Cornucopia: Partially Automated Reverse Engineering of String Manipulation Programs

Manav Malik
Mentor High School
Mentor, OH, USA
manavmalik36@gmail.com

Abstract

As the field of cybersecurity grows ever larger, so does its influence in education. Specifically, recent years have seen a major increase in popularity of capture-the-flag competitions, of which reversing engineering (frequently of string manipulation programs) makes up a major part. These challenges involve determining the necessary inputs for a program to produce a desired output. Reverse engineering Python source code is a non-trivial task so performing it manually frequently becomes infeasible.

This paper presents Cornucopia, a partially automated reverse engineering tool that operates over Python source code to determine necessary inputs to arrive at outputs. Cornucopia takes advantage of symbolic representation to partially automate the process of reverse engineering programs that are affine or include conditional branching. A basic evaluation of Cornucopia finds that it significantly reduces human interaction in reversing, and is composable.

1 Introduction

In recent years, capture-the-flag cybersecurity competitions (CTFs) have been rising in popularity [McDaniel et al. 2016]. Reverse engineering challenges comprise a large aspect of these competitions [Burns et al. 2017]. *Reverse engineering*, or *reversing*, is the process of determining the required inputs for a program to produce a given output [Müller et al. 2000]. For example, given the program $f(x) = x + 2$ and the output 5, reversing this program involves determining the value of x for which the program outputs 5 (in this case, $x = 3$).

However, many such challenges involve more complicated reversing, such as *string manipulation programs* which take in text as input and perform sequences of operations to produce mangled outputs. Figure 1 shows a basic example of a string manipulation program that adds 1 to each letter of the input and swaps adjacent pairs of letters. Reversing this program involves noticing that by subtracting 1 from each letter of the expected output "BQTTPXES" and swapping adjacent pairs, we arrive at the necessary input "PASSWORD." Because this program involves dependences across letter boxes, reversing it is not as simple as solving eight independent algebraic equations.

While the above example is relatively straightforward, it is not difficult to imagine a more complex sequence of operations whose inverses are less obvious. In situations like these, some degree of automated reasoning can be very helpful.

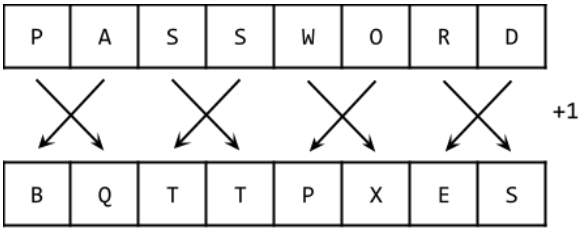


Figure 1. String manipulation program that adds 1 to each letter and swaps adjacent pairs

1.1 Challenges in Automated Reasoning

Automatically reversing programs is, in general, undecidable[1]. By restricting the class of programs that can be reversed, the problem goes from being an impossible one to a possible but difficult one.

There are three major obstacles to manual reasoning that make reversing string manipulation programs particularly challenging: complex algebraic transformations, arbitrary dependences across letters positions, and program control flow.

Algebraic Transformations. In the example in Figure 1 adding 1 to every letter is an instance of an algebraic transformation. While the inverse for this transformation is readily apparent, that may not be true for more complicated expressions. For instance, a function that multiplies each letter by its position in the string and wraps around when necessary is not as straightforward.

Positional Dependences. Again, referring to the example in Figure 1, the swapping induces a dependence between adjacent letter positions. As these dependences become more convoluted, keeping track of them becomes much more difficult.

Control Flow. Programs typically include conditionals (such as if-else statements), which create branching control flow. Each path through the program represents a different set of operations applied to the input, so reversing these

[1] An undecidable problem is one for which it can be proven that no algorithm exists which solves it. A classic example of this is the Halting Problem.

```
def f(x):
    if x > 5:
        y = x * 2
    else:
        y = x + 2
    return y
```

Figure 2. Python program showing conditional control flow

requires knowing a priori which path was taken. For example, consider the Python program in Figure 2. If the expected output is 8, analyzing the two paths through the program yields 4 and 6 as potential inputs. However, looking at the condition shows that 6 is the only correct answer. As the number of paths grows exponentially with the number of conditions, analyzing all of these quickly becomes infeasible to do by hand.

