Semester project
Exhaustive symbolic execution engine for verifying Python programs

Loïc Montandon
EPFL, Switzerland
loic.montandon@epfl.ch
June 2022

***Abstract*—As symbolic execution (SE) engines are usually not for formal verification, this semester projects aims to move an existing SE tool for verifying Python programs towards being exhaustive. Focus will be made on handling non-deterministic functions and implementing it in a concrete case: the SVSHI smart home system[1].**

I. Introduction

While several symbolic execution engines exist, most of them aim for finding as many bugs as possible, in opposition to being exhaustive. The reason for this is that covering all possible cases is hard, in particular for languages such as Python, which allows a large flexibility to the programmer and offers a wide range of features. Additionally, SE analysis is limited by the *path explosion*: every **`if`** statement encountered doubles the number of paths to explore.

The goal of this project is to develop/improve an SE engine towards exhaustively verifying Python programs, for later use in the SVSHI smart home infrastructure.

To address the path explosion problem, code is analyzed on a per-function basis, where each function is annotated with contracts it needs to satisfy. If a function becomes too big to be analyzed in reasonable time, it can be split in multiple smaller functions.
This design is based on [1]. Its main specificity is that it is based on a peer architecture, rather than on defining a new interpreter or compiler (which would be bound to one specific version of the language). Instead, verification is run in the same process as the target code and makes use of object-oriented programming. The key idea is to define classes for each symbolic type. Those classes redefine all methods used to interact with other objects. For example a `SymbolicInt` would define the method `__add__`, which is used for addition with another object. This way all interaction with other objects can be recorded in a state holding all symbolic variables together with the constraints on those variables leading to the current path in the code. To verify contracts, it uses a SMT solver to check if the constraints are satisfied. For more details on this SE design as well as a good introduction to symbolic execution, refer to their article [1].

In this report, I will first discuss the reason of choosing CrossHair as the SE engine to work on (II-A), with a brief description of how it works and its main limitations (II-B and II-C). In part II-D, I will present the design of how to address the main limitation: non-determinism. Next, I will describe some of the implementation details, focusing on the difficulties encountered (III). I will finally present how the same problem of non-determinism was solved for the SVSHI system (IV), followed by a conclusion on the main results and limitations of the proposed solution.

II. Design

A. Choice of the symbolic execution engine

The first step for this project was to chose on top of which SE engine to further improve. One possibility was to build an SE engine from scratch. This would have the advantage of having full control on the design and have it targeted for formal verification. However, by looking at the complexity of existing SE engines, it appeared that in the time of one semester, it would only be possible to build a very basic engine with very limited functionalities. It would have been exhaustive on the few things it is able to check but would not be able to check most of Python's code. So, to

[1]SVSHI - Secure and Verified Smart Home Infrastructure, https://github.com/dslab-epfl/svshi.

take advantage of existing work, I chose to go with CrossHair[2], an SE engine based on [1], with many further improvements.

The main reasons to choose this SE engine is that this is one of the most advanced and active SE tools in Python. Additionally, this is already the tool used in SVSHI (thus leading to less integration effort by the team). And Phillip Schanely[3], the main maintainer of CrossHair was interested in this additional work and available for support. I want to thank him here for his great help in understanding the complexities of CrossHair!

As most SE tools, CrossHair has for goal to find as many bugs as possible, not necessarily covering all paths. To understand its limitations for exhaustive execution, we will first quickly see how it works.

B. *How CrossHair works*

CrossHair is based on [1] and analyzes each function individually based on pre- and post-conditions (predicates) written by the user in the docstring of the function. It creates symbolic variables for each of the arguments of the function, constrains them to satisfy the precondition, executes the function on the symbolic variables and checks if the postcondition is satisfied. Below is an example of a function with a pre- and a post-condition. CrossHair verifies that if both `x` and `y` are negative, then the returned value should be positive.

```
def multiply(x: int, y: int) -> int:
    """
    pre: x < 0 and y < 0
    post: __return__ > 0
    """
    return x * y
```

CrossHair works the following way that it explores all possible paths until having covered all paths or until reaching a timeout. CrossHair has three possible analysis results (run `crosshair check` with the option `--report_all` to get such results):

- *CONFIRMED - "confirmed over all paths"*: All paths were explored exhaustively, the postconditions are always satisfied, on any input satisfying the preconditions.
- *REFUTED - "a counterexample was found"*: CrossHair found a counterexample, where some postcondition is not satisfied. A concrete instance of the counterexample is displayed to the user.
- *UNKNOWN - "not confirmed"*: No counterexample was found; the engine timed out before exploring all paths.

