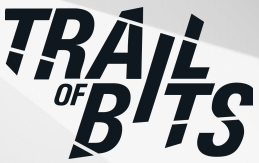

EleutherAI, Hugging Face Safetensors Library

Security Assessment

May 3, 2023

Prepared for:

Stella Biderman

EleutherAI

Nicolas Patry

Hugging Face

Garry Jean-Baptiste

Stability AI

Prepared by: **Fredrik Dahlgren, Suha Hussain, Heidy Khlaaf, and Evan Sultanik**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes, the Linux kernel, and the free AlgoVPN software.

We specialize in software testing, code review, and threat modeling projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

In addition to dedicated teams focusing on application security, cryptography, blockchain security, and emerging platforms security, Trail of Bits has a machine learning (ML) practice that creates tools and techniques for the exploration of new attack surfaces and failures that can lead to the degradation of model performance, exploitation of ML system assets, and manipulation or lack of robustness of resulting ML outputs. Trail of Bits has also created and maintains more than 200 free and open-source tools (available in our GitHub repositories) and offers research and engineering services for the public and private sectors.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, LangSec, the Linux Security Summit, the O'Reilly Security Conference, PyCon, RWC, REcon, and SummerCon.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

EleutherAI engaged Trail of Bits to review the security of the Hugging Face `safetensors` library. From March 20 to March 24, 2023, a team of two consultants conducted a security review of the client-provided source code, with two person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with access to both source code and documentation for the `safetensors` library. We performed static and dynamic testing of the codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	3
Low	3
Informational	3
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	9
Patching	1

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-SFTN-1**
  The tensor metadata in the safetensors file header is not sufficiently validated against the size of the data section, which contains the actual tensor data. As a result, it is possible to append arbitrary data to a safetensors file and have the file remain valid.

  An attacker could create a polyglot safetensors file (i.e., a safetensors file that is simultaneously valid under another file format such as PDF, ZIP, Keras, or TFRecords). This is problematic because two different machine learning pipelines that read safetensors model files could parse the same maliciously crafted file as two different models.

Summary of Recommendations

The `safetensors` library and the corresponding file format help provide users with a safer and more performant file format for machine learning models. Trail of Bits recommends that Hugging Face address the findings detailed in this report and take the following additional steps to further improve the library:

- The safetensors file format is currently underspecified, and multiple issues in the report reflect different aspects of this wider issue. Some questions that are currently not answered by the existing specification include the following:

    a. **Which subset of JSON is permitted?** The JSON RFC and standard are underspecified, and each implementation treats specification ambiguities differently. For example, the `serde_json` crate used to parse JSON rejects JSON with duplicate keys, but other JSON parsers typically allow duplicate keys, with no standard behavior across parsers.

    JSON parsers are inconsistent due to the underspecified nature of the JSON RFC and standard, which can lead to differentials between safetensors implementations. Currently, the Hugging Face safetensors reference implementation delegates JSON parsing responsibility to the `serde` library. If the behavior of `serde` ever changes (e.g., if it changes its default behavior when handling duplicate keys), then this will change the semantics of safetensors parsing. Such a change is not unheard of. For example, the `UltraJSON` library (used by the independent, pure Python implementation `pysafetensors`) has internally inconsistent handling of Unicode escapes throughout its version history.

    b. **Are zero-length tensors allowed?** The current implementation allows equal start and end tensor offsets and zero-length shapes (or alternatively, shapes where one of the dimensions is zero).

    c. **Are not-a-number (NaN) or infinity (Inf) floating point values allowed?** The library does not validate the data in the data section, so floating point values could be exceptional values such as NaN or ±Inf.

    d. **Should each byte in the data section correspond to a tensor defined by the header?** The current implementation checks that tensors are laid out consecutively in the data section but fails to ensure that there is no data following the last tensor. This means that arbitrary data can be appended to a safetensors file without rendering it invalid.

All these issues could lead to parser differentials between different safetensors parsers.

In the short term, Trail of Bits recommends that Hugging Face document the safetensors file format and make this documentation widely available. We also recommend that test cases be made available to ensure that different parsers agree on what is a valid safetensors file.

In the long term, Trail of Bits recommends that Hugging Face abandon the use of JSON in the next version of the safetensors file format and replace it with a custom binary representation. This would avoid the many ambiguities introduced by the JSON format and would also allow the safetensors file format to be fully specified in a DSL such as Kaitai Struct, which automatically generates parsers in any language.

- It is common for file formats to include a signature at the start of the file to make files easier to recognize for applications handling multiple different file formats. Including a signature also helps prevent the construction of polyglot files that could be interpreted differently by different applications. Additionally, it is common to include a file format version that allows the format to be updated and extended over time. The safetensors file format has neither of these protections.

  We recognize that the safetensors file format is already used to serialize and transmit machine learning models. However, we still recommend that Hugging Face add a signature and version number to the format before wider adoption across the community.

- The `safetensors` library contains unit tests with good coverage of the implementation's happy paths. However, there are too few tests exercising the codebase's failure paths, so the behavior of the codebase on adversarial inputs is undertested. We recommend that Hugging Face extend the test suite with tests for various edge cases and invalid inputs. To improve coverage, we recommend using property testing frameworks such as `proptest` and `Hypothesis`. (For an example on how to use `proptest`, see appendix D. For an example on how to use `Hypothesis`, see appendix E)

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Brooke Langhorne, Project Manager
brooke.langhorne@trailofbits.com

The following engineers were associated with this project:

Fredrik Dahlgren, Consultant
fredrik.dahlgren@trailofbits.com

Suha Hussain, Consultant
suha.hussain@trailofbits.com

Heidy Khlaaf, Consultant
heidy.khlaaf@trailofbits.com

Evan Sultanik, Consultant
evan.sultanik@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
March 15, 2023	Pre-project kickoff call
March 27, 2023	Delivery of report draft
March 27, 2023	Report readout meeting
April 26, 2023	Fix review
May 3, 2023	Delivery of final report

Project Targets

The engagement involved a review and testing of the following target.

safetensors

Repository	https://github.com/huggingface/safetensors
Version	`5c1d366813e46c6f9f2c71aa8b89e0c916a92b2f`
Type	Rust, Python
Platform	Multiple

Project Goals

The engagement was scoped to provide a security assessment of the `safetensors` library developed by Hugging Face. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the `safetensors` library contain vulnerabilities that an attacker could leverage to remotely execute code or execute denial-of-service attacks?
- Does the `safetensors` library perform sufficient input validation during deserialization?
- Does the codebase have sufficient test coverage, and are there tests for invalid or adversarial inputs?
- Is the safetensors file format sufficiently specified, or are there unclarities in the specification that could lead to parser differentials or facilitate the creation of polyglots?

