_is_small then add_cheap a b else add_expensive a b
```

For `bls12_381`, `p_is_small` is true, so if suffices to show that `add_cheap` is associative and commutative. This is shown in two ways:

- General functional tests of `neptune-triton` as a whole implies that finite field addition is functionally correct, and any functionally correct definition of addition must be associative and commutative by necessity.
- Specific tests for commutativity and tests for associativity on ten thousand randomly generated applications.

Unsafe code

Futhark supports an `unsafe` attribute (similar to a pragma or annotation in other languages) that can be used for locally turning off bounds checking when accessing arrays. This can trivially be used to subvert memory safety, and thus Futhark's normal safety guarantees do not apply to code that uses `unsafe`. Fortunately, `poseidon.fut` does not use `unsafe`, and provides instructions on how a reviewer of the code can confirm this for themselves.

Safety of genfut

This section reviews whether the code generated by the genfut tool uses the Futhark C API correctly. As mentioned previously, genfut is a tool that automatically generates Rust bindings to the C code emitted by the Futhark compiler. This review does not concern genfut in general, but only the specific Rust code that is produced by genfut for neptune-triton, and which can be found in the library/neptune-triton/src directory. It is possible that genfut has defects that affect other programs, but not neptune-triton.

The C code generated by Futhark is a fairly typical C API, meaning that resource management is manual, and the usual errors (use after free and leaking resources) are easy to make. The main contribution of genfut is wrapping this code in a RAII-style safe Rust API. Note that genfut also exposes the raw C API in the generated file `bindings.rs`. The safety properties are weakened if this module is used directly by application code. Note also that `bindings.rs` is generated not by genfut directly, but by use of rust-bindgen; a standard tool for generating low-level Rust wrappers of C libraries. We assume that rust-bindgen itself is trustworthy, and leave it (and `bindings.rs`) outside the scope of this report.

Genfut works by creating a Rust struct `FutharkContext` representing the Futhark context (encapsulating any necessary GPU state and such), where entry points are then methods defined on the struct. This is a good idea, as it ties the lifetime of the context directly to the lifetime of a Rust object, which prevents calling entry points on a deallocated context. However, the abstraction has some weaknesses.

Fails to free context object (*severity: low*)

The implementation of `FutharkContext` does not at any point call `futhark_context_free()` or `futhark_context_config_free()`. This means that the actual Futhark context object is not freed when the `FutharkContext` Rust object is destroyed. GPU resources may thus be leaked until the process ends entirely and they are reclaimed by the operating system. However, in the context in which neptune-triton is used, the lifetime of the `FutharkContext` is exactly the lifetime of the process, so the practical severity of this issue is low.

To avoid problems in the future, we recommend adding an implementation of the `Drop` trait for `FutharkContext`, which should call `futhark_context_free()` and `futhark_context_config_free()` (in that order). If this is done, then the implementation of the `Copy` and `Clone` traits should probably be removed.

Reported upstream.

General resource management (*severity: low*)

Genfut generates Rust wrappers for all Futhark-level objects exposed through the C API. Specifically, arrays and "opaque objects". The latter are Futhark values that are records, tuples, sum types, or otherwise uninspectable by non-Futhark code. The `Drop` trait is correctly implemented for all wrapper types: `Array_i64_1d`, `Array_i64_2d`, `Array_u64_1d`, `Array_u64_2d`, `Array_u64_3d`, `FutharkOpaqueP11State`, `FutharkOpaqueP2State`, `FutharkOpaqueP8State`, `FutharkOpaqueS11State`, `FutharkOpaqueS2State`, `FutharkOpaqueS8State`, and `FutharkOpaqueT864MState`.

To avoid use-after-free errors, it would be preferable to tie the lifetime of these types to the lifetime of their context.

Error handling

The C API indicates errors by returning an exit code. This is notoriously error-prone. The code generated by genfut automatically checks for these error codes and uses a `Result` type to indicate successful execution of entry points.

