A Search System for Mathematical Expressions on Software Binaries

Ridhi Jain, Sai Prathik, Venkatesh Vinayakarao and Rahul Purandare
Indraprastha Institute of Information Technology, Delhi
New Delhi, India
{ridhij,saip,venkateshv,purandare}@iiitd.ac.in

ABSTRACT

Developers often ask for libraries that implement specific mathematical expressions. A fundamental bottleneck in building information retrieval (IR) systems to answer such mathematical queries is the inability to detect a given expression in software binaries. While we have a few math IR solutions such as EgoMath2 and Tangent-3 that work over text documents, none exist to search over software binaries. Our vision is to build a search system for binaries to answer queries containing mathematical expressions. A wide variety of compilers and differences in the way they optimize the code, pose difficult challenges to solve this problem. In this work, we discuss our preliminary results in detecting mathematical expressions in software binaries. We use a knowledge base assisted approach to solve this problem. We are able to search mathematical expressions with a precision of **80%** and a recall of **53%**. This work opens up interesting research opportunities in areas such as software security and performance, to help analysts in identifying and analyzing binaries for implementations of mathematical expressions.

KEYWORDS

software binaries, information retrieval, mathematical expressions

1 INTRODUCTION

With the growth of mathematical computations as essential building blocks of scientific, analytical and mission-critical applications, use of math libraries such as Eigen [8] and Generic Math Template Library [9] have become ubiquitous. As such, the need for a search system that can fetch relevant libraries containing a given expression has become inevitable. We envision a system for searching mathematical expressions in software binaries. Mathematical expressions can be of different types. In this work, we focus on queries that are algebraic and transcendental expressions. Our system, as shown in Figure 1, takes a mathematical expression, represented using ContentML [4, 10] as input query and returns relevant binaries that implement this expression. For brevity reasons, we use **ME** in the rest of the paper to refer to mathematical expressions.

MSR '18, Gothenburg, Sweden

DOI: 10.1145/3196398.3196413

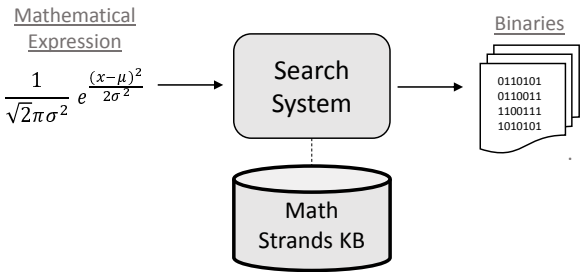

Figure 1: Overview of the search system.

A text search for "C++ Math Library" in StackOverflow[1] results in 4K posts. Developers often search for libraries that implement a certain mathematical expression. For example, a developer asks for a C library that implements Fast Fourier Transform in Quora[2]. We find 39K GitHub[3] projects which use the term "math" in their documentation. According to a study by Zhao et al. [23], users often look for resources such as code or a toolkit with an implementation of a **ME**. Hence, a search system for **ME** in binaries will be useful to developers. Such a search system can be used not only at development time for code reuse, but also be used by other stakeholders such as security analysts to locate vulnerabilities and software testers for bug detection.

Building such a search system is challenging due to several reasons: 1) Not every arithmetic opcode in the binary corresponds to the operator in **ME**. 2) The order in which the operands appear may differ between the binaries and the **ME**. 3) Difference in optimization levels and compilers may lead to different but semantically equivalent assembly forms. 4) Variables and structures in code are reduced to registers and memory locations in assembly. 5) There are multiple ways of representing **ME**. 6) We have a wide variety of operations such as algebraic and transcendental. We propose a simple yet novel fingerprinting approach to match **ME** with the software binaries.

Existing systems such as MathFind [17], Tangent-3 [22] and EgoMath2 [16] focus on the retrieval of documents pertaining to **ME**. In another work, Kamali and Tompa [12] also search for **ME** in documents. Current systems are limited to either searching in textual sources over the web [12, 15, 18], or discussion forums, or in tool documentation. To the best of our knowledge, this is the first work to search for **ME** in binaries.