1.2 Cornucopia

Cornucopia is a partially automated reversing system that uses *symbolic program representation* to determine execution paths and, for each execution path, determine the possible inputs that would result in the desired output. This entails creating a symbolic variable (or *symbol*) for each character of the input and passing it through the given function. Each symbol keeps track of the operations performed on it, and Cornucopia tries to solve for the symbol by setting the final expression equal to the expected output.

Cornucopia can also handle the challenges in automated reasoning given in Section 1.1.

While a program can have arbitrarily complex algebraic transformations, Cornucopia's use of symbolic execution makes it simple to keep track of these. Therefore, as long as the function is affine[2], Cornucopia can reverse it. Cornucopia depends on linear algebra to solve for the values of symbols, so only linear systems of equations are allowed. Thus, it cannot handle functions that are not affine.

Additionally, symbolic execution is entirely position independent. Because Cornucopia depends only on the values of symbols and not their positions, this means that handling positional dependences is no more difficult than handling algebraic transformations.

Finally, Cornucopia uses nondeterminism[3] to deal with conditionals. When a symbol encounters a branch in a program, it gets lifted into a nondeterministic state that holds all of its possible values from both branches. For example, in Figure 2, after the conditional is executed, the symbol y is tracking the fact that its value is $x \cdot 2$ if $x > 5$ or $x + 2$ otherwise. Setting this nondeterministic symbol equal to 8, Cornucopia can solve for x by attempting to solve both equations ($x \cdot 2 = 8$ and $x + 2 = 8$) and only keeping the values that are consistent with the associated conditions.

This paper presents the following contributions:

1. Cornucopia, a partially automated reverse engineering tool.
2. Symbolic variables that represent different types of programs (LinearSymbols and ConditionPacks).
3. Evaluating the effectiveness of Cornucopia involves examining two main factors: how it minimizes user interaction and how it can be composable.

1.3 Related Work

There is a class of reverse-engineering frameworks for decompilation and code analysis [Binary Ninja 2022; Ferguson and Kaminsky 2008; Ghidra 2021; radare2 2022]. The most closely relevant of these tools here is Ghidra. Users input compiled binaries and Ghidra automatically decompiles them into source code. Ghidra also allows users to explore the decompiled source code by, for instance, tracing through function execution and finding particular symbols in the binary. While Ghidra produces source code from a compiled binary, Cornucopia focuses on reversing the source code to determine inputs to a function given an output.

GhiHorn, a Ghidra plugin, performs path analysis on Ghidra's decompiled output [Gennari 2021]. It uses an SMT solver[4] to determine a possible input that triggers the execution of a particular code path. In contrast to GhiHorn, which is designed to reverse engineer specific control flow paths, Cornucopia's goal is to reverse data flow through a program. Cornucopia's specialized focus allows it to incorporate domain knowledge to apply techniques only useful in reversing data flow.

2 Methods

2.1 Symbolic Variables

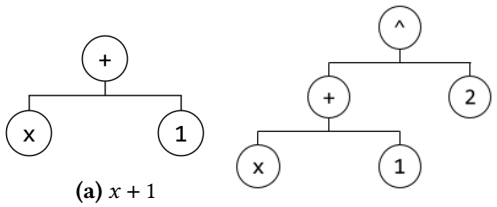
A symbolic variable is a variable that does not have a concrete value. Rather, it represents sequences of computations that are being performed on it. For example, when a function is given symbolic input, it returns a symbolic representation of the operations without actually performing them. Every operation performed accumulates on a *computation graph* and, in the end, this graph represents the entire expression [Torlak and Bodik 2013]. For example, the computation graph for the function $x + 1$ is shown in Figure 3a.

Furthermore, symbolic variables are composable, meaning the computation graph from one function can be passed through a second function and the resulting computation graph would represent the composition of both functions.