No error-checking on deallocation (*severity: low*)

At the C API level, *freeing* a Futhark value can also return an error code. This is not exposed in the Rust code, where the return code of the freeing functions is never checked. This is likely because the implicit resource management implied by the `Drop` trait is incompatible with checking deallocations for errors. Futhark has no documented reason for why deallocation should ever fail, and in practice it will only occur due to compiler bugs, or perhaps driver or hardware malfunction, and never under normal circumstances. Yet, it would be better to check the error code, and perform a Rust `panic` in the event of deallocation error.

Reported upstream.

No error-checking when copying arrays to CPU (*severity: medium*)

In the generated Rust code that copies GPU arrays to CPU memory, the error value returned by the "values" function (in this case for 2D `i64` arrays) is not checked.

Reported upstream.

No error-checking when creating context (*severity: low*)

After creating a Futhark context with the C function `futhark_context_new()`, the caller must immediately call `futhark_context_get_error()` to determine whether initialisation succeeded. Only if `futhark_context_get_error()` returns `NULL` should the context be used.

Reported upstream.

Correct synchronisation when copying arrays to CPU

The generated C functions that copies from GPU to CPU is asynchronous, so explicit calls to the `futhark_context_sync()` functions are necessary to ensure the copies are done, before the values can be relied upon. Neglecting this is an easy way to get nondeterministic errors. Genfut synchronises correctly.

Supply chain risks

Even if neptune-triton itself is correctly written, it is still vulnerable to *supply-chain attacks*, where an attacker compromises the tools, compilers, libraries, or hardware that it uses. In practice, at some point we have to trust our foundations (see Ken Thompson's Reflections on Trusting Trust), and this report will not try to assess the risk of malicious hardware or GPU drivers. However, there are more plausible supply-chain *risks* that are worth briefly identifying.

Malicious changes to the Futhark compiler

Many users of the Futhark compiler will use a binary release, which are available on the Futhark website. There is no mechanically verified guarantee that these binary releases correspond to any source code available in the Git repository. Neptune-triton documents the precise Git commit hash corresponding to the used version of Futhark (currently 0.16.2). This will allow recompilation of the appropriate version of the Futhark compiler, meaning that one does not have to trust opaque binaries. The Futhark compiler is itself compiled with the Glasgow Haskell Compiler, which does not guarantee bit-reproducible builds, and so we cannot expect to completely reproduce the binary releases. However, we can check whether a manually compiled Futhark 0.16.2 generates the same code for `poseidon.fut` as the released binary.

Note that this does not prevent malicious modification of the Futhark compiler itself; it only guarantees that the malicious code is either visible in the Git history, or that we can easily discover that the binary releases have been modified.

Halting development of the Futhark compiler

Malicious changes to the Futhark compiler is very unlikely. A greater risk is for development and maintenance of Futhark to stop altogether. Since Futhark is a small project and a relatively obscure language, this is a plausible risk that we must consider. Noted that all current developers of the Futhark compiler currently have several years stable funding, so there is no *urgent* risk.

If maintenance stops In the short term, the cessation of Futhark maintenance would not be a significant problem for neptune-triton. First, Futhark itself is open source software, so maintenance could be taken over by others. Second, the generated code uses a stable and long-lived subset of the OpenCL API.

In the extreme case, the parts of the generated code that are most likely to possibly need modification for new systems and their platform-specific quirks is the basic initialisation and runtime code. Fortunately, this part of the code is relatively easy to modify, either in the generated code itself or (preferably) in the Futhark compiler itself.

Avenues for migration off Futhark If Futhark were to stop being maintained, neptune-triton would in the medium-to-long term need to migrate to another language, likely hand-written OpenCL. It is not practical to use the Futhark-generated code as a starting point, as it is human-readable only in the strictest sense. A rewrite from scratch is the only realistic option. This is not an impossible task, as the entirety of the neptune-triton-specific Futhark code takes up only 1100 source lines of code (not counting comments and blank lines).