A TRIDENT SCHOLAR PROJECT REPORT

NO. 537

Increasing Application Security Through Interpretation

by

Midshipman 1/C Jack C. Metcalf, USN

UNITED STATES NAVAL ACADEMY
ANNAPOLIS, MARYLAND

USNA-1531-2

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
5-16-23		

4. TITLE AND SUBTITLE
Increasing Application Security Through Interpretation

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Jack C. Metcalf

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Naval Academy
Annapolis, MD 21402

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
Trident Scholar Report no. 537 (2023)

12. DISTRIBUTION / AVAILABILITY STATEMENT

This document has been approved for public release; its distribution is UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In this project, we examine the potential for reducing vulnerabilities in legacy code through program language translation. Our compiler translates source code written in C, a low-level language notorious for security issues, into secure, interpreted Python code. The resulting code is functionally identical, and is produced using C and Python abstract syntax trees. We analyze the effectiveness of this vulnerability reduction by testing the resulting code against known C vulnerabilities, provided by the NIST Juliet Test Suite. Using Juliet, we show that the resulting Python code is less vulnerable to C memory errors such as the buffer overflow and null pointer dereference.

15. SUBJECT TERMS
Vulnerability reduction, Computer language translation, NIST Juliet Test Suite

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**		10	**19b. TELEPHONE NUMBER** *(include area code)*

U.S.N.A. --- Trident Scholar project report; no. 537 (2023)

INCREASING APPLICATION SECURITY THROUGH INTERPRETATION

by

Midshipman 1/C Jack C. Metcalf
United States Naval Academy
Annapolis, Maryland

Certification of Adviser Approval

Professor Daniel S. Roche
Computer Science Department

Acceptance for the Trident Scholar Committee

Professor Maria J. Schroeder
Associate Director of Midshipman Research

USNA-1531-2

1 Abstract

In this project, we examine the role of interpreted, higher level languages in the field of vulnerability reduction. The C programming language, despite its widespread use, has many vulnerable features that create an attack surface for an otherwise secure program. Because interpreted languages have more runtime checks than compiled binaries, they are not typically vulnerable to the same kinds of attacks. Furthermore, insecurities discovered in an interpreted language can be eliminated through patching the interpreter itself, rather than needing to update every program written in the language.

Much of what makes the C language insecure is its 'undefined behavior.' Rather than crash when the programmer does something they shouldn't, C programs continue executing, and many different things can happen depending on the environment, the input to the program, and the timing of the program. This is a known in many C standards as "undefined behavior," and is a common vector for attackers. Because the actual behavior that occurs when a program enters an undefined state varies with the execution environment, attackers are able to manipulate the environment to control the program. Python, however, does not allow this gray area. When a Python program performs an action that is not clearly defined, the program displays an error message and exits. This behavior decreases the attack surface for Python programs, making them more secure.

Additionally, as a compiled language, C cannot perform runtime checks. A compiler is limited in that it can only perform checks when creating a program executable, but an interpreter has oversight on a process while it is running. While compilers for the C language can only statically view a program, the Python interpreter can examine the program dynamically as it executes, giving a view of the environment during execution.

To explore this method of increasing program security, we created a C to Python transpiler. A transpiler is a type of compiler, though instead of compiling a program to machine code it converts source code written in one language into source code written in another. The tool we created for this project is attempting to prove that by translating between C and Python, with no security-centered design choices, programs are less vulnerable to common exploits.

Our transpiler is built using the LLVM project's libclang library and the Python AST library. It uses the Python3 libclang bindings to expose and manipulate the C Abstract Syntax Tree (AST), and then uses the Python AST library to create an equivalent Python tree that is reassembled into source code. While our aim is to achieve functional equivalence and clarity in the produced Python code, key differences in how Python handles memory allocation, variable typing, and structures force us to make design decisions to accommodate.

The resulting Python programs excel in reducing the security impact of the trademark memory errors that have plagued C programs for decades. Array-out-of-bounds accesses, buffer overflows, and integer overflows are all effectively eliminated in the output Python code, creating a safer, albeit slower, program with equivalent functionality to the original. To test the effectiveness of our vulnerability reduction, we tested vulnerable C programs from the NIST Juliet test suite against their Python equivalent, examining whether or not the vulnerabilities were still present.