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

- We ran static analysis tools on the `safetensors` library and the Python bindings to identify code quality issues, outdated dependencies, and dependencies containing known vulnerabilities.
- We reviewed the test coverage for the unit and integration tests shipped with the library.
- We manually reviewed the `safetensors` library and Python bindings, focusing on data validation and memory safety issues.
- We wrote property tests for the `safetensors` library and Python bindings to ensure that the serialization and deserialization processes are inverses of each other.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Clippy	An open-source Rust linter used to catch common mistakes and unidiomatic Rust code	Appendix C
Dylint	An open-source Rust linter developed by Trail of Bits to identify common code quality issues and mistakes in Rust code	Appendix C
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix C
`cargo-audit`	A Cargo plugin for reviewing project dependencies for known vulnerabilities	Appendix C
`cargo-geiger`	A tool that lists statistics related to the use of unsafe Rust code in a Rust crate and all its dependencies	Appendix C
`cargo-outdated`	A Cargo plugin that identifies project dependencies with newer versions available	Appendix C
`cargo-llvm-cov`	A Cargo plugin for generating LLVM source-based code coverage	Appendix C
`pip-audit`	A tool developed by Trail of Bits and Google for	Appendix C

	scanning Python environments for packages with known vulnerabilities	
`mypy`	A Python type checker that can statically infer many type errors	Appendix C

Areas of Focus

Our automated testing and verification work focused on detecting the following issues:

- General code quality issues and unidiomatic code patterns
- Issues related to error handling and the use of `unwrap` and `expect`
- Moderate use of unsafe code
- Poor unit and integration test coverage
- General issues with dependency management and known vulnerable dependencies

Test Results

The results of this focused testing are detailed below.

safetensors library and Python bindings. We ran several static-analysis tools such as Clippy, Dylint, and Semgrep to identify potential code quality issues in the codebase. We then used the Cargo plugins `cargo-geiger`, `cargo-outdated`, and `cargo-audit` to detect the use of unsafe Rust and review general dependency management practices. Finally, we ran `cargo-llvm-cov` to review the unit and integration test coverage for the library and Python bindings.

Property	Tool	Result
The project adheres to Rust best practices by fixing code quality issues reported by linters such as Clippy.	Clippy	**Passed**
The project does not contain any vulnerable code patterns identified by Dylint.	Dylint	**Passed**
The project's use of panicking functions such as `unwrap`	Semgrep	TOB-SFTN-6

and `expect` is limited.		
The project contains a reasonable amount of unsafe code for what the implementation is trying to achieve.	`cargo-geiger`	**Passed**
To avoid technical debt, the project continually updates dependencies as new versions are released.	`cargo-outdated`	**Passed**
The project does not depend on any libraries with known vulnerabilities.	`cargo-audit`	**Passed**
	`pip-audit`	TOB-SFTN-9
All components of the codebase have sufficient test coverage.	`cargo-llvm-cov`	TOB-SFTN-5
The Python bindings do not have any type confusion errors.	`mypy`	Appendix G

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The `safetensors` library relies mainly on unchecked arithmetic, even when performing calculations on untrusted inputs. This could lead to reliability issues or panics when the library is used to parse untrusted data.	**Weak**
Auditing	The library contains no auditing or event logging.	**Not Applicable**
Complexity Management	The codebase is well structured, and the code uses the Rust trait system and Rust macros to avoid code duplication.	**Satisfactory**
Configuration	The library has no configuration.	**Not Applicable**
Data Handling	The `safetensors` library performs some validation of the input data during deserialization, but there are several locations where validation is insufficient or completely missing. This may lead to reliability issues or panics when the library is used to parse untrusted input.	**Weak**
Documentation	The codebase is sufficiently documented, but in some cases, the documentation is outdated and refers to features that are missing from the version under review. We also found that the documentation of the safetensors file format is lacking. This could lead to parser differentials and facilitate the creation of safetensors	**Weak**

	polyglots (files that are at least two valid file types at once) and so-called schizophrenic files (models that are interpreted differently by different safetensors implementations).	
Maintenance	The codebase generates very few warnings when using static analysis tools such as Clippy and Dylint. There are no outdated dependencies or dependencies with known vulnerabilities that are exploitable from `safetensors`. However, the majority of the Python dependencies are not pinned to a version, so depending on the environment, vulnerable versions could be installed.	**Moderate**
Memory Safety and Error Handling	The `safetensors` library contains no unsafe code, and the Python bindings contain only a minimal amount of unsafe code related to memory mapping. Errors are expressive, using custom Rust enums for error types. The library typically propagates errors to the caller and contains limited use of panicking functions such as `unwrap` and `expect`.	**Strong**
Testing and Verification	The library has good test coverage and also contains fuzzers that exercise deserialization and the Python bindings. However, the library does not contain enough tests for adversarial inputs that target the codebase's failing code paths, so the library's behavior on malicious inputs is largely untested.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Tensor offsets are not checked against the total size of the tensor data	Data Validation	Medium
2	Tensor size calculations may overflow in Metadata::validate	Data Validation	Medium
3	The safetensors library allows zero-sized tensors	Data Validation	Informational
4	The SliceIterator type does not validate tensor indexers against the tensor shape	Data Validation	Low
5	Insufficient test coverage against adversarial inputs	Data Validation	Medium
6	Serialization can panic on malformed JSON	Data Validation	Informational
7	Underspecified JSON behavior can lead to parser differentials	Data Validation	Low
8	PyTorch conversion utility is vulnerable to arbitrary code execution	Data Validation	Undetermined
9	Python dependencies are not semantically versioned	Patching	Low
10	The safetensors library does not check for exceptional values	Data Validation	Informational

Detailed Findings

1. Tensor offsets are not checked against the total size of the tensor data	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SFTN-1
Target: `safetensors/safetensors/src/tensor.rs`	

Description

The `Metadata::validate` function validates that the tensors are laid out consecutively by checking the start and end offsets for each tensor. However, these offsets are not checked against the total size of the input buffer, so it may be possible to pass an input buffer that is too large or too small to `Safetensors::deserialize`.

