Dynamic Taint Analysis in Python

Kamran Ahmed

August 13, 2024

Contents

1 Background

Motivation. Bugs in consumer-facing applications can lead to the theft of sensitive data, including personal and financial information. Most of the time, these vulnerabilities result from improper handling of sensitive data within an application or lack of input validation. For example, SQL injection attacks can be used to extract information from a database or modify its contents if an application does not properly sanitize user input.

Analysis Description. To mitigate these risks, developers can employ a technique called taint tracking to identify and track the flow of sensitive data through an application. This works by "tainting" untrusted data and tracking its flow through a program at runtime. If this tainted data ends up in a sensitive sink, such as a database query, the application can immediately raise an alert and exit the program. This allows developers to identify vulnerabilities and take steps to prevent sensitive data from being leaked or modified.

Implementation Strategies. Taint tracking can be implemented in various ways. Static analysis can be used to identify vulnerabilities at compile time, warning developers of issues before the application is ever deployed. Dynamic taint analysis can be used to track the flow of data at runtime, allowing for more accurate information tracking. Static analyses offer a more conservative approach, but dynamic analyses can provide more accurate results based on how the application behaves. In a dynamic approach, taint tracking could be implemented by instrumenting the application's source code or bytecode, or by modifying a scripting language's interpreter to track the flow of data directly.

Tool Overview. In this project, we focus on implementing a simple dynamic taint tracking system using program instrumentation. At a high level, we use Python as our target language and use the `ast` library to parse the source code and insert taint tracking code into the program's abstract syntax tree (AST). Then, we can either directly execute the instrumented code or save it to a file for later execution. We built a comprehensive test suite to validate the correctness of runtime taint tracking. We provide an easy-to-use API for developers to taint data and track its flow through the application and showcase a simple example of how taint tracking can be used to intercept and prevent SQL injection attacks in a Python web application. We also provide a more complex example of how taint tracking can be used to prevent the leakage of sensitive information such as private cryptographic keys in a chat application. The novelty of this project lies in the fact that we created a robust runtime system with minimal overhead without modifying the Python interpreter.

Previous Work. Most Python-based taint tracking tools are very limited in the language features they support. In [1], the authors develop a dynamic taint analysis in which users have to make library calls into the taint tracking library themselves. This leaves the system open to error, as the developer not only has to maintain knowledge of their code base but also the taint tracking library. Furthermore, this means that the taint analysis code is

directly embedded with the application logic, making it difficult to maintain and update. Unfortunately, this library only works on a small subset of the Python language (e.g., strings and integers) and requires the user to manually mark the functions that should propagate taint. We provide a library that just requires labeling taint sources and sinks with type comments, which is less invasive and more maintainable. We also provide a unique taint representation that supports taint tracking in most data types by default and supports the propagation of taint through regular Python functions and methods.

`DynaPyt` is a generic dynamic analysis framework in Python that can be used to implement various dynamic analysis tools [2]. The authors provide a simple interface to easily create custom dynamic analyses and provide an example of a taint tracking analysis. However, `DynaPyt` comes with a few limitations. First, because it is a generic framework, the authors have added many features and runtime hooks that are not necessary for a taint tracking system (e.g., control flow events and memory read and write events). This means that they also have to instrument more code than necessary, which will probably slow down the analysis. The taint analysis example they provide strictly uses the `id` function to track taint which is not a scalable approach because two Python objects with non-overlapping lifetimes can have the same `id`. This means that this analysis will be overly conservative and will not be able to track taint accurately. Lastly, code in third-party libraries will not propagate taint because it has not been instrumented. This is a key feature of our system, as we directly add taint tracking information to objects themselves. And since our representation does not affect the functionality of the object, it can be used to track taint through third-party libraries as well.