[2] An affine function consists of a linear combination of its variables followed by an addition of an optional constant. In other words, an affine function only uses addition and scaling (noting that subtraction is simply addition of a negative).

[3] A nondeterministic variable can have any one of many possible values.

[4] Given a set of variables and constraints, an SMT (Satisfiability Modulo Theories) solver either determines that the constraints are unsatisfiable or instantiates the variables with values that satisfy the constraints.



(a) $x + 1$ **(b)** $(x + 1)^2$

Figure 3. Examples of computation graphs

For example, the resulting computation graph of when the function from Figure 3a is passed through the function x^2 is shown in Figure 3b.

Note that there exists a symbolic variable with no computations attached to it (i.e., a single node in a computation graph, just "x").

Generic symbolic variables (and computation graphs) can represent any function. Because of how general these can be, any automated system cannot determine what these operations are and how to reverse them. By avoiding generality and specializing to a few cases, Cornucopia makes it much easier by using symbolic variables that can only handle certain operations.

2.2 Linear Operations

2.2.1 LinearSymbols. The symbolic variable specialized to linear operations is the LinearSymbol. The only operations a LinearSymbol can track are those that consist of scaling it by a constant integer, or computing the sum of other LinearSymbols. A LinearSymbol is represented as a lookup table that maps a set of names to their coefficients. For example, the lookup table for a LinearSymbol representing $2x - 3y$ looks like this:

$$\{x \to 2, y \to -3, * \to 0\}$$

Here, $* \to 0$ means that any name not explicitly present in the LinearSymbol gets mapped by default to 0.

There are two operations that can be performed a LinearSymbol: summing and scaling. For example, let

$$A = \{x_1 \to n_1, \ldots, x_k \to n_k\}$$
$$B = \{x_1 \to m_1, \ldots, x_k \to m_k\}$$

and let z be an arbitrary integer. Then

$$A + B = \{x_1 \to n_1 + m_1, \ldots, x_k \to n_k + m_k\}$$

and

$$z \cdot A = \{x_1 \to z \cdot n_1, \ldots, x_k \to z \cdot n_k\}$$

(Remember that the default coefficient is 0, so any names present in one symbol but not the other simply get copied over to the sum.)

Additionally, when creating a new LinearSymbol, the lookup table looks like this:

$$\{x \to 1, * \to 0\}$$

Note that constants[5] themselves can be modeled as LinearSymbols by having coefficients associated to the special name 1. For example, the lookup table for the expression $2x + 5$ looks like this:

$$\{x \to 2, 1 \to 5, * \to 0\}$$

2.2.2 Using LinearSymbols. To reverse a linear string manipulation function, the user creates a list of LinearSymbols with one for each character of the string. This list is then passed into the function and the modified LinearSymbols are reversed as described in Section 2.2.3.

For example, consider the Python program in Figure 6. If the desired output is three characters long, the inputted list of LinearSymbols will look like this:

$$[\{x_1 \to 1, * \to 0\}, \{x_2 \to 1, * \to 0\}, \{x_3 \to 1, * \to 0\}]$$

Once this list is passed through case_study_1, the resulting LinearSymbols will be:

$$[\{x_1 \to 2, x_2 \to 1, 1 \to 5, * \to 0\},$$
$$\{x_2 \to 2, x_3 \to 1, 1 \to 5, * \to 0\},$$
$$\{x_3 \to 2, x_1 \to 2, x_2 \to 1, 1 \to 10, * \to 0\}]$$

With these modified LinearSymbols, Cornucopia can use the expected outputs to determine what the required values of x_1, x_2, and x_3 are.

2.2.3 Reversing LinearSymbols. To reverse a set of LinearSymbols, Cornucopia builds and solves a system of linear equations. Referring to the above example and given that the desired output is $[341, 314, 548]$, Cornucopia creates this system of equations:

$$x_1 \cdot 2 + x_2 + 5 = 341$$
$$x_2 \cdot 2 + x_3 + 5 = 314$$
$$x_3 \cdot 2 + x_1 \cdot 2 + x_2 + 10 = 548$$

Using linear algebra, Cornucopia can solve this (or more complicated systems) to determine that the desired values are $x_1 = 116$, $x_2 = 104$, and $x_3 = 101$ [Dumas and Pernet 2012].