If the input buffer is too large, it could allow the construction of polyglots, which could be used to confuse downstream consumers of the API. If the input buffer is too small, it would lead to a panic when the corresponding tensor is accessed in `Safetensors::tensor`.

```
pub fn tensor(&self, tensor_name: &str) -> Result<TensorView<'_>, SafeTensorError> {
    if let Some(index) = &self.metadata.index_map.get(tensor_name) {
        if let Some(info) = &self.metadata.tensors.get(**index) {
            Ok(TensorView {
                dtype: info.dtype,
                shape: info.shape.clone(),
                data: &self.data[info.data_offsets.0..info.data_offsets.1],
            })
        } else {
            Err(SafeTensorError::TensorNotFound(tensor_name.to_string()))
        }
    } else {
        Err(SafeTensorError::TensorNotFound(tensor_name.to_string()))
    }
}
```

Figure 1.1: Since the tensor offsets have not been checked against the total size of the input, indexing into the `SafeTensor` `data` field may panic.

Exploit Scenario 1

A malicious user appends a Keras file to a safetensors file, thereby creating a polyglot file that is simultaneously a safetensors file and a Keras file. The file is recognized as valid but is loaded differently by different applications because some applications recognize and load the file as a Keras file, and others recognize and load it as a safetensors file.

To illustrate this issue, this report is simultaneously a valid PDF and a valid ZIP file. Unzip this report to reveal four example safetensors polyglots with the Keras native, PDF, ZIP, and TFRecords file formats (see appendix F).

Exploit Scenario 2

A malicious user creates a safetensors file with an empty data section. Since the size of the data section is not validated when the file is read, the file is deserialized without issues. However, when the first tensor data is accessed, the implementation indexes out of bounds and panics.

Recommendations

Short term, modify the `Metadata::validate` function so that it validates tensor offsets against the size of the data section during deserialization. The function should ensure that all offsets fall within the data section and that all data in the data section corresponds to tensors defined by the header.

Long term, document the safetensors file format to describe how parsers should handle files with extra data appended to the data section.

2. Tensor size calculations may overflow in Metadata::validate

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SFTN-2
Target: `safetensors/safetensors/src/tensor.rs`	

Description

The `Metadata::validate` method computes the serialized size of each tensor by multiplying the size of the data by the product of the tensor's dimensions. Since Rust does not check for integer overflows in release builds, this computation may overflow and produce the wrong result.

```
fn validate(&self) -> Result<(), SafeTensorError> {
    let mut start = 0;
    for (i, info) in self.tensors.iter().enumerate() {
        let (s, e) = info.data_offsets;
        if s != start || e < s {
            let tensor_name = self
                .index_map
                .iter()
                .filter_map(|(name, &index)|
                    if index == i { Some(&name[..]) } else { None })
                .next()
                .unwrap_or("no_tensor");
            return Err(SafeTensorError::InvalidOffset(tensor_name.to_string()));
        }
        start = e;
        let nelements: usize = info.shape.iter().product();
        let nbytes = nelements * info.dtype.size();
        if (e - s) != nbytes {
            return Err(SafeTensorError::TensorInvalidInfo);
        }
    }

    Ok(())
}
```

Figure 2.1: The computations of `nelements` and `nbytes` may overflow and produce an invalid result.

Exploit Scenario

A malicious user creates an invalid safetensors file where the calculation of the number of elements overflows but the computed number of bytes still matches the given tensor

offsets. As a result, the file is deserialized without issues but panics on an out-of-bounds access when the tensor is used together with the SliceIterator API.

Recommendations

Short term, use checked arithmetic for all arithmetic operations during deserialization, and have the code validate the result whenever possible.

Long term, extend the test suite to test for more types of invalid safetensors files. Consider using property testing to obtain better test coverage.

3. The safetensors library allows zero-sized tensors

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SFTN-3
Target: `safetensors/safetensors/src/tensor.rs`	

Description

The tensor metadata is validated by `Metadata::validate`. This method checks the start and end offsets `(s, e)` specified for each tensor and rejects the input if `e` is less than `s`. However, the function will accept empty tensors where `s` is equal to `e` if the tensor shape contains `0`.

```
fn validate(&self) -> Result<(), SafeTensorError> {
    let mut start = 0;
    for (i, info) in self.tensors.iter().enumerate() {
        let (s, e) = info.data_offsets;
        if s != start || e < s {
            let tensor_name = self
                .index_map
                .iter()
                .filter_map(|(name, &index)|
                    if index == i { Some(&name[..]) } else { None })
                .next()
                .unwrap_or("no_tensor");
            return Err(SafeTensorError::InvalidOffset(tensor_name.to_string()));
        }
        start = e;
        let nelements: usize = info.shape.iter().product();
        let nbytes = nelements * info.dtype.size();
        if (e - s) != nbytes {
            return Err(SafeTensorError::TensorInvalidInfo);
        }
    }

    Ok(())
}
```

Figure 2.1: If `s` is equal to `e` and `info.shape` contains `0`, the tensor will be accepted as valid.

The method also allows empty shapes (with length `0`) if `e` `minus` `s` equals `info.dtype.size()`.

Recommendations

Short term, have the code ensure that `e` is strictly greater than `s` and check that `info.shape()` is non-empty in `Metadata::validate`.

Long term, improve the documentation for the safetensors file format to clarify whether it allows zero-length tensors (and zero-length shapes).

4. The SliceIterator type does not validate tensor indexers against the tensor shape

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SFTN-4
Target: `safetensors/safetensors/src/slice.rs`	

Description

The `SliceIterator::new` method takes a set of intervals (given as `TensorIndexers`) as input. The iterator then iterates through the values in the hypercube defined by the given intervals. However, the `SliceIterator` constructor fails to validate the interval endpoints.

```
// [...]
for (i, &shape) in view.shape().iter().enumerate().rev() {
    if i >= slices.len() {
        // We are  not slicing yet, just increase the local span
        newshape.push(shape);
    } else {
        let slice = &slices[i];
        let (start, stop) = match slice {
            TensorIndexer::Narrow(Bound::Unbounded, Bound::Unbounded) =>
                (0, shape),
            TensorIndexer::Narrow(Bound::Unbounded, Bound::Excluded(stop)) =>
                (0, *stop),
            TensorIndexer::Narrow(Bound::Unbounded, Bound::Included(stop)) =>
                (0, *stop + 1)
            // [...]
        };
        newshape.push(stop - start);
        if indices.is_empty() {
            if start == 0 && stop == shape {
                // We haven't started to slice yet, just increase the span
            } else {
                let offset = start * span;
                let small_span = stop * span - offset;
                indices.push((offset, offset + small_span));
            }
        } else {
            let mut newindices = vec![];
            for n in start..stop {
                let offset = n * span;
                for (old_start, old_stop) in &indices {
                    newindices.push((old_start + offset, old_stop + offset));
                }
            }
```

```
            indices = newindices;
        }
    }
    span *= shape;
}
// [...]
```

Figure 4.1: The interval endpoints `start` and `stop` are not checked against the tensor shape.