[1] [Oct 2017] https://stackoverflow.com/
[2] https://www.quora.com/Are-there-any-libraries-in-C-to-implement-FFTs
[3] https://github.com/

Table 1: Expressions compiled with different optimization levels lead to different instructions in binaries.

Strand	O0	O2	Os
x^2	callq <pow@plt>	mulsd %xmm0,%xmm0	mulsd %xmm0,%xmm0
$x - y$	sub %eax,%edx	sub 0x200af2(%rip),%eax	sub 0x200af2(%rip),%eax
$\frac{x}{y}$	idiv %ecx	idivl 0x200af1(%rip),# 601180	idivl 0x200af1(%rip),# 601180

We compile 20 GitHub projects that contain one of the **ME** using three optimization levels, resulting in a total of 60 binaries for our dataset. To index these projects, first, we disassemble the binaries. Next, we create fingerprints of the disassembled binaries using our knowledge base. These fingerprints are textual by nature and hence can be indexed by a search engine. At query time, we extract and compare the query fingerprint with the binary fingerprints to compute a relevance score. Thus, our system is able to match expressions to binaries with a precision of 80% and recall of 53%.

The key contributions of our work are:

(1) A search system to search for **ME** in binaries.
(2) An approach to compare binaries and **ME**.
(3) A knowledge base of math operators mapped to their implementations in assembly opcodes.

Search in binaries is a less explored research area. While we focus on **ME**, our work opens up exciting opportunities in a variety of domains such as security, bug detection and binary clone detection where semantic analysis on binaries plays a significant role.

2 BACKGROUND AND TERMINOLOGY

In this Section, we provide the necessary background on dealing with **ME** in software binaries and list the associated challenges.

2.1 Working with binaries

Variants. Programs compiled using different compilers or with different optimization levels may result in dissimilar binaries. For example, in Table 1, a program implementing x^2 when compiled without optimization, calls the *pow* function, with binary signature *<pow@plt>*. Whereas, an optimized version uses *mulsd* to multiply the number to itself. We refer these choices available to compilers as *Variants*. The instructions may differ for different compiler optimization levels such as O0, O2 and Os (provided by GNU compiler collection and implemented in gcc and g++).

Ghost Ops. Not all instances of arithmetic opcodes in a binary provide insight about **ME**. Common actions such as passing arguments to a function on the stack and allocating memory make use of arithmetic opcodes too. We call such arithmetic opcodes, that do not have an explicit equivalent operator in source code, *Ghost Ops*. Hence isolating arithmetic opcodes with an equivalent operator in a mathematical expression is a challenge. For example, the presence of the *sub* instruction at assembly level need not imply that there exists a subtraction operation in the source code.

Evaluation Ordering. Compilation may result in a binary where strands of the implemented expression may appear in any order. For example, a compiler may evaluate $a*b+c/d$ as $a*b$ followed by c/d or c/d followed by $a*b$ before finally performing the addition operation. Due to this, the order of operations in **ME** differ from those that surface in the binaries. Hence, to compute similarity, a specific sequence of operations cannot be assumed. We refer to this challenge as *Evaluation Ordering*.

Operand Resolution. Since all operations at the assembly level are performed on registers or on values in memory, resolving the operands to variables is not a straightforward task. We call this challenge *Operand Resolution*. The resolution of operands plays a major role because if expressions were to be compared purely based on structure, the expression $b^2 - 4ac$ would be equal to $b^2 - 4ab$.

2.2 Working with ME

Specification of Expressions. Content MathML [14] (henceforth referred to as ContentML) provides a standardized way to capture **ME**. Yet, mathematical operators may have distinct forms. For example, $x * y$, $x \times y$ and xy represent the same expression. ContentML normalizes the representations, removing ambiguity.