`Pysa` is a static taint analysis tool for Python that uses configuration files and type hints to specify taint sources and sinks [3]. One limitation of this tool is that it does not capture control flow information in code outside of the analyzed repository. To mitigate this, the analysis assumes that if a tainted value is passed to a function that it does not have the source code for, the function will return a tainted value as well. This is a very conservative assumption, and since static analysis is inherently conservative anyway, this tool may produce false positives. Users may become desensitized to the warnings and ignore them, which can lead to critical security vulnerabilities. This type of analysis does not provide the same level of accuracy as a dynamic analysis as aforementioned because it does not have access to runtime state.

2 Technical Approach and Initial Work

In this section, we will describe the high-level strategy of our taint analysis tool, the interface through which developers interact with it, and the internal workings of the tool (i.e., the instrumentation procedure and the runtime library).

2.1 Strategy

The main goals of this taint analysis project are to:

1. Minimize interference with normal program execution.
2. Demonstrate the effectiveness of the tool on real-world, user-facing applications where security is a concern.
3. Support as many language features as possible.
4. Provide a simple and intuitive way for developers to use the tool.

To accomplish these goals in a dynamic analysis, our runtime library will use a taint representation that is lightweight and easily propagated through the program without much intervention. Before that, we will first describe the tool's interface.

2.2 Interface

Basics. We provide a simple interface for developers to use our tool. To instrument the code, you can run:

```
python3 -m tainted.instrument <source_file> -o <output_file>
```

where `<source_file>` is the source file to instrument and `<output_file>` is the file to dump the instrumented code to. This is useful if the program is invoked in a different way than `python3 <source_file>`. For example, if the program is normally invoked via `uvicorn main:app`, then the developer can run:

```
python3 -m tainted.instrument main.py -o main_instrumented.py
```

to instrument the code and then run `uvicorn main_instrumented:app` to start an instrumented version of their web server. We also support directories now too, so an entire Python application can be instrumented with a single command.

Type Comments. Python 3.5 added support for type hints where developers can annotate function parameters and other variables with type information, but the Python interpreter won't enforce these types when loading the program. For example, the following code snippet annotates the `add` function with type hints:

```
def add(a: int, b: int) -> int:
    return a + b

result = add(1, 2)
```

This is mainly for documentation purposes and to aid programmers in writing well-typed, and hopefully more correct, code. Static analyzers like `mypy` can use these type hints to catch type errors, but they are not captured in the abstract syntax tree (AST) of the program when parsing the code with the `parse` function from the `ast` library. Conveniently, Python also supports type comments that can be added to variable assignments. This is useful for our taint analysis tool, as developers can use these type comments to taint variables and mark sensitive regions as sinks. More importantly, they can be examined in the AST. For example, the following code snippet annotates the `user_input` variable as tainted and the `query_db` function as a sink:

```
user_input = get_user_input()  # type: taint[source]
result = query_db(user_input)  # type: taint[sink]
```

so the resulting AST would look like this:

```
...
Assign(
    targets=[Name(id="user_input", ctx=Store())],
    value=Call(
        func=Name(id="get_user_input", ctx=Load()), args=[], keywords=[]
    ),
    type_comment="taint[source]",
),
Assign(
    targets=[Name(id="result", ctx=Store())],
    value=Call(
        func=Name(id="query_db", ctx=Load()),
        args=[Name(id="user_input", ctx=Load())],
        keywords=[],
    ),
    type_comment="taint[sink]",
)
...
```

We look for these type comments when walking over the AST to insert runtime calls to our taint analysis library which we describe in Section 2.4. Currently, we support the following annotations on assignment operations:

```
# type: taint[source]
# type: taint[sanitized]
# type: taint[sink]
```