2 Background

2.1 Significance

Machine code is often a source of vulnerabilities in user applications, with one of the most common exploits being the buffer overflow. Buffer overflow attacks have plagued developers for decades in C and C++ [1], as the two languages have no built-in protection to memory access. There are many insecure functions and practices in C that a bad actor can take advantage of and produce undefined behavior or an unintended line of execution. In fact, C has been found to be the most vulnerable language in modern programming [11]. There are few checks at compile-time in a C program to ensure that a developer is writing code that is not flawed – the language assumes a lot of trust from the developer. While the lack of oversight is a positive for the seasoned kernel contributor that knows exactly how their pointer aliasing is going to affect the environment they are operating in, it is possible for the inexperienced programmer to inadvertently create vulnerable

code. Unsafe array accesses, input buffering, and command timing are all potentially unsafe operations in C, and there is little checking built into the language that safeguards the user.

Because of the speed of C, however, it is still widely used, and developers have found ways to check and identify vulnerabilities in binaries through three main types of analysis: static, dynamic, and hybrid. Static analysis programs will scan through the machine code of an application before it is run to search for well known weaknesses and practices that are known to be malicious. Dynamic analysis, on the other hand, entails executing the machine code for a short period of time in an environment where the behavioral execution patterns are monitored for any techniques that appear in malware. Finally, hybrid analysis is a combination of the two preceding approaches.

These techniques are not perfect, however, and have their cost. According to a study performed on a public C/C++ benchmark suite with 5 memory vulnerability detectors, there was a distinct trend throughout the detectors to have exemplary recall of one type of vulnerability, at the expense of others [9]. There was no detector tested that had a perfect detection score, nor was there a single tool that outperformed the others in every type of vulnerability. While in this particular case the static tools tended to perform better than the dynamic tools, it could be argued that even that advantage could be neutralized, as the dynamic tools ran faster on the tests performed.

2.2 Related Work

This is not the first project to convert C code into Python, but it is the first to do so in order to reduce program vulnerabilities. Much of the other programs that exist to convert C to Python function as a teaching tool, with the goal of making C programs more understandable. These programs operate with very specific rules, and make design decisions not to maximize functionality, but readability [13]. There have been many applications, however, that convert Python to C. Cython, one such program, has the goal of optimizing the speed at which a Python program can compile and run, essentially turning Python into a compiled language. It also allows Python to have explicit type declarations, fundamentally changing one of the main aspects of the language [12].

There have also been many attempts to eliminate potential vulnerabilities in C applications. Given that C is widely considered to be the de facto programming language for lower-level application and OS design, its frequent memory insecurity has been a challenge to developers for decades. While changing coding conventions and avoiding unsafe practices may help the security of an application, it is insufficient in isolation in making a program commercially secure. Instead, many programs seek to analyze existing binaries for insecurities through the two practices mentioned earlier in the paper: static and dynamic analysis.

One of the most popular static analysis tools that attempts to evaluate binary security is the C Bounded Model Checker, or CBMC. CBMC is a bounded model checker that analyzes array bounds, use of pointers, memory-related undefined behaviors for vulnerabilities. This tool is powerful in that the program does not need to be run for the checking to occur, yet it still can alert the user to potential insecurities in their program [9] [7]. Valgrind, a heavyweight dynamic analysis framework [8], is one of the most popular tools for dynamic analysis, providing developers with very robust capabilities for machine code dissection. Memcheck, a tool implemented using Valgrind, attempts to analyze and detect many memory errors, namely undefined value errors. Memcheck works by checking the addressability of every byte in memory, the heap, and memory blocks supplied to functions for undefined references, overlap, and leaks [14]. With memcheck, a user could achieve much of the knowledge about vulnerabilities in their application that we seek to eliminate with our compiler.

