PERFUME: Programmatic Extraction and Refinement For Usability of Mathematical Expressions

Nicolaas Weideman
nweidema@isi.edu
University of Southern California,
Information Sciences Institute
Marina Del Rey, CA, USA

Virginia K. Felkner
felkner@isi.edu
University of Southern California,
Information Sciences Institute
Marina Del Rey, CA, USA

Wei-Cheng Wu
wwu@isi.edu
University of Southern California,
Information Sciences Institute
Marina Del Rey, CA, USA

Jonathan May
jonmay@isi.edu
University of Southern California,
Information Sciences Institute
Marina Del Rey, CA, USA

Christophe Hauser
hauser@isi.edu
University of Southern California,
Information Sciences Institute
Marina Del Rey, CA, USA

Luis Garcia
lgarcia@isi.edu
University of Southern California,
Information Sciences Institute
Marina Del Rey, CA, USA

ABSTRACT

Algorithmic identification is the crux for several binary analysis applications, including malware analysis, vulnerability discovery, and embedded firmware reverse engineering. However, data-driven and signature-based approaches often break down when encountering outlier realizations of a particular algorithm. Moreover, reverse engineering of domain-specific binaries often requires collaborative analysis between reverse engineers and domain experts. Communicating the behavior of an unidentified binary program to non-reverse engineers necessitates the recovery of algorithmic semantics in a human-digestible form. This paper presents PERFUME, a framework that extracts symbolic math expressions from low-level binary representations of an algorithm. PERFUME works by translating a symbolic output representation of a binary function to a high-level mathematical expression. In particular, we detail how source and target representations are generated for training a machine translation model. We integrate PERFUME as a plug-in for Ghidra–an open-source reverse engineering framework. We present our preliminary findings for domain-specific use cases and formalize open challenges in mathematical expression extraction from algorithmic implementations.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**.

KEYWORDS

binary analysis; reverse engineering

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Checkmate '21, November 19, 2021, Virtual Event, Republic of Korea.
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8552-7/21/11…$15.00
https://doi.org/10.1145/3465413.3488575

ACM Reference Format:
Nicolaas Weideman, Virginia K. Felkner, Wei-Cheng Wu, Jonathan May, Christophe Hauser, and Luis Garcia. 2021. PERFUME: Programmatic Extraction and Refinement For Usability of Mathematical Expressions. In *Proceedings of the 2021 Research on offensive and defensive techniques in the Context of Man At The End (MATE) Attacks (Checkmate '21), November 19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3465413.3488575

1 INTRODUCTION

Automatic identification of known functions is critical for a wide range of binary analysis applications, including malware analysis, binary instrumentation, and vulnerability detection. In general, function identification relies on mapping a target binary to a dataset of known functions. For instance, binary analysis tools have built-in capabilities to detect common library functions based on signatures. Advances in data-driven approaches have led to more effective and less rigid techniques that first extract features from source and target functions before mapping the representations. However, depending on the domain, the heterogeneity of an environment may hinder the efficacy of these techniques.

The effectiveness of function mapping depends on the completeness of the dataset and the canonicity of intermediate representations. For instance, prior works have focused on discovering known vulnerabilities in IoT firmware [13, 21, 22]. Static and dynamic features are extracted from known vulnerable functions and compared to functions extracted from a target binary. First, such an approach can only detect vulnerabilities from a given dataset. Second, data-driven approaches assume that machine learning can automatically learn the mapping of function features agnostic to architecture. In reality, there may be an infinite number of permutations between target architecture, compiler optimizations, and customizations of known functions. Intuitively, intermediate representations and function heuristics of a binary program do not capture the semantics of the original algorithm.

This paper aims to recover math expressions of algorithms from their low-level binary representations in a form close to their "natural" human-readable form. We present PERFUME, a framework that maps symbolic output representations of a function to high-level

mathematical expressions. PERFUME works by generating simplified symbolic output representations of a function close to the high-level mathematical expression. We show how these representations can be fed into machine translation models to translate symbolic output representations to high-level math expressions. We initially focus on mathematical domains where the output of a function is a mathematical combination of the inputs, such as cyber-physical systems or cryptographic functions. We integrate PERFUME as a plug-in for Ghidra [17]–a powerful open-source reverse engineering framework. Extracting high-level mathematical operations not only provides more architecture- and implementation-agnostic representations, but also provides a semantically-rich representation that is understandable to domain experts.

Contributions. Our contributions are summarized as follows.

- We formalize the simplification of symbolically executed function outputs to extract a symbolic output expression close to the original algorithmic expression.
- We propose PERFUME, a framework that translates the symbolic output expressions to high-level mathematical expressions from symbolic output.
- We formalize the associated challenges in generating training data corpora for machine translation and present preliminary results in machine translation.
- We integrate our symbolic execution framework into Ghidra [17] as a plug-in and open-source our code.

2 PRELIMINARIES

In this section, we briefly discuss the preliminary information necessary to understand the motivation underlying PERFUME's design.

2.1 Symbolic Execution

Symbolic execution is a static program analysis technique in which execution is simulated on symbolic program states. In a symbolic program state, some data stored in the state are made symbolic. In other words, instead of storing concrete values, symbolic expressions over symbolic variables are stored. As execution is simulated, these symbolic expressions are updated according to the semantics of the instructions. When a branch is encountered during symbolic execution that depends on symbolic data, both paths are followed with separate symbolic program states. Each of these program states receives a constraint on the appropriate symbolic expressions to capture the branching condition. Symbolic execution allows an analyst to reason about the properties of a program with respect to multiple possible configurations of a program state [3]. For example, by evaluating a symbolic expression in a program state that corresponds to the length of an input buffer, it is possible to determine if a buffer overflow can occur. Therefore, symbolic execution has benefits over dynamic analysis, in which individual traces of a program are executed with concrete program states.

