AFRL-RI-RS-TR-2019-072

DERIVATION MINER

KESTREL INSTITUTE

MARCH 2019

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2019-072 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
STEVEN DRAGER
Work Unit Manager

/ S /
QING WU
Technical Advisor, Computing
& Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MARCH 2019	FINAL TECHNICAL REPORT	OCT 2014 – OCT 2018

4. TITLE AND SUBTITLE
DERIVATION MINER

5a. CONTRACT NUMBER FA8750-15-C-0007
5b. GRANT NUMBER N/A
5c. PROGRAM ELEMENT NUMBER 61101E

6. AUTHOR(S)
Eric Smith, Alessandro Coglio, Eric McCarthy, Henny Sipma

5d. PROJECT NUMBER MUSE
5e. TASK NUMBER CK
5f. WORK UNIT NUMBER ES

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Kestrel Institute
3260 Hillview Avenue Palo
Alto, CA 94304

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

DARPA
675 North Randolph St.
Arlington, VA 22203-2114

10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2019-072

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report describes Derivation Miner, an effort under the DARPA Mining and Understanding of Software Enclaves (MUSE) program. The research team included Kestrel Institute, Kestrel Technology, UT-Austin, and Qadium. The team investigated the question of whether large online repositories of open source computer code could be used to assist program synthesis in such a way that the result has high assurance. The results of the effort show that this is indeed possible. A variety of indexing, analysis, and search tools have been developed and it has been demonstrated, in many cases, that these tools can be used to identify code in the MUSE corpus that has the desired functionality. Formal proof tools have also been developed, including the Automated Program Transformation (APT) and Axe toolkits, which have been demonstrated over many examples, verifying code found “in the wild” can be used to synthesize proven programs.

15. SUBJECT TERMS
MUSE, code corpus, analysis by synthesis, APT, Axe, ACL2, Big Code, code search, software similarity, formal verification

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON	19b. TELEPHONE NUMBER (*Include area code*)
U	U	U	UU	44	**STEVEN DRAGER**	**N/A**

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

TABLE OF CONTENTS

LIST OF FIGURES ..... **iii**

1. SUMMARY **1**

2. INTRODUCTION **2**

2.1. BACKGROUND 2
2.1.1. Software Synthesis *2*
2.1.2. The ACL2 Theorem Prover *2*
2.1.3. The CodeHawk Static Analyzer *3*
2.2. RESEARCH QUESTIONS 3

3. METHODS, ASSUMPTIONS, AND PROCEDURES **4**

3.1. ORGANIZING CORPUS CODE 4
3.1.1. Measuring the Corpus Size. *4*
3.1.2. Unpacking the Corpus. *5*
3.1.3. Converting Filenames from Old Encodings to Unicode. *5*
3.1.4. Deduplicating Jar Files and Class Files. *6*
3.2. ANALYZING CORPUS CODE 6
3.2.1. Feature Extraction. *6*
3.3. QUERYING CODE AND ANALYSIS RESULTS IN A DATABASE 8
3.3.1. Titan Graph Database Schema. *8*
3.3.2. A Tool to Load Project, Class and Method Information into the Database *9*
3.3.3. Tools to Load Analysis Results into the Database. *9*
3.3.4. Documentation on Gremlin and the Show Soures Tool *9*
3.4. APPLYING MACHINE LEARNING TO ANALYSIS RESULTS 9
3.4.1. Creating the Sparse Matrix Form. *10*
3.4.2. A Tool to Deduplicate Feature Vectors. *10*
3.4.3. A Tool for Nonnegative Matrix Factorization Clustering. *10*
3.5. FINDING SIMILAR CODE 11
3.5.1. Finding Similar Code Based on Clustering. *11*
3.5.2. Finding Similar Code Based on TF-IDF. *11*
3.6. THE ACL2 THEOREM PROVER 21
3.7. THE AXE REWRITER AND LIFTER 22
3.8. THE AXE EQUIVALENCE CHECKER 23
3.9. THE APT (AUTOMATED PROGRAM TRANSFORMATIONS) TOOLKIT 24
3.9.1. Some Example Transformations. *25*
3.9.2. Publication and Availability. *26*
3.10. ACL2 IN JAVA, AND ACL2 TO JAVA 26
3.10.1. AIJ: The Deep Embedding. *27*
3.10.2. ATJ: The Code Generator. *28*
3.10.3. Publication and Availability. *29*

4. RESULTS AND DISCUSSION **30**

4.1. BRESENHAM EXAMPLE 30
4.1.1. Specification. *30*
4.1.2. Code Search. *30*
4.1.3. Invariant Generation. *31*
4.1.4. Code Lifting. *31*
4.1.5. Top-Down Derivation Steps. *31*
4.1.6. Bottom-Up Derivation Steps. *32*
4.2. MICRO AIR VEHICLE LINK (MAVLINK) EXAMPLE 32

4.2.1. Problem Statement. *32*
4.2.2. MAVLink Specification. *33*
4.2.3. Corpus Code. *33*
4.2.4. Glue Code Generation. *33*
4.2.5. Formal Proof Generation. *33*
4.2.6. Size of the Synthesized Artifacts. *34*
4.3. BITCOIN PUBLIC KEY TO ADDRESS EXAMPLE 34

5. CONCLUSIONS 36

6. REFERENCES 37

7. LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS 38

LIST OF FIGURES

Figure 1. Json File That Encodes The Query For Factorial 12
Figure 2. Relative Weights For The Search Terms For Factorial 12
Figure 3. Ranked Results For The Semantic Search For Factorial 13
Figure 4. Sizes Of The Vocabulary For Different Feature Sets For A Corpus Of 7 Million Documents (Methods) 15
Figure 5. Tf-Idf Score Of D For T 16
Figure 6. Tf-Idf Score Of D For Q 16
Figure 7. Cosine Similarity 16
Figure 8. Relative Weights For The Search Terms For Murmurhash 20
Figure 9. Ranked Results For The Semantic Search For Murmurhash 20
Figure 10. Relative Weights For The Search Terms For Ripemd-160 21
Figure 11. Ranked Results For The Semantic Search For Ripemd-160 21
Figure 12. Bitcoin's Public Key To Address Computation 35

1. SUMMARY

As part of the Defense Advanced Research Projects Agency (DARPA) Mining and Understanding Software Enclaves (MUSE) project, the Derivation Miner team, including Kestrel Institute, Kestrel Technology (KT), the University of Texas at Austin, and Qadium, investigated the question of whether large online repositories of open source computer code could be used to assist program synthesis in such a way that the result has high assurance. Our results show that this is indeed possible. Our work focused on two sub-questions. First, can we search large code corpora and find code that performs specific desired functionality? We developed a variety of indexing, analysis, and search tools and showed that, in many cases, we can use them to identify code in the MUSE corpus that has the desired functionality. Second, can we then verify the found code and incorporate it into larger programs being synthesized, in such a way that the resulting programs can be proved correct? We developed tools, including our Automated Program Transformations (APT) toolkit and our Axe toolkit, to accomplish this, and demonstrated that, for many examples, we can indeed verify code found "in the wild" and use it to synthesize proven programs.

2. INTRODUCTION

In recent years, large repositories of open source code have become available on the web, on sites such as Github. Such "Big Code" corpora represent many person-decades of work and thus may be very valuable resources to programmers. We set out to answer the question of whether such corpora can be useful in the process of *software synthesis*, in particular when a formal, mathematical proof of the synthesized code is desired.