2.3 Conditional Branching

2.3.1 ConditionPacks. Conditional branching is more complicated because it involves keeping track of not just one set of computations but *all possible sets* of computations (corresponding to paths through the program). To keep track of all of these possibilities simultaneously, ConditionPack symbolic variables are used. A ConditionPack consists of:

1. An expected output value
2. A set of possible paths that reach that output

Each path consists of a boolean expression representing the conditions that must be met for that path to be taken (note that this is an accumulation of all the conditionals along that

[5]This is a constant being *added* to a LinearSymbol, not a *scaling* constant.

```
if a:
    if b:
        # 1
    else:
        # 2
else:
    if c:
        # 3
    else:
        # 4
```

(a) Nested branches

```
if a and b:
    # 1
elif a and not b:
    # 2
elif not a and c:
    # 3
elif not a and not c:
    # 4
```

(b) Inline branches

Figure 4. Removing nested branches

```
if a:
    # 1
elif b:
    # 2
else;
    # 3
```

Figure 5. Multiple branches of a conditional

path), and the computation[6] performed on the inputs in that path. For the sake of simplicity, when the computation itself is a ConditionPack, it is "inlined" (i.e., its paths are added to the parent ConditionPack).

Consider again the example program in Figure 2 (again assuming the desired output is 8). The ConditionPack for this program would look like:

$$\{x > 5 : x \cdot 2, x \leq 5 : x + 2\} \rightarrow 8$$

2.3.2 Using ConditionPacks. When creating a ConditionPack for a branch in a program, the user must first remove any nested branching by inlining them into top-level branches. An example of this is shown in Figure 4.

Then, they must determine the condition for each branch, noting that this is the conjunction of its own condition and the negations of the previous conditions. For example, in Figure 5, the condition[7] associated with branch 1 is simply a, for branch 2 it is $\neg a \wedge b$, and for branch 3 it is $\neg a \wedge \neg b \wedge c$.

After determining the computation associated with each branch, the user combines these into a ConditionPack. Multiple sequential conditionals can be compiled into individual ConditionPacks and solved individually by passing contexts between them (Section 2.3.3) or combined into one large ConditionPack.

2.3.3 Contexts. A *context* maps each variable to a value. Cornucopia uses these to track intermediate solutions to a sequence of branches. When given two consecutive branches, Cornucopia can solve them independently (as described in Section 2.3.4), first producing a set of contexts encoding the possible solutions to the first branch (the context passed into the initial branch being empty). Each context is then used in solving the second branch, to ensure that every possible solution Cornucopia comes up with is compatible with the corresponding solution from the first branch. In particular, this lets Cornucopia "prune" solutions to the first branch that result in contradictions in the second branch.

2.3.4 Solving ConditionPacks. To solve a ConditionPack, Cornucopia iterates over all possible paths as shown in Algorithm 1. For each path, Cornucopia creates the following constraints:

1. One for each variable in the context, asserting that the variable must be assigned its value in the context
2. One to assert that the conditions necessary to execute the path are met
3. One to assert that the result of the computation along that path matches the output

To solve this set of constraints, Cornucopia uses the Z3 Theorem Prover [de Moura and Bjørner 2008]. This can have one of two outcomes: *satisfiable* and *unsatisfiable*.

If Z3 determines that the constraints are satisfiable, it returns a *model* that associates a concrete value to each variable used in the constraints. This model represents a possible solution to the branch along the current path which is then packaged into a new context and added to the set of solutions.

If the constraints are deemed unsatisfiable, Z3 returns a proof that there is *no possible solution* to the given set of constraints. This means that in the given context, there is no way for the current path to produce the expected output. (For example, the path consists of doubling an integer but the expected output is odd, or it requires a different value for a variable than the one assigned to it in the context.)

After Cornucopia iterates over all the paths, it returns the resulting set of contexts. These are either possible final solutions or can be used in further parts of the program.