Types of Operations. In this work, we focus on algebraic and transcendental expressions. Algebraic expressions are those which can be represented using only algebraic operations, which consists of addition, subtraction, multiplication and division. Transcendental expressions by contrast, are those expressions that cannot be represented by a finite sequence of algebraic operations. The operations which make up transcendental expressions include exponential, logarithm and trigonometric functions. This is a challenge because we need to consider all the diverse ways of representing these operations and functions at the binary level. For example, in an unoptimized version *log* is represented by *<log@plt>*, while in an optimized version it may get replaced by some precomputed value.

3 APPROACH

Our approach to implement a search system is aimed at addressing the challenges listed in Section 2. The approach is illustrated in Figure 2. It consists of the following components:

- Math Strands Knowledge Base
- Binary Fingerprint Generator
- Mathematical Expression Fingerprint Generator
- Fuzzy Match Scorer

We now describe these components, explain the design rationale, and connect them to the challenges listed in Section 2 of dealing with binaries and **ME**.

Math Strands Knowledge Base. A *math strand* is an individual operator in **ME**. We assume that two expressions are equivalent if they have the same structure. For instance, $x + y + z$ is same as $a + b + c$. Hence, we ignore the variables in **ME**. We define the structure of an expression as the sequence of operators arranged according to their precedence in **ME**. We have chosen to use the term strand instead of operators since some terms (such as e^x) are also of interest to us as they have direct implementations in libraries and therefore, a unique identity in the binary. For example, all programs that want to use the e^x function use the *exp*() function provided in standard math libraries that is uniquely represented by *<exp@plt>* in binary. Figure 2 shows a few rows from the math strands knowledge base.

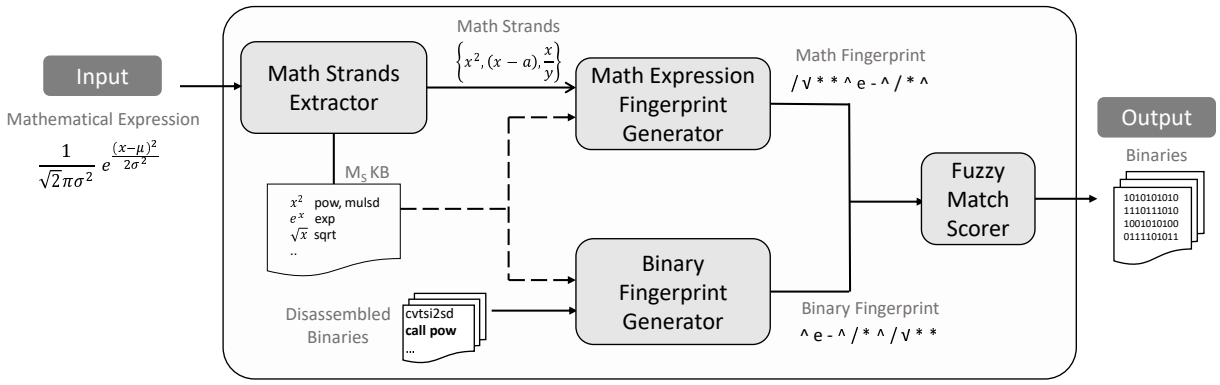

Figure 2: Our approach for searching binaries that implement a given mathematical expression.

The knowledge base is a mapping of top-k operators used in **ME**, and their corresponding assembly equivalents. Since the assembly equivalents for mathematical operators may be either a single instruction or a set of instructions or function calls to common math libraries, we address the problem posed by the type of operations. We refer to the top-k operators as *operators of interest*. We use the NTCIR-11 Math-2 dataset [20], which consists of 592,443 math expressions in Presentation MathML [4] format, to identify the most frequent operators. We use WIRIS[4] web service to convert them to ContentML, count the operators, and rank them by frequency. ContentML has standard tags for representing mathematical operations and hence deals with the challenge presented by variants in specifying **ME**. For each operator in the list thus created, we manually list all possible assembly language equivalents that could perform them. We call this mapping the Math Strands Knowledge Base (M_s KB). For this work, we pick the 15 most frequent operators, spanning both algebraic and transcendental classes, from the list of all operators found in NTCIR-11 dataset.