2.1. Background

2.1.1. Software Synthesis

In stepwise refinement, a program is derived from a formal specification via a sequence of intermediate specifications. Several refinement notions and formalisms and tools exist. In our approach, a *derivation* is a sequence $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n \rightarrow p$, where the specification s_0 captures requirements, s_1 ,…, s_n are intermediate specifications, p is the program that implements the specification, and $\rightarrow$ represents a formal refinement relation – the last step typically involves a code generator that compiles the suitably refined executable specification s_n into a standard programming language like C or Java.

Each derivation step represents a design decision, e.g., the choice of a data structure or algorithm, or the application of a particular optimization. As different design decisions may be taken at each stage, a derivation is one path in a tree whose root and other non-leaf nodes are specifications, whose branches are refinement steps, and whose leaf nodes are implementations. All the leaves are implementations of the root specification; they use different algorithms, data structures, library functions, etc. In practice, the tree may be a graph, as different paths may lead to identical nodes, e.g., if two or more "orthogonal" transformations are applied in different orders.

Kestrel's research has focused on refinement-based program synthesis, where refinement steps are carried out via *automated transformations*. That is, given s_i, instead of writing down s_{i+1} and proving $s_i \rightarrow s_{i+1}$ ('posit and prove'), an automated transformation is applied to s_i to generate both s_{i+1} and a formal proof of the refinement $s_i \rightarrow s_{i+1}$ ('correct by construction'). Applying a transformation may require proving suitable applicability conditions, but these proofs are generally simpler than proving the top-level refinement relation $s_i \rightarrow s_{i+1}$; in other words, transformations help reduce complex proof tasks to simpler proof tasks in a principled way.

When specifications are restricted to particular domains, the application of transformations (and, in particular, the proofs of the applicability conditions) can often be made completely automatic. This results in a domain-specific software synthesis tool that works like a compiler and is usable by people without a background in formal methods, program refinement, or program transformation.

2.1.2. The ACL2 Theorem Prover

A Computational Logic for Applicative Common Lisp (ACL2) is a state-of-the-art, industrial-strength theorem prover, developed by our teammates at the University of Teaxs at Austin. ACL2 combines powerful proof procedures with a sophisticated environment to develop not only

formal theories and proofs, but also efficient programs and meta-programs. ACL2 has been used in a variety of applications, in particular hardware verification, but it is increasingly used in software verification as well.

2.1.3. The CodeHawk Static Analyzer

Kestrel Technology's CodeHawk analyzer performs static analysis of C programs, Java (bytecode) programs, and x86 binaries. It aims to perform sound analysis and is based on abstract interpretation. In this project, we used CodeHawk to extract features for machine learning.

2.2. Research Questions

In this project, we investigated two main research questions:

1. Can we search large code corpora and find code that performs specific desired functionality?
2. Can we then verify the found code and incorporate it into larger programs being synthesized, in such a way that the resulting programs can be proved correct?

The first question was initially posed by the MUSE program announcement, and was therefore investigated by the other teams in the MUSE program as well. The second question was instead unique to our team and approach. It addresses the issue of whether code found in the wild can be *safely* used in a development. There are obvious risks in including dubious code into one's own development. Our approach to mitigate these risks is to formally prove that the code has the desired functionality. In this project, we focused on functional requirements, but our approach can also address non-functional requirements, security requirements (e.g., the code does what it is expected to do, and critically nothing else), and so on.

3. METHODS, ASSUMPTIONS, AND PROCEDURES

The Kestrel team's research for the MUSE program focused on safely reusing Java code from the corpus. Specifically, we analyzed Java bytecode present in the corpus. The tools we developed can be divided into two groups: corpus-related tools and proof-related tools. The corpus-related tools include tools for:

(1) organizing corpus code;

(2) analyzing corpus code;

(3) querying code analysis results in a database;

(4) applying machine learning to analysis results; and

(5) finding similar code based on machine learning results

The proof-related tools include:

(1) the underlying ACL2 theorem prover, upon which our proof tools are built

(2) the Axe Rewriter, for applying simplification rules

(3) the Axe Lifter, for lifting code into logic

(4) the Axe Equivalence Checker, for proving equivalence of programs and specifications

(5) the APT toolkit, for constructing derivations linking code and specifications and for synthesizing code from specifications

(6) the ACL2 in Java (AIJ), and ACL2 to Java (ATJ) tools, for generating Java code from ACL2 code.

(7) Various formal specifications and supporting libraries of definitions and theorems.

The rest of this section describes the major tools that we developed and improved over the course of our research, motivated by the challenge problems.

3.1. Organizing Corpus Code

For organizing corpus code, Kestrel has developed tools for:

(a) measuring the corpus size, as total expanded size, and as numbers and sizes of Java-related items

(b) unpacking the corpus in a fault-tolerant way

(c) removing non-Java-related files

(d) converting filenames from old encodings to Unicode, and

(e) deduplicating jar files and class files in the file system while retaining origin information

3.1.1. Measuring the Corpus Size.

The corpus started out as projects on open source sites such as Github, SourceForge, Google Code, and the University of California at Irvine (UCI) Maven repositories. Leidos and other

performers selected C/C++ and Java projects, assigned a Universally unique identifier (UUID) to each project, attempted to do builds, and packaged the sources and builds into various tarballs.

When we (Kestrel Institute) were given access to the tarballs, each project could contain tarballs for metadata, code, UCI_build, C_build, and buildbot. We assigned corpus version numbers to specific lists of tarballs (along with their cksum hashes). The following discussion will refer to version V4_2 of the corpus unless otherwise specified.

To get an idea of the quantity of code that we needed to process, we wrote the script gather-tarball-java-data. For each tarball, this script records the tarball file size and cksum, the expanded size in the filesystem, and the numbers and size in bytes of java-related (java, class, and jar) files in the expanded tarball.

For MUSE Corpus version V4_2, there are 1,462,668 tarballs, taking 10.7 TB. The total expanded size of these tarballs is 23.0 TB. Kestrel's focus for the MUSE program was on Java projects. When we filtered out projects without Java content, the total expanded size of those tarballs was 7.1 TB.

Within the projects with Java content, there are many other files such as code in other languages, image files, etc. Just counting the Java-related files, there is a total of 1.3 TB of Java-related files in the expanded tarballs. 1.11 TB of that is in jar files, 182 GB is in java source files, and 16 GB in separate class files. Note: these numbers include a lot of duplication, and the jar files can contain non-java-related files.

3.1.2. Unpacking the Corpus.

Unpacking the corpus, removing non-Java-related files, and deduplicating are all done in the script make-jcorpus. To fully unpack the Java-related corpus, we expanded the tarballs with Java-related content and removed the non-Java-related files, and expanded the jar files. All expansion was done after checking the archive files for directory traversal vulnerabilities (e.g., when the archive contained a file name starting with "../").

For expanding the jar files, we developed the program "jarextractor". Jar files are in a zip format, but there are some inconsistencies between versions of zip. jarextractor calls the Linux "unzip" program first, and if it gets an error, tries to use "7za". Any errors are recorded in the filesystem parallel to the jar file, and the expanded jar is also recorded parallel to the jar file. Within the expanded jar, non-Java-related files are removed and recursive jars expanded, until no jars remain.

3.1.3. Converting Filenames from Old Encodings to Unicode.

Some projects in the MUSE corpus were created prior to widespread use of UTF-8 for file names. Since we have many downstream tools that operate on file names, we wanted to make sure those file names were valid Unicode strings. We identified 206 different files whose names were not valid UTF-8.

