EXTRACTING ICS MODELS FROM MALWARE VIA CONCOLIC ANALYSIS

A Thesis
Presented to
The Academic Faculty

By

Fabian Kilger

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Computer Science

Georgia Institute of Technology

August 2020

EXTRACTING ICS MODELS FROM MALWARE VIA CONCOLIC ANALYSIS

Approved by:

Prof. Brendan Saltaformaggio, Advisor
School of Electrical Engineering
Georgia Institute of Technology

Prof. Raheem Beyah
School of Mechanical Engineering
Georgia Institute of Technology

Prof. Paul Pearce
School of Computer Science
Georgia Institute of Technology

Date Approved: July 23, 2020

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my supervisor Prof. Brendan Saltaformaggio of the Cyber Forensics Innovation Laboratory at the Georgia Institute of Technology for his support, enthusiasm and our insightful discussions. From the Technical University of Munich, I would like to thank my supervisor Prof. Claudia Eckert and advisor Fabian Franzen for their support, encouragement and immensely helpful feedback. I am also particularly thankful for my labmate Mingxuan Yao. No matter what problem I faced, he was always there to discuss it and those discussions led to new unique ideas.

Next, I would like to thank Tohid Shekari and Qinchen Gu of the Communications Assurance & Performance Group at the Georgia Institute of Technology led by Prof. Raheem Beyah for providing me with test code for my evaluation.

I would also like to thank the international offices from both the Georgia Institute of Technology and the Technical University of Munich for helping me in my remote studies and making this research possible. Especially, my sincere thanks go to the coordinators of my Double Degree Program: Dawn Rutherford from the Georgia Institute of Technology and Prof. Michael Gerndt from the Technical University of Munich for their encouragement, enthusiasm and advise.

Finally, I thank the Federation of German-American Clubs (VDAC) and the Fulbright program for their scholarships that allowed me to study at the Georgia Institute of Technology. The previous year has been a challenging but immensely enriching one. I was able to meet a lot of awesome people and am thankful for every eye-opening experience I had.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

While there has been significant progress in automated malware analysis, the focus of prior work has been mostly on programs written in C/C++. Advanced malware such as the Triton malware, however, also employ Python which imposes additional challenges to the automated malware analysis. Motivated by this example, we design and implement a concolic execution framework that is capable of extracting models of the targeted industrial control systems (ICS) based on the Python malware's communication with the system. Our approach first transforms the Python malware to C and then utilizes a symbolic execution engine to analyze the resulting C code. We prove the functionality of our framework on a set of test programs and evaluate it on two ICS-related samples including the Triton malware. Finally, we discuss how the results of our analysis can be used to identify potentially targeted ICS of a Python malware.

CHAPTER 1

INTRODUCTION

Attacks on Industrial Control Systems (ICS) are especially severe as they often target critical parts of infrastructure to either retrieve critical information or significantly harm a specific organization and/or state. Recently, an ICS malware called Triton made the news and was labeled ‘’the world’s most murderous malware‘’[1] after targeting a petrochemical plant in Saudi Arabia. It received its label because it disables safety systems in industrial control systems what then leads to catastrophic accidents. And, while it was first discovered in 2017, there have been traces of it spreading further in the middle east[1]. To prevent such further spread, one would first need to analyze the malware as to what systems are potential targets.

However, analyzing which systems are potentially vulnerable and require additional protection is a time-expensive manual task that needs to repeated several times: First, the malware needs to be reverse-engineered to extract the targeted ICS communication protocol and used commands. Then, every ICS needs to be inspected on whether it supports this protocol and the issued commands. Automating the process of extracting such information from the malware will considerably ease the process of reverse-engineering and considerably speeds up possible responses to new malware.

While previous work has extracted commands and helped reversing the protocol for other domains of malware[2, 3] by leveraging the binary symbolic execution tool `angr`[4], such tools would not be applicable to advanced malware such as Triton: It uses the Python language which makes the analysis significantly more difficult. Specifically, the dynamic features of interpreted languages such as Python generally limit the applicability of static analysis. Additionaly, interpreted languages make heavy use of optimizations and data structures that cannot be handled well by symbolic executors[5].

Previous research on symbolic execution for Python[6, 5, 7] focuses on maximizing code coverage and generating test-cases to detect bugs. However, for malware, one is generally interested in analyzing the behavior of the commonly taken paths inside the malware ignoring rare edge-cases. Furthermore, these tools are limited in that they require modifications of the python source[5], cannot symbolically execute arbitrary types and features of the language[6, 7], or cannot handle Python code which is either written or compiled into C.

To protect against future ICS malware written in Python, we develop an analysis framework that is able to efficiently run concolic execution for Python malware and extract information that helps in classifying the targeted ICS software. For this, we extract control flow information that enables us to target interesting locations in the malware (e.g. communications with the machines), develop a search strategy to efficiently reach points of interest, and output symbolic models that represent the ICS communication protocols used by the malware to communicate with the ICS. For this purpose, we first compile the Python Code into C and, then, implement our approach using an open-source symbolic executor. Our contribution can be summarized as follows:

- We propose a new method for concolic execution of Python by transforms it to C.
- We design and implement a concolic execution framework that is able to discover common paths in python ICS malware and produce models of the ICS based on the malware's communication.
- We employ history-based branch prediction to efficiently and repeatedly pass validation functions in network protocols.
- We evaluate our approach on real malware (Triton) and an open-source python implementation of the Modbus protocol used in ICS applications.

In the following, Chapter 2 will, first, discuss necessary background to understand this work. Then, in Chapter 3, we explain the design of our framework; specifically, why and how we compiled Python to C code and integrated it into a concolic execution framework. Next, Chapter 4 discusses implementation details such as the code structure, modifications to used tools and details about our symbolic function models for Python. Following, we evaluate our framework on a set of test and real-world ICS-related samples in Chapter 5. We discuss limitations in Chapter 6, related work in Chapter 7 and conclude our work in Chapter 8.

CHAPTER 2
BACKGROUND

2.1 Symbolic Execution

After initially introduced in 1976 by James C. King[8], symbolic execution has seen a rise in applications of program analysis such as automated vulnerability testing[9, 10, 11, 12] where it has achieved high coverage and was able to discover many 0-day vulnerabilities. Furthermore, it has also been applied to automatically generate exploits[13], reverse engineering and reconstructing command and control servers[3].

The general idea in symbolic execution is that instead of letting the program run with concrete values, the execution operates on symbolic expressions that may represent arbitrary (or constrained) values. For this, the program is initially supplied symbolic input variables which will be operated on instead of concrete input. The program is then executed as it would under normal execution. When a branching instruction is reached, the execution splits into one path for each branch and the condition for the branch is added to the *path condition*. A constraint satisfaction solver can then check whether the current path is still feasible. This way, vulnerabilities can be found by formulating their conditions into logical formulas and the behavior and semantics of a program path can be analyzed by inspection of the path constraints and resulting symbolic formulas.

Common problems in symbolic execution are *path explosion* and expensive constraint solving. *Path explosion* describes the phenomena where paths inside a program grow exponentially. Specifically, each branch creates a new path and, as a consequence, when there is a branching instruction inside a loop with n iterations, the loop itself already has 2^n paths. To fight against this problem, previous work has suggested the use of techniques such as *state merging*[14], which will merge two similar paths by combining their expressions, and the use of more advanced search strategies such as ones trying to maximize coverage[9, 11]. Constraint solving, however, is expensive by its nature: It is an NP-complete problem and the solution time can exponentially grow as the query size grows. To ease the strain on the constraint solver, previous work has focused on two objectives: First, reduce the size and cost of each query. For example, KLEE[9] uses *constraint independence* to reduce the query only to formulas that have a dependence relation with the branch condition. The second objective is to reduce the number of necessary queries. This has been achieved by caching query results[9, 15], using concolic execution[9, 11, 12] and employing hybrid approaches where fuzzing is combined with symbolic execution[10, 16].

Function modeling is a technique that can be used for two purposes. First, it is used to introduce symbolic variables into the program (e.g. when a socket is read), and, second, to optimize frequently used functions to mitigate the path explosion problem. The first purpose is clear: Parts of the program that cannot be analyzed (for example when they are external) need to be modeled for the symbolic execution to continue. The second purpose is not as clear: Since library functions are frequently used, great efforts are taken to optimize them for the concrete execution. This, however, can often hurt the performance of symbolic execution. One such optimized function is the `memcpy` function: A naive implementation would copy each byte by itself, but the optimized ones will copy several bytes with each instruction and will branch depending on whether the number of bytes to be copied is still greater than what the target processor can copy with any instruction. This significantly speeds up the native execution of the function but it will also introduce multiple branches leading to further path explosion.

2.1.1 Concolic Execution

Concolic execution[9, 11, 12] tries to improve symbolic execution further by reducing the load on the constraint solver: Instead of symbolically executing the whole program, all values are traced concretely and symbolically. As a consequence, not every path condition of a branch needs to be sent to the constraint solver: The concrete values are already a proof for the satisfiability of the branch taken by the concrete state. Furthermore, complex functions and expressions can be concretized if solving them would no longer be feasible. For example, cryptographically secure hash functions cannot be efficiently reversed and concretizing them would allow us to continue to discover further branches whose conditions do not depend on the hash function.

2.1.2 Search Strategies

For an efficient symbolic execution framework it is crucial to choose an appropriate search strategy. As mentioned before, the number of paths generally explode and, therefore, not every path can be explored. The search strategy will, as a consequence, decide which path will be explored. In the following, we will describe common search strategies used in symbolic execution.

Depth-First Search (DFS) is a traversal algorithm commonly used in graphs that first tries to reach the deepest parts in the program and only backtracks when a path is fully explored. As a consequence, the memory consumption scales linearly with the length of the path and deeper parts are reached fast. However, DFS easily gets stuck in loops as it will try to fully explore them first.

Breadth-First Search (BFS) is another traversal algorithm commonly used for graphs and is the default search-strategy for *angr*[4]. It tries to reach out to all branches at the same time. As a consequence, BFS generally does not reach too deeply into the program as the number of paths are exploding. Furthermore, it will also lead to an exponential use of memory as information about each active state needs to be maintained. Compared to DFS however, it does not immediately get completely stuck in loops as it will simultaneously explore paths in- and outside the loop.