Binary Fingerprint Generator. To generate the binary fingerprint, we first disassemble the binary using the *objdump* tool from the GNU binutils[5]. The returned dump is segmented into function definitions and sections. We leverage this to extract each function, and generate a *math fingerprint* for each of them by passing it through a sieve constructed with the mapping from (M_s KB). The motivation for this is the assumption that an expression would be implemented in a single function for modularity. We reverse look-up each assembly instruction on the M_s KB, and if found, we add the strand corresponding to this instruction in M_s KB to an output string, which we call the math fingerprint of binary (B_{fp}). The output of the sieve is a binary fingerprint representing the concatenation of math strands.

Mathematical Expression Fingerprint Generator. We represent the expression fingerprint as a concatenation of math strands corresponding to the mathematical operators in **ME**, from M_s KB . We parse the ContentML input query to retrieve the *operators of interest* in their order of precedence in **ME**. The order of evaluation of any two operations in an expression cannot be swapped, unless the operations are independent of each other. This implies the presence of a partial order which, for preliminary analysis, is approximated by ordering the operators according to precedence. We then create the math fingerprint by replacing the resulting sequence of operators with the corresponding math strand from the M_s KB. We call this the mathematical expression fingerprint (M_{fp}).

Fuzzy Match Scorer. Our final step involves comparing B_{fp} with M_{fp}. This is non-trivial due to two factors: *Evaluation Ordering* and *Ghost Ops*. We employ the Longest Common Subsequence (LCS) [1] matching algorithm where the shorter M_{fp} is expected to be a subsequence of the B_{fp} (due to the presence of Ghost Ops). To convert the LCS score to a binary result indicating the presence or absence of the expression in the binary, we use a threshold σ. Therefore, if LCS(M_{fp}, B_{fp}) > σ, we declare that the **ME** was found in the binary and include it in the output set. To improve precision we have to deal with ghost ops and length normalization. So, we use two heuristics:

- *Length Heuristic*: If $|M_{fp}| < \alpha|B_{fp}|$, we ignore the LCS score and declare that the **ME** was not found in the binary. The main purpose of this rule is to enhance the precision for short **ME**.
- *Relevance Heuristic*: We count the number of strands (Irr(B_{fp}, M_{fp})) in B_{fp} that are absent in M_{fp}. We drop the binary if Irr(B_{fp}, M_{fp}) > $\beta|B_{fp}|$.

4 EVALUATION

4.1 Dataset Description

We need a dataset of binaries that contain at least one math expression each. Towards this purpose, we explore the GitHub projects, for the **ME** listed in Table 2. We convert the **ME** to ContentML using Visual Math Editor[6] which is a publicly available editor.

Selection of Expressions. In this work, we focus on expressions whose fundamental building blocks are generally available in the in-built language features of high-level languages such as C, C++ and Java. We select four **ME** which are listed in Table 2.

[4]http://www.wiris.com/editor/docs/content-MathML
[5]https://www.gnu.org/software/binutils/
[6]http://visualmatheditor.equatheque.net/

Table 2: The expressions and sample binaries that implement them which we use in our evaluation.

Name	Expression	Type	Git Author/Repository	Description
Heron's Formula	$\sqrt{s*(s-a)*(s-b)*(s-c)}$	Algebraic	SchraJa/Shape	Calculates area and perimeter
Sigmoid Function	$\frac{1}{1+e^{-x}}$	Transcendental	WayneTran/nn	Experiments with neural networks
Compound Interest	$K_n = K_0 * (1 + \frac{p}{100})^n$	Algebraic	Myerly/Interest	Calculates all forms of interest
D1-Black-Scholes	$\frac{\log \frac{S_0}{X} + t(r - q + \frac{\sigma^2}{2})}{\sigma\sqrt{t}}$	Transcendental	RyanKennedyio/black-scholes-cpp	Exercise in Black-Scholes formula for option pricing

Table 3: Precision, Recall and F_1 Score for each expression.