Linux allows filenames that contain any character other than slash or null. There was one Brazilian project that had filenames in Code Page 850 (CP-850), and two other projects with filenames (and directories) in International Organization for Standardization (ISO) 8859-1. We

used "convmv" to convert these files and directories to UTF-8, as part of the "make-jcorpus" script.

3.1.4. Deduplicating Jar Files and Class Files.

After make-jcorpus expands a top-level jar file (including all nested jars), if there are no java source files in the expansion, then the jar file and its expansion are moved to "jarcentral" and renamed to the MD5 hash of the jar file, and replaced by symlinks in the original project directory. Parallel to the jar file and its expansion in jarcentral there is a file which we call the "original-files" file containing the original jar file name. When make-jcorpus sees another copy of this same top-level jar file, it does not need to expand it again. Instead, it replaces the jar file by a symlink to the same jar file in jarcentral, adds a directory symlink to the expanded jar in jarcentral, and adds the original file name to the "original-files" file in jarcentral. This allows any tool to look up a jar file by its MD5 hash and to find out all the projects that it was in.

In a similar way, the class files in the project directories and the class files in the expansions of jar files are deduplicated to "classcentral".

For MUSE corpus version V4_2, the size of jarcentral is 682 GB, and the size of classcentral is 280 GB.

gather-tarball-java-data and make-jcorpus have been made open source and published on Github under the Berkeley Software Distribution (BSD) 3-clause license. The documentation there goes into much more detail about the file organization and other aspects. See https://github.com/KestrelInstitute/big-code-corpus.

3.2. Analyzing Corpus Code

For analyzing corpus code, one simple step is to collect metadata information on each project; another step is to parse all the class files in classcentral using the ObjectWeb ASM library to gather a list of all the methods in the corpus.

The collect-project-metadata script scans all the project directories created by expanding the corpus tarballs, collects certain property-value pairs from the project-level Javascript Object Notation (JSON) metadata files, and outputs them to a summary file for later uploading project-level information into a graph database.

The collect-metadata script, besides calling collect-project-metadata, also parses all the class files in classcentral using the ObjectWeb ASM library to gather a list of all the classes and methods in the corpus. This list is used later for creating the class and method vertices in a graph database.

3.2.1. Feature Extraction.

Kestrel Technology developed a variety of analyses to extract features from corpus code, including features for loops, methods, and classes. Loop features currently include:

bc: bytecode frequencies of instructions within the loop, with conditional jumps that jump out of the loop distinguished from conditional jumps that stay within the loop;

bc-cat: bytecode frequencies grouped by category (arithmetic, comparison, control, etc);

libcalls: count of library calls made from within the loop

sizes/counts:

– number of instructions within the loop

– depth of the loop

– number of exits out of the loop

Features extracted from methods currently include:

bc: byte code frequencies of instructions within the method

bc2: frequencies of byte code pairs (two consecutive instructions)

bc3: frequencies of byte code triples (three consecutive instructions

libcalls: count of library calls made within the method

api-types: count of types of arguments and return value

op-types: frequency of types operated on by instructions

ksubgraphs: frequency of 4-subgraphs of the control flow graph

literals: count of integral and floating point literals used in the method

attrs: boolean (1/0) values for method attributes such as static, final, synchronized, etc.

sizes/counts:

- instrs: number of instructions in the method
- i-calls: number of interface calls made
- v-calls: number of virtual calls made
- loops: number of loops in the method
- args: number of arguments passed to the method
- max-depth: maximum depth of the loops in the method
- exn-handlers: number of exception handlers
- checked-exns: number of checked exceptions thrown by the method
- subgraphs: number of 4-subgraphs of the control flow graph
- cfg-edges: number of edges in the control flow graph
- cfg-nodes: number of nodes in the control flow graph
- complexity: control flow graph complexity (edges - nodes + 2)

Features extracted from classes are mostly an accumulation of method features and currently include:

bc: accumulated byte code frequencies over all methods

bc2: accumulated byte code pair frequencies over all methods

Approved for Public Release; Distribution Unlimited.

bc3: accumulated byte code triple frequencies over all methods

libcalls: accumulated library call counts over all methods

api-types: frequency of all method argument and return value types

op-types: frequency of all types operated on by instructions

ksubgraph: accumulated 4-subgraph frequencies over all methods

literals: accumulated integral and floating point literals over all methods

attrs: class attributes such as final, interface, immutable, dispatch, etc.

sizes/counts:

- methods: number of methods with byte code defined in the class
- native-methods: number of native methods defined in the class
- fields: number of fields defined in the class
- max-depth: maximum loop depth over all methods in the class
- loops: number of loops in all methods combined
- instrs: number of instructions in all methods combined
- subgraphs: number of distinct subgraphs in all methods combined
- max-complexity: maximum control flow graph complexity over all methods

The features for each artifact analyzed are output in Extensible Markup Language (XML) format for easy consumption by other tools. The features used for the method similarity tool based on Non-negative Matrix Factorization (NMF) clustering (described below) are the bytecode frequencies and the instrs, v-calls, i-calls, loops, max-depth, subgraphs, cfg-nodes, and complexity features for each method.

3.3. Querying Code and Analysis Results in a Database

For querying code analysis results in a database, Kestrel has developed

(a) a Titan graph database schema

(b) a tool to load project, class and method information into vertices and edges in the database

(c) tools to load analysis results into properties in the database

(d) documentation on how to use Gremlin to query the database, and

(e) a tool that relates query results back to Java source files in the file system

3.3.1. Titan Graph Database Schema.

TitanDB is a graph database that can use various storage backends and various search backends. (Since we started the project, the company that developed TitanDB was purchased by Aurelius and TitanDB was used to build the commercial product DataStax Enterprise Graph, while the original TitanDB was forked into JanusGraph, now a Linux Foundation project.) We use

Cassandra for the storage backend and Elasticsearch for the search backend. Titan also supports the Apache TinkerPop stack including the Gremlin graph query language.

The schema we developed defines Project, Class, and Method vertices along with various properties on them. The schema also defines edges connecting methods to their classes. Project vertices are primarily indexed by their UUIDs. Class vertices are primarily indexed by the MD5 hash of the class file. Method vertices are primarily indexed by "methodidx", which concatenates the classhash, method name, and method signature. In addition, there are Elasticsearch full text search indexes (for doing searches using regular expressions) for project name, method name, and method signature.

3.3.2. A Tool to Load Project, Class and Method Information into the Database.

After defining the schema, we run the "ingest-metadata" script to create the project, class, and method vertices in the Titan graph database.

3.3.3. Tools to Load Analysis Results into the Database.

The CodeHawk Java analyzer creates an XML output file for each class file, with the results of analysis (the features described above) for all methods in that class file.

To load these results into Titan, we developed the tool "ingest-features". Given a user-specifiable number N of processes, it organizes the XML files into N chunks and processes the chunks of files in parallel. The XML files are parsed to extract the analysis feature values we want, and then the feature values are ingested into the database.

In particular, for each method we load frequencies for the 256 Java bytecodes, along with 8 other features: number of instructions, number of virtual calls made, number of interface calls made, number of loops in the method, maximum depth of the loops in the method, number of 4-subgraphs of the control flow graph, number of nodes in the control flow graph, and control flow graph cyclomatic complexity.

For the MUSE corpus V4_2, there were 26,521,669 analysis results files processed, containing data for a total of 243,293,975 methods.

3.3.4. Documentation on Gremlin and the Show Soures Tool

Gremlin is a graph query language that has been implemented for many graph databases. However, it is not similar to Structured Query Language (SQL). Kestrel has developed notes on how to use Gremlin on our particular graphs, along with examples. We developed a tool, "show_sources" in Groovy Shellscript that can be loaded into a Gremlin interactive session that looks up query results (bytecode methods) in the filesystem to find possible matching source files.