Distinguishing these three categories is fundamental for formal verification, as one would expect *CONFIRMED* to be returned iff the function was exhaustively verified.

Of course, CrossHair is much more complex and while we will see some more details in section III, you can refer to the official documentation[4] (particularly section The Details: Contracts), if you wish to learn more about it.

C. *Limitations of CrossHair*

From a formal verification perspective, one wants to ensure the SE engine never returns *CONFIRMED* when this is not the case, and it should ideally not return *UNKNOWN* too often to still be useful. The first point is the most critical one, therefore, a necessary step was to identify such cases in CrossHair (v0.0.22), which are listed below:

1) **Non-determinism** - All functions, directly or indirectly invoked by the function (or its pre/post conditions) should exhibit the same behavior when given the same arguments. Examples of non-determinism are:
   - Calls to functions such as `random` or `time`. In the example below, CrossHair will wrongly output *CONFIRMED* if verification is done during another month than July.

```
from datetime import datetime
def never_July() -> int:
    """post: __return__ != 7"""
    return datetime.now().month
```

   - Reading a global variable or reading a cache.
   - Functions raising exceptions depending on the state of the system (e.g. if memory is full). More generally, the system state should ideally be the same for verification and for production.

[2]CrossHair, *An analysis tool for Python that blurs the line between testing and type systems*, https://github.com/pschanely/CrossHair/.

[3]Phillip Schanely, https://github.com/pschanely.

[4]CrossHair Documentation, https://crosshair.readthedocs.io/en/latest/index.html.

2) CrossHair assumes **code termination**. In the code below, CrossHair will infer from the postcondition that the inner call to `recurs` returns a value larger than 10. This implies that the outer function will also return a value greater than 10. CrossHair makes a recursive proof without proving the base case and outputs *CONFIRMED*, even if the function never terminates.

```
def recurs(x: int) -> int:
    """
    pre: x > 0
    post[]: _ > 10
    """
    return recurs(x)
```

3) CrossHair requires all arguments to have **directly constructable states**. For example, a class having an integer attribute should allow assigning a value to it through the `__init__` method. The reason for this is that when a function has an object as an argument, CrossHair will create this object by giving symbolic values to `__init__`. The created object can only be seen as symbolic if any of its states can be reached through the constructor.
4) Another assumption made by CrossHair is that the **type hierarchy is known** and closed. At initialization, CrossHair will load the whole class hierarchy of all modules and packages available in the current environment. However, if another module somewhere else in the world defines a subclass to a type used in your code, and that this subclass makes the postcondition fail, CrossHair cannot know that the subclass exists and will not report a counterexample.
5) **Approximated exceptions** - To prevent a premature realize of symbolic variables, CrossHair might simplify information contained in exceptions thrown by the standard library. So, code catching an exception and relying on its details might behave wrongly during verification. For example, a call to `x.to_bytes(2, "little")` for `x` a negative integer should raise `OverflowError: "can't convert negative int to unsigned"`. While CrossHair still raises the correct exception type, the message has been removed. This only affects exceptions raised by the standard library, not those in user code or external libraries.
6) **Parallelism** - CrossHair assumes a single-threaded execution of the code and will therefore not be able to detect concurrency problems.

This list of limitations has been shortened to only keep most important information for the purpose of this report. You can find the complete list with more examples (including cases where it always returns *UNKNOWN*) in appendix A.

One simple way to have exhaustive SE is to assume all the above properties hold on the analyzed code. However, this is not realistic for most projects, particularly when external libraries are used.

Of course, addressing all limitations in such a small amount of time is not possible, as most of them would be individual research subjects. So, after discussing with the team working on SVSHI, it appeared that item 1) (non-determinism) was the most useful point to address.

D. Handling non-determinism

In order to handle non-determinism during the symbolic execution, the proposed solution is to annotate functions which are source of non-determinism, so that the SE engine knows their possible outcomes.