ME	M_{fp}	Precision	Recall	F_1 Score
Heron's Formula	$-*-*-*q$	0.69	0.56	0.61
Sigmoid Function	e+/	1.00	0.54	0.70
Compound Interest	/+^*	0.50	0.62	0.56
D1-Black-Scholes	/l^/+-*+q*/	1.00	0.38	0.55
Average		0.80	0.53	0.61

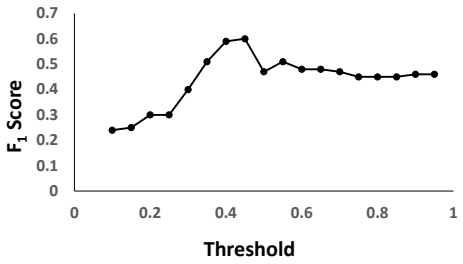

Figure 3: A threshold of 0.45 works best for this dataset.

Selection of Projects. We pick 20 projects from GitHub on disparate domains such as Finance, Algebra and Machine Learning. For each expression listed in Table 2, we pick five projects from GitHub such that each project implements the expression of interest. We choose libraries from multiple authors. While choosing implementations of the corresponding expressions on GitHub, we ensure that our methods are not biased towards a particular tool, release or community.

4.2 Results

We use a dataset of 60 binaries corresponding to 5 implementations for each of the expressions listed in Table 2. We compile them with the optimization levels O0, O2 and Os. Table 3 gives the results of our experiments. Here, M_{fp} shows the sequence of operators. For brevity, we use q for square root, e for exponent and l for logarithm. For each of these expressions and over the complete dataset of 60 binaries, we check if we retrieve the correct binary (precision) and all the correct binaries (recall). We compare our results against manually annotated ground truth for various values of threshold. Figure 3 shows that the F_1 score peaks for a threshold value, $\sigma = 0.45$. Further, we derive the parameters α and β empirically to be 0.25 and 0.40 respectively. We find that our approach is able to identify with an average precision of 80% and recall of 53% ($F_1 = 61\%$) at a threshold value of $\sigma = 0.45$. The evaluation ordering of instructions accounts for a low value of recall.

5 RELATED WORK

The work closest to ours is of Kamali and Tompa [12]. They present a **ME** search engine which represents the structure of mathematical expressions as trees and matches based on tree edit distance. This approach does not work with binaries due to the challenges such as *Ghost Ops* and *Evaluation Ordering*. Our approach is inspired by the simplicity and effectiveness of the fingerprinting approach as demonstrated in the existing works [2, 3, 11] on binary analysis.

Decompiling binaries is not always an option for mainly three reasons: 1) A decompiler may not exist [21] for the combination of programming language, compiler version, optimization levels, and OS architecture. 2) Decompilers do not always result in error-free code which can be recompiled for dynamic analysis. Even static analysis may not be possible in cases where type information is required. 3) Decompilers are not bug-free and we become dependent on the correctness of the decompiler.

BinGo [5] is a binary search engine which works across architectures and various compiler optimizations. It claims that relevance is higher for optimized code (compared to O0 non-optimized level). They have also modeled functions (not math, but code procedures) in a structure agnostic way suitable for binary search. We leverage these ideas in our work. There are several other binary search or similarity systems [6, 7, 13, 19] tuned to various purposes. However, none of them retrieve **ME**.

6 CONCLUSION AND FUTURE WORK

Mathematical expressions (**ME**) and binaries pose several hard problems such as *types of operations* and *variants*. To solve these problems, we use a data driven approach supported by a Math Strands Knowledge Base (M_s KB). With a simple fingerprinting based indexing, we show that **ME** can be located in binaries with 61% F_1 score for algebraic and transcendental expressions.

We envision automating the creation of M_s KB for multiple system architectures. Apart from the classes of operations considered, there are other classes of operations such as logical (&& for AND, $\rightarrow$ for implies), and relational (such as $\leq$). Summation ($\sum$) and product ($\prod$) are examples of *iterative operations* that require applying an expression over a range of values. Precision can be improved by keeping track of the operands in the **ME**. We will address these in our future work. Our work opens up a wide range of opportunities to attack problems on searching domain specific (such as music, medical and finance) content in binaries. We find that knowledge base assisted solution is promising to address such problems.