3 Results

There are two main criteria on which to evaluate Cornucopia:

1. **Does it minimize user interaction?** That is, does using Cornucopia to reverse a program make it less difficult?
2. **Is it composable?** In other words, can smaller, individually reversible parts be combined into larger, more complex programs?

To assess these, two case studies were used.

3.1 Minimizing User Interaction

First, refer to Case Study 1, shown in Figure 6. Given the

[6] This computation can either be an arbitrary computation graph (as described in Section 2.1) or itself a specialized symbolic variable such a LinearSymbol (as described in Section 2.2.1).

[7] $\neg$ is logical NOT, $\wedge$ is logical AND

Algorithm 1: Solving ConditionPacks

```
Algorithm SolvePack(paths, output, context)
    answers = [];
    foreach condition, computation ∈ paths do
        constraints ← ∅;
        AddConstraint(constraints, context);
        AddConstraint(constraints, condition);
        AddConstraint(constraints, computation == output);

        if solution ← Solve(constraints) then
            answers.add(context ∪ solution);
    return answers
```

```python
def case_study_1(x):
    for i in range(len(x)):
        # double it...
        x[i] = x[i] * 2
        # ...add the value of the next element...
        x[i] = x[i] + x[(i + 1) % len(x)]
        # ...and finally, add five.
        x[i] = x[i] + 5
    return x
```

Figure 6. Python program showing linear transformations

```python
import solver
# create LinearSymbols x1, x2, x3
symbols = [LinearSymbol(f'x{i+1}')
           for i in range(3)]
# pass the symbols through the function
x = case_study_1(symbols)
# solve the system of equations
n = solver.System(x)
n.solve([341, 314, 548])
```

Figure 7. Cornucopia reversing case_study_1

desired output, [341, 314, 548], reversing this program manually would entail writing out the inverse of the function. This becomes difficult because the user must recognize that, because the values of the characters influence each other, it is necessary for the inverse to iterate backwards through the list. Additionally, determining the inverses of each computation — especially the ones involving multiple characters at a time — can be non-trivial.

Instead, because the only computations in this function are linear and positional, Cornucopia can reverse it using only LinearSymbols. The relatively simple code required to do this (shown in Figure 7) entails creating LinearSymbols for each character, passing them through the function, and building and solving the resulting system of equations.

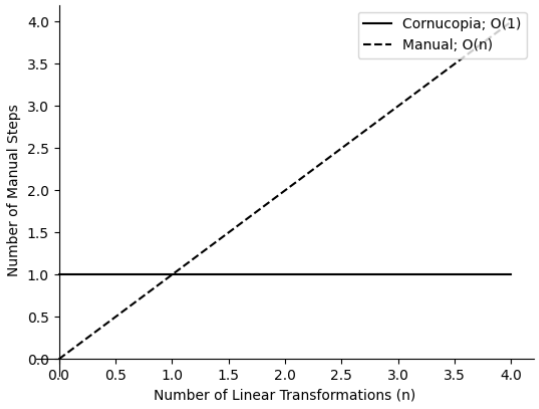


Figure 8. Problem complexities for reversing linear programs

Furthermore, while programs including combinations of linear transformations can become arbitrarily complex, the code required to reverse the program with Cornucopia remains more or less the same, provided the function remains entirely linear. Essentially, this means that the problem complexity of manually reversing a linear program is $O(n)$ while reversing it with Cornucopia is $O(1)$ (where n is the number of individual linear transformations). In other words, reversing a linear program manually means that the number of manual steps required increases linearly as the number of linear transformations does, while the number of manual steps required in reversing it with Cornucopia remains constant. This difference is shown in Figure 8.

Additionally, looking at conditional statements, the problem complexity of manually reversing these is $O(2^n)$ (as each branch has two possibilities that must be addressed) while reversing it with Cornucopia is $O(n)$ (where n is the number of conditionals). This is shown in Figure 9.