Iterative-Deepening tries to combine the efficient memory consumption of DFS with the broad coverage of BFS. Instead of continuing DFS until the path is fully explored, one sets a limit on the depth for each exploration and increments it after each iteration. This way, only the number of states used for the current DFS need to be saved. While it may seem that the execution is slower because states are explored multiple times, the asymptotic runtime is the same as for DFS and BFS. For these reasons, it has been used in multiple applications for program testing[17, 18].

Targeted or directed search[19]. This is a search strategy that tries to find a path reaching a specific point in the program. It will use a distance metric to determine which path reaches the program point in the shortest time. For this, it relies on static analysis information to extract a control flow graph of the program. In this strategy, the choice of the correct metric is essential as it has been shown that using the shortest path on the control flow graph might fail crucially[19].

In the *coverage guided search* strategy, the goal is to maximize coverage of the program and it has been proven to lead to high coverage inside a program[9]. Similar to the targeted search, this strategy uses a metric to estimate the distance of one state to the nearest uncovered program point. Therefore, this will also require the use of static analysis to retrieve a control flow graph where the metric can then be computed on.

2.1.3 Triton Concolic Engine

Not to be confused with the malware Triton, Triton[12] is an open-source dynamic binary analysis framework for the x86 and ARM architectures, consisting of a taint and concolic execution engine. Compared to other concolic execution frameworks like KLEE[9] and angr[4], its engine does not use an intermediate representation and operates directly on the instructions themselves. Furthermore, Triton exposes python bindings while its core is written in C++ for performance.

Additionally, compared to other concolic execution engines, Triton itself is very lightweight: It does not contain any search strategies and, instead, it symbolically executes instructions of one path that is given to it. Therefore, one will need to manage the symbolic states and implement a *tracer* which then will feed the instructions to the symbolic execution engine. Conveniently, one such *tracer* is shipped with Triton. It uses the dynamic binary instrumentation (see Section 2.2) framework Intel Pin[20]. However, according to their documentation, it only supports Linux versions with a kernel of version < 4.x.

As reflected by a capability benchmark[21], Triton currently has additional limitations compared to other concolic execution engine frameworks: It cannot deal with symbolic arrays and pointers[22], and, therefore, support for these would need to be manually added. Furthermore, Triton does not provide any function modeling for the C library functions and does not have any internal support for it. However, function modeling can still be implemented by directly modifying the symbolic state.

2.2 Dynamic Binary Instrumentation

Dynamic Binary Instrumentation (DBI) is a popular method to dynamically add custom made instrumentation to a binary. While for programs whose source code is available, compiler level instrumentation works very well, static instrumentation of closed-source binaries is prone to crashing the program because data references are lost during the compilation process and are not easily recoverable[20].

Dynamic instrumentation does not suffer from the same problem as it does not directly modify the program and keeps all references in tact. For example, the DBI framework Intel Pin[20] dynamically compiles instrumented code, saves it in *code caches* and, then, executes the code cache. This way, the DBI framework makes sure that no value or address reference is modified and hence avoids crashes.

2.3 Python

Python is a dynamically typed scripting language which has evolved over the years with continuous updates. While there are multiple implementations of Python interpreters[23, 24, 25], in this work, we will focus on the most commonly used one, CPython[24]. In the following, we will shortly discuss the usage of dictionaries inside the Python language, CPython's internal optimizations and, lastly, the implementation of exceptions.

Dictionaries. The most discerning feature of Python is the very common use of the dictionary (dict) type. It implements a typical key-value mapping. Programmers use them frequently as they offer an easy way to store structured data whose elements are accessible by name. Additionally, the language itself uses dictionaries to store local and global variables, attributes of objects, and members of classes and modules.

Dynamic name resolution. Python dynamically resolves each name (variables and functions). As every function call is dynamic, statically reasoning about the call-graph and global control flow graph of a Python program becomes increasingly difficult. While the symbolic execution engine is running in a dynamic context, some components usually rely on some form of static analysis. For example, the coverage-guided and targeted search strategies rely on control-flow information to deduce the shortest path to a specific block. Therefore, targeting strategies cannot be directly used.

Optimizations. To boost native performance, Python makes heavy use of optimizations. However, some of these optimizations hurt the performance of symbolic execution engines. For example, Python uses fast paths, caching and *interning*. *Interning* makes sure only one instance of a specific value exists in the program. Fast paths lead to more branches inside a function and, therefore, increase path explosion. The logic for caching and *interning* generally result in symbolic pointers, i.e. pointers whose value is a symbolic expression, and therefore might point to multiple locations. These symbolic pointers are either not supported (as with Triton) or are supported and tremendously slow down the execution of a symbolic engine[26].

Exceptions. Python is a language with exception support and its standard library makes use of it to communicate unforeseen events including syntax and name errors. In CPython, exceptions are roughly implemented as follows: If an exception is raised, specific thread-local variables will be set to point to the exception raised and the function raising the exception will return an error value. This type of implementation, where each function is responsible for cleaning itself up on a raised exception, can be easily managed by common search strategies since raised exceptions are reflected in the program's control flow graph.

CHAPTER 3

DESIGN

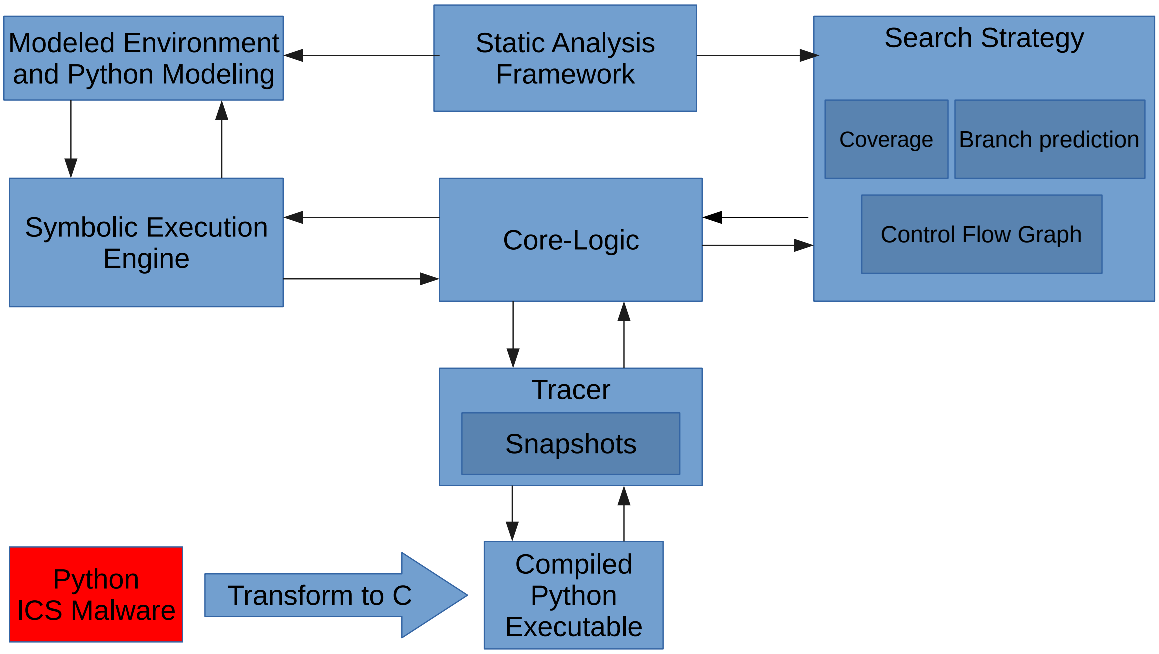
Modeled Environment and Python Modeling
Static Analysis Framework
Search Strategy
Coverage
Branch prediction
Control Flow Graph
Symbolic Execution Engine
Core-Logic
Tracer
Snapshots
Python ICS Malware
Transform to C
Compiled Python Executable

Figure 3.1: Design of our framework

In Figure 3.1, we present an high-level overview of the design of our concolic execution framework for Python. First, we will be transforming the Python malware into a C executable and employ a tracing tool to direct the concrete execution of the malware and allow snapshotting of the concrete states. At the center of our design is the core logic which receives a callback by the tracer for each instruction executed. It will forward the information about the executed instructions to the symbolic execution engine and search strategy. The symbolic execution engine will, in addition to symbolically executing instructions, check whether a called function is part of the modeled environment. The modeled environment represents all functions that are symbolically modeled as described in Section 2.1. If a modeled function is encountered, the core logic will then communicate with the tracer to synchronize the effects of the modeled function with the concrete environment. The search strategy will use the program's control flow graph produced by the static analysis framework, coverage information, and branch prediction to inform the core logic about which branches need to be covered next.

In the following sections, we will first describe why and how we transform the Python code to C. Then, we will explain our choice of tooling for the symbolic execution engine, our tracer and our static analysis framework. Then, we will discuss the design of our concolic execution including the search strategy and several optimizations. Lastly, we present how we generate the output of our tool.

3.1 Python transformation to C

The first step in our design is to transform the Python code to C. We chose this because building a symbolic execution engine directly for Python's byte or source-code would incur a high maintenance cost for each Python update. Additionally, malware could protect itself from such an engine by transforming selected parts of the program into C code[27, 28]. Symbolically executing also generally fails to discover relevant paths of Python programs as it lacks understanding of Python interpreters program counter. Therefore, we chose to use an existing tool to transform C to Python and use an open-source symbolic execution engine to analyze the transformed code. The Nuitka[28] project allows us to perform this transformation.

Nuitka wraps each of Python's operators into operator functions that are specialized depending on the statically determined type. Wrapper for comparisons will either return a boolean or a Python object. Operators will return the resulting object. For function calls, the function's name is first dynamically resolved to a function object and, then, the function object is called with the help of a wrapper with the name `CALL_FUNCTION_X` where `X` is a string representing the number of arguments.