3.4. Applying Machine Learning to Analysis Results

For applying machine learning to analysis results, Kestrel has developed

(a) a tool that assembles analysis results into sparse matrix form

(b) a tool that deduplicates feature vectors while retaining origin information

(c) a tool that uses Smallk NMF to reduce each unique feature vector to a shorter cluster vector

3.4.1. Creating the Sparse Matrix Form.

In order to convert the analysis data to matrix form, we developed the build-matrix tool. The analysis data is initially in the form of XML files generated by the CodeHawk analyzer, but the dimension-reduction tools we use expect Matrix Market Coordinate Format, which is a text format for matrices that is sparse in both dimensions. build-matrix spins up N processes to parse the XML files in parallel to generate N bare matrix files and then combines them into a single full sparse matrix file. build-matrix also creates row label (method index) and column label (feature index) files.

After building the matrix file, build-matrix normalizes the feature values (all of which are in a range [0 .. N]) by scaling them to the range [0 .. 1]. The maximum raw value for each feature is recorded in a file.

For the MUSE corpus V4_2, there were 26,521,669 analysis results files processed, containing data for a total of 243,293,975 methods. The resulting matrix has 3,122,724,156 nonzero feature values.

3.4.2. A Tool to Deduplicate Feature Vectors.

Many methods have identical feature vectors. This does not necessarily mean the methods are identical, since differences in the constant pool or in the order of bytecode instructions will not affect the feature vector. However, feature vector duplicates are not useful, and it slows down downstream tools. To deduplicate the feature vectors in the matrix, we developed the "remove-duplicate-documents" tool. This tool creates a new matrix file and a new row label (method index) file, as well as augmenting the old row label file with a new column that maps each old method to the representative new method with the same feature vector.

For the MUSE corpus V4_2, the 243,293,975 method vectors were deduplicated to 21,026,181 different method vectors. The 3,122,724,156 nonzero feature values in the original matrix were reduced to 507,343,064 nonzero feature values in the new matrix.

3.4.3. A Tool for Nonnegative Matrix Factorization Clustering.

Nonnegative Matrix Factorization is a class of dimension reduction methods similar to Principal Component Analysis, but resulting in nonnegative weights and derived features that can be more obvious in meaning. Kestrel Institute developed a Docker image that drives the open source Smallk "fuzzy clustering" implementation of NMF. The number of clusters is a parameter N that defaults to 32.

The result is a new matrix in dense Comma Separated Value (CSV) format with one row for each method and one column for each cluster. The cluster weights are all in the range [0 .. 1], and the cluster weights for a method sum to 1.0. The cluster weights can be viewed as a probability mass function. This also means that the shape of cluster space (the reduced data) is an N-dimensional simplex rather than an N-dimensional cube, so distance measures and nearest-neighbors calculations within are less impacted by the "curse of dimensionality".

3.5. Finding Similar Code

For finding similar code based on machine learning results, our team has developed

(a) a Nearest-Neighbors tool that shows Java methods with high similarity to a given query Java method, using the cluster vectors from NMF

(b) a tool based on term-frequency inverse document frequency (TF-IDF) for detecting similar code

3.5.1. Finding Similar Code Based on Clustering.

The similarity tool, "find-similar-methods", is a tool that, given either an existing method in the corpus or a point in cluster space, lists the closest N methods in cluster space, from nearest to farthest. The default distance metric is L1, the sum of the coordinate differences. The tool has parameters for restricting the results to those within a certain distance, for specifying the distance metric, and for limiting the number of results shown.

To improve efficiency for later calls, the first time the tool is called on a cluster matrix CSV file, it creates an equivalent array in a binary file that can be memory-mapped into the process for quick lookups.

The "find-similar-methods" tool has had extensive use finding interesting methods in the Corpus that would not have been findable using text-based methods. For example, methods for which there was no source code in the Corpus and/or whose names were obfuscated.

The similarity tool has been made open source and published on Github under the BSD 3-clause license. See https://github.com/KestrelInstitute/big-code-similarity

3.5.2. Finding Similar Code Based on TF-IDF.

Introduction: An Example

The primary purpose of the kt-semantic-code-search tool is to enable searching for an implementation of an algorithm for which we have some description of its operations. Let us start with a simple example. Suppose we need an iterative implementation of factorial. As a first approximation of the tests and operations involved in factorial we can take:

$$i < 0 \quad (1)$$

$$i := i \quad (2)$$

$$i := 1 \quad (3)$$

$$i := (i * i) \quad (4)$$

where i stands for any integer variable. The first term is a guard against a negative input value. The second and third terms indicate that the computation can start either with a variable (presumably the input value) or with the value 1, indicating a decrementing or incrementing computation, respectively (we leave out the increment/decrement operations themselves, as these have not been canonicalized at this point and therefore show too much variation). The last term indicates an assignment of the product of two variables.

Furthermore we want an integer implementation, so we specify a method signature of (I)I, which takes an integer as argument and returns an integer. Finally, we want an iterative implementation, so we specify the presence of a loop. From these six elements we construct a query, shown in Figure 1 (the feature set keys will be explained later),


```
{
   "branch-conditions-vmcfsi": {
      "i<0": 1
   },
   "method-assignments-vmcfsi" : {
       "i:=(i * i)": 1,
       "i:=i": 1,
       "i:=1": 1
   },
    "signatures" : {
      "(I)I": 1
    },
   "sizes": {
     "loops": 1
   }
}
```


Figure 1. JSON file that encodes the query for factorial

and submit it to the semantic-code-search tool with an indexed corpus of 7 million methods. The results are shown in Figure 2.

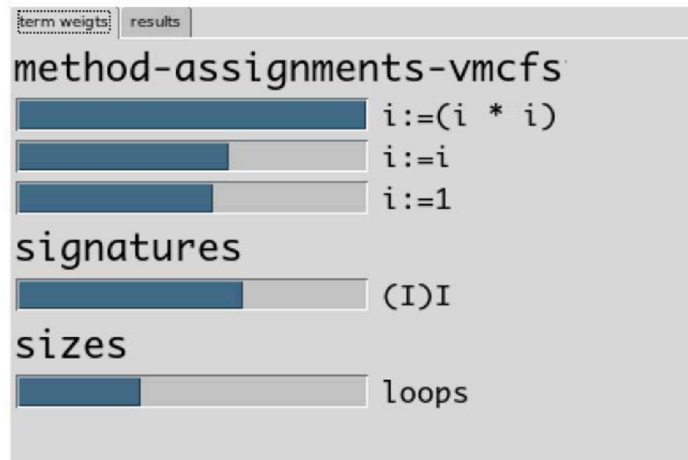

Figure 2. Relative weights for the search terms for factorial

Approved for Public Release; Distribution Unlimited.

Figure 3. Ranked results for the semantic search for factorial

Figure 2 shows the weights assigned to each of the six search terms: a higher weight is assigned to a term that is less common. Figure 2 shows that the assignment of the product has the highest weight, thus assigned because it is the least common term in the corpus, relative to the other terms used in the query; the presence of the loop contributes the least to the result, as loops are fairly common (they are present in roughly 8-10% of all methods).

Figure 3 shows the ranked results of the query. Figure 3 shows that the query is surprisingly specific, considering its simplicity: the first twelve results are all, judging by their method name, implementations of the factorial function. Of course, we do not know how many implementations of factorial were not reported, so we can say we have high precision, but unknown recall.