This could lead to a panic during iteration in `SliceIterator::next` or result in invalid data being passed back to the user.

```
impl<'data> Iterator for SliceIterator<'data> {
    type Item = &'data [u8];

    fn next(&mut self) -> Option<Self::Item> {
        // TODO We might want to move the logic from `new`
        // here actually to remove the need to get all the indices
        // upfront.
        let (start, stop) = self.indices.pop()?;
        Some(&self.view.data()[start..stop])
    }
}
```

Figure 4.2: Since the original interval endpoints are not checked against the dimensions of the tensor, `SliceIterator::next` may panic when indexing into the tensor data.

Since both `start` and `stop` may be arbitrarily large in `SliceIterator::new`, this issue could also lead to overflows when the offsets into the data sections are calculated in `SliceIterator::new`.

The same type of issue is present in the `get_tensor` implementation in the Python bindings.

```
pub fn get_tensor(&self, name: &str) -> PyResult<PyObject> {
    let tensors = self.metadata.tensors();
    let info = tensors.get(name).ok_or_else(|| {
        SafetensorError::new_err(format!("File does not contain tensor {name}",))
    })?;

    match &self.storage.as_ref() {
        Storage::Mmap(mmap) => {
            let data =
                &mmap[info.data_offsets.0 + self.offset..
                      info.data_offsets.1 + self.offset];

            let array: PyObject = Python::with_gil(|py|
                PyByteArray::new(py, data).into_py(py));

            create_tensor(
```

```
                &self.framework,
                info.dtype,
                &info.shape,
                array,
                &self.device,
            )
        }
        // [...]
```

Figure 4.3: The Python bindings slice into the tensor data without validating the interval offsets.

Recommendations

Short term, for the interval endpoints (`start, stop`), have the `SliceIterator` constructor ensure that `start` is less than `stop` and that `stop` is less than the relevant dimension. Have the `get_tensor` function return an error if the validation fails.

Long term, replace all arithmetic on untrusted values with their checked counterparts.

5. Insufficient test coverage against adversarial inputs

Severity: **Medium**	Difficulty: **Not Applicable**
Type: Data Validation	Finding ID: TOB-SFTN-5
Target: `safetensors/safetensors/src/slice.rs`	

Description

The average line coverage for the `safetensors` crate is over 87%, which is sufficient, but the crate contains very few test cases exercising the implementation's failure paths. This means that the library's behavior on adversarial input is largely untested.

```
261  8      pub fn read_metadata<'in_data>(
262  8          buffer: &'in_data [u8],
263  8      ) -> Result<(usize, Metadata), SafeTensorError>
264  8      where
265  8          'in_data: 'data,
266  8      {
267  8          let buffer_len = buffer.len();
268  8          if buffer_len < 8 {
269  0              return Err(SafeTensorError::HeaderTooSmall);
270  8          }
271  8          let arr: [u8; 8] = [
272  8              buffer[0], buffer[1], buffer[2], buffer[3], buffer[4], buffer[5], buffer[6], buffer[7],
273  8          ];
274  8          let n: usize = u64::from_le_bytes(arr)
275  8              .try_into()
276  8              .map_err(|_| SafeTensorError::HeaderTooLarge)?;
277  8          if n > MAX_HEADER_SIZE {
278  1              return Err(SafeTensorError::HeaderTooLarge);
279  7          }
280
281  7          let stop = n
282  7              .checked_add(8)
283  7              .ok_or(SafeTensorError::InvalidHeaderLength)?;
284  7          if stop > buffer_len {
285  0              return Err(SafeTensorError::InvalidHeaderLength);
286  7          }
287  7          let string =
288  7              std::str::from_utf8(&buffer[8..stop]).map_err(|_| SafeTensorError::InvalidHeader)?;
289  7          let metadata: Metadata = serde_json::from_str(string)
290  7              .map_err(|_| SafeTensorError::InvalidHeaderDeserialization)?;
291  7          metadata.validate()?;
292  6          Ok((n, metadata))
293  8      }
```

Figure 5.1: Many of the failing code paths of `Safetensors::read_metadata` are untested.

Exploit Scenario

A user uploads a maliciously crafted safetensors model to the Hugging Face hub. When the model is parsed, the `safetensors` library panics because of a previously undetected edge case in the implementation.

Recommendations

Short term, write test cases that exercise the failure paths of the `safetensors` crate.

Long term, regularly run `cargo llvm-cov` to get test coverage data. Ensure that all functions are covered and that the crate includes ample tests against adversarial inputs.

6. Serialization can panic on malformed JSON

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SFTN-6
Target: `safetensors/safetensors/src/tensor.rs`	

Description

The serialization data preparation function `prepare` performs an unchecked call to `unwrap` on an option that will be `None` if the `data_info` mapping cannot be serialized to JSON.

```
fn prepare<
    S: AsRef<str> + Ord + std::fmt::Display,
    V: View, I: IntoIterator<Item = (S, V)>>
(
    data: I,
    data_info: &Option<HashMap<String, String>>,
) -> Result<(PreparedData, Vec<V>), SafeTensorError> {
    // [...]

    let mut hmetadata = Vec::with_capacity(data.len());
    // [...]

    let metadata: Metadata = Metadata::new(data_info.clone(), hmetadata)?;
    let mut metadata_buf = serde_json::to_string(&metadata).unwrap().into_bytes();
```

Figure 6.1: The call to unwrap on line 179 will panic if the JSON in `data_info` is malformed. (`safetensors/safetensors/src/tensor.rs#150–179`)

This finding's severity is set to informational because we could not devise an input to `data_info` that could not be serialized to JSON with `serde_json::to_string`. However, if such an input were discovered, the serialization would panic on line 179 of figure 6.1. A future update to `serde` could also change its behavior, inducing a panic. This is described in more detail in the following finding (TOB-SFTN-7).