To show how the transformation from Python to C looks like, we depict an example function in Figure **??** and the result of its transformation in Figure 3.2. Note that we removed exception handling and reference counting for the sake of simplicity. The if condition in the example program is transformed to a call to a RICH_COMPARE wrapper in line 6. Nuitka was able to infer that the type of the second argument is an int and chose the corresponding specialization. The control flow of the python program is transformed into multiple blocks that are jumped to with goto statements as the following if statement in the C code shows. In line 17, the dynamic name resolution of Python is made explicit; the name gcd is looked up in the module's dictionary. Then, the modulo operation takes place in line 18 and the recursive call happens in line 20. Note that for the modulo operation, Nuitka did not perform any specialization since the types of the parameters are not known statically.

```
1   PyObject* gcd(struct Nuitka_FunctionObject const* self, PyObject **python_pars) {
2       PyObject* retval;
3       PyObject* a = python_pars[0];
4       PyObject* b = python_pars[1];
5
6       nuitka_bool cond_eval = RICH_COMPARE_EQ_NBOOL_OBJECT_INT(b, const_int_0);
7       if(cond_eval == NUITKA_BOOL_TRUE)
8           goto branch_yes_1;
9       else
10          goto branch_no_1;
11
12  branch_yes_1:
13      retval = a;
14      goto return_exit:
15
16  branch_no_1:
17      PyObject* gcd_fun = GET_STRING_DICT_VALUE(moduledict___main__, const_string_plain_gcd);
18      arg_element_2 = BINARY_OPERATION_MOD_OBJECT_OBJECT_OBJECT(a, b);
19      PyObject* call_args[] = {a, arg_element_2};
20      retval = CALL_FUNCTION_WITH_ARGS2(gcd_fun, call_args)
21      goto return_exit;
22  return_exit:
23      return retval;
24  }
```

Figure 3.2: Simplified transformed version produced by Nuitka (excluding exception handling and reference counting)

In Figure 3.3, we show the exception handling code which is added after each operation. After a wrapper function, Nuitka checks whether the returned value is `NULL` to indicate an exception. If an exception was raised, Nuitka will fetch the exception information from the thread-local storage (see Section 2.3) and set the `exception_lineno` value to the line where the exception occurred. Then, it will jump to the exception handling code that checks whether an exception handler catches the thrown exception. If the exception is not caught, it will propagate the exception further.

```
arg_element_2 = BINARY_OPERATION_MOD_OBJECT_OBJECT_OBJECT(a, b);

if(arg_element_2 == NULL) {
    FETCH_ERROR_OCCURRED(&exception_type, &exception_value, &exception_tb);
    exception_lineno = 4;
    type_description_1 = ”oo”;
    goto frame_exception_exit_1;
}
```

Figure 3.3: Example of Nuitka's exception checking code

This type of exception handling will help us in designing our search strategy (Section 3.3.1): Exception handling is directly embedded in the control flow and, therefore, control flow graph based strategies are aware when exceptions are thrown and can use this information to find more paths of the program.

3.2 Choice of Tooling

After having explained the first step of our design, we present what tools were chosen as the building blocks of our design. In the following, we will explain the chosen symbolic execution engine, the chosen dynamic tracer and, finally, the used static analysis framework.

3.2.1 Choice of dynamic symbolic execution engine

As discussed in Section 2.1, there are multiple dynamic symbolic execution engines available for C programs. Note that because Nuitka compiles Python into C, we would also be able to use source-code level concolic execution engines like KLEE[9]. However, we argue that to evade our analysis the malware author could either also use Nuitka to compile the binary to assembly or obfuscate important parts into a Python module written in C.

Angr[4] is also a very powerful symbolic execution engine which could have been used in our approach. However, it was shown shown that angr imposes a huge overhead in the symbolic emulation[10]and specifically because we expect a lot of instructions to be purely concrete, we feared this might become the bottleneck of our approach. Instead, we use the more light-weight symbolic execution engine Triton[12] whose core is, contrary to angr, written in C.

One downside of using Triton, however, is that it does currently not support floating point operations while Python programs might freely use floating point data types. For example, floating point data types are nearly always used in the context of sockets to specify timeouts. Therefore, it is required for our analysis framework to handle floats in a reasonable manner. To solve the problem of Triton's lacking float support, we decided to always concretize on any float related functions. While this might lead to under-approximation and missed paths when analyzing floats, we will argue in Section 6.4 that this leads to no practical problem in our approach. Lastly, as previously mentioned, Triton does not implement support for symbolic pointers and we had to implement this ourselves (see Section 4.2.1 for implementation details).

3.2.2 Choice of static analysis framework

We rely on static analysis information to implement the function modeling and search strategy. Specifically, the static analysis framework will need to compute a control flow graph, identify modeled functions and retrieve the location of function arguments. On top of that, we will use the type information stored in the Python interpreter's debug information to compatibly access and modify Python-specific data structures.

As one of the biggest and most wide-spread frameworks, we decided to use IDA[29] as our main framework for static analysis. To integrate it into our framework, we created a server inside an IDA headless session and implemented several commands that will export IDA's static analysis information. Since IDA itself generally only supports static analysis of a single binary file, we implemented an additional PE-file loader script using `LIEF`[30] so that we are able to retrieve static analysis results of the compiled binary, Python library and dynamically loaded Python modules without having to start multiple IDA instances.

3.2.3 Choice of dynamic tracer

For our concolic execution, we require some component to concretely execute the program. First, we considered to use IDA's debugger interface together with WinDbg[31] because we already employ IDA for static analysis. However, we observed that there is an extremely huge overhead per instruction and found out that WinDbg's and IDA's performance do not scale with bigger programs. Therefore, we switched to a more efficient way to trace the program.

Considering that out-of-process tracing with debuggers like WinDbg[31] and GDB[32], imposes a higher overhead because more context switches (between processes and kernel) are required, we decided to use in-process tracers. This means that the tracer program executes in the same process (and address space) as the analyzed program. Such tracers can be implemented with the help of Dynamic Binary Instrumentation (DBI) frameworks, which can dynamically analyze and alter the behavior of a program. We decided to use Triton's tracer that utilizes Intel Pin[20], port it to windows and implement the additional functionality required by our framework (see Section 4.2.2 for the details of the tracer modifications).

3.3 Concolic execution

After the transformation step and used tools were described, we now explain the inner workings of our concolic execution engine. First, we will start by explaining the foundation of our search strategy and, then, explain how branch prediction can further improve this strategy. This is followed up by an explanation of our snapshot mechanism that is used to reset to a previously explored state. At the end of this section, we will then discuss further optimizations that our concolic execution framework uses.

3.3.1 Search strategy

Search strategies are essential for successful symbolic execution since they determine, which states/branches of a program will be explored. In Section 2.1.2, we discussed several common search-strategies used for symbolic execution. However, the discussed strategies are not optimal for our use-case: Our goal is not to explore all edge cases of a program and achieve high coverage; instead, we try to find the path(s) through the program that are *most likely* to occur in a real world scenario.

As a base, we use DFS since our goal is to find a path that finds the usual exit of the program. To not get stuck at loops or wrong branches, we detect loops and compute basic blocks from which we can no longer reach any point of interest so that we can avoid them.

Generally, points of interest would be functions that might be part of a communication with the ICS device such as receiving and sending messages. However, as discussed in Section 2.3, there is no sound static way to compute any function transitions and hence we cannot easily compute whether we can reach a Python function from another. Instead, we will try to dynamically discover the function transitions of the Python program and compute reachability based on the dynamically detected transitions. Therefore, the points of interest are Python function calls that have not been covered yet.

The search logic can then be summarized as follows: First, we compute the control flow graph of the program and find all blocks that can reach any uncovered Python call in the program. If we reach a block that can no longer reach these points of interest, we will apply DFS to revert to the first block that can reach an uncovered Python call and solve for the correct successor. Then, whenever we reach an uncovered Python call, we will recompute the reachable Python calls for each block.

By avoiding blocks that cannot reach any point of interest, we can significantly improve the performance of the DFS. However, we still encounter problems inside loops: First, in any loop, any block that is reachable from inside the loop can also be reached from every point inside the loop. Therefore, the search strategy cannot lead the execution to our points of interest. To mitigate this problem, we dynamically try to detect loops and will not use the back-edge for our reachability analysis.

The second problem that can occur inside loops is a symbolized branching condition: Because in each iteration the symbolized branching condition will split the execution in two, the total number of paths in such a loop would be equal to 2^n where n is the number of iterations. Since our base DFS strategy would try to fully explore all paths inside the loop, we would get stuck here.

An example of one such loop is depicted in Figure 3.4. There, we show the function `dump` which prints a hexdump of a sequence of bytes. This function is called by the Triton malware whenever an invalid response from the ICS is received - probably to help the authors with debugging. In line 19, there is a symbolized branch condition that is inside a loop. The condition checks whether a specific character is printable and will replace it with a dot in case it is non-printable. While this is a loop that is not interesting for our analysis in any way, it causes the DFS approach to fail and, therefore, we need to add something to our strategy so that we can prevent this case.

```
1  def dump(data, text=None):
2      if isinstance(text, basestring):
3          if len(text) > 0:
4              print text
5      if not isinstance(data, basestring):
6          print ’BAD DATA’
7          return
8      for i in xrange(0, len(data) / 16 + 1):
9          seq = data[i ∗ 16:min(i ∗ 16 + 16, len(data))]
10         hexes = (’ ’).join((a.encode(’hex’).upper() for a in seq))
11         sys.stdout.write(hexes)
12         for i in xrange(0, 16 − len(seq)):
13             sys.stdout.write(’ ’)
14
15         sys.stdout.write(’ ’)
16         for i in xrange(0, len(seq)):
17             c = seq[i]
18             sr = ’’
19             if ord(c) < 32 or ord(c) >= 128:
20                 c = ’.’
21             sys.stdout.write(c)
22
23         sys.stdout.write(’\n’)
24
25     sys.stdout.write(’\n’)
```

Figure 3.4: Extract of Triton-Malware dump function

As a solution, we track which basic blocks we already covered and, instead of exploring the deepest of all branch, we will first explore the deepest of all branches that directly lead to an uncovered block. That way, after solving the condition in line 19 once, we will revert further and first explore other paths.

3.3.2 Branch prediction

When we ran our initial tests using our search-strategy, we encountered a path-explosion problem that could not be solved using the previously mentioned coverage-based approach. Specifically, network-protocol implementations often utilize a parsing routine that checks whether the header contains valid information. Since this validation function will already be explored and fully covered after the first time, the coverage-based approach does not know to prioritize the exploration of the validation function over the exploration of an path-exploding loop like in the previously mentioned `dump` function.