More concretely, this allows users to register contracts (pre- and postcondition) on any function, including external libraries. Then, when CrossHair encounters a registered function, it will check that its preconditions are met and assume that the returned value can be any value satisfying the postcondition.

As an example, suppose you are using `random.randint`, which takes to integers `a` and `b` and outputs an int at random between `a` and `b`. This is clearly not deterministic. Then you can register the function the following way:

```
from crosshair.register_contract import
↪  register_contract
from random import Random
register_contract(
    Random.randint,
    pre=lambda a, b: a <= b,
    post=lambda __return__, a, b: a <=
     ↪  __return__ <= b,
)
```

Then, when CrossHair encounters `randint(a, b)` during verification, it will create a new symbolic variable for the return type, which can be of any value between `a` and `b`.

This way, as long as non-deterministic functions are registered, CrossHair is told how non-deterministic functions behave and can correctly think about them.

III. Implementation

A. *Faking a condition parser*

In order to implement this feature of registering contracts for external functions, the key idea was to make use of the *condition parsers* of CrossHair. Those parsers are used to find the pre- and post-conditions of a function. We have already seen the *PEP 316* syntax based on the docstring, but other syntaxes based on decorators, like *icontract* or *deal*, are also supported by CrossHair. See the documentation[5] for supported syntaxes.

Knowing this, implementing contract registration is done by adding a new parser, hidden to the user. This parser simply returns the registered contract for the current function (from a hashmap containing all registered contracts).

With that, CrossHair knows contracts for registered functions. However, this does not mean that such functions will be skipped. Indeed, CrossHair skips functions only if the following conditions are met: either the function is annotated with `# crosshair: specs_complete=True`, or all arguments to the function are symbolic and are marked as not mutable[6] This means that skipping functions already exist, the solution is to force it when the function has a registered contract.

B. *Parsing from the stubs*

For the above solution to work, the registered function needs to be type-annotated. Otherwise, CrossHair does not know its return type and cannot create a symbolic variable of that type when skipping the function.

Unfortunately, even the standard library and the builtins are not type-annotated. Instead, separate files, called *stub files* (`.pyi` extension), are available in the *typeshed* repository[7]. Proposed solution is that whenever registering a contract for a function, CrossHair would first verify if the function is type-annotated. If this is not the case, it would parse the corresponding stub file to get the signature of the function.

Parsing stub files is not easy for one main reason: those files are not meant to be executed or used at runtime. Even importing such a file would likely fail. For instance, you might encounter the code below (for the random module), which raises the following: `AttributeError: "'ellipsis' object has no attribute 'randint'"`.

```
_inst: Random = ...
randint = _inst.randint
```

Furthermore, some functions might be annotated with some types which are internal to typeshed and not available at runtime. An example is the type `SupportsLenAndGetItem[T]`, which represents a type implementing both `__len__` and `__getitem__`. This is to reflect Python's *duck typing*, but those types have no runtime equivalent.

Another difficulty with stub files is that those are written using Python 3.11's latest syntax, which is not backwards compatible. So, to use it in earlier versions of Python, conversions need to be done.

It quickly appeared that parsing stub files reliably would not be possible for all functions. Instead, the proposed solution is a best-effort parser working the following way: It opens the stub file for the corresponding module and loads it with `ast`[8]. Then it iterates over the `__qualname__` of the function (the path inside the module), to find each enclosing object one by one until the function definition is found. On the way, it executes all assignments encountered, in case those are needed for the function signature. It finally parses the signature itself and if an error occurs, the signature is not used.

To support cases where the stub parser does not succeed, as well as functions which have no stub file, the contract registration API was augmented with an optional signature the user can provide. See the documentation[9] on how to specify a signature.

[5]Contract syntaxes, https://crosshair.readthedocs.io/en/latest/kinds_of_contracts.html.

[6]Mutable arguments are defined inside square brackets after the `post` keyword. Empty bracket means no argument is mutated by the function. https://crosshair.readthedocs.io/en/latest/kinds_of_contracts.html#pep-316-contracts.

[7]https://github.com/python/typeshed.

[8]Abstract Syntax Trees, https://docs.python.org/3/library/ast.html

[9]Adding Contracts to External Functions, https://crosshair.readthedocs.io/en/latest/plugins.html#adding-contracts-to-external-functions.