Exploit Scenario

A metadata hashmap that cannot be encoded in JSON by the `serde` library is serialized. The `prepare` function panics on line 179, rather than returning an error.

Recommendations

Short term, have the `prepare` function check the output of `serde_json::to_string` for errors before unwrapping it.

Long term, consider abandoning the use of JSON in a subsequent version of safetensors. Since the JSON schema used by safetensors is relatively simple, use a custom binary encoding to allow safetensors to be fully specified in a DSL such as Kaitai Struct, which automatically generates parsers in any language.

7. Underspecified JSON behavior can lead to parser differentials

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SFTN-7
Target: `safetensors/safetensors/src/tensor.rs`	

Description

The JSON RFC, the JSON standard, and the safetensors documentation do not explicitly specify how to handle duplicate keys in the JSON dictionary. The safetensors reference implementation uses `serde` for JSON parsing, which rejects JSON inputs with duplicate keys. This behavior is not common; most other JSON parsers silently keep either the first or last duplicate. Python's built-in JSON parser does the latter and keeps the last key-value pair in the event of a collision. On the other hand, the popular high-performance Golang JSON parser `buger/jsonparser` has first-key precedence, so it would result in different values than the Python parsers.

There are already independent implementations of the safetensors file format. For example, there is a pure Python parser and associated safetensors JSON schema validator. As shown in figure 7.1, these implementations use either the built-in Python JSON parser or `UltraJSON` library, both of which accept JSON with duplicate keys without raising a warning.

```
 7    try:
 8          import ujson as json
 9    except ImportError:
10          import json
```

Figure 7.1: The pure Python safetensors parser uses different JSON semantics than the reference implementation. (pysafetensors/pysafetensors/kaitai/safe_tensors_parsed_header.py#7–10)

The risk of differentials between safetensors implementations is not solely based on discrepancies in duplicate key handling. JSON implementations have historically differed on interpretation of Unicode escapes, large numbers, and representations of infinity (Inf) and not-a-number (NaN) (see references below).

Exploit Scenario

A safetensors model that contains duplicate keys is created. This model is rejected by the Hugging Face reference implementation but accepted and even validated by third-party tools such as `pysafetensors`. Another third-party parser uses `buger/jsonparser` for

JSON deserialization, allowing users to construct safetensors model files that load one set of tensors in one implementation and a completely different set of tensors in another implementation.

Recommendations

Short term, create a rigorous file format specification that explicitly prohibits duplicate JSON keys. File bug reports with third-party tools that do not conform to the specification.

Long term, consider abandoning the use of JSON in a subsequent version of safetensors. Since the JSON schema used by safetensors is relatively simple, use a custom binary encoding that allows safetensors to be fully specified in a DSL such as Kaitai Struct, which automatically generates parsers in any language.

References

- Jake Miller. *An Exploration of JSON Interoperability Vulnerabilities*, Bishop Fox blog. February 25, 2021.
- Nicolas Seriot. *Parsing JSON is a Minefield*, Nicolas Seriot's blog. October 26, 2016.

8. PyTorch conversion utility is vulnerable to arbitrary code execution

Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-SFTN-8
Target: `safetensors/bindings/python/convert.py`	

Description

The `convert_file` function uses the PyTorch `torch.load()` method, which is known to be insecure. The method's `weights_only` parameter is not set to `True`, which means that the `pickle` module is used to load the model in an unrestricted fashion. It is thus possible to construct malicious pickle data that will execute arbitrary code during unpickling when converting a PyTorch model to a safetensors format.

```
112    def convert_file(
113        pt_filename: str,
114        sf_filename: str,
115    ):
116        loaded = torch.load(pt_filename, map_location="cpu")
117        if "state_dict" in loaded:
118            loaded = loaded["state_dict"]
```

Figure 8.1: The `torch.load()` method is used unsafely since the `weights_only` parameter is not set to `True`. (`safetensors/bindings/python/convert.py#112–118`)

Exploit Scenario

An attacker uploads a maliciously crafted PyTorch model to the Hugging Face hub. Since the conversion script invokes `torch.load()` in an unsafe manner, the model allows the attacker to execute arbitrary code on the hub when the model is converted to a safetensors file.

We do not know if the script is currently deployed on the Hugging Face hub or, if it is deployed, what security mitigations are in place to protect the system against this type of attack, so the severity of this issue is marked as undetermined.

Recommendations

Short term, add documentation warning users not to run the `convert.py` script on untrusted data.

The current documentation for this script notes that the new safetensors file "is equivalent to `pytorch_model.bin` but safe in the sense that no arbitrary code can be put into it."

However, it does not warn users that it is still possible for an attacker to execute arbitrary code on the system where the conversion tool is used.

Long term, if possible, change the use of `torch.load()` to enable `weights_only`. Alternatively, remove the script from the repository.

9. Python dependencies are not semantically versioned

Severity: **Low**	Difficulty: **Low**
Type: Patching	Finding ID: TOB-SFTN-9
Target: `safetensors/bindings/python/setup.py`	

Description

The majority of dependencies for the Python bindings are not constrained to a minimum version.

```
# IMPORTANT:
# 1. all dependencies should be listed here with their version requirements if any
_deps = [
    "black==22.3",
    "click==8.0.4",
    "flake8>=3.8.3",
    "flax",
    "h5py",
    "huggingface_hub",
    "isort>=5.5.4",
    "jax",
    "numpy",
    "setuptools_rust",
    "pytest",
    "pytest-benchmark",
    "tensorflow",
    "torch",
    "paddlepaddle",
]
```

Figure 9.1: The majority of Python dependencies are unversioned. (`safetensors/bindings/python/setup.py#7–25`)

The package resolution engine (e.g., `pip`) is free to install older, vulnerable versions of dependencies as necessary due to the Python environment's external requirements. For example, all but the latest version of `paddlepaddle` are vulnerable to a critical arbitrary code execution vulnerability.

Exploit Scenario

The `safetensors` library is installed with an older, vulnerable version of a dependency that is exploitable through the safetensors API.

Recommendations

Short term, pin to a version of each dependency that has no known vulnerabilities.

Long term, integrate `pip-audit` (e.g., using its official GitHub action), as well as Dependabot, into the `safetensors` CI pipeline.