Document Indexing

The kt-semantic-code-search tool is based on the theory and practice of Information Retrieval, described by Christopher Manning and others, which was developed primarily for text retrieval. Below we describe how we adapt each of the stages and concepts described there to semantic code search. We closely follow the organization and description in Manning's book *Introduction to Information Retrieval* [6].

From Manning: "Information retrieval is finding documents without a clear, semantically overt structure that satisfy an information need from within large collections."

The information need is encoded in a query; a document from the collection is relevant if it contains information of value with respect to the need. The first task is to decide what terms can be used in a query and what constitutes a document. In the context of text retrieval the terms used in the query are usually words that correspond (not necessarily one-to-one) to words in the documents; documents are pieces of text, ranging from paragraphs to chapters or books, or other forms of text such as blogs or web pages. In the context of semantic code search, we choose Java methods as our document units and expressions (and a few other features) as our terms. Below we describe the four stages of constructing an inverted index to facilitate the search.

Document Collection: In the context of text retrieval, the documents are typically bytes in a file or on a web server, where the bytes correspond to American Standard Code for Information Interchange (ASCII) characters or perhaps Unicode points. In the context of code search the documents are the streams of bytes that constitute the Java class files (Java bytecode). In contrast with text retrieval, these sequences of bytes do not directly map to meaningful entities related to the information need. A class file can contain multiple methods (our chosen document unit), each of which consists of a sequence of byte code instructions. These instructions, however, refer to a shared constant pool, which adds to the meaning of the instructions and thus must also be considered part of the document. Even with the constant pool information included, byte code instructions are still too low level to directly map to reasonable query terms.

The purpose of the semantic code search is to search for implementations of algorithms for which we have a high-level description. Suitable terms in a query for such an algorithm are then the terms that appear in its high-level description, which are typically expressions that occur in assignments and expressions that occur in conditions (decisions). Thus we raise the level of interpretation of the bytecode from a sequence of single bytecode instructions to a sequence of assignments and branch conditions to match the terms likely to appear in a query.

Tokenization: In the context of text search, tokenization is the process of chopping character streams (that make up the documents) into tokens. In the context of code search, the tokens are the assignments and branch conditions that are implied by the bytecode. These assignments and branch conditions can span from one to more than a dozen bytecode instructions and need to be reconstructed from the bytecode.

Normalization: Manning defines token normalization as the process of canonicalizing tokens so that matches occur despite superficial differences in the character sequences of the tokens. It is typically achieved by creating equivalence classes of tokens that are then named after one member of the set. In the context of code search, there is a similar need for canonicalization. Whereas in text retrieval one may want to map the words "email" and "e-mail" both to the token `email`, in code search we may want to map the integer assignments a := a + 1 and b := b + 1 to the assignment i := i + 1, as variable names are irrelevant for the behavior of an algorithm. However, we may want to go beyond variable names in our normalization to reduce the size of the term vocabulary. For example, we could also normalize on method names and map the branch condition expressions a.isLower() and b.isHigher() to the token b.m(), with b standing for the generic object.

We have chosen to have the option of multiple levels of normalization where terms can be either fully explicit (apart from variable names) or where we reduce one or more components of the expression to the generic version of that component. The expression components that can thus be reduced are:

variable names (v): always reduced: integer variables are denoted by i, float and double variables are denoted by x, and object variables are denoted by b.

method names (m): can either be present explicitly or reduced to m.

class names (c): can either be present explicitly (with fully qualified name) or reduced to c.

field names (f): can either be present explicitly or reduced to f.

string literals (s): can either be present explicitly or reduced to s.

integer literals: can either be present explicitly or reduced to n, with the exception of 0 and 1, which will always be included explicitly.

We currently generate three different feature sets: v (which only reduces variable names), vmcfs (which reduces variable names, method names, class names, field names, and string literals), and vmcfsi (which reduces all components).

Stemming and Lemmatization: Part of the normalization process in text retrieval is to group families of words with similar meanings that have a shared core stem, for example algorithm and algorithmic. Stemming and lemmatization reduce inflectional forms and derivationally related forms to a common base form, where stemming is a rather crude form that chops off characters and lemmatization is a more informed approach based on language characteristics.

Semantic code search can benefit from a similar process based on algebraic equivalences such as i+1 = 1+i, reducing terms to a canonical form. The present version of our code search does not yet perform this form of canonicalization.

Indexing: Indexing is the process of creating a data structure that contains the relationship between documents and tokens present in those documents that facilitates efficient document retrieval based on a query. The most common data structure used in information retrieval is the *inverted index*. An inverted index keeps, for each term in the vocabulary, a list of documents in which that term appears (also called a posting list).

At the indexing stage, the semantic code search is mostly indistinguishable from text retrieval with the only difference being that we use multiple vocabularies, one for each expression reduction level, as described above (and a few more). Figure 4 gives an idea of the sizes of the different vocabularies for a corpus of just over 7 million documents.

vocabulary (feature set)	distinct terms
api-types	8
attrs	8
branch-conditions-v	1,495,108
branch-conditions-vmcfs	172,444
branch-conditions-vmcfsi	91,792
libcalls	726,924
libcalls-sig	801,299
literals	86,762
method-assignments-v	2,791,355
method-assignments-vmcfs	491,832
method-assignments-vmcfsi	266,823
signatures	462,967
sizes	8

Figure 4. Sizes of the vocabulary for different feature sets for a corpus of 7 million documents (methods)

Approved for Public Release; Distribution Unlimited.

Document Retrieval

In response to a query, the expectation is to receive a list of documents ranked by relevance to the query. Hence we need a mechanism to rank documents based on how well they match the terms in the query, that is, we need to compute a score for each document based on the query. Several such mechanisms have been developed and studied in the field of Information Retrieval. We choose to adopt one of those, named term-frequency – inverse frequency (TF-IDF) weighting. We briefly explain the motivation and the calculation, based on Manning's approach.

TF-IDF: Term frequency (tf) counts how often a term from the query appears in a document; the assumption is that if a term appears more frequently in a document, that document is more relevant than documents in which the term appears less frequently or not at all. Document frequency (df) counts how many documents in the collection contain at least one occurrence of a term in the query. The reason to use document frequency rather than collection frequency (cf), the total number of occurrences of the term in the document collection, is that document frequency is a more effective document discriminator: a term that appears once in every document does not have any discriminative power whereas a term that has the same number of occurrences, but all of them in the same document, has high discriminative power and thus should be given higher weight. The actual weighting in the computation of the score uses the inverse document frequency (idf): the higher the document frequency, the lower the discriminative power of that term.

The TF-IDF score of a document d for a given term t is thus given by

$$\text{tf-idf}_d(t) = \text{tf}_d(t) * \text{idf}_d(t)$$

Figure 5. TF-IDF score of d for t

and the score of a document d for a query q is the sum of the scores for each term:

$$\text{tf-idf}_d(q) = \sum_{t \in q} \text{tf-idf}_d(t)$$

Figure 6. TF-IDF score of d for q

Cosine Similarity: A problem with the TF-IDF score is that it is biased towards larger documents because of the term frequency factor. One way to reduce this effect is to represent each document as a vector in term space (one dimension per term in the query), where each element of the document vector is the TF-IDF score for the corresponding term: $v_d[t] = \text{tf-idf}_d(t)$. Two documents can then be compared using cosine similarity:

$$\text{cosine-similarity} = \frac{\vec{v}_{d1} \cdot \vec{v}_{d2}}{|\vec{v}_{d1}|\,|\vec{v}_{d2}|}$$

Figure 7. Cosine Similarity

The denominator provides a form of document length normalization: it cancels out the effect of larger term frequency for larger documents.