3.2 Composability

Next, consider Case Study 2, shown in Figure 10a. This program shows a combination of linear operations and conditional branching. Because Cornucopia is composable, these can be split into two distinct functions (as in Figure 10b) and reversed separately given the expected output of [334, 301, 322].

function2 is applied second so it must be reversed first. To do this, it can be written as a ConditionPack. First, the change function must be rewritten inline like in Figure 10c. The resulting ConditionPack for the first character is shown in Figure 11. The only possible value that satisfies the given constraints for all paths is $x_1 = 368$. This means that the first character of the output of function1 must be 368.

Now, reversing function1 is relatively simple because it is entirely linear. Figure 12 shows the LinearSymbol created

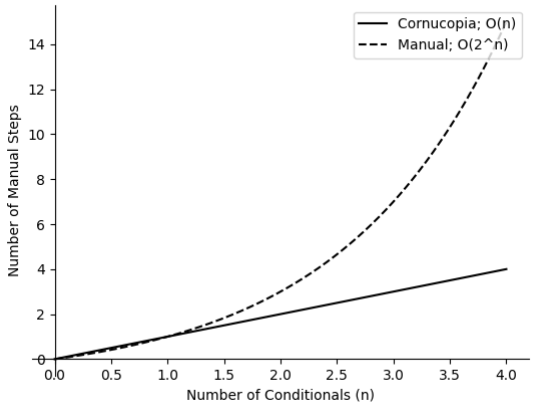


Figure 9. Problem complexities for reversing conditional programs

for the first character of the input. The only solution to this is $x_1 = 195$. This same principle can be applied to the other characters of the string as well, showing how Cornucopia is composable.

4 Discussion

Cornucopia is a reverse engineering tool that operates over Python source code to determine the necessary inputs to a function to produce a desired output. It does this primarily using specialized symbolic variables. Using these variables grants Cornucopia all the benefits of symbolic execution without the pitfalls of arbitrary generalization.

Regardless of its length or complexity, Cornucopia can easily reverse a program consisting of only linear operations with very little user interaction. Though the complexity of the code required to invoke Cornucopia on an affine program remains constant despite the complexity of the program, the same does not hold true for manually writing out the inverse of the program. As a linear program becomes more and more complex, so does its inverse which means that Cornucopia allows the user to skip this non-trivial task and easily reverse the program.

Furthermore, Cornucopia can reverse programs that contain branching control flow. Instead of having to manually analyze each possible path in the program and determine all the possible solutions, the user simply needs to translate the branches into a ConditionalPack. Once this is created, Cornucopia handles the analysis and determines possible solutions to satisfy the branches.

From the examples in Section 3, it becomes evident that Cornucopia is useful in handling linear operations and conditional branching. Its ability to significantly decrease user interaction makes it an effective tool. Additionally, its composability allows it to handle arbitrarily complex combinations

```python
def change(x):
    if x < 82:
        x = x * 4 - 2
    elif x > 143:
        if x > 195:
            x -= 34
        else:
            x = x * 2 - 1
    else:
        x += 24
    return x

def case_study_2(inp):
    out = map(lambda x: change(2 * x - 4),
              inp)
    return list(out)
```

(a) Original program

```python
def function1(inp):
    out = map(lambda x: 2 * x - 4, inp)
    return out

def function2(inp):
    out = map(lambda x: change(x), out)
    return out

def case_study_2(inp):
    out = function1(inp)
    out = function2(out)
    return list(out)
```

(b) Divided program

```python
def change_inlined(x):
    if x < 82:
        x = x * 4 - 2
    elif x > 143 and x > 195:
        x -= 34
    elif x > 143 and not x > 195:
        x = x * 2 - 1
    else:
        x += 24
    return x
```