10. The safetensors library does not check for exceptional values

Severity: **Informational**	Difficulty: **Not Applicable**
Type: Data Validation	Finding ID: TOB-SFTN-10
Target: `safetensors/safetensors/src/tensor.rs`	

Description
The `safetensors` library does not validate the individual tensors during deserialization. This means that floating point tensors may contain NaN or ±Inf values. This is not documented anywhere and may lead to issues for downstream consumers of the safetensors API.

Recommendations
Short term, document which security guarantees are provided by the library to downstream consumers.

Long term, provide documentation for the safetensors file format, as well as a compatibility test suite of examples that could be used to ensure that third-party parsers are compatible with the Hugging Face implementation.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Automated Testing

This section describes the setup of the automated analysis tools used during this audit.

Clippy

The Rust linter Clippy can be installed using `rustup` by running the command `rustup component add clippy`. Invoking `cargo clippy` in the root directory of the project runs the tool.

Dylint

Dylint is a linter for Rust developed by Trail of Bits. It can be installed by running the command `cargo install cargo-dylint dylint-link`. To run Dylint, we added a `Cargo.toml` file to the root of the repository with the following content.

```
[workspace.metadata.dylint]
libraries = [
  { git = "https://github.com/trailofbits/dylint", pattern = "examples/general/*" },
]
```

Figure C.1: Metadata required to run Dylint

To run the tool, run `cargo dylint --all --workspace`.

Semgrep

Semgrep can be installed using `pip` by running `python3 -m pip install semgrep`. To run Semgrep on a codebase, run `semgrep --config "<CONFIGURATION>"` in the root directory of the project. Here, `<CONFIGURATION>` can be a single rule, a directory of rules, or the name of a rule set hosted on the Semgrep registry.

We ran several custom Semgrep rules on the `safetensors` library. Because support for Rust in Semgrep is still experimental, we focused on identifying the following small set of issues:

- The use of panicking functions such as `assert`, `unreachable`, `unwrap`, and `expect` in production code (i.e., outside unit tests)

```
rules:
  - id: panic-in-function-returning-result
    patterns:
      - pattern-inside: |
          fn $FUNC(...) -> Result<$T> {
              ...
          }
      - pattern-either:
```

```
          - pattern: $EXPR.unwrap()
          - pattern: $EXPR.expect(...)
    message: |
      `expect` or `unwrap` called in function returning a `Result`.
    languages: [rust]
    severity: WARNING
```

Figure C.2: panic-in-function-returning-result.yaml

```
rules:
- id: unwrap-outside-test
  patterns:
  - pattern: $RESULT.unwrap()
  - pattern-not-inside: "
        #[test]
        fn $TEST() {
            ...
            $RESULT.unwrap()
            ...
        }
    "
  message: Calling `unwrap` outside unit test
  languages: [rust]
  severity: WARNING
```

Figure C.3: unwrap-outside-test.yaml

```
rules:
- id: expect-outside-test
  patterns:
  - pattern: $RESULT.expect(...)
  - pattern-not-inside: "
        #[test]
        fn $TEST() {
            ...
            $RESULT.expect(...)
            ...
        }
    "
  message: Calling `expect` outside unit test
  languages: [rust]
  severity: WARNING
```

Figure C.4: expect-outside-test.yaml

- The use of the `as` keyword in casting, which can silently truncate integers (e.g., casting `data.len()` to a u32 can truncate the input length on 64-bit systems)

```
rules:
- id: length-to-smaller-integer
  pattern-either:
```

```
  - pattern: $VAR.len() as u32
  - pattern: $VAR.len() as i32
  - pattern: $VAR.len() as u16
  - pattern: $VAR.len() as i16
  - pattern: $VAR.len() as u8
  - pattern: $VAR.len() as i8
  message: |
    Casting `usize` length to smaller integer size silently drops high bits
    on 64-bit platforms
  languages: [rust]
  severity: WARNING
```

Figure C.5: `length-to-smaller-integer.yaml`

- Unexpected comparisons before subtraction (e.g., ensuring that x is less than y before subtracting y from x), which may indicate errors in the code

```
rules:
- id: switched-underflow-guard
  pattern-either:
    - patterns:
        - pattern-inside: |
                if $Y > $X {
                    ...
                }
        - pattern-not-inside: |
                if $Y > $X {

                } else {
                    ...
                }
        - pattern: $X - $Y
    - patterns:
        - pattern-inside: |
                if $Y >= $X {
                    ...
                }
        - pattern-not-inside: |
                if $Y >= $X {

                } else {
                    ...
                }
        - pattern: $X - $Y
    - patterns:
        - pattern-inside: |
                if $Y < $X {
                    ...
                }
        - pattern-not-inside: |
                if $Y < $X {
```

```
            } else {
                ...
            }
    - pattern: $Y - $X
- patterns:
    - pattern-inside: |
            if $Y <= $X {
                ...
            }
    - pattern-not-inside: |
            if $Y <= $X {

            } else {
                ...
            }
    - pattern: $X - $Y
- patterns:
    - pattern-inside: |
            if $Y > $X {

            } else {
                ...
            }
    - pattern: $Y - $X
- patterns:
    - pattern-inside: |
            if $Y >= $X {

            } else {
                ...
            }
    - pattern: $Y - $X
- patterns:
    - pattern-inside: |
            if $Y < $X {

            } else {
                ...
            }
    - pattern: $X - $Y
- patterns:
    - pattern-inside: |
            if $Y <= $X {

            } else {
                ...
            }
    - pattern: $X - $Y
- patterns:
    - pattern: |
            if $X < $Y {
            }
            ...
```

```
message: Potentially switched comparison in if-statement condition
languages: [rust]
severity: WARNING
```

Figure C.6: `switched-underflow-guard.yaml`

cargo-audit

The `cargo-audit` Cargo plugin identifies known vulnerable dependencies in Rust projects. It can be installed using `cargo install cargo-audit`. To run the tool, run `cargo audit` in the crate root directory.

cargo-geiger

The `cargo-geiger` Cargo plugin provides statistics on the use of unsafe code in the project and its dependencies. The plugin can be installed using `cargo install cargo-geiger`. To run the tool, run `cargo geiger` in the crate root directory.

cargo-outdated

The `cargo-outdated` Cargo plugin identifies project dependencies with newer versions available. The plugin is installed by running `cargo install cargo-outdated`. To run the tool, run `cargo outdated` in the crate root directory.

cargo-llvm-cov

The `cargo-llvm-cov` Cargo plugin is used to generate LLVM source-based code coverage data. The plugin can be installed via the command `cargo install cargo-llvm-cov`. To run the plugin, simply run the command `cargo llvm-cov` in the crate root directory.

pip-audit

The `pip-audit` utility was developed by Trail of Bits and Google to detect known vulnerabilities in dependencies in Python environments. Install it with `pip3 install pip-audit` and run it in the root of the Python project with `pip-audit`. The output is a table enumerating all packages with known vulnerabilities installed in the environment.

mypy

The `mypy` Python type checker can statically infer many type errors. It uses both static type inference and optional Python type hints. Install it with `pip3 install mypy` and run it in the root of the Python project with `mypy`. It will emit potential issues such as type confusion and operation on an optional variable without first checking if it is `None`.