2.3 Runtime Library

2.3.1 Taint Representation

In our system, we represent taint on a thin wrapper class around the original data. Since we cannot possibly predict and know all object types at runtime, including those outside of the program being instrumented, we will use a just-in-time translation of pure objects to taintable objects. To create a taintable object, we first have to create a taintable class that inherits from the original object's class. This allows us to add an `is_tainted` attribute to the base object and convenience methods to propagate taint or sanitize the object. This function, `create_taintable_class` can be found in `runtime.py`. We also add taint propagation hooks to all methods of the taintable class which will be described further in Section 2.3.2. However, objects will not be marked as taintable by directly invoking this function. Instead, we provide a `make_taintable` convenience function that converts a pure object to a taintable object:

```
def make_taintable(obj: Any) -> Any:
    _type = type(obj)

    # If the object is not taintable, we can just return it as is
    if _type in NON_TAINTABLE_TYPES:  return obj
    if _type in taint_class_cache: return taint_class_cache[_type](obj)

    # Build a new class that inherits from the original
    if _type in BUILTIN_TAINTABLE_TYPES:
        taint_class = create_taintable_class(_type)
        taint_class_cache[_type] = taint_class
        return taint_class(obj)

    # Otherwise, initialize the hooks directly on the object
    try:
        create_taintable_object(obj)
    except AttributeError as e:
        tainted_object_ids.add(id(obj))
    return obj
```

This allows us to prevent "non-subclassable" objects (e.g., `bool`, `None`) from being converted to this representation. Since we rely on the type of the object to perform this conversion, we can maintain a cache of taintable classes to avoid creating the same class multiple times. Furthermore, we now support custom and third-party class instances via the function `create_taintable_object` which will directly add taint tracking methods and propagation hooks to the class instance itself. In fact, this function could replace `create_taintable_class` altogether, but the overhead of adding the taint tracking code is non-negligible so this seems to be a good combination of flexibility and performance. One limitation of this representation is that any function that calls `type` on a taintable object will not return the original type of the object, but rather the taintable class. There does not seem to be an easy way around this without modifying the interpreter, which would require a significant amount of work. Lastly, to support strictly immutable objects, we use the built-in `id` function to add an object to a set of tainted `id`s as a fallback mechanism. Unfortunately, this means that our current propagation mechanism will not work for these objects, so more work would be needed to support this behavior in the future.

2.3.2 Taint Propagation

For the most critical aspect of our runtime taint system, we devised a simple way to propagate taint information across function boundaries and method calls. To accomplish this, we heavily exploited function and method call interposition to propagate taint and raise exceptions when tainted data is found at a sink. To propagate taint through method calls, we add a `_propagate_taint` function hook to most methods found on the class in `_create_taintable_class` or class instance in `create_taintable_object`. `_propagate_taint` will wrap every taintable object's method calls and will examine the object and method arguments in order to set the result of the call to be tainted if any of the arguments are. This can be seen in the function snippet below.

```
def propagate_taint(method):
    def inner(self, *args, **kwargs):
        result = make_taintable(method(self, *args, **kwargs))
        if type(result) in NON_TAINTABLE_TYPES: return result
        if is_tainted(self):
            result = taint(result)
        for arg in args:
            if is_tainted(arg):
                return taint(result)
        for kwarg in kwargs.values():
            if is_tainted(kwarg):
                return taint(result)
        return result
    return inner
```

For regular sinks (e.g., simple assignment operations), we raise a `RuntimeError` if the data is tainted just by checking its `is_tainted` attribute or seeing if its `id` is in the tainted set. At a function sink, we raise an exception if any of the arguments are tainted before the function is called as seen in the next snippet.

```
def function_sink(fn: Callable, *args, **kwargs) -> Any:
    for arg in args:
        raise_if_tainted(arg)
    for kwarg in kwargs.values():
        raise_if_tainted(kwarg)
    return fn(*args, **kwargs)
```

This requires tight cooperation with the instrumentation procedure to properly convert a function call at a sink into the format expected by `function_sink`, which we describe next.

2.4 Instrumentation

Our instrumentation code reads in a single Python file or a folder of Python files, parses their ASTs, and visits each AST node with a custom `ast.NodeTransformer` class. Python literals such as strings, integers, and floats and primitives like lists, tuples, and sets are wrapped with a call to `make_taintable` in the runtime library.