To compute the relevance of a document to a query we can simply reduce the vector space to the terms in the query, represent the query by the vector with all elements equal to 1, and compute the cosine similarity value between this vector and the document vector in the query term space. Note that documents that do not include all terms can still have a high score if some of the terms they contain have a very low document frequency (and thus a very high idf, and hence a high scaled TF-IDF value).

Implementation

The semantic code search implementation consists of three stand-alone components with artifacts passed between them via a file interface. We describe each of the components below.

Feature Extraction: The CodeHawk Java Analyzer is used to extract the features from java class files (bytecode). It reconstructs expressions from expression stack operations, canonicalizes the expressions, and classifies them as assignments, conditionals, and function arguments. The analyzer also extracts structural features such as loops and various other statistics about methods. All of these features are saved in textual form in XML files.

Feature extraction has to be performed only once and can be done off-line. Currently the CodeHawk Java Analyzer can process about 1 million methods per hour on a single processor. No interaction between class files, however, is present in the extraction of features, and thus this step can be parallelized over as many processors as are available (memory usage is low), or probably better, distributed over many different systems, as on a single system the file system may become a bottleneck. Computation time is essentially linear in the number of documents.

Feature Indexing: The goal of the second step, feature indexing, is to produce an index of all documents in the collection to enable efficient query processing. This step is implemented in python. It takes as input the XML files produced by the feature extraction step and creates the following files:

vocabulary files: for each feature set, indices are assigned to all terms in the feature set, captured in JSON files;

document cross-references: documents (java bytecode methods) are indexed by MD5 hash (to exclude duplicates) and cross-referenced to jar, package, class, method name, and method signature, captured in JSON files.

postings: for each package, for each feature set, a postings (JSON) file is created that maps document indices to a mapping of term indices to term frequency. These files are saved in a hierarchical directory structure according to package MD5.

package digest: for each feature set, a list of the terms present in each package is constructed to allow selective loading of postings files: only postings files belonging to packages with a non-empty intersection with the set of terms in the query need to be loaded. Depending on the query, this may reduce the amount of data to be loaded for a query by a factor of 20 or more.

All of these files are compressed into a jar file, which is then used as the input for query processing.

The construction of the index is incremental. New feature files can be integrated into an existing feature index, producing a new index file. This step, also performed off-line, cannot be easily parallelized because of the need to construct single indexes for all components Computation time

is slightly super-linear due to the effect of dictionaries increasing in size; the implementation, however, can still efficiently handle indexes of millions of methods (documents).

Query Processing: This step computes the response to a query, and is the only step performed on-line. It takes as input a query, consisting of a set of terms, each associated with a particular feature set, and a pre-computed index of the document collection. The query can be provided in the form of a JSON file, or the terms can be entered directly via command-line arguments.

This step produces a list of methods (identified by jar, class file, method name, and method signature) with their associated computed similarity score to the query.

This step is also implemented in python; it makes use of the python library sklearn for text feature extraction, part of the scikit-learn machine learning library, to compute the TF-IDF values and cosine-similarity scores. For an index of a million methods, response times vary from a few seconds for queries with only a few terms, to several minutes for queries containing several hundred terms.

Vocabulary Generalization and Scaling Up: The implementation described above limits the code search to a set of predefined feature sets that are determined by the feature extraction step. The reason is that the feature extraction step converts expressions to text strings and assigns these text strings to a particular feature set (e.g., conditional or assignment). This is attractive, as the downstream process can be treated more or less as a standard text retrieval procedure with feature sets playing the role of zones. The disadvantage, however, is that there is very little opportunity in the downstream process for customization, because the expression strings have lost their meaning as expressions: they have become the elements of a vocabulary in which each element only has its name, without any other associated properties. This is perfectly acceptable in text search where, in general, the search engine does not need to know that a table and a chair are both types of furniture. For code search, however, this is rather limiting, as the feature extraction essentially limits the types of queries that can be handled, because that step determines the feature set and the vocabulary. In particular, the feature extraction step was implemented to support the search for algorithms, given a high-level description of the algorithm. The features thus generated, however, do not very well support a search for other code patterns, e.g., the use of database access procedures or patterns of resource usage. Furthermore the CodeHawk Java Analyzer is proprietary, thus making it impossible for an end user to change the features extracted.

To lift this limitation, we decided to split the feature extraction step into two steps. The first step, performed by the CodeHawk Java Analyzer, extracts a large number of code constructs from the Java byte code, including expressions, control flow graph, method calls, arguments to method calls, but instead of saving these constructs as text, it saves them as indexed structured data, thus preserving the possibility to take them apart and recombine them later in any way necessary.

The second step, implemented in python, takes as input the structured pre-features saved by the CodeHawk Java Analyzer, and a python object that defines how these structured pre-features are to be transformed into features. The python object allows the user to define:

the feature sets, and

the vocabulary of the feature sets, and

for each indexed pre-feature, whether and how the pre-feature is to be transformed into a feature as part of one or more of the feature sets

This approach provides an end user much more flexibility to customize the search to a particular area of interest, e.g., SQL-queries, or use of measurement units, or resource-use patterns, etc. Example implementations of this transformation are provided for algorithms (recreating the features that were originally created directly by the CodeHawk Java Analyzer), for SQL-calls, and for resource consumption patterns.

The new approach also contributes to increased scalability. The original approach required all possible features of interest to be explicitly (as text) saved and indexed. The pre-features are much more concise, and the transformation step can be selective about what to include in the different feature sets when tailoring for a particular end-user information need, thus resulting in a smaller and likely more relevant index as a basis for the search.

The new python implementation including the transformers mentioned are provided open-source in the public GitHub repository `kestreltechnology/kt-semantic-code-search`.

Examples

We illustrate the code search on a few example algorithms for which we obtained a high-level description from the internet. For each we experimented by creating a few different queries to determine the terms that served best as discriminators. The results shown in Figures 8, 9, 10 and 11 are obtained from a corpus of about half a million methods that was constructed with a bias towards applications with high algorithmic content.

Murmurhash: An example implementation (in C) of MurmurHash can be found at https://en.wikipedia.org/wiki/MurmurHash.

Approved for Public Release; Distribution Unlimited.