D. Property Testing with Proptest

During the review of the `safetensors` library, we wrote several property tests for the library based on the `proptest` framework.

Property testing is used to test whether a property holds true for arbitrary inputs to a function or component of the codebase. If a failure case is found, the framework automatically finds a minimal test case that violates the property.

The `proptest` crate uses strategies to generate random inputs. There are simple strategies such as `any::<usize>()`, which can be used to generate primitive data types. Strategies can also be composed to generate more complex types.

During the engagement, we wrote strategies to generate arbitrary instances of the `Dtype` and `Metadata` types defined by the `safetensors` library. We then used these instances to validate that the serialization and deserialization processes are inverses of each other (i.e., that we got the same data back if we first serialized and then deserialized an existing `SafeTensors` structure).

```
proptest! {
    #![proptest_config(ProptestConfig::with_cases(100))]

    #[test]
    fn test_roundtrip(metadata in arbitrary_metadata()) {
        let data: Vec<u8> = (0..data_size(&metadata))
            .map(|x| x as u8).collect();
        let before = Safetensors { metadata, data: &data };
        let tensors = before.tensors();
        let bytes = serialize(
            tensors.iter().map(|(name, view)| (name.to_string(), view)),
            &None
        ).unwrap();
        let after = Safetensors::deserialize(&bytes).unwrap();

        // Check that the tensors are the same after deserialization.
        assert_eq!(before.names().len(), after.names().len());
        for name in before.names() {
            let tensor_before = before.tensor(name).unwrap();
            let tensor_after = after.tensor(name).unwrap();
            assert_eq!(tensor_before, tensor_after);
        }
    }
}
```

Figure D.1: This test case runs `test_roundtrip` 100 times with random inputs.

This property test uses the `arbitrary_metadata` strategy to generate random `Metadata` instances.

```
fn arbitrary_metadata() -> impl Strategy<Value = Metadata> {
    // We generate at least one tensor.
    (1..MAX_TENSORS)
        .prop_flat_map(|size| {
            // Returns a strategy generating `size` data types and shapes.
            (
                prop::collection::vec(arbitrary_dtype(), size),
                prop::collection::vec(arbitrary_shape(), size),
            )
        })
        .prop_map(|(dtypes, shapes)| {
            // Returns a metadata object from a (length, dtypes, shapes) triple.
            let mut start = 0;
            let tensors: Vec<TensorInfo> = dtypes
                .iter()
                .zip(shapes.into_iter())
                .map(|(dtype, shape)| {
                    // This cannot overflow because the size of
                    // the vector and elements are so small.
                    let length: usize = shape.iter().product();
                    let end = start + length * dtype.size();
                    let tensor = TensorInfo {
                        dtype: *dtype,
                        shape,
                        data_offsets: (start, end),
                    };
                    start = end;
                    tensor
                })
                .collect();
            let index_map = (0..tensors.len())
                .map(|index| (format!("t.{index}"), index))
                .collect();
            Metadata {
                metadata: None,
                tensors,
                index_map,
            }
        })
}
```

Figure D.2: This `proptest` strategy generates arbitrary `Metadata` instances and relies on the simpler strategies `arbitrary_dtype` and `arbitrary_shape`.

To ensure that the test passes, we made some assumptions about the file format that are currently not in the specification. For example, we assumed that data offsets should never index out of bounds and that tensor shapes should be non-empty and never contain zero.

To compare `TensorView` instances in `test_roundtrip`, we derived the `PartialEq` and `Eq` traits for the `TensorView` type.

References

1. Proptest GitHub page
2. The Proptest Book

E. Property Testing with Hypothesis

During the review, we wrote several property tests using the Python `Hypothesis` library. As with `proptest`, these tests can be used to specify whether a property holds true for arbitrary inputs to a function or component and automatically finds a minimal failing test case if such an example exists.

We used strategies to generate arbitrary inputs. In particular, we used strategies specifically designed for libraries using the `NumPy` library and tested properties such as the absence of crashes and the integrity of round-trip serialization and deserialization. We also extended these property tests to incorporate fuzzing through `python-afl`.

The following test checks whether `NumPy` arrays saved in a safetensors file are the same when they are deserialized. In other words, the property specified by this test is the integrity of round-trip serialization and deserialization. This test substantiates TOB-SFTN-3. To make the test effective, we made multiple assumptions that are not included in the specification. These assumptions are explicitly outlined in the tests.

This Python Hypothesis test checks that the data is the same after serialization and deserialization. The decorator specifies the strategies used to generate arbitrary data, and multiple assumptions are made in the function body.

```
import safetensors.numpy
import numpy as np
from hypothesis import assume, given, strategies as st
import hypothesis.extra.numpy as hen

@given(
    tensor_dict=st.dictionaries(
        st.text(min_size=1),
        hen.arrays(
            dtype=hen.unsigned_integer_dtypes(endianness="<"),
            shape=hen.array_shapes())),
    metadata=st.dictionaries(st.text(), st.text()))
def test_roundtrip_serialize_deserialize(tensor_dict, metadata)-> None:
    assume(bool(tensor_dict) == True)
    assume(bool(metadata) == True)
    assume(tensor_dict.keys() != (['']))
    serialized_bytes = safetensors.numpy.serialize(tensor_dict=tensor_dict,
                                                   metadata=metadata)
    deserialized_dict = safetensors.numpy.deserialize(bytes=serialized_bytes)
    assert tensor_dict == deserialized_dict
```

Figure E.1: The Python `Hypothesis` test

References

1. Python Hypothesis

F. File Format Polyglots

The safetensors file format allows the creation of polyglots, which are files that can be interpreted validly as multiple different file formats. This is particularly impactful for downstream applications that rely on parsing such files, as polyglots can enable steganography and cause invalid format detection, parsing bugs, and other issues. For instance, an attacker can upload a file that is a valid machine learning (ML) model in the safetensors file format and a backdoored model in the Keras native file format. An attacker could also upload a file that is both a valid ML model in the safetensors file format and malware as a ZIP archive.