Finally, note that some functions might have multiple signatures (using the decorator `@typing.overload`[10]). In such a case, the stub parser will return all signatures. The correct one will be chosen at runtime, according to the type of the arguments given to the function.

C. Special case of builtins and C modules

One thing to note in Python is that builtins and C modules[11] often behave differently. For example, while all other functions have attributes `__name__`, `__qualname__` and `__module__`, this is not the case for builtins and C functions. Similarly, `inspect.signature(fn)` raises an exception if `fn` is a builtin or a C function. This is why it was important to keep this in mind when using such attributes and to test all features with builtins and C functions as well. Such objects are often wrapped into a descriptor[12] and a workaround is to extract information from that descriptor. For example, one can get the module name of function `fn`, wrapped into a `MethodDescriptor` or a `WrapperDescriptor`, through `fn.__objclass__.__module__`. This was needed in the stub parser, where the module name is necessary to know which stub file to parse.

D. An infinite recursion problem

When analyzing a function, CrossHair copies all arguments using `copy.deepcopy` to make their initial value available through the `__old__` keyword in the postcondition, even if the arguments were mutated by the function. An infinite recursion occurs when copying an argument (indirectly) calls a registered function. As an example, consider the function `Random.getstate(self)`, where `self` is an instance of the `Random` object. If one registers this function, when CrossHair analyzes it, it calls `copy.deepcopy` implicitly calling `Random.__reduce__`, which in turn calls `Random.getstate(self)`, thus causing an infinite recursion. A solution would be to avoid registering functions which are directly or indirectly called by the copying process. A better solution, which Phillip Schanely (the main contributor of CrossHair) will implement is to not do the copy when it is not needed (i.e. when the `__old__` keyword is not used in the postcondition). An intermediate solution is currently implemented, where CrossHair will not try to analyze the recursive call to the function if it occurs inside the copying process.

E. Functions returning objects

A final difficulty was to handle functions returning non-trivial objects. For example, `datetime.now()` returns a `datetime` object. So, if one registers the method `now`, CrossHair would skip it at runtime by returning a symbolic `datetime` object (feeding symbolic arguments to its `__init__` method). However, CrossHair needs to construct a **valid** `datetime` object. For instance, its `hour` attribute must be between 0 and 23, otherwise an exception is raised. Additionally, a `timezone` object needs to be created as well and also has constraints on its values.

To allow CrossHair successfully building such classes, a solution is to register a contract to their `__init__` method. This contract has a precondition, specifying the constraints on the expected arguments. However, we don't want CrossHair to skip executing the `__init__` method at runtime, so the contract registration API was augmented with one more argument: `skip_body=True`, which specifies if the function should be skipped or not.

IV. Handling non-determinism in SVSHI

After adding support to non-determinism in CrossHair, the goal was to use the new feature into SVSHI, a toolchain for developing and running formally verified smart infrastructure[13]. In SVSHI, apps are run using their *iteration* function. This function takes as input the current state of the system and might update the state and notify other devices of the changes. We want to allow users to import and use any library they want, while still ensuring that the specifications of the system are formally verified. As third party libraries might behave non-deterministically, this is where registering contracts would be useful.

[10]typing.overload, https://docs.python.org/3/library/typing.html#typing.overload

[11]Code compiled in C, but available as a python module. A well-known example is *numpy*.

[12]https://docs.python.org/3/library/types.html#types.WrapperDescriptorType

[13]For more information about SVSHI (Secure and Verified Smart Home Infrastructure), see https://github.com/dslab-epfl/svshi_private.

SVSHI uses CrossHair to verify that invariants of the system still hold after executing the *iteration* functions of the different apps. However, SVSHI recently switched to use `crosshair cover` instead of the regular `crosshair check`. The contract registration feature is not meant to be used for `crosshair cover`, so we took a different approach here.

The first thing to note is that calls to external libraries are unsafe, as they might crash (i.e. network or credential error) or they might have non-determinism. At the same time, those calls are usually slow. And since we want to keep SVSHI being responsive, it is necessary to isolate such functions.