For assignments, if the right-hand side of the assignment is a function call and the type comment `type: taint[sink]` is present, then we wrap the function call with a call to `function_sink`. The execution of the original function will be delayed until the taint analysis library has determined that all arguments supplied to the function are untainted. If the right-hand side of the assignment is not a function call, then we will wrap it with the appropriate runtime call as follows:

- If type comment is `type: taint[source]`, then call `taint`.
- If type comment is `type: taint[sink]`, then call `raise_if_tainted`.
- If type comment is `type: taint[sanitized]`, then call `untaint`.
- Otherwise, leave the node unchanged.

Lastly, since standalone functions cannot be annotated with a type comment, we provide a convenience function `taint_sink` that takes a function as an argument and during instrumentation is replaced with a call to `function_sink`. This serves as an easy way for developers to mark function calls as sinks, preventing them from having to be manipulated to the format expected by `function_sink`.

2.5 Setup

A detailed setup guide can be found at https://github.com/kamodulin/tainted.

3 Evaluation Methodology

In this section, we outline the metrics that we used to evaluate the efficacy of this taint analysis approach.

3.1 Validation

To validate our tool at a basic level, we created numerous unit tests to test our runtime library with most Python language features. Part of this included ensuring that taintedness is propagated correctly across simple operations, method calls, and function boundaries. These tests can be found in `test_runtime.py`. Here is a snippet from one of test case that demonstrates some features of our runtime library with lists:

```
def test_list():
    l = _taint([1, 2, 3])
    assert_tainted(l)

    # Indexing and slicing preserve taint because the whole list is tainted
    assert_tainted(l[0])
    assert_tainted(l[:2])

    # Commutative
    assert_tainted(_taint([1, 2]) + [3, 4])
    assert_tainted([1, 2] + _taint([3, 4]))

    # Tainted elements do not taint the whole list
    assert_untainted([_taint(1), 2, 3])
```

As we can see, this test checks that the taint is propagated correctly when indexing and slicing lists and that individual elements do not taint the whole list. Furthermore, when concatenating lists, the taint is propagated correctly regardless of the order of the operands. We also have tests for other built-in types, such as tuples, dictionaries, and sets. We have not tested every single method of built-in types, but we believe that we have tested enough to ensure that the core components of our runtime library are working as expected.

3.2 Performance Testing

Ideally, this tool extends beyond the scope of toy examples and can be used in real-world applications, so we measured the overhead of taint tracking and its impact on application performance. The specific tests can be found in `test_perf.py`. Several speed tests were conducted along with a peak memory usage test. The baseline version of the tests was run without taint tracking and used pure Python objects. The tainted version of the tests adds calls into the taint tracking runtime. Lastly, the ratio is calculated as the time or memory

usage of the tainted version divided by that of the baseline version. The results are presented in the table below.

Test Name	Test Summary	Baseline	Tainted	Ratio
Basic Arithmetic	Performs simple arithmetic operations for 500,000 iterations.	0.016s	6.788s	419.25
Methods	Performs 500,000 `set.add` operations.	0.010s	0.323s	32.83
Functions	Performs 500,000 function calls and in the tainted version wraps them with a call to `function_sink`.	0.023s	0.259s	11.21
Memory Usage	Performs 10,000 math operations and append the result of each to a list to track memory usage.	83.2 KiB	2451.1 KiB	29.45

It is no surprise that the tainted versions of the tests are slower than the baseline versions. This is most likely due to taint propagation logic that is invoked on most method calls. Memory usage is also unsurprisingly higher in the tainted version due to the creation of wrapped classes and objects, but we did not expect the discrepancy to be as large as it is. It might be worthwhile to identify critical hotspots in the code using a profiler, but we did not have time to do so. In Section 5, we discuss potential optimization strategies including static analysis preprocessing to reduce the overhead of taint tracking. Ultimately, the performance overhead is significant, but the tradeoff may be worthwhile in small to medium-sized applications where security is a top priority.