In our context, we are specifically interested in the output of a function that we have performed symbolic execution on. This output is a symbolic expression capturing the relationship between the function's input, symbolic variables passed as input parameters, and the return value. Binary analysis frameworks with symbolic execution engines such as angr [1] allow for creating symbolic states of a binary program and performing symbolic execution. Specific to symbolic execution on a binary program, the symbolic expressions are stored in the registers and memory locations of the symbolic program state. The symbolic variables used in these expressions are usually bit vectors, as opposed to integers–which are more natural to humans. Symbolic expressions over symbolic bit vectors usually involve low-level bit operations, such as bit shifts. These operations hinder the readability of the expressions significantly. Thus, our goal is to simplify this symbolic expression as much as possible to produce human-readable math expressions. However, in reality, deterministic simplification of symbolic expressions is intractable for modern programs–even for simple embedded firmware. Thus, the simplified representations will need to be fed into more powerful translation mechanisms such as machine translation models to obtain a human-readable expression.

2.2 Sequence to Sequence Translation

Today's machine translation (MT) systems are sequence-to-sequence neural networks. These networks first encode the source sentence in an intermediate vector space, then predict a target sentence from the intermediate representation. The dominant MT model architecture is the transformer [24]. The unique power of the transformer comes from its *attention mechanisms*. Attention allows the model to learn which surrounding words provide the most important contextual information for translating a particular word. While transformers were originally developed for translating between natural languages, they are also an important component of large language models like BERT [7] and GPT-3 [5] and have even performed well on non-linguistic tasks like image classification [8].

In our context, we are interested in translating binary program representations to mathematical expressions. Recent works have shown that machine translation can be successfully applied to symbolic mathematics. For example, [15] used transformers to integrate functions and solve first and second order differential equations. In our case, we would like to explore the feasibility of translating a symbolic output representation to a human-readable mathematical expression.

3 PERFUME OVERVIEW

We first present the system model before describing the design goals, challenges, and overview.

3.1 System Model

We assume that a binary analyst is provided a stripped binary file. We initially focus on analyzing a single binary with fully satisfied dependencies, i.e., we assume the program behavior is self-contained within the binary–which is common in embedded firmware binaries.[1] We also assume the analyst can correctly load the binary address and is provided the target function locations in the binary. Function classification and boundary identification are outside of scope. Our goal is to extract a mathematical expression for a given target function.

[1] Although we limit ourselves to these conservative assumptions, future work can explore recent advances in fusing more complex binaries for simplified analyses [12].

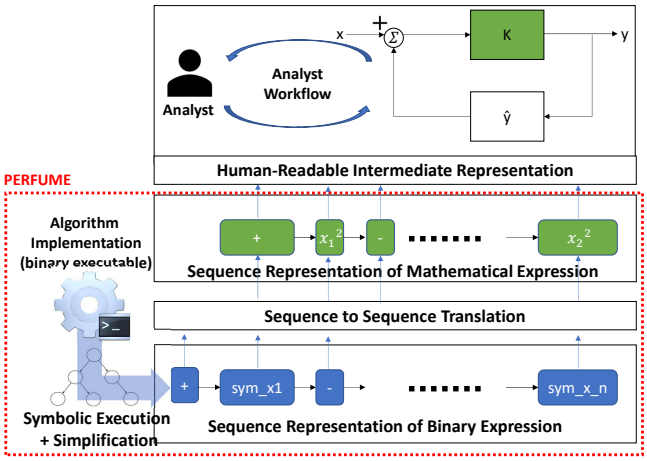

Figure 1: Overview of PERFUME. A binary implementation of an algorithm is symbolically executed and simplified to generate a sequence representation of a binary expression. The sequence representation is then translated (via machine translation) to its respective mathematical expression. The output of PERFUME is interpreted by an analyst or an analyst's toolset directly integrated into a workflow environment.

3.2 Design Overview

Figure 1 depicts an overview of PERFUME. A binary implementation of an algorithm is symbolically executed and simplified to generate a symbolic expression representation. The extracted symbolic expression is fed into a sequence to sequence translation model, which outputs a corresponding mathematical expression. An analyst interacts with the human-readable output expression and directly integrates the representation into the workflow. For instance, Figure 1 depicts an analyst integrating an output expression into a control systems model, i.e., providing semantics to the reverse-engineered algorithm.

Goals and challenges. We summarize the goals and challenges of PERFUME as follows.

- Our symbolic expression extraction should simplify expressions as much as possible, i.e., by deterministically translating bit vector representations to human-readable form.
- For readability, PERFUME should be able to replace known functions with their respective symbols, i.e., provide a mechanism to replace known functions with their symbols in the output expression.
- Ideally, machine translation is not needed. In reality, we need to identify when symbolic execution breaks down and formalize an appropriate sequence representation.

4 APPROACH

4.1 Source Representations: Binary Function Level Symbolic Output

The first step of PERFUME is to extract symbolic expressions from the target binary in order to convert them to a human-digestible representation. To this end, we use symbolic execution. In this section, we discuss the approach we use to extract symbolic expressions from binary executable files. This module is used both in the context of data generation, to create pairs of mathematical expressions and symbolic expressions, as well as in the reverse engineering phase, to extract symbolic expressions to be translated. We follow a binary function based approach, in which we perform symbolic execution on a particular function, the *target function*, recovered from the target binary, both supplied by the analyst. The resulting symbolic expression captures the relationship between the input of the function (its parameters) and its output (the return value). We assume this relationship captures the entire mathematical computation implemented in the function. In other words, we assume no component of the computation is retrieved via a memory access, either to global memory or to a pointer passed as a parameter. We discuss the complications caused by such memory accesses in Section 8.2.

The benefit of performing symbolic execution, as opposed to analyzing the binary instructions directly, is that the symbolic expression obtained is relatively free from the noise found in the binary implementation of a mathematical computation. Such noise, for example, can be introduced by instructions manipulating memory addresses or moving values between registers. While these instructions are necessary for computation, they do not contribute to the semantics of the program.

4.1.1 Symbolic Program State Creation. In preparation of symbolic execution, we create a symbolic program state with the instruction pointer set to the entry point of the target function. We also modify the program state to include symbolic variables in the registers and memory locations that correspond to the function's parameters. This can be achieved by using information regarding the calling convention employed by the target binary. The calling convention dictates which registers and memory locations should be used to pass function parameters and return values between a caller and callee function. Using a liveness analysis on these registers and memory locations at the callsite of the callee function, PERFUME can identify which function parameters exist for the callee function. We allow the analyst to provide a name for each of the symbolic variables used as function parameters. For machine translation, we use this to ensure that the variable names in the mathematical and symbolic expressions are consistent. Also, in the reverse engineering phase, this allows the analyst to choose whichever variable names aid in improving understandability of the extracted expression.