The proposed solution is therefore to allow calls to external libraries only in two kind of functions, whose name starts with `periodic` or `on_trigger`. Both are executed asynchronously, so they have no impact on the main iteration. Periodic functions are not allowed to have any argument and are executed on a periodic basis, with the period in seconds provided by the user in the docstring. On_trigger functions can be executed using `svshi_api.trigger_if_not_running(fn, arg1, arg2, ...)` and may have arguments. To get the returned values of such functions, the user can use `svshi_api.get_latest_value(fn)`. Below is an example of both types of function and their usage.

```
from external_library import weather,
↪   send_message
def on_trigger_send_message(message:
↪   str) -> bool:
  return send_message(message)

def periodic_get_weather_forecast() ->
↪   float:
  """period: 60"""
  return weather.rain_probability()

def iteration():
  rain = svshi_api.get_latest_value(
    periodic_get_weather_forecast
  )
  if rain is not None and rain > 0.5:
    # some code to close the windows...
    svshi_api.trigger_if_not_running(
      on_trigger_send_message,
      "Closing windows because of rain"
    )
```

For verification, we assume such function can return **any** value of the correct type (or `None` if the function has never been executed). This is the reason why users have to sanitize the returned value, by checking it is not `None` and it is in the expected range. At verification time, we replace calls to `svshi_api.get_latest_value` by a symbolic variable of the correct type[14].

The strength of this design is that we do not assume anything about what happens inside those functions, which is good, since calls to external libraries cannot be verified and might be unsafe. Such functions are allowed to do anything or even crash, which will not impact any of the installed apps. The only assumption made is that they return a value of the correct type. This property is checked at runtime and if it happens to not be the case, a warning if thrown and the returned value is ignored.

V. Results

The feature of registering contracts has now been merged into CrossHair and conduced to version 0.0.23 of CrossHair[15]. Furthermore, this feature is already in use by default for the non-deterministic functions of the `random` and `time` modules (v0.0.24). The documentation is available online[16] and we offer it as a plugin. This means anyone can write patches for non-deterministic functions in a plugin and publish it to other people. Then plugins can be installed with pip and CrossHair will automatically use them.

VI. Further work

A possible improvement to registering contracts would be to make the stub parser handle more complex cases that it currently does. While the type checker *Mypy*[17] does not offer an API to parse stubs, it might be interesting to see how they parse stub files and if it is possible to inspire from that. Mypy is a huge project, so it may not be easy to do it.

Further work could be done on CrossHair in the direction formal verification to address (or at least

[14]To be more exact, an object containing one symbolic value for each periodic or on_trigger function is given as an argument to the iteration function.

[15]Release notes: https://pschanely.github.io/2022/05/17/handling-nondeterminism.html.

[16]Adding Contracts to External Functions, https://crosshair.readthedocs.io/en/latest/plugins.html#adding-contracts-to-external-functions.

[17]Mypy, https://github.com/python/mypy.

detect and cap the result to UNKNOWN) other of the identified limitations (see appendix A). While this is hard to do exhaustively, due to the complexity of Python, there remains some steps to do in this direction to cover more cases.

Another approach would be to limit available features of the language, in order to have easier assumptions where exhaustive symbolic execution would be possible. This would however significantly reduce utility of the program and disallow using any library which is not compliant.

VII. Conclusion

To conclude, we have seen that implementing an exhaustive SE engine in Python is hard, due to the complexity of the language. Furthermore, we saw with the limitations listed in part II-C that being exhaustive highly depends on the assumptions made about the code analyzed. The proposed way of handling non-determinism is to tell the verifier the possible values a function can return, so that it can skip the function and create appropriate symbolic return values. A similar approach was chosen for SVSHI, with the addition that unsafe code is isolated to ensure it has no impact on the core code of the system.

References

[1] A. M. Bruni, T. Disney, and C. Flanagan, “A peer architecture for lightweight symbolic execution,” 2011.

Appendix A. Limitations of CrossHair as of May 11th 2022 (CrossHair v0.0.22)

Document converted from markdown, available online[18].

The aim of this document is to list the limitations of CrossHair[19] as of May 11th 2022 (CrossHair v0.0.22). You can find an up-to-date version of this list here[20].

Edit (June 7th 2022): Note that since some of the limitations have been addressed in more recent versions of CrossHair. So, some of the examples below might not work anymore (for example non-determinism has been partially addressed in v0.0.24).