The creation of polyglots with the safetensors file format is chiefly enabled by the ability to append arbitrary data to the file without affecting the validity of the file. As a result, an attacker can append any file in a format that accepts prepended data, such as ZIP or PDF. This should be disallowed by the parser. ZIP is a particularly common container format for other ML file formats, such as the Keras native format.

Using the header size as the starting element, followed by the variable-length `metadata` component, expands the set of safetensors polyglots that can be constructed. An attacker can set the header size in the safetensors file format to be the magic signature of another file format, thus changing the size of the `metadata` component to match the magic signature. As a result, polyglots can be created even if the other file format does not accept prepended data. This capability is somewhat restricted by the maximum header size; when interpreted as a header size, some magic signatures are greater than the maximum header size specified by the library and are therefore rejected during parsing. This should be a consideration if the maximum header size is ever changed.

We created valid polyglots from the safetensors file format with the Keras native, PDF, ZIP, and TFRecords file formats.

References

1. Evan Sultanik, *Two New Tools that Tame the Treachery of Files*, Trail of Bits Blog. November 1, 2019.

G. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- The expression `filter_map(..).next()` could be replaced with `find_map(..)` (line 454 in `safetensors/src/tensor.rs`).

```
454 |                   let tensor_name = self
    |  ___________________________________^
455 | |                     .index_map
456 | |                     .iter()
457 | |                     .filter_map(|(name, &index)| if index == i {
                                Some(&name[..])
                            } else {
                                None
                            }
                        )
458 | |                     .next()
    | |____________________________^
```

- `Metadata::new` could return `Metadata` instead of `Result<Metadata, ...>` (line 425 in `src/tensor.rs`).
- The `slices` argument to `SliceIterator::new` is passed by value but is not consumed in the function (line 214 `src/slice.rs`). Consider changing the type to `&[TensorIndexer]` instead.
- The example from the README is missing the line `import torch`.

```
from safetensors import safe_open
from safetensors.torch import save_file

tensors = {
   "weight1": torch.zeros((1024, 1024)),
   "weight2": torch.zeros((1024, 1024))
}
save_file(tensors, "model.safetensors")

tensors = {}
with safe_open("model.safetensors", framework="pt", device="cpu") as f:
   for key in f.keys():
       tensors[key] = f.get_tensor(key)
```

- The following line (line 165 in `safetensors/src/tensor.rs`) could be removed since `data` already has type `Vec<(S, V)>`.

```
let data: Vec<_> = data.into_iter().collect();
```

- The `filename` argument to `serialize_file` is passed by value but is not consumed in the function (line 123 `bindings/python/src/lib.rs`). Consider changing the type to `&PathBuf` instead.
- The type hint for the function `_is_little_endian` should be `bool` (on line 166 of `bindings/python/py_src/safetensors/numpy.py`).

```
def _is_little_endian(tensor: np.ndarray) -> str:
```

H. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On April 26, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Hugging Face team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

Hugging Face's fixes span a number of commits and pull requests. We list the associated location of each fix in the Detailed Fix Review Results section. There was only one finding left unresolved, of informational severity (TOB-SFTN-3**)**. Our suggestions were implemented, but it was later discovered that a fix would break compatibility with the models produced by other libraries, so the fix was reverted to maintain compatibility.

In summary, of the 10 issues identified in this report, Hugging Face has resolved 9. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Tensor offsets are not checked against the total size of the tensor data	Medium	Resolved
2	Tensor size calculations may overflow in Metadata::validate	Medium	Resolved
3	The safetensors library allows zero-sized tensors	Informational	Impracticable
4	The SliceIterator type does not validate tensor indexers against the tensor shape	Low	Resolved
5	Insufficient test coverage against adversarial inputs	Medium	Resolved
6	Serialization can panic on malformed JSON	Informational	Resolved
7	Underspecified JSON behavior can lead to parser differentials	Low	Resolved

8	PyTorch conversion utility is vulnerable to arbitrary code execution	**Undetermined**	**Resolved**
9	Python dependencies are not semantically versioned	**Low**	**Resolved**
10	The safetensors library does not check for exceptional values	**Informational**	**Resolved**

Detailed Fix Review Results

TOB-SFTN-1: Tensor offsets are not checked against the total size of the tensor data
Resolved in PR #206. Safetensors now fails to load a file if there is extraneous data at the end.

TOB-SFTN-2: Tensor size calculations may overflow in Metadata::validate
Resolved in PR #207. Safetensors now checks whether metadata contains information that will lead to an arithmetic overflow.

TOB-SFTN-3: The safetensors library allows zero-sized tensors
Resolved in PR #215 and PR #216, but subsequently reverted in PR #221 and PR #226. Other libraries such as PyTorch emit models with zero-sized tensors, so `safetensors` needs to maintain its previous behavior for compatibility.

TOB-SFTN-4: The SliceIterator type does not validate tensor indexers against the tensor shape
Resolved in PR #216 and PR #218. The tensor shape is now validated.

TOB-SFTN-5: Insufficient test coverage against adversarial inputs
Resolved in PR #214, PR #216, PR #225, PR #228, and PR #235. Additional test coverage and input validation was added.

TOB-SFTN-6: Serialization can panic on malformed JSON
Resolved in PR #209. The unchecked `unwrap` was removed.

TOB-SFTN-7: Underspecified JSON behavior can lead to parser differentials
Resolved in PR #215. The documentation was improved.

TOB-SFTN-8: PyTorch conversion utility is vulnerable to arbitrary code execution
Resolved in PR #219. A warning and confirmation message were added to the utility.

TOB-SFTN-9: Python dependencies are not semantically versioned
Resolved in PR #204, PR #227, and PR #233. All Python dependencies are now semantically versioned, and the Cargo lockfile has been added to the repository.

TOB-SFTN-10: The safetensors library does not check for exceptional values
Resolved in PR #215. This behavior was documented.