4.1.2 Symbolic Execution. Symbolic execution is then performed with this program state as the initial state until all branches reach a return instruction of the function. We refer to these states as the *final states*. From each final state, we extract the return value, also according to the calling convention. This return value is a symbolic expression that captures the relationship between the input and

output of the function for a single execution path. This path is subject to path constraints placed on the registers and memory by branching instructions. We merge the symbolic expressions of all final states into a single expression on the path constraints.

4.1.3 Simplification. After symbolic execution, we have a raw bit vector representation of the mathematical expression. The next step is to simplify this expression, before passing it to the MT phase, in order to improve its understandability. To this end, we perform a number of deterministic steps to strip away information from the bit vector representation that is unnecessary when only trying to gain a higher level understanding of the computation being performed. This information, for example, includes the operations to increase the width of a bit vector by appending zero bits. We also convert constant bit vectors that represent integers in two's complement notation into signed integer values. Similarly, we convert constant bit vectors representing floating point numbers to rational numbers.

Symbolic execution example. We show an illustrative example of how PERFUME extracts a symbolic expression from a function in Figure 2. The source code of the function in question, `f`, is shown in Listing 1. PERFUME starts by constructing a symbolic program state that corresponds to calling the function `f` with input parameters a and b, which are symbolic variables. We assume the calling convention for this example dictates that the first and second integer arguments to a function are passed in the registers `rdi` and `rsi`, respectively. After creating the start state, PERFUME performs symbolic execution. The conditional statement, shown on Line 2 of Listing 1, causes a branch in execution. One branch has the path constraint $a < b$, while the other has $a \geq b$. As there are no further branches in execution, we have two final states. We assume, according to the calling convention, an integer return value is passed in the register `rax`. We merge the symbolic expressions extracted from this register into a single expression, while using the path constraints. This yields the raw symbolic expression `<BV64 if a[31:0] <s b[31:0] then (0x0..0x1 + a[31:0])else (0x0..0x2 + b[31:0])>`. We perform the simplification step on this symbolic expression, which yields the simplified symbolic expression `If(a<b,(1+a),(2+b))`. Referring back to Listing 1, it is clear that the simplified symbolic expression is a more understandable representation of the source code.

```
1 int f(int a, int b) {
2     if (a < b) {
3         return a + 1;
4     } else {
5         return b + 2;
6     }
7 }
```

Listing 1: An example function to illustrate how PERFUME extracts symbolic expressions.

4.1.4 Symbolic Expression Summarising. The default mathematical operators provided by a processor architecture are usually limited to a small number of rudimentary operations (e.g., addition, subtraction, and multiplication). In order to compute more complicated mathematical expressions, programmers frequently rely on mathematical libraries. This presents challenges and opportunities for a symbolic execution based approach. As the mathematical computations inside the functions of these libraries are often complicated, this can significantly increase the execution time of the symbolic execution engine and complicate the resulting symbolic expression. However, since these are usually well-known mathematical functions, we can summarize the computations of these functions in the symbolic expression, by using a symbolic function, instead. The function arguments passed to this library function during symbolic execution are now used as input to this new symbolic function. This significantly improves the understandability of the symbolic expression. Note that since this is only a symbolic function, it is not possible to evaluate it on input arguments using an SMT solver. However, this is not a limitation of PERFUME, because we are only interested in the readability of symbolic expressions and not in evaluating them.

If the binary is stripped, it is necessary for PERFUME to have information regarding the prototype of the function (how it receives input and yields output), as well a sensible name to use for its summary in the resulting symbolic expression. We delegate the decision of whether to perform symbolic execution on a callee function, or to summarise it, as well as choosing the name, to the analyst.

We illustrate the idea of either including or summarising callee functions in the resulting symbolic expression with the functions shown in Listing 2. The target function, `f0`, makes a call to a callee function `f1`. PERFUME can summarise the computations within this function, by replacing them with a symbolic function. The resulting expression is `a + f1(b + 0.5, c)`. We assume the name `f1` of this function is provided to PERFUME. On the other hand, if PERFUME traverses `f1`, the resulting expression includes the subexpressions found therein, that is `a + 3(b + 0.5)+ c`. Depending on the context and program under analysis, an analyst may either choose to include or summarize the expressions of a callee function to improve understandability.

```
1 float f0(float a, float b, float c) {
2     return a + f1(b + 5.0, c);
3 }
4
5 float f1(float b, float c) {
6     return 3.0 * b + c;
7 }
```

Listing 2: Example functions to illustrate the capability of PERFUME to summarise the mathematical computations found in functions.

4.2 Target Representations: Mathematical Expressions

Our target representations are human readable high-level mathematical expressions. As demonstrated in Figure 2, we currently focus on transferring low-level operations into simple math expressions. Such expressions contain simple arithmetic operations, such as addition and subtraction, as well commonly used mathematical functions, such as sine and cosine. We discuss the details of synthesizing expressions in Section 4.3.

4.3 Dataset Synthesis

Target representation synthesis. To synthesize target representations for MT training, we instrumented the implementation of the random math expression generator from a previous work [15]. The process is illustrated in Figure 3. We first generate a random tree structure. The internal nodes are then filled with random unary/binary operators and functions. We subsequently fill in the leaves

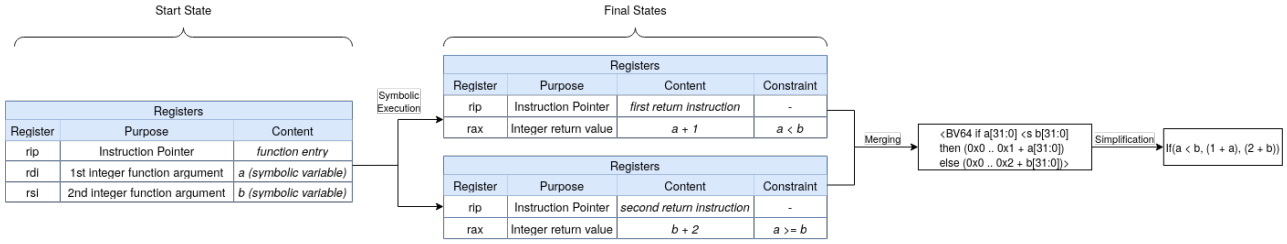

Figure 2: A visual illustration of how PERFUME uses symbolic execution to extract a symbolic expression from the function shown in Listing 1.