To explain in more detail as to how this can fail, let us take a look at the specific case inside the Triton malware. An extract of one such validation function used by the Triton malware is depicted in Figure 3.5. This function will process received UDP data (from `udp_result`) in 3 steps. First, in line 8, the length of the packet is checked to be at least the minimum size of 6. Afterwards in lines 12 to 14, the first header field is checked, which apparently denotes the length of the packet. In the last step, in lines 18 to 20, the saved `crc16` checksum at the end of the message is checked to match the computed one.

```
 1  def tcm_result(self):
 2      if self._tcm_result != None:
 3          return self._tcm_result
 4      self._perror = −1
 5      data_received = self.udp_result()
 6      while True:
 7          self._tcm_result = (0, None)
 8          if data_received == None or len(data_received) < 6:
 9              print 'bad tcm size'
10              self._perror = 10
11              break
12          type, size = struct.unpack('<HH', data_received[0:4])
13          packet = data_received[4:−2]
14          if len(packet) != size:
15              print 'bad tcm size'
16              self._perror = 10
17              break
18          checksum = struct.unpack('<H', data_received[−2:])[0]
19          test_cksum = crc.crc16(data_received[:−2])
20          if checksum != test_cksum:
21              print 'bad tcm crc'
22              self._perror = 11
23              break
24          self._perror = 0
25          self._tcm_result = (type, packet)
26          break
27
28      return self._tcm_result
```

Figure 3.5: Extract of Triton-Malware message parsing

When we now enter this validation function a second time, we will first fail at the condition in line 12, because the received data is unconstrained and very likely does not match the size. Then, following our current strategy, we will continue to execute until we can no longer reach any point of interest. This means that, eventually, we will also again reach the loop inside `dump`. This time however, because we already covered every path in the validation function, the coverage metric cannot prevent the search-strategy to get stuck in exploring the `dump` function.

To solve this problem and to improve our search strategy on such validation functions, we used a construct commonly built into our processors: Dynamic Branch Prediction[33]. Specifically, we will try to predict which successor of a branch will lead us to better results based on the previous exploration of that branch. Applied to validation functions, this means that when we successfully found the correct path through this function we want to remember this path and follow it in the second exploration as well. To achieve this, we remember, for each branch, the previously explored successor and follow that same successor the next time as well.

3.3.3 Snapshotting

Whenever we want explore a new branch in our search strategy, we will need to reset the program to a previous point of time. Remembering this previous state of the program is what we refer to as snapshot. There are multiple ways to achieve this: First, one could restart the program and stop at the desired point. Second, it is possible to trace all modifications and then revert them to the desired point. Lastly, one could also save all the whole memory space and reset it to that point. The second option is generally the best option as it requires the least amount of memory and doesn't incur any overhead of restarting the program. Therefore, we chose use this option to snapshot our concrete state. However, for the symbolic execution engine *Triton*, we did not find an efficient way to record and revert changes in the symbolic expressions and, therefore, always replay the symbolic execution to the point of interest. While this certainly is not optimal for the performance, this has not shown to be a performance bottleneck yet.

3.3.4 Other optimizations

We also employ several optimizations introduced in previous work. Specifically, we try to reduce the overhead of symbolic emulation and dynamic binary instrumentation by only selectively enabling them. In the following, we will explain the techniques that achieve this.

When we use *selective symbolic execution*[11] and disable symbolic emulation for certain functions, it is important to correctly synchronize the symbolic state so that the concrete state still represents an instance of the symbolic one. Unlike previous work[11], we do not automatically convert concrete return values to symbolic ones because we noticed a huge performance overhead when we did. Specifically, more symbolic values will increase the number of symbolized branch conditions. As a consequence, we can observe a significant and exponential slowdown as exponentially many more paths exist that the search strategy will try to explore. Furthermore, the additional feasible paths are often rare edge-cases such as being out-of-memory or operating system failures. Therefore, they also do not serve the goal of our search strategy.

Instead of symbolizing return values, we carefully selected functions that in practice do not need to be symbolically executed. Such functions are, for example, the import of additional modules, the initialization of a module and the dynamic name resolution of object members. By default, we will also concretely execute any library functions that we do not whitelist. For library functions with relevant symbolic effects, we manually specify how the symbolic information is propagated by the function and use it to synchronize the symbolic state afterwards. One such example would be Python's internal conversion from a long to a Python int object which is implemented by the function `PyInt_FromLong`. This function uses its own routines to allocate space for a new Python int object and, then, copies the C long into the object's memory. In case the parameter was symbolized, we will have to copy the symbolic expressions saved in the parameter to the new object's memory. However, we would not need to symbolically execute the allocator.

To reduce the overhead of the dynamic binary instrumentation, we employ *selective tracing*. There, we disable the instrumentation for functions where we can foresee that its execution will not affect any symbolic state. Afterwards, the symbolic state is lazily synchronized with the concrete state. Lazy in this context means that we only synchronize the changes when the symbolic state is actually using any of the changed concrete state. One such example where this can be used is to significantly improve the performance is the lookup of attributes in a Python object whose attribute names are purely concrete.

3.4 Output

After successfully exploring the target Python malware, we will output unique sequences of a *interesting* system functions that our framework was able to discover. Those sequences are accompanied by the generated path constraint which represents a model for the behavior the malware expects from the ICS. Additionally, for each message the malware sends to the ICS, we output symbolic expressions which describe those sent messages.

The set of *interesting* system functions can be flexibly changed to include more functions if a more fine-grained analysis would be required. By default, we record the access of files, sent network messages, and every function that introduces input to the program as, for example, the socket receive function does.

CHAPTER 4
IMPLEMENTATION

After having explained our design choices, this chapter will discuss some notable implementation details of our framework. Specifically, we explain the module structure of our framework, modifications made to the Triton dynamic analysis framework's tracer and, finally, details about the modeling of Python functions.

4.1 Framework code structure

The following section describes the components of our framework and how they interact with each other. The symbolic execution component is responsible for interfacing with Triton. This component synchronizes the symbolic state with the concrete state including changes after pure concrete execution (see Section 3.3.4). For this, it hooks the memory accesses via Triton's API and checks whether there was an untracked change in the concrete execution or whether it is the first access of this address. Furthermore, it exposes an API to reset the symbolic state as well as to solve branch conditions.

We also constructed an abstraction layer for the tracer and implemented our logic on top of that. This abstraction layer would allow us to easily exchange the tracer for a different one without having to modify any other logic in our code. An implementation of this abstract interface would need to incorporate functions common for a debugger, including setting/getting memory and register contents, querying the currently loaded modules and hooking the beginning of an instruction.

The *Trace* component stores a list of executed instructions and the symbolized branches encountered. The implementation of the search strategy (excluding the static analysis) and snapshotting are realized inside this component as well. Furthermore, it generates backslices and detects loops in the program base on the recorded trace.

The *Modeling* component exposes an API to hook into function calls and potentially return a modeled function. This hook allows us to add multiple different kinds of modeled functions such as dynamic models (see Section 4.3.2), manually coded models and models generated from a database. Then, the *ModeledFunction* class of this component exposes API's to comfortably interact with the concrete and symbolic state, def-use information for back-slices, and function arguments. Furthermore, it is responsible for generating the output of the framework (see Section 3.4).

The static analysis component is split in two parts. One part resides inside the IDA[29] framework, responsible for the extraction and sending of static information. The other part, residing in the tracer, manages the data and employs caching to reduce the number of queries to IDA. Furthermore, it stores the control flow graph and uses it to compute the path to the possible targets in our search strategy.

The Python modeling component (described in more detail in Section 4.3) consists of a wrapper module that exposes user-friendly API's to the Python object structures so that they can be used as if they would be normal Python objects. For this, it utilizes the extracted type information from the static analysis module. Finally, the component encompasses several modeled Python functions including sockets, files, and the dict type.

4.2 Modifications to the Triton Dynamic Analysis Framework

4.2.1 Symbolic pointers

As discussed in Section 2.1.3, the Triton dynamic analysis framework currently does not handle symbolized pointers and always concretizes them. However, as discussed in Section 2.3, optimizations inside Python result in symbolic pointers. Specifically, small int values and strings of length 1 are optimized so that there exists only one copy of them at all times. As a consequence, the pointer for an int object will be symbolized and when the program tries to access the stored integer value and compare it with another, we will run into a symbolic memory access.

To work with Triton's missing support of symbolic pointers, we manually implemented logic to deal with symbolic memory access. First, we added a callback to the Triton framework for symbolic memory accesses. Then, we used this callback to implement the logic in the codebase of our framework. For symbolic accesses, we create a chain of `ITE` (if-then-else) expressions where we match each possible address the pointer can point to with the value stored at that location. However, for this to work, we need to reliably compute which values a pointer can point to. To achieve this, we first try to extract the used base address of the memory access from the symbolic expression and then use IDA's analysis (and the symbols in the executable file) to guess the start and end of the accessed array. This way, we can compute all possibly accessed elements and build our `ITE`-chain. However, if the extraction of the base address or the guessing of the array bounds fails, we concretize the symbolic memory access instead.

4.2.2 Tracer modifications and porting

To implement features such as selective symbolic execution and selective tracing(see Section 3.3.4), and to port the tracer to Windows, we had to perform some modifications to Triton. First, we had to make the tracer compile on Windows. Doing so, we discovered that Intel Pin and Python have conflicting type declarations on Windows. To fix those, we had to put all Python include statements inside a separate namespace and be careful to include all standard library files included by Python outside of Python's namespace because otherwise they would not be accessible outside from the global namespace.

Next, we encountered library dependency errors because Intel Pin removes the system library search path and only allows to link against the `kernel32.dll` and `ntdll.dll` dynamic libraries. However, Python and its modules are linking against several other system libraries. To solve this problem, instead of implicitly linking against the system libraries, we wrote wrappers around the library functions and explicitly linked against them. This means, that instead of instructing the linker to link against the system libraries, we loaded the system libraries via the `LoadLibrary` call and used `GetProcAddress` to find the addresses of the required system calls.