2.3 Compilation vs Transpilation

A compiler is a program that takes source code through multiple states of analysis and optimization in order to create a machine code executable that, when run, executes the commands specified in the original program. This compilation is generally done in the following 5 steps:

1. Scanning / lexical analysis: Lexical analysis, or simply scanning, is the first step in the compilation process. When given raw source code, the scanner tokenizes the program. Each programming language has its own language of tokens that represents all possible code segments that can appear in a program. Tokens are defined with regular expressions, a standardized representation of rule-based string patterns.

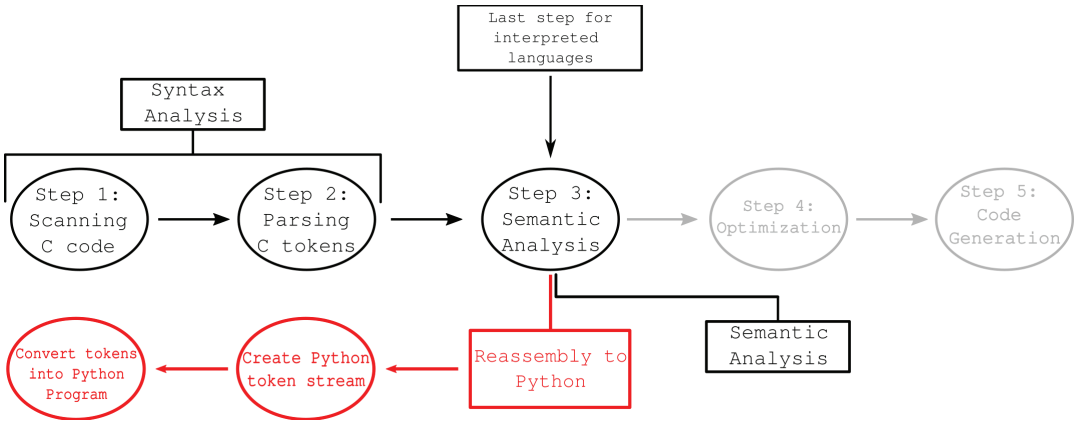

Figure 1: Compilation Process (black/gray) and Transpilation Process (black/red)

To accomplish the tokenization, scanner uses the regex definitions to sequentially run through the program and translate each piece of code to its corresponding token, generating a token stream of the program.

2. Parsing / syntax analysis: Given the token stream generated by the scanner, the parser's goal is to generate a parse tree, or abstract syntax tree (AST) of the program. The parser is dictated by a Context Free Grammar, a set of progressively generated rules, that defines the possible statements of a language. The AST is composed of terminals and non-terminals generated from logical chunks of meaning in the source code, and is the input to the semantic analysis of the program detailed below.

3. Semantic analysis: Once the AST has been generated for a program, the compiler can begin to analyze the semantics of the source code. Both scanning and parsing only deal with the syntax of the program, but semantic analysis is where the compiler can start to translate the meaning of the program that is independent of the language that it was written in. The Abstract Syntax Tree is generally populated with generic labels, in which case it is considered undecorated, but if the compiler chooses to fill in the values that each token represents, it is considered decorating the tree.

4. Optimization: The only optional part of the compilation process, optimization is a step that each compiler does slightly differently. In general, however, each compiler makes a couple of optimizations to the AST in order to make the output slightly faster or more space efficient. Simple computations, unused variable declarations, and redundant loops are often optimized away to create a smaller, more efficient AST.

5. Code Generation: Code generation is the final step in the compilation process, and it is where the Abstract Syntax Tree is either transformed into machine code or some Intermediate Representation (IR), once again independent of application language. After the code generation step, the resulting code is only dependent on the target architecture, assembly language, or intermediate representation.

In this project, however, we succeed by modifying the compilation process. In order to output source code, different steps are taken. The process is called transpilation, and is shown in red in Figure 1. Instead of performing the last two steps of the compilation process, a transpiler reverses course and does steps 2 and 1 in reverse. Instead of outputting an executable or intermediate representation, our transpiler outputs Python source code.