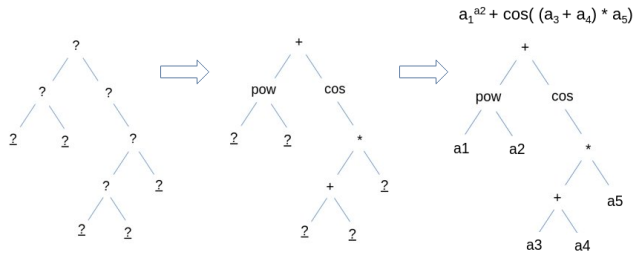

Figure 3: The process of synthesizing target representation.

Math expression	a1+(a2-a3)
Target representation (infix)	a1 + (a2 - a3)
Source representation (infix)	a1 + (a2 + (-1 * a3))
Target representation (prefix)	+ a1 - a2 a3
Source representation (prefix)	+ a1 + a2 * -1 a3

Table 2: Data synthesized.

Unary operators	-
Binary operators	+, -, *, /, %
Binary bit operators	or, xor, left-shift, right-shift
Unary functions	abs, acos, asin, atan, ceil, cos, cosh, cbrt, exp, fabs, floor, log, log10, sin, sinh, sqrt, tan, tanh, acosh, asinh, atanh, atanh, exp2, log2, tgamma
Binary functions	pow

Table 1: A list of supporting arithmetic operators and functions in our target representation.

with random variables or constants. Although we initially focus on generating expressions consisting of simple arithmetic operators, $(+, -, \times, \div, \%)$, the mathematical expression synthesizer supports more complex functions and operators such as sine and cosine. In Table 1 we list out the operations and functions we include for synthesizing. The previous implementation was modified to support consistent variable renaming, i.e., the first variable is named `a1`, the second variable is `a2`, and so on. This normalizes the variable names prior to translation.

The math expression synthesizer currently supports both infix and Polish notation, although we will initially focus on infix expressions since they are more natural. We will also initially ignore monotonic ordering of expressions as we anticipate the machine translation models will learn commutative, associated, and distributive properties.

Source representation data synthesis. In our preliminary experiments, we try to answer two questions:

- What is the best training model for translating our representations?
- Which target representation is easier to train with, infix or prefix? Is MT able to reason about the parentheses of math expressions?

In order to answer these two questions, we further simplify our target representation data synthesis to only "+" and "-" over integer variables. On the source representation, we disable an option in angr, which will turn a simple math expression `a - b` into `a + (-1 * b)` in symbolic execution. Although it looks like we are getting further away from the target representations, such difference is actually a nice entry problem for MT to solve. We can fine tune our model well before trying to translate more complex expressions. In Table 2 we give an example of target and source representations for a simple math expression.

4.4 Machine Translation for Symbolic Math

In the next step of the PERFUME framework, we use machine translation to convert symbolic execution output to human-readable expressions. There are several challenges associated with using MT in a mathematical setting. First, the source and target languages in our domain are much more constrained than natural languages. While an MT model on natural language might see tens or hundreds of thousands of unique vocabulary words, our preliminary dataset has a total vocabulary size of 40. This huge difference in vocabulary size causes natural language translation models like the base transformer [24] to overfit and perform abysmally for our application. To combat this overfitting and achieve acceptable translation performance, we conducted preliminary experiments to determine the best Transformer hyperparameters for our specific application. The results of these experiments are summarized in Table 3 and our model implementation is described in 5.2.

model	layers	model dim.	FF dim.	Dev. BLEU	Test BLEU
Transformer Base [24]	6	512	2048	3.6	3.6
Transformer Slim	6	128	512	3.6	3.6
Transformer Shallow	2	512	2048	8.0	8.0
Transformer Slim, Shallow	2	128	512	8.0	8.0
Transformer Slimmer, Shallow	2	64	256	27.9	27.9
Transformer Dim=Vocab, Shallow	2	40	160	40.3	40.6
Transf. Dim=FF Dim=Vocab, Shallow	2	40	40	22.7	22.8
Transformer Vocab=Dim.	6	40	160	10.4	10.4
Transformer Slimmest Shallow	2	20	80	6.9	6.9

Table 3: Results from MT experiments. Model dimension was found to be the most important hyperparameter for scaling to our use case.

5 IMPLEMENTATION

5.1 Symbolic Expression Extraction

We implement the symbolic execution module of PERFUME as a plug-in to the Ghidra software reverse engineering suite [17]. Ghidra allows an analyst to explore a binary executable using a polished graphical-user interface (GUI) and gather information this way. However, while Ghidra is a user-friendly reverse engineering tool, its out-of-the-box features do not include some of the more advanced program analysis techniques, such as symbolic execution. We create a separate module, to be used by the plug-in, that is built on the angr [1] binary analysis framework. This module handles the symbolic execution.

Our Ghidra plug-in contains a GUI that allows the analyst to enter the information required by the angr module to perform symbolic execution. This information is passed to the angr module, which performs symbolic execution and passes the simplified symbolic expression back to Ghidra, to be displayed to the analyst.

In the angr module, we configure the symbolic execution engine to optimize the readability of the created symbolic expressions. In particular, we noticed that disabling all options to simplify symbolic expressions yields the most readable results. We use the hooking feature of angr to achieve the symbolic expression summarising. This feature allows one to replace specific instructions during symbolic execution. In its default configuration, the symbolic execution engine uses this feature to replace the return value of unavailable library functions with an unconstrained symbolic variable. Using such a variable is unsatisfactory for our purposes, since this discards all information regarding the symbolic expressions passed to this function as input parameters. Instead, we use the hooking feature to replace the return value with a symbolic expression consisting of a symbolic function and the appropriate symbolic expressions as input parameters. We create these symbolic functions dynamically, during symbolic execution, using a name provided by the analyst. We use the Decompiler Parameter ID analysis of Ghidra to determine the prototype of the function in order to select the correct symbolic expressions as the input parameters to the function.