After solving library dependency errors, we encountered several crashes in the initialization routines before the `main` function was called. To solve this, we first statically linked Python as per the recommendations of the Intel Pin user guide[34]. This allowed us to use Pin and Python together on Windows. Afterwards, we still faced problems running the tracer. We tracked the problem down to the Z3 library: Pin does not support throwing exceptions and Z3 heavily uses exceptions. However, we currently cannot explain why the Z3 library did not cause problems on the Linux version of the Triton tracer as well. To solve this incompatibility, we create another process for Z3 and send our requests to it via the Windows socket API.

Furthermore, to add our optimizations (Section 3.3.4) to the tracer, we had to add Python callbacks to stop and restart the analysis routines. Before, it was only possible to statically select a range of instructions that would be symbolically executed by the Triton framework. For this change, we had to significantly restructure the way Triton allocates memory for instructions, and how it handles conditional instrumentation. After the change, memory for the analysis of an instruction analysis will only occur when it instruction is first symbolically executed. Additionally, we followed Intel Pin's recommendations to inline the routines that conditionally enable and disable the instrumentation for symbolic execution.

Finally, our approach requires the Appcall feature that is present in common debuggers and Intel Pin. Appcall allows the analysis routines to call functions of the analyzed program. This is used in the realization of our Python modeling, e.g. to allocate additional Python objects. To use Intel Pin's Appcall feature inside our Python code base, we built a wrapper around Intel Pin's `PIN_CallApplicationFunction` function. Since C/C++ does not allow to compatibly pass an array of arguments to a *varargs* function, we did not implement the wrapper in C. Instead, we modeled the used enums and structs for Intel Pin's Appcall in Python and used the `ctypes` module to call the program's function. Furthermore, Pin is written in C++ and, therefore, the symbol for `PIN_CallApplicationFunction` is mangled in a platform-dependent way. To access the function with `ctypes` in a platform-independent way, we stored the function pointer in a global variable and exported the variable using the `extern C` directive.

4.3 Python modeling

As discussed in Section 4.1, the Python modeling component is split into two parts: One that is made of several wrappers to expose a comfortable API and one that implements the modeled functions and operators. In the following, we will first explain the internals and structure of Python's object system. Then, using this, we will explain internals of our modeling.

4.3.1 Python's object system

In Figure 4.1, we depict a class diagram of the core types related to the object system. Each value in Python is a `PyObject` that stores a `PyTypeObject` to denote its type. Note that even types are objects themselves and the type of the base `PyTypeObject` is itself. Types store several functions for arithmetic operations, allocation logic, and attribute lookup as well as a list of defined attributes (`tp_members`) and methods (`tp_methods`). The `tp_dictoffset` describes the offset of the `ob_dict` pointer from its object and defines whether an object contains this pointer in the first place.

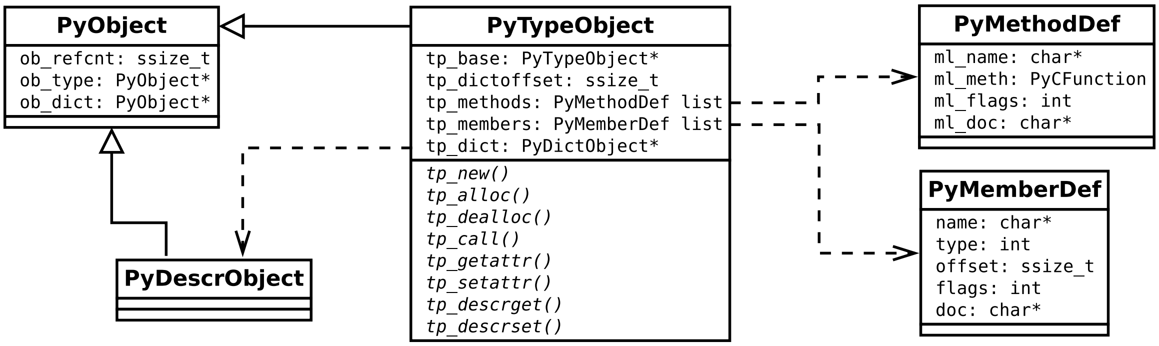

Figure 4.1: Class diagram of Python's object system

The frequent use of dictionary objects (dict/`PyDictObject`) is, as mentioned in Section 2.3, one of the core features of the Python language and is also very frequently used for Python's internal bookkeeping. Specifically, on top of a dictionary to store the object's attributes, each type also contains a dict (`tp_dict`) that stores its class attributes and *descriptors*. *Descriptors* describe how an attribute of an object can be modified and retrieved. Internally, they are used by calling its type's `tp_descrget` and `tp_descrset` functions. Descriptors are added to the type's directory for each attribute that is declared with Python's *slot* features. Using *slots*, the programmer is able to reduce the memory consumption of objects by telling the interpreter to allocate storage for the attribute directly adjacent to the PyObject's memory. In this case, descriptors store the offset of the attribute to the beginning of the object.

The access of an attribute can then be summarized as depicted in Figure 4.2. If there exists a dictionary in one of the base type's dictionaries, Python will use it to access the attribute. In the other case, Python will check whether the object contains a dictionary and will use that for the attribute access.

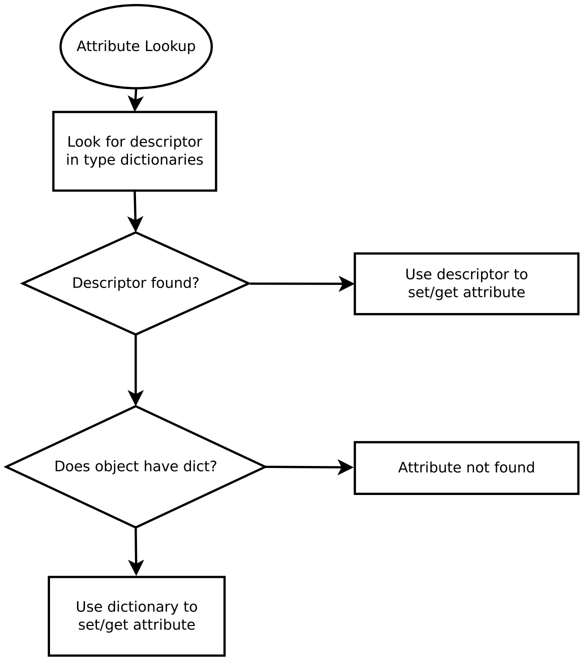
Attribute Lookup
Look for descriptor in type dictionaries
Descriptor found?
Use descriptor to set/get attribute
Does object have dict?
Attribute not found
Use dictionary to set/get attribute

Figure 4.2: Flow of Python's attribute lookup

4.3.2 Modeling

We built our wrappers for Python types similar to Python's internal structure. First, we wrap the `PyObject` itself and implement all generic functions on it. Then, for each wrapped type, we inherit from this class and add routines to extract the type's information from memory. To detect the type of a dynamic Python Object, we extract the type address and look up the symbol corresponding to that address. Using this, we can directly map from the name to the corresponding wrapper.

For the modeling and selective symbolic execution (see Section 3.3.4) of attribute lookups, we will first check whether the looked up attribute is accessed with a descriptor or through the object's dictionary. In case a descriptor is used whose behavior we do not know (e.g. user-defined ones), we do not perform any modeling or *selective symbolic execution* and, instead, symbolically execute the interpreter itself. In every other case, it is safe to disable symbolic execution and just observe the object that was returned by the attribute lookup.

Similarly, the modeling of Python's operators requires dynamic inspection as well: Python is a dynamically typed language, which means the types are not known statically. If we then would like to model an operator function such as `BINARY_OPERATION_MOD_OBJECT_OBJECT_OBJECT` (as previously seen in Figure 3.2), we need to be able to model all possible types. If we would try to build an symbolic expression that supports all types, it would have to include cases and logic for every possible type. As a consequence, the symbolic expression grows very large and will slow down the constraint solving. Instead, we dynamically decide how we model the function based on the types of the passed objects. In case we do not have a model for the specific types, we fall back to concolic execution of the operator function itself. Additionally, since symbolized types are never returned by any of our modeled functions, we can always accurately determine the type of the arguments.

CHAPTER 5
EVALUATION

After describing the technical details of our approach, this chapter focuses on the evaluation of our framework. We will describe the metrics and experiment setup used in our evaluation, then perform experiments on a variety of programs. Specifically, we will first run experiments on a selected number of programs that showcase the correct functioning of our framework. Then, we will evaluate on the Triton malware and one other real world ICS-related sample. Lastly, we will discuss the results of our approach.

5.1 Metrics

Lines of Code (LOC) is the number of lines inside all Python files used by Nuitka, excluding the Python's standard library modules. For this purpose, we use the `cloc`[35] utility to count the lines of code. With the help of this tool, we exclude whitespaces and comments and only count actual lines of code

Executable size is the file size of the executable generated by Nuitka. Nuitka compiles the Python code into C and then uses the compiler (on Windows: MSVC) to build an executable. In this process, it generates and adds a lot of glue code, which will be added to the executable. This generally bloats the binary to a huge size and, therefore, together with the LOC, gives some insight on the efficiency of the transformation from Python to C.

Analysis time is the amount of time passed for the analysis to finish. It is calculated by taking the difference of the time when the analysis was started and when the analysis has finished. To retrieve the time, we use Python's `time.time()` function. The unit of time is minutes.

Reached Python calls is the number of dynamically resolved calls in the binary that we were able to reach. Note, that we exclude dynamically resolved calls inside Python's standard modules such as the `socket` module.

Total Python calls is the total number of dynamically resolved calls that were compiled into the binary with Nuitka. We exclude calls inside Python's standard modules (e.g. `socket`) module.

#Call-Chains is the number of call-chains that we were able to retrieve. A Call-Chain is a sequence of *interesting* calls as defined in Section 3.4.

Depth Call-Chain records the minimum/average/maximum depth of all retrieved unique call-chains.

Constraint queries is the number of all constraint queries to the constraint solver. We increment our counter each time we query the symbolic execution engine.

Constraint time is the number of time spent solving constraints. This is computed by recording the start and end times of a query to the constraint solver and then taking the difference of those. We will then add up the time of all queries and report it. The unit of time is minutes.