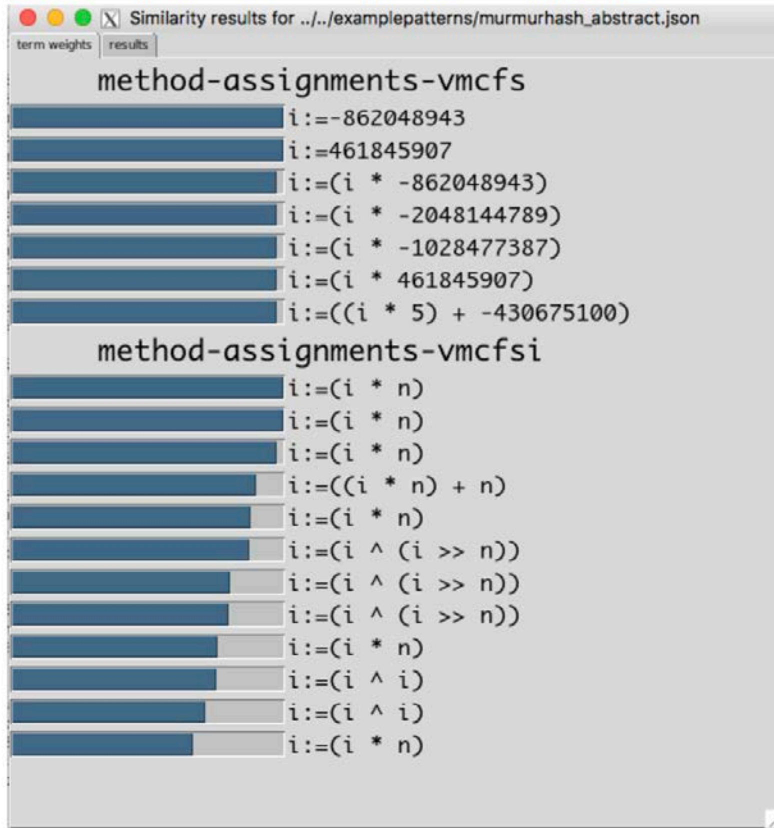

Figure 8. Relative weights for the search terms for murmurhash

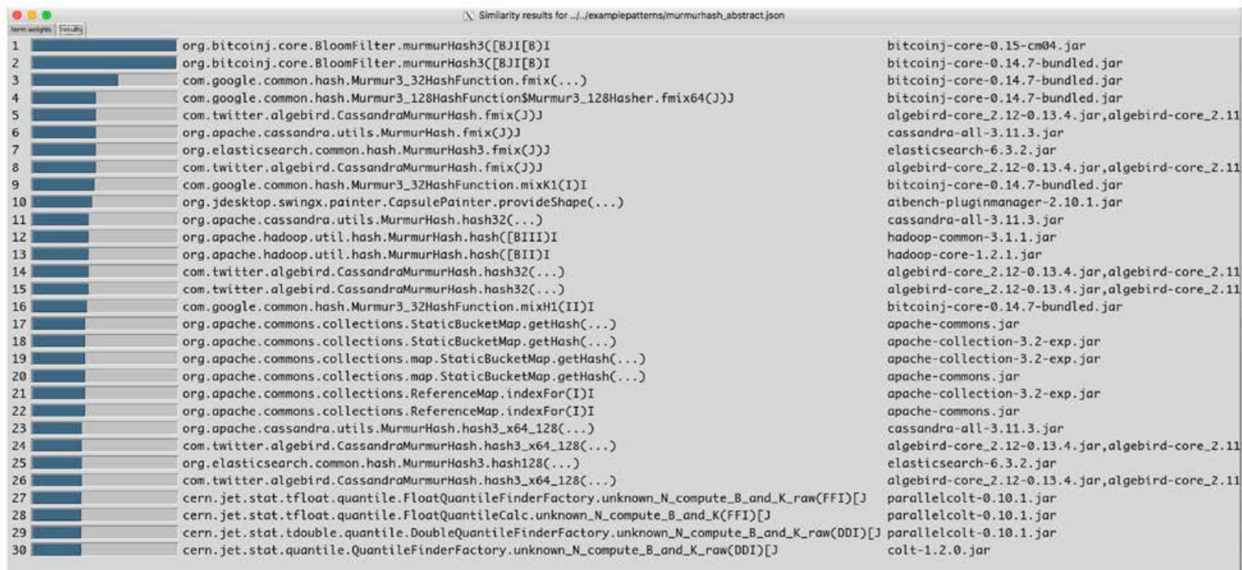

Figure 9. Ranked results for the semantic search for murmurhash

RIPEMD-160: A description of the RIPEMD-160 hash function is provided at https://en.bitcoin.it/wiki/RIPEMD-160.

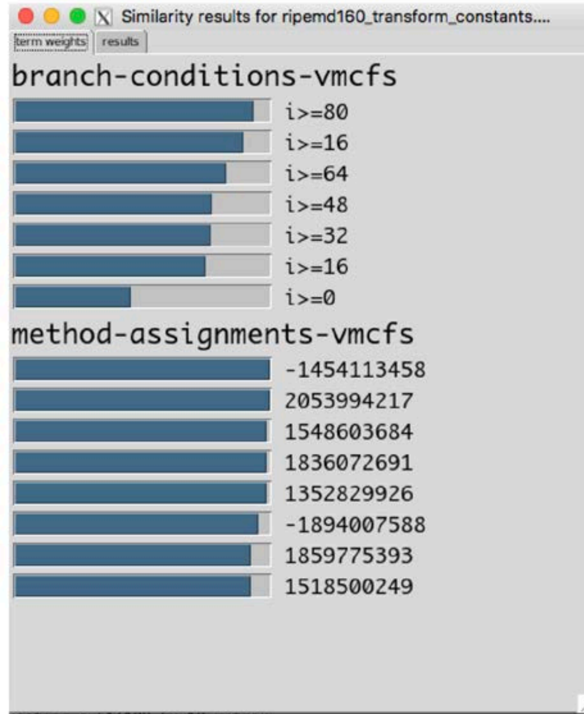

Figure 10. Relative weights for the search terms for RIPEMD-160

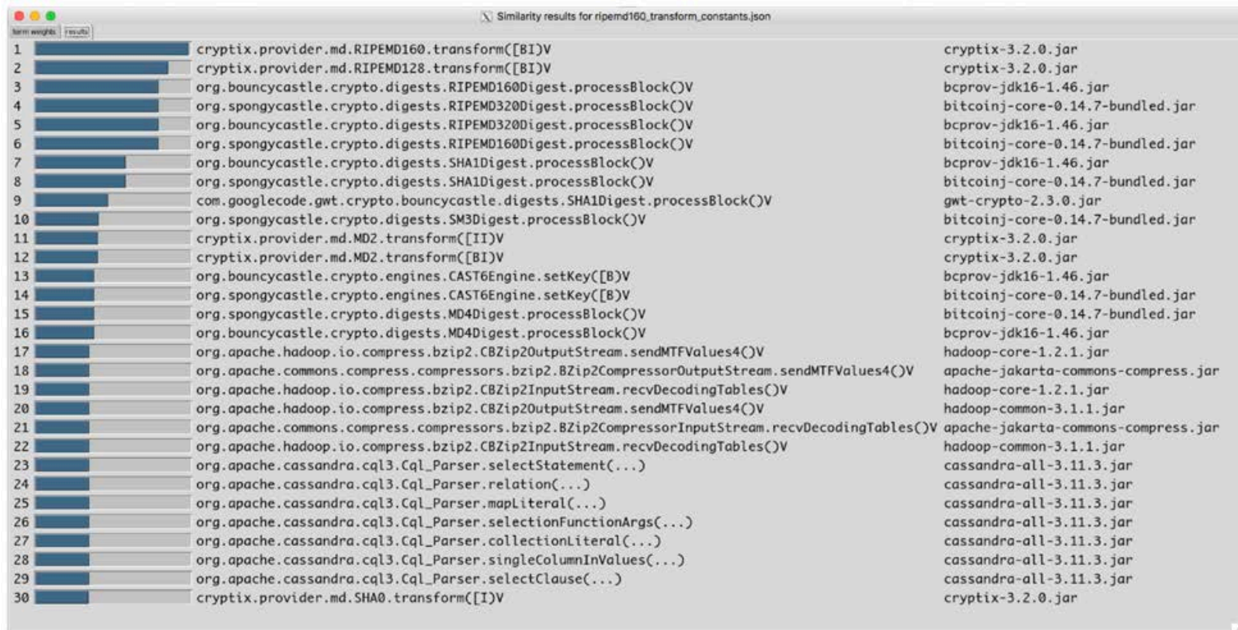

Figure 11. Ranked results for the semantic search for RIPEMD-160

3.6. The ACL2 Theorem Prover

The main focus of Kestrel's research is on proving existing code correctness as well as synthesizing provably-correct code. Kestrel has developed and improved many tools that work within the ACL2 theorem proving framework. ACL2 is a powerful, mature, open-source, industrial-strength theorem prover. It has been developed at UT-Austin since 1989. The current

version, Version 8.1, was released in September, 2018. See http://www.cs.utexas.edu/users/moore/acl2/.