5.2 Machine Translation Experiments

In Table 3, we report BLEU [19], the standard evaluation metric in machine translation. BLEU is reported on a scale of 0 to 100, with higher scores being better. Details of evaluation metrics are discussed in 6.2. Our experiments were conducted using the fairseq [18] Transformer implementation. All experiments used infix notation for source expressions (line 2 of Table 2), 8 attention heads, vocabulary size of 40, batch size of 4096 tokens, 8000 warmup steps, inverse square root learning rate schedule with maximum LR of 0.05, and ADAM optimizer with $\beta_1 = 0.9, \beta_2 = 0.98$. The vocabulary was constructed using whitespace tokenization. While MT models usually split tokens into subwords in the natural language setting, the tokens in our mathematical expressions are so short that there are no meaningful subword units, and whitespace tokenization is sufficient. Models were trained using cross-entropy loss, stopping when no improvement in validation loss was achieved at 10 consecutive checkpoints. Training took about 1 hour per model on one NVIDIA Tesla P100 GPU.

Our experiments successfully tuned hyperparameters for machine translation of symbolic expressions to parenthesized mathematical expressions. We found that the best hyperparameters are: 2 Transformer layers deep on both encoder and decoder side and model dimension of 40 (equal to the vocabulary size), with corresponding feedforward network dimension of 160. Conceptually, model dimension is the "width" of the model, or the number of hidden units at each layer. Using the best hyperparameter settings, we achieved a BLEU score of 40.3 on validation data and 40.6 on testing data. As shown in Table 3, the model dimension, i.e. the number of hidden units per self-attention layer, is the most important hyperparameter for scaling transformers down to mathematical expressions. However, our results also show that model depth is important and that optimal performance is only achieved when both depth and width of the model are scaled down considerably from the base transformer. The use of a smaller model also means that training takes less time, money, and energy than a full-size transformer.

Fig. 4 shows the training and validation loss curves for our best experiment. Training loss is shown in red and validation loss is shown in blue. The x-axis is the number of training steps (gradient updates) conducted, and the y-axis is a unitless loss value representing the current "wrongness" of the model's predictions. As training continues, both training and validation loss curves level off and begin to converge, showing that the model is fitting properly to the data. This indicates that we have a reasonable set of hyperparameters and we can expect good performance in future MT experiments.

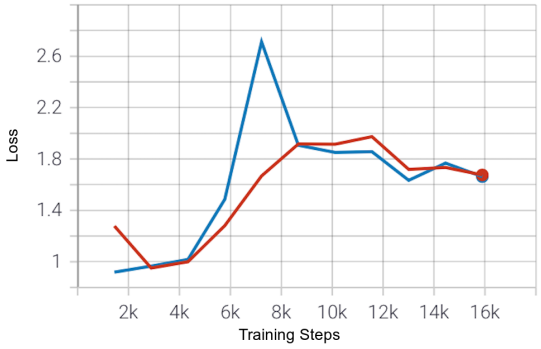

Figure 4: Training (red) and validation (blue) loss curves for our best MT model. Convergence of the two curves indicates model is fitting properly to data.

6 EVALUATION

In this section, we evaluate the two phases of PERFUME, separately. In order to evaluate the symbolic execution phase, we extract the symbolic expression of a number of handcrafted functions and compare these expressions, manually, to the source code. Afterward, we motivate the applicability of PERFUME to real-world scenarios by presenting a case study. Here, we extract a symbolic expression from a function from a binary executable found within the ArduPilot project [2]. Finally, we discuss the shortcomings of standard machine translation evaluation metrics and alternative metrics that are better-suited for symbolic mathematics.

6.1 Symbolic Execution

6.1.1 Evaluating on Synthetic Data. We start the evaluation of the symbolic execution phase of PERFUME on a number of handcrafted functions. We create these functions such that the input parameters are used in a mathematical computation, which is then returned, in order to conform to the assumptions discussed in Section 4.1. We show a number of these functions in Table 4, as well as the corresponding symbolic expression. In this table, we can see that the symbolic expressions extracted for the functions `f1`, `f2`, `f3` and `f4`, match the source code nearly identically. For function `f4` a slight difference is introduced, transforming `2.0 * a` into `a + a`. The symbolic expressions extracted for the functions `f5` and `f6`, on the other hand, are almost unrecognizable from the source code. For both these functions, the difference originates from how the source code instructions are encoded into binary instructions. For function `f5`, note that calculating `(a*1717986919)>>1)-(a>>31)` is mathematically equivalent to calculating `a / 5` in modulus 2^{32} (32 is the bit width of the integer type of this program). The compiler opted for this encoding of the mathematical computation, as it is more computationally efficient. Finally, the complexity in the symbolic expression for function `f6` is introduced by how floating point values are compared.

6.1.2 Symbolic Expression Summarising. Next, we evaluate the ability of PERFUME to summarise symbolic expressions. Listing 3 shows a function using a function from the SX library [23] to calculate the mathematical floor function of a number.

Source	Extracted Symbolic Expression
`float f1(float a, float b) { return 3.0 * a / b; }`	`(a * 3.0)/ b`
`float f2(float a, int b) { if (b) { return a / 2; } else { return 3.0 * a; } }`	`If(b!=0,(a/2.0),(3.0*a))`
`float f3(float a, float b, float c) { return pow(a * b, 4) + cos(c); }`	`(cos(c))+(pow(a*b,4.0))`
`float f4(float a, float b) { return 2.0 * a + b; }`	`b+(a+a)`
`int f5(int a, int b) { return (a / 5) * b; }`	`b*((a*1717986919)>>1)-(a>>31)`
`float f6(float a, float b) { if (a <= b) { return a; } else { return b; } }`	`If(((If(b<a,1,(If(b>a,0,(If(b==a,64,69)) ))))&&1)==0,a,b)`

Table 4: A number of handcrafted functions we use to evaluate the symbolic execution phase of PERFUME and the corresponding simplified symbolic expression.

```
1 float f(float a, float b) {
2     return a * sx_floor(b);
3 }
```

Listing 3: An example program to show the difference in understandability of two symbolic expression extracted by PERFUME of a function using the `sx_floor` function of the math library, SX [23]

We use this function to show the significant difference symbolic expression summarising can make in the understandability of a symbolic expression. Figure 5 shows the resulting symbolic expression when the subexpressions generated during symbolic execution of the `sx_floor` function is included. It can be seen that the expression is complicated and difficult to understand. On the other hand, when we instruct PERFUME to summarise the function, `sx_floor` with the symbol `floor`, the resulting expression is reduced to: `(floor(b))* a`.