3.3 Microbenchmarks

To validate that this approach is practical, we tested it on a few toy applications in Section 4. The SQL injection example is a simple, canonical measure of the efficacy of the approach. If our approach can detect SQL injection accurately across function boundaries and intercept tainted data before executing a sensitive operation, then it is likely to be effective elsewhere. The other synthetic microbenchmark we created was an encrypted chat application. This application performs a Diffie-Hellman key exchange to establish a shared secret, which is then used to encrypt and decrypt messages between a client and server. We will demonstrate that our approach can detect secret key leakage before it is sent erroneously over the network. We initially planned to examine a more complex application, such as a messaging client built on the Signal Protocol, to see how well our approach scales to a more intricate program. However, the chat application we created was sufficient to demonstrate the approach's effectiveness on a more complex example with multiple files and function calls.

4 Microbenchmarks

Here, we showcase the ability of our taint tracking to work with multiple files, functions, and methods. The following examples demonstrate how the taint tracking analysis could be used in real systems and indicate that our taint analysis is portable and tractable. Setup guides for both of these microbenchmarks can be found here.

4.1 SQL Injection

In this benchmark, we will demonstrate how our dynamic taint analysis can be used to mitigate SQL injection attacks. We will use a simple Python web server that uses FastAPI to expose an endpoint for querying a sensitive database. Listing 1 shows the implementation of the server and its interaction with the database. The code is simplified for the sake of clarity. Furthermore, the tainted source, `query` is annotated with a type comment to indicate that it is tainted, and the `execute` function is also annotated with a type comment to indicate that it is a sink.

```python
import sqlite3

from fastapi import FastAPI, HTTPException, status

conn = sqlite3.connect(":memory:")
app = FastAPI()

def fetch_one(query: str):
    cursor = conn.cursor()
    result = cursor.execute(query)  # type: taint[sink]
    user = result.fetchone()
    return user

@app.get("/users/{user_id}")
async def get_user(*, user_id: str):
    query = f"SELECT * FROM users WHERE id = {user_id}"  # type: taint[source]
    user = fetch_one(query)
    if not user:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="The requested user does not exist",
        )
    return user
```

Listing 1: Python code vulnerable to SQL injection attacks.

After instrumenting the server with our taint analysis (Listing 2), it will be able to intercept a malicious input and raise an error before the query is even executed. For example, if an attacker issues an HTTP GET request to the `/users/user_id` endpoint with a malicious parameter (e.g., `user_id="https://example.com/users/0 OR 1=1"`), a `RuntimeError` exception will be raised. The server and database will continue to run normally, but, more importantly, the attack will be mitigated.

```
import sqlite3

from fastapi import FastAPI, HTTPException, status

from tainted import *

conn = sqlite3.connect(_make_taintable(":memory:"))
app = FastAPI()

def fetch_one(query: str):
    cursor = conn.cursor()
    result = _function_sink(cursor.execute, query)
    user = result.fetchone()
    return user

@app.get(_make_taintable("/users/{user_id}"))
async def get_user(*, user_id: str):
    query = _taint(_make_taintable(f"SELECT * FROM users WHERE id = {user_id}"))
    user = fetch_one(query)
    if not user:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=_make_taintable("The requested user does not exist"),
        )
    return user
```

Listing 2: Instrumented server code.

Internally, the type comments are converted to specific function calls during instrumentation. For example, the type comment `# type: taint[source]` will be converted to a call to `_taint` to mark the underlying value as tainted which can be seen in Listing 2. Similarly, `# type: sink` will be converted to a call to `_function_sink` to check if any arguments to the original function are tainted. If so, this will raise a `RuntimeError` exception.

It is important to note that in this example, even regular users will trigger the runtime error if they provide an input that is not sanitized, which would make for a poor user experience. To address this, we provide a mechanism to sanitize tainted objects. Developers

can use the `type: taint[sanitized]` type comment to indicate that the input is no longer tainted.