2.4 Memory in C

The C programming language is vulnerable to attack in many ways, but at the core of many of them is an attackers exploitation of the conventions in which memory is laid out. In C, one of the primary working spaces a process has to operate within is called *the stack*. The stack is a dynamic section of memory where

local data defined and altered throughout the course of a program's execution is stored. When a function is called, a *stack frame* is assembled, and all variables used in that function are defined there. It is a powerful concept that allows local variables to operate with the proper scoping, but when an attacker is able to find an error in a C program, one of the most common vectors to escalate that error into an exploit is to leverage the predictability of the stack. As shown in Figure 2, immediately beneath the main function's stack frame are saved registers, which are pointers to locations in memory. One of those pointers, called the saved instruction pointer, points to where the program should return execution to after the working function is completed. An attacker with enough control over the stack frame can overwrite that address, causing the program to begin executing instructions at an arbitrary location.

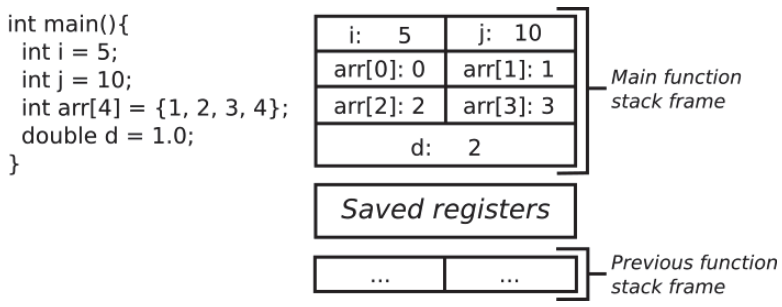

Figure 2: C source code (left) with the run-time memory diagram (right)

This behavior can not be changed. Every C executable will operate with a stack and stack frames at run-time, which means that when an attacker can break the bounds of the stack frame they are in, they know exactly where they are in process memory. By using a compiled executable, the user loses valuable control over the execution of the program in exchange for speed. Through use of an interpreter, the potential exists to examine and restrict run-time behaviors, such as when the bounds of a stack frame is broken.

3 Transpiler Creation

3.1 Abstract Syntax Trees (ASTs)

In order to translate C code into Python, it is critical to understand the meaning of the program to be translated. It is important to be able to look at what a program actually does, not the specific code blocks used, when translating between languages. Similar to how different written languages structure semantically identical sentences differently, programming languages have constructs and quirks that, while functionally similar, are formatted completely differently. As such, we use Abstract Syntax Trees (ASTs) as our main vehicle for transpilation. The AST of a program is the semantic meaning of a program, represented in a tree-like structure. An AST is made up of nodes to represent small units of meaning, with each node connected to both the previous and next one in the program. It appears to be the perfect vehicle to use to translate source code between languages, as the ASTs for identical programs written in different languages have, in theory, no difference. In practice, however, they are often very different. Even though the AST for a program should be language independent, most ASTs are still tied not only to the language it comes from, but to the specific AST generation program used to make it. As such, most of the work on the transpiler has been translating between the C AST and the Python AST.

3.2 Tools Used

We access the C AST through the LLVM Project's libclang library. The LLVM Project is a large library of tools for C-esque languages, a low-level language for compilers, as well as libraries for interacting with C programs in other langauges. The LLVM Project has also created bindings for libclang in Python, allowing Pythonic access to much of Clang's functionality. This meant that we were able to write our compiler in Python, using the libclang library as the main driver to access the C AST [5].

The ASTs that libclang produced for input C programs were different from the format that Python's native AST library accepted [3], so one of the largest challenges we faced throughout the project was bridging the gap between the two. In the C AST generated by libclang, each block of code is represented as the child of the node that contains it. That means that every line contained within an if statement would be a child of the if statement node itself. Conversely, in a correctly formatted Python AST, all AST nodes are contained within 'body' arrays of their parents. In some cases, such as variable assignment, however, the Python AST stores the nodes of the variables to be assigned in a 'targets' array. This complexity is enblematic of the differences between the two languages, as well as the difficulty of creating a comprehensive transpiler.