6.1.3 Case Study: ArduPilot. To test the effectiveness of the symbolic execution of PERFUME, we apply it to a target function from a binary executable found within the ArduPilot project [2]. We identify a function that corresponds to the assumptions we make regarding the target function, discussed in Section 4. In particular, we use the function shown in Listing 4, from the rover module in ArduPilot, as target function.

```
1 float get_steering_out_rate(float desired_rate, bool
      motor_limit_left, bool motor_limit_right, float dt);
```

Listing 4: The prototype of the function within the ArduPilot project [2], we use as target function, to evaluate symbolic expression extraction.

This function calculates a steering servo output, which it returns, from a given desired rate, given as function input parameter.

We perform symbolic expression extraction on this function. When naming the symbolic variables for the input parameters, we use the same names as used for the variables in the source code. Note that at the binary level, the C++ language passes a pointer

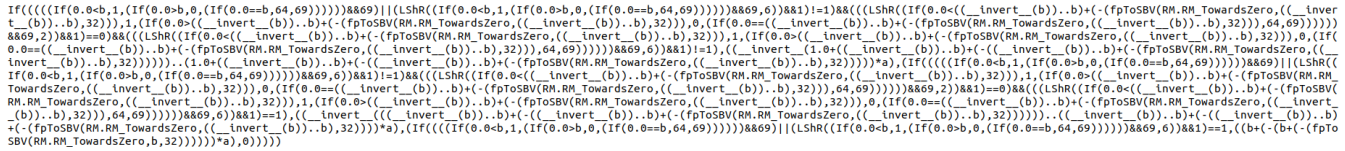

Figure 5: The simplified symbolic expression extracted from the program show in Listing 3, when including the subexpressions from the `sx_floor` function.

to the specific instance to a called instance method as the first function parameter. We use the name `this` for the symbolic variable corresponding this function parameter. We instruct PERFUME to replace the symbolic expressions of the functions `update_all`, `get_yaw_rate_earth` and `get_ff` with symbolic function calls where each function has the same name as in the source code. As the extracted symbolic expression has 296 leaves, we only show a subexpression below.

```
...
update_all(this+16,
  desired_rate,
  get_yaw_rate_earth(mem_fffffffffff80000_1675_64),
  0
) +
get_ff(this+16)
...
```

The symbol `mem_fffffffffff80000_1675_64` in this subexpression is a symbolic variable caused by a symbolic memory access. We discuss the challenge these present in Section 8.2. Finally, we show a portion of the source code to which this subepxression corresponds in Listing 5. The correspondence between the original source code and the symbolic expression can clearly be seen.

```
1 float output = _steer_rate_pid.update_all(_desired_turn_rate,
2   _ahrs.get_yaw_rate_earth(),
3   (motor_limit_left || motor_limit_right)
4 );
5 output += _steer_rate_pid.get_ff();
```

Listing 5: Two lines of source code of the `get_steering_out_rate` function of the rover module from ArduPilot [2].

6.2 Machine Translation

The standard evaluation metric in machine translation is BLEU [19]. BLEU measures the proportion of 1-grams through 4-grams that match between the output translation and the reference translation. While this is a useful metric for natural language, it does not capture mathematical equivalence well. Therefore, the most pressing goal for the MT part of our work is to establish a better metric than BLEU for translation quality in the mathematical domain.

BLEU was originally proposed in the context of natural language translation, as a metric for similarity between a MT system's output and one or more "correct" reference translations produced by human translators. BLEU measures n-gram-wise matching between the output and the reference, but it has no notion of overall "correctness" or conveyance of meaning. Additionally, BLEU's reliance on n-grams means it is very sensitive to the order of output tokens. This is a desirable property for natural language where word order affects meaning. However, in a mathematical domain where associativity and commutativity apply, it has the effect of scoring two mathematically equivalent outputs differently.

More generally, BLEU is primarily concerned with surface-level string similarity rather than semantic correctness. This problem has been recognized in the NLP community since BLEU became widely used [6]. It poses particular problems for our use case. Ultimately, we care much more about whether the MT system's output is algebraically equivalent to the reference expression than about how many n-grams overlap between the two expressions. To choose models that can produce mathematically correct output and to teach them that they should do so, we need an evaluation metric that encodes the type of matching and correctness that is most important for our use case.

This evaluation metric question is actually a subproblem of a larger problem encountered using models optimized for human language to produce symbolic math output. Mathematical expressions have different syntax and semantics than natural language sentences. The major syntactic issue we have encountered is the issue of balancing parentheses. While [11] showed that self-attention networks cannot be theoretically guaranteed to recognize or generate balanced parentheses, [10] showed that they can generally perform well enough to be useful in practice. In our use case, we have found empirically that most of our transformer output is only off by one or two closing parentheses (either missing or extra inserted at the end) and is therefore easily corrected during postprocessing.

While unbalanced parentheses cause expressions to differ from the reference by only a few tokens and therefore do not affect the BLEU score too much, they cause significant downstream problems. Because we care about mathematical equivalency more than token-wise matching, we plan to use the SymPy symbolic mathematics library [16] to simplify and check equivalence of generated expressions. Initial tests of this method failed because SymPy fails to process expressions with unbalanced parentheses. After applying heuristic postprocessing to rebalance the parentheses in MT output, we found that SymPy simplification and equivalence checking can be used as an auxiliary evaluation metric on validation and test data.

7 RELATED WORK

We provide a brief overview of related works. We broadly categorize the works into signature-based function approaches and semantic-pattern matching approaches. **Signature-based approaches.** A plethora of works exist that focus on identifying signatures of functions within a binary program. State-of-the-art binary analysis tools such as IDA Pro [9], Ghidra [17], and angr [1] are equipped with the capability of detecting when common libraries are used

by a binary, e.g., standard math functions. Similarly, tools such as Byteweight [4] that extract feature signatures provide more robust signature-detection, but they do not provide a generalizable mechanism for recovering math expressions for unknown functions.