(c) Change function inline

Figure 10. Python program showing conditional branching and linear transformations

of linear operations and branches in programs. Cornucopia's use of symbolic subunits (LinearSymbols and ConditionalPacks) allows it to reverse some specific cases of programs that would not have been otherwise reversible because of their generalness.

```
x1 = ConditionPack(
    334, ['x'],
    [(lambda x: x * 4 - 2, lambda x: [x < 82]),
     (lambda x: x - 34,
      lambda x: [x > 143, x > 195,
                 z3.Not(x < 82)]),
     (lambda x: x * 2 - 1, lambda x:
      [x > 143, z3.Not(x > 195),
                z3.Not(x < 82)]),
     (lambda x: x + 24,
      lambda x: [z3.Not(x > 143),
                 z3.Not(x < 82)])])

x1.solve()
```

Figure 11. ConditionPack for function2

```
symbols = [LinearSymbol('x1')]
x = function1(symbols)
n = System(x)
n.solve([386])
```

Figure 12. LinearSymbol for function1

4.1 Limitations

Though Cornucopia is powerful in these respects, it also has two major limitations:

1. **Limited use cases.** Because of the highly specific nature of the symbolic variables Cornucopia uses, it can only handle a small subset of string manipulation programs (those that consist only of linear operations — adding and scaling — and branching control flow). This means that it is only useful in certain specialized cases rather than being generally applicable to all string manipulation programs.
2. **Cornucopia is not fully automated.** Reversing programs that contain conditional branching requires a quite a bit of user interaction and this amount scales up as the branches progressively become more complex. That is, though ConditionalPacks are still simpler than manual analyses of a program's branches, they become more complex as more branches and conditions are added. Additionally, the user first needs to manually recognize the types of symbols that must be created for each program.

4.2 Future Work

To address these limitations, future work extending or building upon Cornucopia is necessary.

Additional types of symbolic variables can be created that can handle a wider range of programs. Some such examples are structures that handle higher degrees of transformations as well as structures that use nondeterminism to handle the modulus operator. These would expand the possible functionalities of Cornucopia to be able to reverse more diverse types of programs. Furthermore, Cornucopia's existing composability means that these additional types of programs can be reversed individually and put together.

Addressing the second limitation could entail creating a function that automatically creates ConditionPacks for a set of else and elif clauses. This would mean that the user would not need to manually enumerate all the conditions that must be met for the path to be taken. A further expansion to this could be analyzing the source code automatically and creating ConditionPacks based on the syntax. Both of these additions would make Cornucopia more automated and would decrease the amount of user interaction necessary. Source code analysis (such as through the implementation of a typesystem or DSL) could automatically recognize where different types of symbols need to be used and translate them without requiring user input.

References

Binary Ninja 2022. Binary Ninja. https://binary.ninja. Vector 35.

Tanner J. Burns, Samuel C. Rios, Thomas K. Jordan, Qijun Gu, and Trevor Underwood. 2017. Analysis and Exercises for Engaging Beginners in Online CTF Competitions for Security Education. In *2017 USENIX Workshop on Advances in Security Education (ASE 17)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/ase17/workshop-program/presentation/burns

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

Jean-Guillaume Dumas and Clément Pernet. 2012. Computational linear algebra over finite fields. *arXiv preprint arXiv:1204.3735* (2012).

Justin Ferguson and Dan Kaminsky. 2008. *Reverse engineering code with Ida Pro*. Syngress Pub.

J. Gennari. 2021. GhiHorn: Path Analysis in Ghidra Using SMT Solvers. Carnegie Mellon University's Software Engineering Institute Blog. http://insights.sei.cmu.edu/blog/ghihorn-path-analysis-in-ghidra-using-smt-solvers/

Ghidra 2021. Ghidra (release 10.1.1). https://github.com/NationalSecurityAgency/ghidra. National Security Agency.

Lucas McDaniel, Erik Talvi, and Brian Hay. 2016. Capture the Flag as Cyber Security Introduction. In *2016 49th Hawaii International Conference on System Sciences (HICSS)*. 5479–5486. https://doi.org/10.1109/HICSS.2016.677

Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Kenny Wong. 2000. Reverse Engineering: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering* (Limerick, Ireland) *(ICSE '00)*. Association for Computing Machinery, New York, NY, USA, 47–60. https://doi.org/10.1145/336512.336526

radare2 2022. radare2 (release 5.6.8). https://github.com/radareorg/radare2.

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586