3.3 Libclang limitations

Though libclang was an excellent tool for dissecting the function of a C program, it is not a comprehensive solution. For example, for arithmetic, there is no built-in way to extract the operator (addition, subtraction, etc) from the AST node that contains the operation. For the longest time, we had no way short of examining the tokens of the source code itself to get that information, and the transpiler was limited in what it could translate as a result. Interestingly, there have also been others who have run into the same issue, particularly LLVM contributer Arthur Peters. He and a few others created a patch for this issue that, while not merged into the main branch of the LLVM Project, is quite functional [10]. We build our own libclang shared object file for the transpiler, and expose new functionality to extract the operator for binary arithmetic operations.

4 Transpiler Capabilities

The bulk of the work on this project was building out the transpiler to understand a wide variety of programming principles from integer addition to heap allocation. The development cycle was to expand the scale of program until the compiler crashes due to an unknown AST node, expand its capabilities until it can translate the program in question, then repeat. In the end, we ended up with a project that spanned just over 1800 lines of code, over many source files. The full source code is available at `https://github.com/Jmetcalf26/trident_project`. The readme documents explains how to run the transpiler, and the bulk of the work is stored in the `parser.py` file. The transpiler is able to translate the following:

Literals	Expressions	Statements	Control Flow	Functions	Classes
Integers	Array subscripts	Var declaration	If	Definitions	Definitions
Characters	List initialization	Var assignment	For	Calls	Member references
Floats	Unary operators	Return	While		Member Assignment
Strings	Binary operators	Type references	Break		
	Explicit casting	Enums	Continue		
	Implicit casting		Switch		
	sizeof Operator				

Figure 3: C to Python Transpiler Capabilities

4.1 Pointers

In C, a pointer is a value that represents an address in memory. When a program is being run, variables are stored in the memory of the running program in a stack frame. To the programmer writing the code, however, this convention is abstracted away. Variable names provide the programmer with easy access to where their variables are going to be stored. With pointers, however, the memory location of variables are exposed. You are able to perform operations on pointers to retrieve variable offsets, index into arrays, and more.

This functionality is fundamental to C, but has no equivalent in Python. In order to duplicate that functionality, we created the Pointer class. This custom class stores the data that the C pointer references, the offset from the initial address the pointer originally referenced, and the size of the data type the pointer references. We do not save the actual type of the variable, as we are able to rely on the Python run-time

type system to resolve typing. The latter two fields enable both pointer arithmetic and aliasing, two core C language mechanics. Different from C, however, is the restrictions on pointer arithmetic. In C, it is possible to access the values of other variables from a memory address, simply by adding to the pointer beyond the size bounds of the original value. Our Python Pointers and simulated memory space restricts this possibility, meaning that variables and addresses in memory are isolated from one another.

4.2 Variables

All variables have a location in memory in C, and that location must be able to be exposed. You should be able to modify the value contained at a certain memory address, and that should be reflected in the variable stored at that location. As such, the translation of a variable declaration such as `int x = 5;` into `x = 5` does not work, as the Python has no sense of it's location in memory. Such a translation would work for a learning-level translation, but not for a fully functional C environment. The previous translation would break when the user took the address like `int *p = &x;`, and then attempted to change the value the pointer references: `*p = 20;`. According to C semantics, the original variable x will be updated to 20. In Python, however, that information would be lost; there is no way for one primitive type to effect another (in this case two integers). In order to fully encapsulate the way C variables behave, we treat all variable declarations as dereferenced pointers. A dereferenced pointer could be addressed to receive its memory location Pointer object, and the original variable can be operated upon to change the pointer within. To that end, we created a Deref class to wrap the Pointer class, and the variable function as an alias to a call to Deref and Pointer. The transpiler would translate `int x = 5;` into `x = variable(5, 4)`, which is equivalent to `x = Deref(Pointer(5, 0, 4), 0)`.