Number of solvable symbolized branches (NoSSB) is the number of branch instructions whose condition was dependent on a symbolic input variable and where both branches were deemed solvable by the constraint solver.

Number of branches reverted (NoBR) is the number of instructions with symbolic branch conditions that were *reverted* during the execution. *Reverted* in this context means that the taken branch was not leading to a path to one of the targets and, therefore, we reverted back to that branch instruction to solve for a different successor.

Branches predicted records the total number of branches that were taken because of our branch prediction strategy.

Branches falsely predicted records the number of times the branch prediction was wrong and needed to be updated. Note that a branch can be counted in *Branches falsely predicted* without being counted in *Branches predicted*. Specifically, when no other uncovered Python call reachable is reachable from the predicted branch and it was therefore not symbolically explored.

Instructions executed records the total number of instructions that we concolically executed. This does not include instructions that were executed with *selective tracing* or *selective symbolic execution* (refer to Section 3.3.4).

Block Coverage is the percentage of covered basic blocks to all basic blocks in compiled functions. Note, that since Nuitka adds blocks to check and deal with exceptions after every Python instruction, we can expect lower coverage as would be usually reported for test generating concolic execution frameworks. Furthermore, in libraries there might be more functionality implemented than one can trigger with input.

5.2 Experiment Setup

All tests were run on a Intel Core i7-7700 CPU with 8 cores, 3.5 GHz per core, 14 GB of memory and running a 64-Bit Windows 7. For the functioning of the symbolic analysis, we have to choose the maximum size for received network messages. We chose to use 100 bytes as the maximum size so that one message can contain the full header and also additional data. For the program arguments, we allowed up to 5 program arguments with a maximum length of 16 bytes.

5.3 Functionality Test Programs

We first evaluated the functionality of our concolic execution engine and Python modeling with a manually crafted test set. For this, we first have to define which functionality we want to check. On top of testing the basic symbolic execution, we test features of the Python language that we explicitly modeled as well as parts of the Python language that we deemed possibly challenging for symbolic execution engines. Specifically, we test the following features of Python programs: First, we make sure that a single path execution can finish without problems. Second, we test whether the symbolic input (program arguments, socket input) is properly used and conditions can be solved. Third, we check that no loss of symbolic information occurs when storing variables in a dictionary such as in a class. Fourth, we test that our function models properly work as intended.

In Figure 5.1, we depict the list of our test programs. In Figure 5.1a, we test scenarios related to the program arguments. We check whether we can correctly solve for the length of the arguments and cases related to `IndexError` exceptions when accessing the arguments. Furthermore, it tests the `format` function and string concatenation in Python. Next, in Figure 5.1d we test whether we lose any symbolic information when storing variables inside a class (internally a dict). Then, Figure 5.1b tests whether the modeled hex encoding function is working as intended. Following, Figure 5.1e checks whether the execution of a single path containing float operations is working. Finally, Figure 5.1c tests the socket operations and Figure 5.1f checks whether symbolic checksum expressions over input strings can be built with our modeled operators and, then solved.

```
import sys

def dummy():
    return 1

if len(sys.argv) <= 1:
    print("Hello world!")
    dummy()
else:
    try:
        print("Argument 2: {}"
                .format(sys.argv[2]))
        dummy()
    except IndexError:
        print("IndexError")
        dummy()

    if sys.argv[1] == "haxor":
        print("W3lc0m3 fr13nd, format: {}!"
                .format(sys.argv[1]))
        print("W3lc0m3 fr13nd, addition: "
                + sys.argv[1] + "!")
        dummy()
    else:
        if len(sys.argv) == 2:
            print("Wow, hello! len(argv): {}!"
                    .format(len(sys.argv)))
            dummy()
        else:
            print("Hello {}! len(argv): {}!"
                    .format(sys.argv[1], len(sys.argv)))
            dummy()
```

(a) Argv-Test

```
import sys


def dummy():
    return 1

msg = sys.argv[1]

if msg.encode('hex') == "7365637265746b6579":
    sys.stdout.write("Success!\n")
    dummy()
else:
    sys.stdout.write("Try again!\n")
    dummy()
```

(b) Encoding test

```
import sys
import socket

DEFAULT_PORT = 31337

def dummy():
    return 4 + 5

def run(port):
    sock = socket.socket(socket.AF_INET,
            socket.SOCK_STREAM)
    server_address = ('localhost', port)
    sock.bind(server_address)
    sock.listen(1)
    print("Setting up connection on port: " + str(port))
    print("Waiting for connection")
    connection, client_address = sock.accept()
    connection.settimeout(20)
    print("Connection from " + str(client_address))
    try:
        data = connection.recv(4096)
        if data == "1337h4x0r":
            print(data)
            sock.connect((data, 1337))
            dummy()

        sys.stdout.write(data)
        connection.sendall(data)
        connection.send(data)
    except socket.timeout:
        print("Timeout received")
        dummy()
    connection.close()

def do_stuff():
    if len(sys.argv) > 1:
        port = int(sys.argv[1])
        run(port)
    else:
        port = DEFAULT_PORT
        dummy()

do_stuff()
```

(c) Socket Test

```
import sys

class Test(object):
    def __init__(self, val):
        self.val = val
        self.str = ""

def dummy():
    return 1


t = Test(3)

msg = sys.argv[1]

t.val = int(msg)

if t.val == 3:
    sys.stdout.write("Success!\n")
    dummy()
else:
    sys.stdout.write("Try again!\n")
    dummy()

msg = sys.argv[0]

t.str = msg

if t.str == 'd':
    sys.stdout.write("Success string!\n")
    dummy()
else:
    sys.stdout.write("Try again string!\n")
    dummy()


t.stri = sys.argv[2][3]

if t.stri == 'g':
    sys.stdout.write("Success string_subscript!\n")
    dummy()
else:
    sys.stdout.write("Try again string_subscript!\n")
    dummy()
```

(d) Class Test

```

f = 3.5
f = f / 2.0
f = 1 / f
f = f / 3

f = int(f) / 1.
f = int(f) * f
f = int(f) + 2.5
f = -2.5 + int(f)
f = long(f) - 2.5

print "Success: {}".format(f)
```

(e) Float Test

```

import socket

result = 2139

def dummy():
    return 4+5

s = socket.socket()

recv = s.recv(100)

i = 0
r = 1
for c in recv:
    r += ord(c)

if r == result:
    dummy()
    print("Correct")
    dummy()
    dummy()
else:
    dummy()
    print("Wrong")
    dummy()
```

(f) Operator test

Figure 5.1: List of test-programs

Table 5.1: Analysis result of functional test programs

Metric	Argv	Encoding	Socket	Class	Float	Operator
LOC	29	10	37	31	10	19
Executable Size	632KB	630KB	763KB	635KB	629KB	756KB
Total Python Calls	10	5	16	14	1	6
Analysis time	1.38	1.32	1.70	1.49	1.16	1.69
Instructions executed	2522	1170	13726	3829	1674	23052
Block coverage	46.69%	45.78%	34.79%	41.55%	45.26%	48.74%
Reached Python Calls	10	5	16	14	1	6
Constraint time	0.05	0.02	0.04	0.08	0.00	0.03
Constraint queries	37	7	59	30	0	4
NoSSB	4	2	3	6	0	1
NoBr	4	2	3	4	0	1
Branches predicted	0	0	0	0	0	0
Branches falsely predicted	1	0	0	0	0	0

Results. We depict the gathered metrics in Table 5.1. We exclude metrics related to call-chains as they hold no purpose in the context of functional tests. First, note that our framework was able to correctly analyze all test programs and discover all paths. This can be seen when comparing the *Reached Python Calls* with the *Total Python Calls*. Respecting this, the *block coverage* metric gives us some insight about how many unreachable blocks are added during the transformation to C. The test programs with only one function generally achieved a *block coverage* of around 45% while programs with more than one defined function (*Socket*, *Class*) achieve lower coverage. We attribute this observation to the additional blocks that are inserted in each function to handle possibly thrown exceptions. Specifically, because exceptions in the functions of *Socket* and *Class* are not feasible, our framework is also not able to cover them, leading to a lower coverage in total. For the branch prediction, we can observe that it is generally not used for these test programs. This makes sense: The programs do not call any validation routine multiple times for which our branch prediction was designed for. For *Argv*, there exists one falsely predicted branch and zero predicted branches. This is because the exploration will reach the check for "haxor" twice and remember the taken branch on the first encounter. Therefore, the second time we encounter this branch, it will be counted as a falsely predicted one. For every program apart from *Class*, we can observe that *NoSB* equals *NoSBr*. This is the case because for all those programs, all encountered branches need to be eventually solved. For *Class* however, there are two branches that do not need to be solved to discover every function call: Specifically, they occur during the accesses of `sys.argv[1]` and `sys.argv[2][3]`. The first statement can throw an `IndexError` exception and the second statement can throw it on two occurrences. Since these exceptions would abort the program, a thrown exceptions cannot lead to any uncovered Python Call. Therefore, our framework also does not try to cover them. However, of those 3, the access of `sys.argv[2]` needs to be reverted once because the concrete execution will initially throw an exception because the number of arguments is initialized to 1. For the executable sizes, we can observe that all sizes are similar and only two programs are significantly larger. Those programs use the `socket` module which is then also transformed to C and linked to the final executable.

5.4 ICS-related Samples

After evaluating our approach on manually created functionality test programs, this section focuses on the evaluation on ICS-related samples. For this purpose, we will first give a short description of the samples and then explain the results of our experiments.

5.4.1 Triton Malware

The Triton malware was originally deployed as an executable created by Py2EXE[36]. Py2EXE compiles all required Python modules for a program to byte code (`.pyc` files), bundles them, and creates an executable that will later call the Python Interpreter to run the byte code. The byte code can be retrieved using unpy2exe[37] and decompiled using uncompyle6[38].

Next, we will give an overview of the Python code of Triton. In Figure 5.2 we depict an overview of Triton's components and their functions. We excluded functions that are present in the modules but were not used by the payload. The goal of the Python malware is to inject its own binary program into the ICS controller. The main module can achieve this by using the SafeAppendProgramMod function of the TsHi module. After injection, it waits and queries the program status with GetProjectInfo.