7 ACKNOWLEDGMENT

The authors are supported by Visvesvaraya and PM fellowships.

REFERENCES

[1] L. Bergroth, H. Hakonen, and T. Raita. 2000. A survey of longest common subsequence algorithms. In *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000.* 39–48. DOI: http://dx.doi.org/10.1109/SPIRE.2000.878178

[2] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. 2007. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium (SS'07).* USENIX Association, Berkeley, CA, USA, Article 15, 16 pages. http://dl.acm.org/citation.cfm?id=1362903.1362918

[3] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. 2006. Towards Automatic Generation of Vulnerability-Based Signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (SP '06).* IEEE Computer Society, Washington, DC, USA, 2–16. DOI: http://dx.doi.org/10.1109/SP.2006.41

[4] O Caprotti and D Carlisle. 1999. OpenMath and MathML: semantic markup for mathematics. *Crossroads* 6, 2 (1999), 11–14.

[5] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: Cross-architecture cross-OS Binary Search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016).* 678–689.

[6] Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of Binaries Through Re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017).* 79–94.

[7] Yaniv David and Eran Yahav. 2014. Tracelet-based Code Search in Executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14).* 349–360.

[8] Eigen. 2018. http://eigen.tuxfamily.org/. Last accesed: 23-Jan-2018. (2018).

[9] GMTL. 2018. http://ggt.sourceforge.net/html/main.html. Last accesed: 23-Jan-2018. (2018).

[10] Patrick DF Ion. 1999. MathML: A key to math on the Web. *Mathematical Reviews, PO Box* 8604 (1999).

[11] Jiyong Jang, David Brumley, and Shobha Venkataraman. 2011. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11).* ACM, New York, NY, USA, 309–320. DOI: http://dx.doi.org/10.1145/2046707.2046742

[12] Shahab Kamali and Frank Wm. Tompa. 2013. Retrieving Documents with Mathematical Content. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '13).* 353–362.

[13] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based Obfuscation-resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014).* 389–400.

[14] Content Markup. 2018. https://www.w3.org/TR/MathML3/chapter4.html. Last accesed: 23-Jan-2018. (2018).

[15] Robert Miner and Rajesh Munavalli. 2007. An approach to mathematical search through query formulation and data normalization. In *Towards Mechanized Mathematical Assistants.* Springer, 342–355.

[16] Jozef Mišutka and Leo Galamboš. 2011. System Description: EgoMath2 As a Tool for Mathematical Searching on Wikipedia.Org. In *Proceedings of the 18th Calculemus and 10th International Conference on Intelligent Computer Mathematics (MKM'11).* Springer-Verlag, Berlin, Heidelberg, 307–309.

[17] Rajesh Munavalli and Robert Miner. 2006. Mathfind: a math-aware search engine. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval.* ACM, 735–735.

[18] Tam T. Nguyen, Kuiyu Chang, and Siu Cheung Hui. 2012. A Math-aware Search Engine for Math Question Answering System. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM '12).* 724–733.

[19] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. 2009. Detecting Code Clones in Binary Executables. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA '09).* 117–128.

[20] Moritz Schubotz, Abdou Youssef, Volker Markl, and Howard S. Cohl. 2015. Challenges of Mathematical Information Retrievalin the NTCIR-11 Math Wikipedia Task. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '15).* 951–954.

[21] Katerina Troshina, Alexander Chernov, and Yegor Derevenets. 2009. C decompilation: Is it possible. In *Proceedings of International Workshop on Program Understanding, Altai Mountains, Russia.* 18–27.

[22] Richard Zanibbi, Kenny Davila, Andrew Kane, and Frank Wm. Tompa. 2016. Multi-Stage Math Formula Search: Using Appearance-Based Similarity Metrics at Scale. In *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '16).* 145–154.

[23] Jin Zhao, Min-Yen Kan, and Yin Leng Theng. 2008. Math information retrieval: user requirements and prototype implementation. In *Proceedings of the 8th ACM/IEEE-CS joint conference on Digital libraries.* ACM, 187–196.