4.3 Strings

Another interesting design choice came up when translating strings, or character sequences. In C, a string is nothing more than an array of characters, which are in turn represented by integer ASCII values, while in Python a string is a more complicated object with special properties. Unfortunately, given the operations that can be done with pointers and C arrays, we decided not to use the special properties of Python strings, and instead opted to represent C strings as pointers to character arrays in the output Python code. We created a special class called Trigger to allow the code to still bear a resemblance to what a string would look like, as the Trigger class itself has the XOR operator overloaded to convert the right-hand-side string argument of the operation into a character sequence. Such a choice, while functionally not necessary, was similar to the choice to create the variable function in the previous section, and was for the purpose of retaining code readability with the output Python code.

```c
int main(){
  int a = 1;
  int b = a;
  int * c = &a;
  return 0;
}
```

```python
from helper_classes import *
trig = Trigger()
def main():
    a = variable(1, size=4)
    b = variable(a, size=4)
    c = variable(a.pointer, size=8)
    return 0

if __name__ == '__main__':
    main()
```

Figure 4: Example program translated from C (left) to Python (right)

This combination of classes allows us to effectively simulate a stack frame in C, while eliminating much of the insecurity that comes along with C stack frames. At the beginning of every translated file, there is a call to import the helper class file, which contains the Pointer, Deref, and Trigger classes. Here is the first introduction to what has been a very powerful concept in our development of the compiler: writing our

own Python code manually for the automatically generated Python to reference. Shown broadly in Figure 5, importing a custom file of our own allows the output Python program to reference functions beyond its own scope, a powerful mechanic to enhance our functionality. We have used this technique to implement pointers, the math behind pointer aliasing, as well as the focus of the following section, library calls.

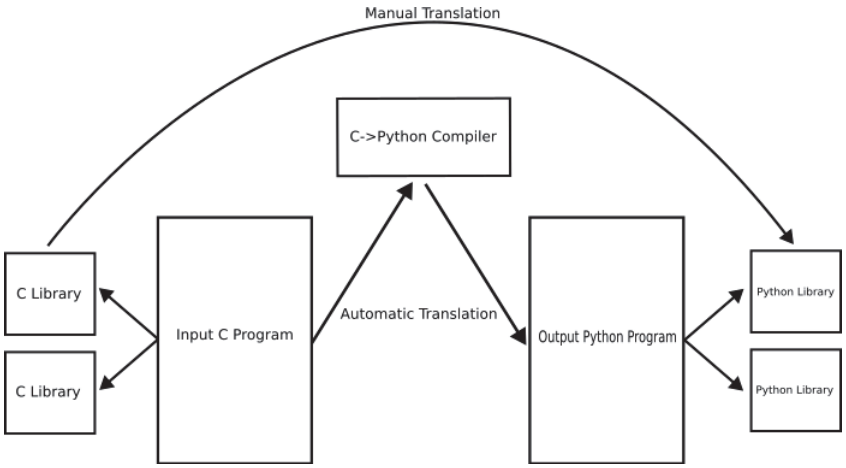

Figure 5: The scheme used to translate C programs into Python

4.4 Libraries

One of the largest challenges in this project has been library calls. C programs are built on library calls. A lot of the core functions of the C language, such as I/O and networking, are based on calls to their respective libraries. There were multiple approaches that we could have taken when attempting to use library calls. For one, the ctypes Python library[4] allows Python programs to call C library functions. This would have been an excellent solution, however, in many cases the library function itself is vulnerable. The classic example of one such function is `gets`, a method that reads input from the user into a buffer until it encounters a new-line character. The base functionality of `gets` is vulnerable, as the user can simply input more characters than the buffer is long and overwrite memory. There is no way to secure `gets` and maintain it's functionality, and as such many C compilers warn the programmer when it is used. Thus, for this project we implement our own version of functions such as `gets`, which takes advantage of Pythons memory system to dynamically resize the buffer as more characters are read in. We manually translate library functions into Python equivalents, securing vulnerable ones in the process. Additionally, however, the decision to manually translate library calls was also made out of necessity, as we did not have the resources to build out the transpiler to translate libraries automatically.