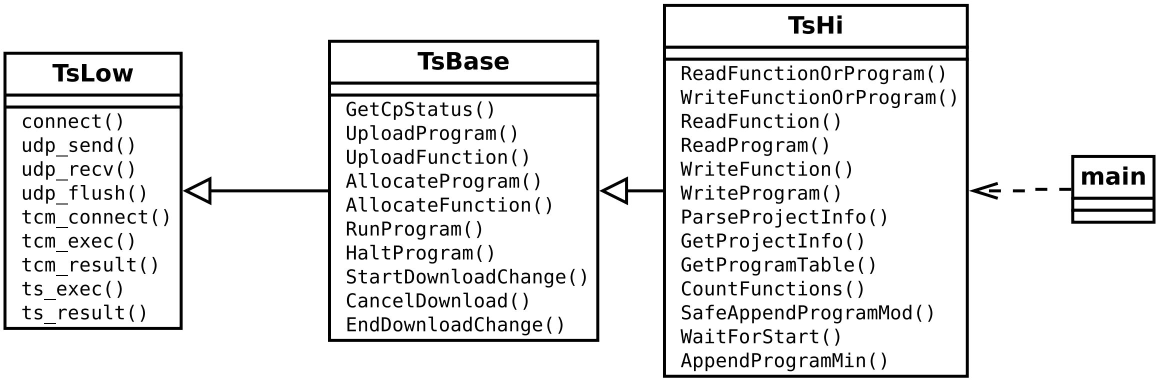

Figure 5.2: Components of the Triton Malware's Python code

The TsLow module implements the low level layer of the network protocol. The protocol uses UDP and consists of two layers again: The tcm and the ts layer. The TsBase layer wraps the low level commands sent via ts_exec in functions that are named after the intended purpose. These functions are then used by the TsHi layer to implement the high-level functionality.

The SafeAppendProgramMod function of the high level module will first query the state of the machine via GetProjectInfo, then enumerate all stored programs and functions. Using this information, it will then append its program to the program list with AppendProgramMin and wait until the ICS is in a running state.

5.4.2 PyModbus

For the evaluation on another ICS-related sample, we chose the widely used and openly published Modbus protocol[39]. Modbus is an application layer protocol and is used on top of other communication protocols such as TCP, UDP and serial RTU. The basic functionality allows to read and write *coils*, which are represented by single bits, and to read and write 16-bit registers. Registers and Coils can both be addressed with 16-bit values.

For our evaluation, we received a GUI controller program from the Communications Assurance and Performance (CAP) group from the School of Electrical Engineering at the Georgia Institute of Technology. The program could be split into two parts: one controller class and one class for managing the GUI. Regarding the latter, the concolic execution of GUI programs is out of the scope of this work, and so we removed the GUI and instead built a test routine which calls each of the controller's function after another. The controller class wraps, similar to Triton's `TsBase` class, the low level Modbus commands in several functions with meaningful names. It uses the Modbus protocol over TCP to communicate with the PLC and uses the PyModbus[40] library as an implementation thereof. For our evaluation, we used the most recent version of PyModbus, 2.3.0.

5.4.3 Results

Summary. In Table 5.2, we depict the results of our experiments. There, we can first observe that the PyModbus program is larger in size as can be seen by *LOC*, *Executable Size* and *Total Python Calls*. For Triton, the analysis took 36.98 minutes and for PyModbus 69.39 minutes. We were able to cover far more branches in Triton than we were able to in our PyModbus sample. For Triton, our branch prediction was applied 12 times and only one prediction needed correction. In PyModbus, the number of of Call-Chains is only one with a size of 15. In Triton, we discover 3 call-chains and the maximum depth is 13.

Table 5.2: Analysis result of the Triton Malware and PyModbus

Metric	Triton Malware	PyModbus Sample
LOC	961	7028
Executable size	1021KB	2873KB
Total Python calls	184	1100
Analysis time	36.98	69.39
Instructions executed	149993	509322
Block coverage	16%	4%
Reached Python calls	74	126
Constraint time	6.13	8.11
Constraint queries	1340	6294
NoSSB	47	414
NoBr	20	3
Branches predicted	12	0
Branches falsely predicted	1	0
#Call-Chains	3	1
Depth Call-Chain (min/avg/max)	10.00/11.00/13.00	15.00/15.00/15.00

Analysis Progress. For Triton, we were able to successfully analyze all low-level functions, extract symbolic expressions for the sent data and extract path constraints for the received data. In the high-level class, we reached the function `SafeAppendProgramMod` that queries the state of the ICS and appends malicious code to the targeted controller. We were successful in passing Triton's validation of the ICS state and were able to extract the path constraints for it. However, our framework then got stuck in the function `GetProgramTable` which repeatedly calls `UploadProgram` to extract all stored programs in the ICS controller. We are able to analyze the UploadProgram function but currently fail inside the loop of GetProgramTable due to a bug in our loop detection. The *Depth Call-Chain* metric gives us more specific insight of how much communication has occurred as each sent and received messages increases it by 1. In fact, we could observe five received and five sent messages. The other three elements in the chain are two file accesses and the symbolization of the program arguments. For PyModbus, the analysis failed parsing the responses because we currently lack support to index a dictionary with a symbolic value. Specifically, PyModbus extracts the Modbus *function code* value from the received message header and uses a dictionary to select the correct class to store the response data. Therefore, we were only able to extract the sent messages from PyModbus but not any meaningful path constraints.

Branch prediction. For PyModbus, our branch prediction could not be used because the analysis failed. For Triton, however, the branch prediction was used and shows good results as 12 predicted branches were taken and only once a branch was falsely predicted. The falsely predicted branch is the timeout branch in the udp_recv function. Specifically, the udp_flush receives data until it times out and, therefore, requires a timeout to progress. After leaving the loop, the branch prediction remembered the branch to the timeout which then led to one falsely predicted branch. When we compare the number of branches that needed to be reverted (*NoBR*) with the number of predicted branches, we observe that a relative high amount of branches did not need to be reverted thanks to our branch prediction. This shows that our branch prediction has significantly improved the search-strategy for this application.

Performance. As depicted in Table 5.2, the *constraint time* is only a relatively small fraction of the overall *analysis time*. To further analyze which part of our framework uses up the most time, we used cProfile[41]. as a profiler. With the help of this tool, we found out that we spend a relatively large amount of time in two routines. One routine checks whether the current instruction is in the set of blocks that can reach another undiscovered Python call. The other one returns the basic block given an address. Even after applying multiple optimizations to our code and significantly speeding up those functions, they remain the most expensive in total.

5.5 Discussion

While we could not fully explore the Triton malware, we argue that our extracted information can serve as models for the targeted ICS software. Specifically, we were able to record several valid messages sent from the malware to the ICS and extract constraints of data expected by the malware. Most notably, we were able to extract these for the low-level protocol functions. We argue that since the header of most protocols uniquely identifies them, our extracted information would also already be sufficient to identify the protocol of the targeted ICS.

While one could argue that then also the message sent to initiate a connection to the ICS should be enough to identify the protocol, the Triton malware proofs the opposite: While the targeted ICS uses the TS and TCM protocol, only the TCM protocol is used to establish a connection. Therefore, analyzing the initial connection request, one would only be able to detect the use of the TCM protocol. With our approach, however, we are also able to extract information regarding higher level protocols such as the TS protocol. This shows that our framework, even with limited results, can significantly contribute to identifying potential attack targets of Python ICS malware.

CHAPTER 6
LIMITATIONS

6.1 Symbolic dictionary lookups

Currently, the check whether a specific key exists in a dictionary is concretized. As a consequence, we were not able to completely pass the parsing routine of the PyModbus library. Complete symbolic modeling of dictionaries is a huge challenge as keys can have arbitrary types including classes defined in the analyzed program's code. For limited applications such as dictionaries with only integers and strings, this can be implemented analogously to our implementation of symbolic pointers (Section 4.2.1). This is up to future work.

6.2 Python version

Our framework currently only supports the Python version 2.7. This is due to the fact that the Triton malware was written in that Python version and the other Python samples we found were also either written or compatible with this version. For the support of Python versions 3.x we do not expect significant overhead. However, some further modeling will need to be performed for the distinction between the `bytes` and `str` types and the respective encoding functions. This support is up to future work.

6.3 Long values

Python allows its integer values to hold values of any size. In Python 2, there are two different integer types: `int` and `long`. The former is a 32-bit or 64-bit value depending on the platform and the latter is an integer of arbitrary size. If the `int` type is no longer able to hold the value, it will automatically be converted to the `long` type. Since this conversion is non-trivial to model and we could not observe the usage of arbitrary size integers yet, this is part of future work.

6.4 Floating point values

Since the Triton dynamic analysis framework currently does not support operations for floating operators, we inherit this limitation. However, as our evaluation has shown, the analysis of malware protocols did not require floating point values. Furthermore, constraint solvers do have support for floating point operations and previous work has shown that it is possible to add this support to symbolic execution engines[26]. Therefore, adding floating point support to Triton is up to future work.

6.5 Exec-Statements

One crucial limitation of our approach is that our search strategy cannot efficiently deal with control flow inside an exec statement. This is due to the fact that exec statements are not compiled into C code and, therefore, our framework does not have any control flow information about it. In this case, our strategy will fall back to pure DFS. However, we argue that this generally does not limit the applicability of our framework. For one thing, it should be possible extract the argument of the exec function and replace the exec statement in the Python codewith the executed string. In case the exec argument depends on some input, it should be possible to extract the control-flow deciding components inside the statement and wrap only the dynamic parts inside an exec statement. For another, in a more advanced approach it should also be possible to use Nuitka's approach to compile the dynamically executed statement to assembly just-in-time and extract control-flow information from there.

6.6 Operating system

Our framework was currently only tested and run on the Windows operating system. This stems from the fact that our motivation was to analyze malware targeting industrial control systems, which, from our experience and analyzed samples, usually run inside a Windows operating system. However, our components are platform-independent and our main codebase is written in Python and, therefore, porting the framework to other operating systems is possible in future.