A. Cases where CrossHair might return `CONFIRMED` instead of `REFUTED`

The following is a list of cases were CrossHair might report a wrong analysis result because it made some assumptions which your code does not satisfy.

- **Non-determinism**: All functions, directly or indirectly invoked by the function (or its pre/post conditions) should exhibit the same behavior when given the same arguments. Here are some examples[21] of non-determinism. For cases raising `NotDeterministic`, see section below[22]. Examples of non-determinism are:
- Calls to functions with randomized output (example: `random` library), functions depending on the system’s state (example: `time` and `datetime` libraries), or functions whose output depends on previous queries.
- Reading a global variable or using a cache.
- Functions raising exceptions depending on the state of the system (e.g. if memory is full). CrossHair will only trigger exceptions which depend on the function’s args.
- Self-modifying code is also a kind of non-determinism, as different executions of the same function might lead to different results.

[18]Limitations of CrossHair as of May 11th 2022 (CrossHair v0.0.22), https://gist.github.com/lmontand/4108e889070a857dfce0b3f4f76282f2.

[19]<https://github.com/pschanely/CrossHair>

[20]<https://github.com/pschanely/CrossHair/discussions/156>

[21]<https://crosshair-web.org/?crosshair=0.1&python=3.8&flags=report_all&gist=30122ad2cae84d22ca38d8c36acfe7ce>

[22]<#cases-where-an-exception-is-raised-and-analysis-aborted>

If you still wish to use such functions in your code, you might want to register contract for those to help CrossHair understand them. See plugins and contract registration[23].

- **Code termination**: Verification requires that the code terminates on all inputs. (why? Think about it like an inductive proof - the inductive step only works if you also have a "base case"). A real verification tool will have you write a termination proof before reasoning about the property to be verified; CrossHair simply assumes termination. If your code does not terminate, CrossHair may declare a contract confirmed over all paths when it isn't. Here is an example[24].
- **All arguments have states that are directly constructable**: Problematic examples: classes with mutable private members like this[25]. A list containing itself as a member[26] (this is also an example of the aliasing problem below). To ensure CrossHair correctly understand your classes, please follow these guidelines[27].
- **Aliases**: Aliasing problems might happen when you are using containers/collections holding non-trivial objects. CrossHair remembers instances created before and is able to detect some cases. However, CrossHair is not able to correctly reason about all cases: see this issue[28] which contains two interesting examples. More to this topic, you should avoid comparing objects with the `is` keyword. Here is an example[29] of what could go wrong. The only cases you might use it are for `is None` or for comparison with the value of an `Enum`.
- **Closed type hierarchy**: CrossHair will detect and attempt to use subclasses that have already been defined in the interpreter (warning, see exceptions below under "IMPORTANT"), but will not consider the possibility of additional subclasses. Path exhaustion will happen after all known subclasses have been attempted. As an example, suppose you run CrossHair on some module `module1` containing the class `class Class1`. Suppose another module `module2` contains `class Class2(Class1)` (i.e. a subclass of `Class1`). If `module2` is not loaded by the interpreter, CrossHair will have no way of knowing that `Class2` exists. So, the analysis result might be "confirmed over all paths" even if `Class2` breaks some postconditions.

IMPORTANT: classes defining either `__copy__`, `__deepcopy__`, `__reduce__` or `__reduce_ex__` are not considered in the possible subclasses, because they have a non-trivial implementation of copying. Here is an example[30].

Another similar and important note is that CrossHair won't detect subclassing that is revealed via dynamic isinstance hooks, e.g. `__subclasscheck__` and `__subclasshook__`.

- **Catching exceptions**: If you do some "advanced" exception handling - not only based on the exception type itself, but on other information carried with the exception - your code handling the exception might not behave the correct way when CrossHair runs it. The reason for this is that CrossHair simplifies information of some exceptions of the standard library, to avoid unnecessary realization of symbolic variables. This only concerns exceptions thrown by the standard library, if your code throws some exceptions, they will remain untouched. Here is an example[31] where handling the exception depends on the exception's message, which is missing when executing CrossHair.
- **Callables with TypeVar**: If your function takes as input a `Callable[X, Y]`, where `X` and/or `Y` is/contains a `TypeVar`, CrossHair might return "confirmed over all paths", even when the postcondition is wrong. See this issue[32] as an example.