Creation of custom library files also gives the ability to adapt C functions to behave in a more Pythonic way while still retaining identical functionality. For example, in C, string comparison requires the string library's `strcmp` function. `strcmp` compares strings lexicographically, meaning that they are compared character by character for equality [6]. Comparing strings using the equals operator does not work. Conversely, in order to compare strings lexicographically in Python, the equals operator is used. There are small differences in behavior between the two forms of comparison, but for the most part manually considering the optimal way to implement C libraries in Python allowed the simplification of the library definition of `strcmp` and many more similar functions, reducing the translation workload and AST size.

5 Transpiler Evaluation

5.1 Evaluation Metrics

The first metric that the transpiler can be evaluated on is it's robustness, or ability to handle a variety of programs and code blocks. There are a finite number of constructs within the C language that can appear,

but nearly infinite ways to arrange them to produce new behavior for the transpiler. By this metric, the transpiler is at the level of a 2nd-year Computer Science undergrad. It can handle many self-contained, simple programs that are small in scale, but it is not able to translate production-grade code like we had anticipated. The goal was to translate programs from the Linux core utilities repository such as `cat` and `ls`, a goal which would require a significantly more robust transpiler. Some of the biggest obstacles we ran into were large dependency trees, convoluted variable declarations and function pointers.

The second metric to test the effectiveness of the transpiler is the degree of vulnerabilities it can reduce. To evaluate this point, we used the NIST Juliet 1.3 Test Suite [15]. The suite is based around the concept of Common Weakness Enumerations (CWEs) in C. CWEs are a "community-developed list of common software and hardware weakness types that have security ramifications" [2], and are compiled by the MITRE Corporation. They run the full spectrum in terms of severity, as well as applicability to this project. The CWEs that the transpiler has been proven to be effective on in some capacity are as follows:

- CWE 121 - Stack Based Buffer Overflow
- CWE 242 - Use of Inherently Dangerous Function
- CWE 476 - NULL Pointer Dereference
- CWE 190 - Integer Overflow

The transpiler is judged effective on a CWE class if it can translate a C file containing an example of the CWE, and eliminate the vulnerability present in the original C source. We were able to meet this threshold for 4 CWEs, with over 25 Juliet tests translated. Despite the fact that the transpiler cannot translate every example of each CWE type listed, we are able to confidently say that there is evidence that the transpiler is effective at vulnerability reduction. Should the scope of the project continue to increase, and the transpiler ability expanded, it would only be able to eliminate a broader variety of CWEs. Chief among the CWEs that the transpiler could handle with minimal retooling are ones related to the heap, such as CWE 122 - Heap based buffer overflow, CWE 415 - Double free, and CWE 416 - Use after free.

The final consideration for the evaluation of the transpiler is the output programs speed. The biggest trade-off when translating a program from C to Python is the reduction in speed. C has long been used in systems programming for its speed, while Python has made a home through its powerful scripting capabilities and extensive libraries. Considerations should be taken to ensure that the output Python code takes steps to stay as fast as possible. However, for the intended use case of the transpiler, speed is less important than it may seem. We are not attempting to translate bulky, industrial programs, rather lightweight and small C utilities. The speed reduction we would experience would be imperceptible, the difference between microseconds and milliseconds. As such, we do not change the output Python code in any way to affect the speed. The transpiler can not translate large enough programs where it would really be a factor, and for smaller programs the difference for a single run is negligible.

5.2 Example Vulnerability Elimination - CWE 121

In the following example, we demonstrate a vulnerability elimination in a program made by a student in the IC210 Introduction to Computer Science course. After the translation, we run both versions of the program against some strange input, in this case hundreds of 'A' characters, and attempt to cause a memory error. In the C program, the program crashes, and there is evidence that we have corrupted one of the variable values. In the Python program, however, the program simply crashes. The latter case is far preferable, as a crashing program can not cede execution control to a malicious attacker. In the case of the vulnerable C program, it is possible for a malicious actor to gain full control of the environment the program is running in using binary exploitation techniques.

```
m234158@lnx1080267govt:~/trident_project$ ./convert
HelloWorld!
H3lloWorld!
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
DONE
DONE
1094795585 conversions for "H3lloWorld!"
Segmentation fault
m234158@lnx1080267govt:~/trident_project$ python -c "print(bytearray.fromhex(hex(1094795585)[2:]))"
AAAA
m234158@lnx1080267govt:~/trident_project$
```