CHAPTER 7

RELATED WORK

7.1 Malware Analysis

Prior work uses symbolic execution in a variety of applications including automatic discovery of bugs[9, 10, 11, 15], malware analysis[42, 3, 2, 43] and malware classification[44]. Yadegari et al.[45] study symbolic execution on obfuscated code including obfuscations that transform the program into an interpreting virtual machine[46]. Our approach analyzes malware written in Python, which is usually compiled into byte-code and then interpreted as well. Moser et al.[42] and Brumley et al.[43] use symbolic execution to explore multiple paths of a malicious binary and identify malicious behavior that is triggered only on certain conditions (e.g. on a specific day). On the contrary, our work focuses on deeply exploring the common paths and does not actively try to discover trigger-based behavior. If the malware hides its common behavior with such triggers, they might need to be discovered first so that our approach can further explore the deeper parts of the malware. Baldoni et al. explore the use of symbolic execution to extract sequences of commands in Remote-Access-Trojans (RAT) and Borzacchiello et al.[3] extend this work to reconstruct the respective C2 (Command & Control) server. The goal of this work is similar to our approach as we also try to extract sequences of commands but from the malware to the targeted ICS.

7.2 Concolic execution of Python

Conpy[7] is a concolic execution engine for Python that allows symbolic execution of `str` and `int` types. Contrary to our work, it is directly implemented inside Python and does not rely on the analysis of the interpreter. Instead, Conpy relies on manual overloading of

Python's internal functions (such as ord and chr). CHEF[5] is a framework for prototyping symbolic execution engines for interpreter languages and has been used to build concolic execution engines for Lua and Python. Similar to our approach, it uses the interpreter itself to implement the language's semantics. CHEF relies on manual modifications to the interpreter, such as disabling previously optimizations, modifying hash-functions and instrumenting the interpreter loop to communicate the high-level program counter to the framework.

CHAPTER 8
CONCLUSION

Motivated by the potential catastrophic consequences of the ICS malware Triton, we proposed a new method for concolic execution of Python by transforming it to C. To this end, we have employed the tool Nuitka to generate the C code and developed a tool based on the Triton symbolic execution engine to analyze the malware.

To adapt the symbolic execution engine to efficiently analyze the Python interpreter that is used by the generated code, we implemented multiple function models that describe the behavior of Python's operations. We specifically focused on modeling the lookup of attributes and accesses to dictionaries as they are the most commonly used constructs in Python.

To speed up the exploration of network protocol code, which often employs validation functions that check the structure of a message before the semantic content is retrieved, we have proposed a history-based branch predictor. This branch predictor remembers the correct path in the validation function and uses it when it explores it next.

We have shown that we can successfully symbolically execute Python programs with a variety of test programs. Then, we have used our framework on the Triton malware and a PyModbus sample to show its capabilities of extracting ICS information. While the analysis of PyModbus failed due to a missing feature in our symbolic modeling, we were able to extract significant information about the Triton malware's communication with the targeted ICS.

REFERENCES

[1] M. Giles, *Triton is the world's most murderous malware, and it's spreading*, `https://www.technologyreview.com/s/613054/cybersecurity-critical-infrastructure-triton-malware/`, Retrieved June 21, 2020.

[2] R. Baldoni, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Assisting malware analysis with symbolic execution: A case study," Jun. 2017, pp. 171–188, ISBN: 978-3-319-60079-6.

[3] L. Borzacchiello, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Reconstructing c2 servers for remote access trojans with symbolic execution," in *Cyber Security Cryptography and Machine Learning*, S. Dolev, D. Hendler, S. Lodha, and M. Yung, Eds., Cham: Springer International Publishing, 2019, pp. 121–140, ISBN: 978-3-030-20951-3.

[4] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[5] S. Bucur, J. Kinder, and G. Candea, "Prototyping symbolic execution engines for interpreted languages," *SIGARCH Comput. Archit. News*, vol. 42, no. 1, 239–254, Feb. 2014.

[6] S. Sapra, M. Minea, S. Chaki, A. Gurfinkel, and E. M. Clarke, "Finding errors in python programs using dynamic symbolic execution," in *Testing Software and Systems*, H. Yenigün, C. Yilmaz, and A. Ulrich, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 283–289, ISBN: 978-3-642-41707-8.

[7] T. Chen, X.-s. Zhang, R.-d. Chen, B. Yang, and Y. Bai, "Conpy: Concolic execution engine for python applications," in *Algorithms and Architectures for Parallel Processing*, X.-h. Sun, W. Qu, I. Stojmenovic, W. Zhou, Z. Li, H. Guo, G. Min, T. Yang, Y. Wu, and L. Liu, Eds., Cham: Springer International Publishing, 2014, pp. 150–163, ISBN: 978-3-319-11194-0.

[8] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[9] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008.

[10] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, “QSYM : A practical concolic execution engine tailored for hybrid fuzzing,” in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761, ISBN: 978-1-939133-04-5.

[11] V. Chipounov, V. Kuznetsov, and G. Candea, “The s2e platform: Design, implementation, and applications,” *ACM Trans. Comput. Syst.*, vol. 30, no. 1, Feb. 2012.

[12] *Triton: A Dynamic Symbolic Execution Framework*, SSTIC, 2015, pp. 31–54.

[13] J. Honig, “Autonomous exploitation of system binaries using symbolic analysis,” 2017.

[14] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, “Efficient state merging in symbolic execution,” in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’12, Beijing, China: Association for Computing Machinery, 2012, 193–204, ISBN: 9781450312059.

[15] S. Cha, T. Avgerinos, A. Rebert, and D. Brumley, “Unleashing mayhem on binary code,” pp. 380–394, May 2012.

[16] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Krügel, and G. Vigna, “Driller: Augmenting fuzzing through selective symbolic execution,” in *NDSS*, 2016.

[17] K. L. McMillan, “Lazy annotation for program testing and verification,” in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 104–118, ISBN: 978-3-642-14295-6.

[18] S. Khurshid, C. S. Păsăreanu, and W. Visser, “Generalized symbolic execution for model checking and testing,” in *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS’03, Warsaw, Poland: Springer-Verlag, 2003, 553–568, ISBN: 3540008985.

[19] K.-K. Ma, K. Yit Phang, J. S. Foster, and M. Hicks, “Directed symbolic execution,” in *Static Analysis*, E. Yahav, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 95–111, ISBN: 978-3-642-23702-7.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” *SIGPLAN Not.*, vol. 40, no. 6, 190–200, Jun. 2005.

[21] H. Xu, Z. Zhao, Y. Zhou, and M. R. Lyu, “Benchmarking the capability of symbolic execution tools with logic bombs,” *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2018.

[22] J. Salwan, *Smt array issue #806*, `https://github.com/JonathanSalwan/Triton/issues/806`, Retrieved June 21, 2020.

[23] C. F. Bolz and A. Rigo, "How to not write virtual machines for dynamic languages," 2007.

[24] *The python programming language*, `https://github.com/python/cpython`, Retrieved June 21, 2020.

[25] *The stackless python programming language*, `https://github.com/stackless-dev/stackless`, Retrieved June 21, 2020.

[26] D. M. Perry, A. Mattavelli, X. Zhang, and C. Cadar, "Accelerating array constraints in symbolic execution," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017, Santa Barbara, CA, USA: ACM, 2017, pp. 68–78, ISBN: 978-1-4503-5076-1.

[27] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science Engineering*, vol. 13, no. 2, pp. 31 –39, 2011.

[28] K. Hayen, *Nuitka*, `http://nuitka.net`, Retrieved June 21, 2020.

[29] C. Eagle, *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. USA: No Starch Press, 2011, ISBN: 1593272898.

[30] R. Thomas, *Lief - library to instrument executable formats*, https://lief.quarkslab.com/, Apr. 2017.

[31] D. Vostokov, *WinDbg: A Reference Poster and Learning Cards*. Opentask, 2008, ISBN: 190671729X.

[32] *Gdb: The gnu project debugger*, `https://www.gnu.org/software/gdb/`, Retrieved June 21, 2020.

[33] S. Mittal, "A survey of techniques for dynamic branch prediction," *Concurrency and Computation Practice and Experience*, Apr. 2018.

[34] *Pin: Pin 2.14 user guide*, `https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/index.html`, Retrieved June 21, 2020.

[35] *Cloc - count lines of code*, `http://cloc.sourceforge.net/`, Retrieved June 21, 2020.

[36] *Py2exe*, `https://www.py2exe.org/`, Retrieved June 21, 2020.

[37] M. Bordese, *Unpy2exe*, `https://github.com/matiasb/unpy2exe`, Retrieved July 9, 2020.

[38] R. Bernstein, *Uncompyle6*, `https://github.com/rocky/python-uncompyle6`, Retrieved July 9, 2020.

[39] I. Modbus, “Modbus application protocol specification v1. 1a,” *North Grafton, Massachusetts (www. modbus. org/specs. php)*, 2004.

[40] RiptideIO, *Pymodbus*, `https://github.com/riptideio/pymodbus`, Retrieved July 12, 2020.

[41] *The python profilers*, `https://docs.python.org/2/library/profile.html#module-cProfile`, Retrieved June 21, 2020.

[42] A. Moser, C. Kruegel, and E. Kirda, “Exploring multiple execution paths for malware analysis,” Jun. 2007, pp. 231–245, ISBN: 0-7695-2848-1.

[43] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, “Automatically identifying trigger-based behavior in malware,” in *Botnet Detection: Countering the Largest Security Threat*, W. Lee, C. Wang, and D. Dagon, Eds. Boston, MA: Springer US, 2008, pp. 65–88, ISBN: 978-0-387-68768-1.

[44] S. Sebastio, E. Baranov, F. Biondi, O. Decourbe, T. Given-Wilson, A. Legay, C. Puodzius, and J. Quilbeuf, “Optimizing symbolic execution for malware behavior classification,” *Computers & Security*, vol. 93, p. 101 775, 2020.

[45] B. Yadegari and S. Debray, “Symbolic execution of obfuscated code,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’15, Denver, Colorado, USA: Association for Computing Machinery, 2015, 732–744, ISBN: 9781450338325.

[46] H. Fang, Y. Wu, S. Wang, and Y. Huang, “Multi-stage binary code obfuscation using improved virtual machine,” Oct. 2011, pp. 168–181.