[23] <https://crosshair.readthedocs.io/en/latest/plugins.html>

[24] <https://crosshair-web.org/?crosshair=0.1&python=3.8&flags=report_all&gist=b5ed75907e16742939965211163ffa95>

[25] <https://crosshair-web.org/?crosshair=0.1&python=3.8&flags=report_all&gist=9ec95667e2965ed7a2ddca237c0922ff>

[26] <https://crosshair-web.org/?crosshair=0.1&python=3.8&flags=report_all&gist=e5f3f25ec8886a9dae92b591d0c67a6f>

[27] <https://crosshair.readthedocs.io/en/latest/hints_for_your_classes.html?highlight=dataclass#hints-for-your-classes>

[28] <https://github.com/pschanely/CrossHair/issues/47>

[29] <https://crosshair-web.org/?gist=fe18a54ca442aa0be028dbf3b7b98732&crosshair=0.1&python=3.8&flags=report_all>

[30] <https://crosshair-web.org/?crosshair=0.1&python=3.8&flags=report_all&gist=6260ee385c2d993e91b70963b18674b5>

[31] <https://crosshair-web.org/?crosshair=0.1&python=3.8&flags=report_all&gist=684dda56d3741c6619925a69a697d189>

[32] <https://github.com/pschanely/CrossHair/issues/85>

- **Parallelism**: The code is assumed to be single-threaded and to not have any other kind of parallelism. Therefore, CrossHair provides no detection for deadlocks, concurrent writes, racing conditions, ... Here is a sample example[33] which works perfectly fine in a single-threaded setting, but could be wrong when adding parallelism.
- **Pointers**: Using the `ctypes` library and modifying pointers, it's easy to have CrossHair return incorrect results. To demonstrate the power of `ctypes`, one could even redefine the value of 1[34], thus totally breaking the behavior of Python.
- **Python environment**: Similarly, do not expect any guarantees if you try to modify part of the standard library. Results only apply to a fresh and unmodified python interpreter and library.

Additionally note that if you supply wrong pre- or postconditions or that you write incorrect type annotations to your functions, analysis results have high chances to be incorrect as well.

B. *Cases where the result is caped to* `UNKNOWN`

Below is a list of cases CrossHair has troubles handling with and the result is often `UNKNOWN`. You can find an example for most of them in this gist[35].

- Because CrossHair approximates float values as real numbers, it will never report "Confirmed over all paths" when it uses a symbolic float.
- If CrossHair needs to realize a symbolic value of type `object` or `Any` (both are treated the same way), it will only try the subtypes `int` and `str` and it will cap the result to `UNKNOWN`. This is because trying **all** subclasses of `object` is clearly not feasible in a reasonable amount of time. Note that in some rare circumstances where only some trivial operations are performed on symbolic objects, CrossHair does not need to realize the object and can still prove your result. Here is an example[36].
- Using `TypeVar` with constraints (for example `TypeVar("T", int, str)`) is currently not supported and the result is caped to `UNKNOWN`. Note that this is different from using `TypeVar("T", bound=Union[int, str])`, which is supported. For more information about the difference, see this post[37].
- Symbolic values are simulating real values. However, this illusion is not perfect. And some function might raise a `TypeError` when receiving a `SymbolicInt` instead of a regular `int`, for example. If CrossHair detects such a case, the result is caped to `UNKNOWN`. If this was not detected, the result will be `REFUTED` and it will directly report you the `TypeError`.
- If the signature of a constructor is not found or incomplete, CrossHair will not be able to create a proxy for it (it will not know what types of symbolic values to feed to the constructor). Here again, the result is caped to `UNKNOWN`. This can happen for some of the builtins and for some external C-based modules.
- Currently, CrossHair does not support symbolic functions with an ellipsis as argument type (i.e. `Callable[..., <some_return_type>]`) or with `typing.Paramspec` or `typing.Concatenate`. In such cases, the result is caped to `UNKNOWN`.
- Slices with a step different from `1` are currently not supported and the result is caped to `UNKNOWN`.
- Most operations over inputs of arbitrary size (lists, strings, dicts, etc) explicitly or implicitly trigger an infinite number of execution paths. In such cases, CrossHair timeouts and the result is `UNKNOWN`.
- Similarly, if you write very complicated constraints, the SMT solver might have a hard time solving them and will timeout. In such cases, the result cannot be better than `UNKNOWN`, as some paths are not explored.