Figure 6: Vulnerable C program demonstrates a Buffer Overflow

```
m234158@lnx1080267govt:~/trident_project$ python3 convert.py
HelloWorld!
H3lloWorld!
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Traceback (most recent call last):
  File "/home/mids/m234158/trident_project/convert.py", line 54, in <module>
    main()
  File "/home/mids/m234158/trident_project/convert.py", line 22, in main
    scanf([Pointer('%s', 0, 1)], [curr_word[0]])
  File "/home/mids/m234158/trident_project/pheaders/stdio.py", line 59, in scanf
    return fscanf(stdin, fmt, *args)
  File "/home/mids/m234158/trident_project/pheaders/stdio.py", line 278, in fscanf
    dest[i] = rhs[i]
  File "/home/mids/m234158/trident_project/helper_classes.py", line 31, in __setitem__
    self.array[i] = a
IndexError: list assignment index out of range
m234158@lnx1080267govt:~/trident_project$
```

Figure 7: Secured Python program is not vulnerable

6 Conclusion

Our C to Python transpiler explores the possibility that interpreted languages offer increased security implicitly. Throughout the course of this project, we did not build any security measures into the transpiler, and yet it shows that translating programs into Python increases their security.7 This, coupled with the additional oversight an interpreter gives over the execution of a program, builds a strong case for security through interpretation. To fully realize this security principle, more work must be done on the topic, but the transpiler we have created demonstrates that the concept has merit.

References

[1] S. Gupta. "Buffer Overflow Attack". In: *ISOR Journal of Computer Engineering* (2012).

[2] The MITRE Corporation. *Common Weakness Enumeration*. URL: https://cwe.mitre.org/about/index.html.

[3] Python Software Foundation. *ast - Abstract Syntax Tree*. URL: https://docs.python.org/3/library/ast.html.

[4] Python Software Foundation. *ctypes - A foreign function library for Python*. URL: https://docs.python.org/3/library/ctypes.html.

[5] Tao He. *Clang Indexing Library Bindings*. URL: https://libclang.readthedocs.io/en/latest/.

[6] Michael Kerrisk. *strcmp(3) — Linux manual page*. URL: https://man7.org/linux/man-pages/man3/strcmp.3.html.

[7] Tautschnig M Kroening D. "CBMC – C Bounded Model Checker". In: *TACAS 2014. Lecture Notes in Computer Science, vol 8413.* (). DOI: https://doi.org/10.1007/978-3-642-54862-8_26.

[8] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *SIGPLAN Not. 42, 6 (June 2007)* (), pp. 89–100. DOI: https://doi.org/10.1145/1273442.1250746.

[9] Y. Nong and H. Cai. "A Preliminary Study on Open-Source Memory Vulnerability Detectors". In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (), pp. 557–561. DOI: 10.1109/SANER48275.2020.9054851..

[10] Arthur Peters. *Retrieve BinaryOperator::getOpcode and BinaryOperator::getOpcodeStr via libclang and its python interface*. URL: https://reviews.llvm.org/D10833?id=39176.

[11] Aqsa Rasool. *Which is the Most Vulnerable Programming Language?* URL: https://www.digitalinformationworld.com/2019/03/searching-for-the-most-secure-programming-language.html.

[12] D. S. Slejebotn S. Behnel R. Bradshaw C. Citro L. Dalcin and K. Smith. "Cython: The Best of Both Worlds". In: *Computing in Science and Engineering, vol. 13, no. 2* (), pp. 31–39. DOI: 10.1109/MCSE.2010.118.

[13] Daniel Lester Saldanha. *C to Python Translator*. URL: https://github.com/DAN-329/C_to_Python_translator.

[14] J Seward and N Nethercote. "Using Valgrind to Detect Undefined Value Errors with Bit-Precision". In: *USENIX Annual Technical Conference, General Track* ().

[15] National Institute of Standards and Technology. *NIST Juliet Test Suite*. URL: https://samate.nist.gov/SARD/test-suites/112.