4.2 Cryptograhpic Key Leakage

In this benchmark, we demonstrate how to use our dynamic taint analysis to detect and prevent cryptographic key leaks in a toy messaging application. If an attacker were able to intercept private keys used in key exchanges, they would be able to decrypt all messages sent between users. With taint tracking, we can mitigate this attack by preventing the key from being leaked in the first place. So, we can label specific cryptographic keys as tainted (see crypto.py) and mark network calls like `sock.send` as sinks. For example, in `crypto.py`, we mark the secret key as tainted:

```
def generate_key_pair():
    """Create a public/private key pair for use in a Diffie-Hellman key
    exchange."""
    sk = ec.generate_private_key(ec.SECP256R1())  # type: taint[source]
    pk = sk.public_key()  # type: taint[sanitized]
    return pk, sk
```

We also label the call to `sock.send` as a sensitive sink in main.py:

```
def handle_client(sock: socket.socket) -> None:
    """Handle a single client connection by exchanging keys and printing
    messages."""
    pk, sk = generate_key_pair()
    _ = sock.send(serialize_key(pk))  # type: taint[sink]
    ...
```

This program is in fact correct, but let's suppose that a developer misinterprets the variable `pk` as **p**rivate **k**ey instead of the **p**ublic **k**ey. This would be a reasonable mistake to make because there is little documentation and the two-letter variable name is ambiguous if you are not familiar with the code. If the developer wrote the following code instead:

```
def handle_client(sock: socket.socket) -> None:
    """Handle a single client connection by exchanging keys and printing
    messages."""
    sk, pk = generate_key_pair()
    _ = sock.send(serialize_key(pk))  # type: taint[sink]
    ...
```

the program may now leak the private key just by changing two characters! Now, with taint tracking enabled, we can prevent this mistake from causing a security vulnerability. When we run the instrumented program by creating a server and connecting a client, we will

observe that the taint analysis runtime will raise a `RuntimeError` exception immediately before the private key is sent over the network. While this seems like a contrived example, it demonstrates how such a small mistake can lead to severe security implications and highlights how taint tracking can be a powerful tool for preventing such vulnerabilities in production software.

5 Future Directions

5.1 User-Defined Taint Flow

Instead of viewing taint as a binary property, we could allow users to define custom taint levels. This would allow for finer-grained control over taint propagation and would allow for more complex taint policies to be enforced. For instance, a user could define taint classes as "secret", "top secret", and "confidential", and then define policies that only allow data to flow from "secret" to "top secret" if it has been sanitized. This could be implemented as a lattice of taint labels, with some partial order relation to define the taint hierarchy and flow policies. This lattice model has been previously described in the context of secure information flow and access control mechanisms [4].

5.2 Static Analysis Optimization

Clearly, the instrumented code could be optimized to reduce the overhead of taint tracking. Most literals and constants are wrapped in a custom object class. While this interposition is necessary for a correct system, it is not necessary for every object and function if it can be statically determined that the object does not need to be tracked. Moreover, if a tainted object can *never* reach a sink, then it does not need to be tracked.

Performing a static analysis, such as a "reachability" interprocedural dataflow analysis, could allow us to remove some of the taint tracking overhead. This would require a sophisticated interplay between the dataflow analysis and the instrumentation but could result in a significant performance improvement.

5.3 Interpreter Modification

There are some obvious limitations to the data representation of tainted objects. Every single object is wrapped in a custom object class, which is convenient for this project, but is not scalable. Additionally, any function that introspects on a wrapped object will see that it has the type `taintable_class`, not the wrapped type. To make this class as transparent as possible, we most likely need to modify the Python interpreter to include a taint attribute on every object. This would yield a more scalable and transparent solution but would require a significant amount of work.

References

[1] J. J. Conti and A. Russo, "A taint mode for python via a library," *Information Security Technology for Applications*, 2010.

[2] A. Eghbali and M. Pradel, "Dynapyt: A dynamic analysis framework for python," *Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.

[3] "Pysa: An open source static analysis tool to detect and prevent security issues in python code." https://engineering.fb.com/2020/08/07/security/pysa/.

[4] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, 1976.