C. *Cases where an exception is raised (and analysis aborted)*

Below is a list of operations which are not blocked by CrossHair. When detected, analysis is directly stopped with an exception. Cases are listed below according to the type of exception they raise.

[33] <https://crosshair-web.org/?crosshair=0.1&python=3.8&flags=report_all&gist=7f0cebe03912ba551be4670d4891e496>

[34] <https://www.reddit.com/r/Python/comments/2441cv/can_you_change_the_value_of_1/>

[35] <https://crosshair-web.org/?crosshair=0.1&python=3.8&flags=report_all&gist=fa89d612822941d23f109659a2bb4b2c>

[36] <https://crosshair-web.org/?crosshair=0.1&python=3.8&flags=report_all&gist=427739cf93932cf43e7626f9ce1eefe1>

[37] <https://stackoverflow.com/questions/59933946/difference-between-typevart-a-b-and-typevart-bound-uniona-b>

Here also, you might find contract registration[38] useful, if you still want to use such functions.

- raise `NotDeterministic`: If a function is detected to have a different behavior between two executions, a `NotDeterministic` error is raised. This is detected by CrossHair when the condition or the path to reach some part of the code changed between two executions. This detection is not exhaustive and you should also read the ”non-determinism” bullet of this section[39]. You can have a look at `crosshair/statespace.py` if you wish to see the implementation details.
- raise `SideEffectDetected`: CrossHair has a list of side-effect events which will be blocked. The main reason for that is that CrossHair **does** execute your program and such side effects might cause undesirable changes on your computer. Upon detection, code execution is directly stopped. Note that this protection is only applied for python ¿= 3.8 and that it is not perfect (notably, it will not prevent actions taken by C-based modules). So, your code should better not have any side-effect.
- When using `open`, write operations are blocked (`os.O_WRONLY`, `os.O_RDWR`, `os.O_APPEND`, `os.O_CREAT`, `os.O_EXCL` and `os.O_TRUNC` are blocked). An exception is made for the file `/dev/null`.
- The following events are blocked: `"winreg.CreateKey"`, `"winreg.DeleteKey"`, `"winreg.DeleteValue"`, `"winreg.SaveKey"`, `"winreg.SetValue"`, `"winreg.DisableReflectionKey"`, `"winreg.EnableReflectionKey"`.
- The following events are allowed: `"os.putenv"`, `"os.unsetenv"`, `"os.listdir"`, `"os.scandir"`, `"os.chdir"`, `"os.fwalk"`, `"os.getxattr"`, `"glob.glob"`, `"os.listxattr"`, `"os.walk"`, `"pathlib.Path.glob"`, `"socket.gethostbyname"`, `"socket.__new__"`.
- Other events (not listed above) in the following list are blocked: `"os"`, `"fcntl"`, `"ftplib"`, `"glob"`, `"imaplib"`, `"msvcrt"`, `"nntplib"`, `"os"`, `"pathlib"`, `"poplib"`, `"shutil"`, `"smtplib"`, `"socket"`, `"sqlite3"`, `"subprocess"`, `"telnetlib"`, `"urllib"`, `"webbrowser"`.

D. Others

A current limitation of CrossHair is that you cannot use the new python 3.10 syntax for `Union` (PEP604[40]). An issue[41] is open for this. In the meantime, you should stick to the python ¡= 3.9 syntax for unions.

Also note that Consuming values of an iterator or a generator in a pre- or post-condition will produce unexpected behavior[42]. So you should avoid doing it.

E. Note on the execution environment

Code analysis should be run on the same platform and with the same interpreter where the code will be used for production. This is because some python code depend on the platform and/or on the python version.

[38] <https://crosshair.readthedocs.io/en/latest/plugins.html>
[39] <#cases-where-crosshair-might-return-confirmed-instead-of-refuted>
[40] <https://peps.python.org/pep-0604/>
[41] <https://github.com/pschanely/CrossHair/issues/161>
[42] <https://github.com/pschanely/CrossHair/issues/9>