Semantic pattern-matching. Recent works have similarly used pattern-matching or signatures to map binary representations to their embedded cyber-physical system contexts. Although the aforementioned binary analysis tools are equipped with decompilers, they do not recover the high-level algorithmic expressions we are targeting. ICSREF [13] used signature-based techniques to recover the entire semantics of an embedded cyber-physical system controller. In theory, such an approach could provide a deterministic mapping of functions to high-level expressions. However, this approach does not generalize for unknown binary functions. Similarly, recent works in binary similarity analysis have focused on first extracting and identifying semantic patterns for known mathematical functions, e.g., cryptographic functions [14, 26]. However, these works are still limited to the known set of patterns, and would be complementary to PERFUME. The Mismo [22] framework is closest to our approach as it performs semantic pattern-matching between the abstract syntax trees of known control algorithms to the abstract syntax trees of symbolically executed functions. However, this approach relies on the completeness of the known function templates. Moreover, the approach is not robust to changes in the ordering of arithmetic operations. However, future work can focus on mapping PERFUME's extract mathematical expressions to the set of known mathematical functions. Establishing a better metric for formal equivalence of symbolic expressions would further facilitate such semantic pattern-matching.

8 DISCUSSION & CHALLENGES

8.1 Fine-grained math sequences extraction

In the current version of PERFUME, the analysis is performed on a target function, which we assume follows a very particular format in terms of prototype. There is room for improving PERFUME, to allow us to extract and translate any mathematical computations. To move away from a function-level based approach, the next logical step is to define an analysis that performs symbolic execution on a sequence of instructions. Ideally, PERFUME would identify the instructions that contribute to a mathematical computation, automatically. A road to achieve this, could be to identify the result of the mathematical computation in the binary code. This could be, for example, the instruction that saves the final result of the computation to memory or a register. Then, to identify the instructions that make up the mathematical computation, PERFUME can calculate a backward slice from this identified result instruction. As a result, PERFUME only needs to perform symbolic execution on the slice instead of the entire function. This will allow analysts to use PERFUME on a finer granularity and also yield significant improvements in terms of symbolic execution time. However, calculating an accurate backward slice via static analysis is an open research problem orthogonal to PERFUME.

8.2 Scalability

While translation works well for simple functions, as shown in Section 6, we expect to encounter a number of challenges when scaling up to functions used in real-world programs. In Section 4, we make the assumption that the sought after mathematical computation can be expressed solely in terms of the input parameters of the function. In real-world programs, this may not necessarily be the case, as computations often rely on memory accesses to retrieve values. These can be either to a memory address passed as pointer via the input parameters of the function, or to global memory. In the symbolic expression, these will manifest as symbolic variables, unrelated to the input of the function. This, will complicate the understandability of the expression for the analyst. Furthermore, we expect computations implemented in real-world programs to be significantly larger. As symbolic execution is a computationally expensive process, this may pose a significant limitation to analyzing complicated functions. Even if a symbolic expression is extracted successfully, such an expression may be very complicated, in terms of sequence length. For example, in Figure 5 we show the expression for calculating the floor of a number. There is a notable contrast between the simplicity of the mathematical concept and the length of its expression. This presents a challenge if the analyst does not have sufficient information to propose a meaningful name to summarize such a function, as long expressions are a challenge for MT. Generally, the semantic complexity of the mathematical expressions is not a significant concern for MT models, because the math expressions are still less complex than natural language sentences. Translating more complex expressions will require larger transformers than we are currently using, but these transformers will still be small enough to be trainable on our hardware in a reasonable amount of time. Our main concern with scalability of our MT models is the increase in sequence length for more complicated functions. In particular, MT models face practical limitations on the number of tokens per sequence they can process. While there is no theoretical limit on the input length for transformers, the memory requirement of the self-attention mechanism is quadratic in sequence length. Additionally, longer sequences mean that fewer examples will fit in GPU memory, and batch sizes must be small. Too small a batch size can increase training time and lead to suboptimal model performance.

8.3 Pointer & data references

While PERFUME has the ability to abstract away the symbolic expressions contained within the callee functions of the target function, these callee functions are also subject to strict assumptions, similar to the target function. In particular, in order to use a symbolic function in lieu of the symbolic expressions extracted from a callee function, it is necessary that this function follows a specific format. This format is similar to the restrictions we place on the target function, in the sense that we assume the input parameters are used directly in the mathematical computation. Also, we assume the callee function only passes data back to the caller function via the return value. It becomes significantly more complicated to do the replacement when these assumptions no longer hold. It is possible, for example, that the callee function receives pointers as its input parameters, which are then dereferenced and used in the mathematical computations. Similarly, it is also possible that the callee function does not pass the result of the mathematical computation back to the caller function via return value. Instead, this can also be

```
void Quaternion_multiply(Quaternion* q1, Quaternion* q2,
     Quaternion* output) {
    ...
    output.v[0] = q1->v[0] * q2->w
                + q1->w * q2->v[0]
                + q1->v[1] * q2->v[2]
                - q1->v[2] * q2->v[1];
    output.v[1] = ...;
    output.v[2] = ...;
}
```

Listing 6: An example of program starting and returning with data reference [25]

done by modifying the memory at these pointers directly. This is common in code implementing vector and matrix mathematics. We provided an example of such function in Listing 6. How to capture this behavior in a symbolic expression is not immediate.

8.4 Semantic gaps introduced by compiler optimization

As we have seen in Section 6, besides the natural gap between high-level mathematics and its implementations, there are several cases where compiler introduces differences in the binary code. We listed a few examples in Table 5. These cases cannot be naively translated with the current PERFUME pipeline and require more sophisticated approaches to match the mathematics equivalent.

Integer (a * 5)	(a « 2) + a
Float (a * -1)	a XOR 0x80000000

Table 5: Examples of compiler introduced differences.

8.5 Limitations of Symbolic Expressions Summarising

The ability of PERFUME to automatically include highlevel symbolic functions currently requires the analyst to select which functions to summarise and to supply a meaningful name for the replacement symbolic function. A direction to overcome these limitations could be to perform the replacement of subexpressions only after symbolic execution has been completed. The idea is to analyze various subexpressions of the final symbolic expression in order to identify any of these that perform a well-known mathematical computation. These can then be replaced with an appropriate symbolic function to create a new higher level symbolic expression with improved understandability.

Another problem we encountered when symbolic executing binaries from ArduPilot is handling indirect function calls, caused by invoking class functions. In the C++ programming language, class functions are often stored as function pointers in a class data structure. Invoking such a function, involves calculating the target of the call operation dynamically. Resolving such indirect jumps is an open research problem.

8.6 Higher Level Representation

While PERFUME has shown promise in converting low-level mathematical computations to a human-readable form, the output representation is still not at a level you would find, for example, in a text book. We are exploring the possibilities of two advanced target representations. The first is to generate more complex math expressions like integration, differentiation, matrix arithmetic, etc. The other is to combine multiple parts of math expressions into flow charts of controlling diagrams.

8.7 Next Steps for Machine Translation

We have two current goals within the MT section of the project. Our first goal is machine translation directly from assembly code to mathematical expressions. We are currently working on hyperparameter tuning, using the values established in our previous work as a starting point. So far, we are finding that we need a much larger translation model to handle assembly code, and we have not yet achieved a useful translation directly from assembly to mathematical expressions. However, based on our preliminary results, we are optimistic that direct assembly-to-math translation will be possible.

Our second and more general goal is to further improve our architecture, training, and evaluation to be well-suited for mathematical expressions, with evaluation as our first priority. Ideally, our evaluation would go beyond binary equivalence checking to a metric that gives partial credit for partially correct expressions. Therefore, we plan to use tree edit distance (TED) as a metric for how close to equivalent two unequal expressions are. TED is a natural choice for several reasons. First, mathematical expressions can be easily expressed as trees, where leaves are variables and internal nodes are operators. Second, edit distance metrics encode a notion of how far from the reference the output actually is; intuitively, this is similar to the cross-entropy loss value used during training. Finally, TED allows for a cost function that depends on where the node is in the tree, so it is possible to penalize mistakes that affect a greater proportion of the output more heavily. Intuitively, we would like to penalize an incorrect negative sign that is at a low tree depth, meaning it distributes over several terms, more heavily than a deeper sign that only affects a few terms. While we initially had some concerns about the efficiency and practicality of incorporating TED into our evaluation, the AP-TED+ algorithm of [20] uses time and space linear in the size of the input trees and has been shown to run in a few milliseconds for trees comparable in size to our data. In order to incentivize mathematical correctness during training, we plan to introduce a TED term and a binary correctness term to the cross-entropy loss function. We also plan to explore task-specific refinements to the attention mechanism to better capture the syntax of mathematical expressions.

9 CONCLUSION

In this paper, we presented PERFUME, a framework to extract high-level mathematical expressions from binary programs. PERFUME symbolically executes target functions while simplifying the output representation from bit vector arithmetic to integer arithmetic, i.e., a more human-readable representation. The simplified representation is then fed into a machine translation model to translate the symbolic output to “natural" math expressions. We presented preliminary findings for candidate functions within real-world embedded firmware code, and presented quantitative results from the subsequent machine translation. We have integrated the symbolic execution component as a plug-in for Ghidra. We enumerated future directions for both the symbolic and machine translation

components, which include validating our approach in the context of analyst workflows.

10 ACKNOWLEDGEMENT

This research is based upon work supported by DARPA's ReMath program, Contract HR00112190020. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

REFERENCES

[1] angr. 2021. The Angr binary analysis platform. http://angr.io.
[2] ArduPilot. 2009. ArduPilot. https://github.com/ArduPilot/ardupilot.
[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.
[4] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)*. 845–860.
[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf
[6] Chris Callison-Burch, Miles Osborne, and Philipp Koehn. 2006. Re-evaluating the Role of Bleu in Machine Translation Research. In *11th Conference of the European Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, Trento, Italy. https://aclanthology.org/E06-1032
[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423
[8] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).
[9] Chris Eagle. 2011. *The IDA pro book*. no starch press.
[10] Javid Ebrahimi, Dhruv Gelda, and Wei Zhang. 2020. How Can Self-Attention Networks Recognize Dyck-n Languages?. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 4301–4306. https://doi.org/10.18653/v1/2020.findings-emnlp.384
[11] Michael Hahn. 2020. Theoretical Limitations of Self-Attention in Neural Sequence Models. *Transactions of the Association for Computational Linguistics* 8 (2020), 156–171. https://doi.org/10.1162/tacl_a_00306
[12] Alexander Heinricher, Ryan Williams, Ava Klingbeil, and Alex Jordan. 2021. Weldr: fusing binaries for simplified analysis. In *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. 25–30.
[13] Anastasis Keliris and Michail Maniatakos. 2019. ICSREF: A Framework for Automated Reverse Engineering of Industrial Control Systems Binaries. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/icsref-a-framework-for-automated-reverse-engineering-of-industrial-control-systems-binaries/
[14] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2020. Revisiting Binary Code Similarity Analysis using Interpretable Feature Engineering and Lessons Learned. *arXiv preprint arXiv:2011.10749* (2020).
[15] Guillaume Lample and François Charton. 2019. Deep Learning for Symbolic Mathematics. *CoRR* abs/1912.01412 (2019). arXiv:1912.01412 http://arxiv.org/abs/1912.01412
[16] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, Amit Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (Jan. 2017), e103. https://doi.org/10.7717/peerj-cs.103
[17] NSA. 2021. Ghidra. https://ghidra-sre.org/.
[18] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A Fast, Extensible Toolkit for Sequence Modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*.
[19] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Philadelphia, Pennsylvania, USA, 311–318. https://doi.org/10.3115/1073083.1073135
[20] Mateusz Pawlik and Nikolaus Augsten. 2016. Tree edit distance: Robust and memory-efficient. *Information Systems* 56 (2016), 157–173. https://doi.org/10.1016/j.is.2015.08.004
[21] Pengfei Sun, Luis Garcia, Gabriel Salles-Loustau, and Saman Zonouz. 2020. Hybrid firmware analysis for known mobile and iot security vulnerabilities. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 373–384.
[22] Pengfei Sun, Luis Garcia, and Saman Zonouz. 2019. Tell me more than just assembly! reversing cyber-physical execution semantics of embedded iot controller software binaries. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 349–361.
[23] Sepehr Taghdisian. 2018. sx. https://github.com/septag/sx.
[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 5998–6008.
[25] Martin Weigel. [n. d.]. MartinWeigel/Quaternion. https://github.com/MartinWeigel/Quaternion/blob/master/Quaternion.c
[26] Dongpeng Xu, Jiang Ming, and Dinghao Wu. 2017. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 921–937.