During the MUSE project, Kestrel's partner UT-Austin made a variety of improvements to the ACL2 theorem prover and its supporting libraries, to support the team's work on MUSE. The ACL2 system, and its supporting libraries, is available from https://github.com/acl2/acl2.

3.7. The Axe Rewriter and Lifter

The Axe Lifter lifts the functionality of Java code into a logical representation. It does so by applying the Axe Rewriter to perform symbolic simulation of the Java Virtual Machine (JVM) model as its executes the program on symbolic input. The result is a mathematical term that represents the effect of the code, expressing the code's output in terms of its symbolic inputs. For compactness, terms are represented as structure-shared directed acyclic graphs (DAGs) in which each unique node is represented only once. Each node in a DAG is either a constant, a variable, or a function application of a named function to arguments represented by other nodes.

There are two main approaches to lifting code, depending on how loops are handled. In some cases, loops can be fully unrolled, because their iteration counts are fixed or bounded by a known bound (either in all cases, or under assumptions supplied by the user to the lifter). The result of lifting is then a loop-free representation of the code's functionality in logic. This is implemented in the tool unroll-java-code. When loops cannot be unrolled, they can often be lifted into tail recursive functions. This style of lifting is implemented in the tool lift-java-code, which we call the non-unrolling lifter. Note that this style of lifting is less automatic, because typically loop invariants are needed, e.g., to show that exceptions do not occur in the loop body or that different storage locations manipulated by the loop do not alias. The lifter can automatically guess and check simple invariants.

Normally, unroll-java-code and lift-java-code effectively inline any subroutines encountered. It may be desirable to instead lift code compositionally (one subroutine at a time). This is implemented for the unrolling lifter as the tool unroll-java-code2 and for the loop lifter as lift-java-code2. A related tool, lift-java-code-segment can lift a fragment of Java code that does not form a complete method.

Some restrictions apply to lifting. First, recursion is only supported in the unrolling lifters (though we may add support for it in the non-unrolling lifters as well). Second, the non-unrolling lifters can only lift loops that touch (read or write) a finite set of heap objects. This is because each field touched by the loop becomes a parameter of the function that represents the loop. Furthermore, the lifter must be able to show that the object fields touched by the loop cannot alias. This is eased somewhat by the fact that fields whose names or types differ cannot possibly alias each other.

The lifters allow the user to indicate which part of the JVM state holds the output of interest after execution. Typically, this is the return value of the function or the final value of some field (e.g., the contents of one of the array objects passed into the function). This becomes the return value of the function produced by the lifter.

The lifters, perhaps with guidance from the user, determine which JVM state components affect the output of the program to be lifted. These become parameters of the function produced by the lifter.

The lifters allow assumptions about the inputs to be supplied and allow rewrite rules to be passed in for simplifying the terms that arise during lifting. The loop lifters allow the user to supply information about the loops to be lifted, including invariants, type information, and termination measures. The user can also supply arbitrary properties (in the form of ACL2 theorems called loop postludes) to be proven after one loop before lifting the next loop; these are needed because values from one loop may flow into a subsequent loop, and the lifter may need to show that such values satisfy the invariant of the second loop.

The loop lifter requires a loop body to provably never throw an exception or error (assuming that the loop invariant holds), but settings are available to discard execution paths that result in exceptions or errors. Doing so is unsound but sometimes helpful in quickly lifting a loop.

The loop lifter attempts to lift a loop using a set of candidate invariants (some generated by tool and some supplied by the user). This includes lifting any nested loops. It then tries to show that the invariants assumed are in fact preserved by the loop body (assuming the loop does not exit). If this proof fails, the lifter discards the invariants that failed to prove and attempts again to lift the loop. The process continues until an inductive set of loop invariants is found.

The function that represents a lifted loop contains an exit test (representing the conditions under which the loop terminates) and an update function (representing the change made to the loop variables by the loop body). The loop function repeatedly executes the update function until the exit test is true, whereupon it returns the loop parameters as a tuple.

After lifting, the term (actually a DAG) representing the lifted program can be subjected to further analysis (e.g., theorem proving, equivalence checking, or test generation). Often the result of the unrolling lifter contains only simple operators over bit-vectors and arrays, allowing highly automated reasoning using the Axe Equivalence Checker and its connection to the Simple Theorem Prover (STP) Satisfiability Modulo Theories (SMT) solver. The output of the non-unrolling lifter contains recursive functions that model the loops; these can be transformed using our APT transformation toolkit and subjected to analysis by theorem proving. When a loop contains no nested loops, the loop body itself is loop-free and can thus be subjected to highly automated SMT-based reasoning.

3.8. The Axe Equivalence Checker

The Axe Equivalence Checker is a tool for proving equivalence of terms (actually DAGs) representing (lifted) programs and their specifications, given a set of assumptions about their inputs. If the checker reports success, it guarantees that the two DAGs have equal output values for all input values that satisfy the assumptions. The equivalence checker operates by forming a single DAG (called a "miter") representing the equality of the two input DAGs. The goal of equivalence checking is then to prove that this miter always evaluates to true (for any input that satisfies the assumptions). The following description describes the basic equivalence checking process. If loop functions are present in the miter, extra steps are performed, as discussed in Dr. Smith's Ph.D. thesis [3].

Equivalence checking begins by rewriting the miter DAG. If rewriting fails to transform the miter into true, random test cases are used to discover probable equalities involving internal nodes. These internal equalities are used to break down the equivalence proof into a sequence of smaller proofs, which are attempted via rewriting and calls to the STP solver

(http://stp.github.io/) for bit-vectors and arrays. The Axe Equivalence Checker sweeps up the DAG, attempting to prove the probable equalities and, when successful, using the proven facts to transform the DAG. If the sweeping process fails to reduce the miter to true, the equivalence checker attempts to split the miter into two cases and then attempts to perform equivalence checking on each case, recursively.

During the MUSE project, we also developed a new, tactic-based equivalence checker which is a variant of the Axe Equivalence Checker that provides more user control. Supported tactics include rewriting, pruning unreachable branches (by rewriting and by calling STP), splitting into cases, calling STP on the current goal, and calling ACL2 on the current goal.

We also developed a formal code query tool that can answer questions about simple programs, such as "Is there any input that can cause X to occur?". This can be applied to code found in the corpus to begin exploring its behavior, before a formal proof is attempted.

3.9. The APT (Automated Program Transformations) Toolkit

Under this and other projects, we have been building APT, a library of tools, built on the ACL2 theorem prover, to transform programs and program specifications with a high degree of automation. The APT transformation tools operate on artifacts (e.g. functions) and generate corresponding transformed artifacts along with formal proofs of the relationship (e.g. equivalence) between old and new artifacts; all the proofs generated by APT are checked by the theorem prover. APT includes transformations to apply algorithm schemas, turn data into isomorphic representations, apply rewrite rules, incrementalize computations, turn recursion into tail recursion, and many others.

APT can be used in *program synthesis*, to derive provably correct implementations from formal specifications via sequences of refinement steps carried out via transformations. The specifications may be declarative or executable. The APT transformations can synthesize executable implementations from declarative specifications, as well as optimize executable specifications or implementations. The APT transformations can also be used to generate a variety of diverse implementations of the same specification.

APT can also be used in *program analysis*, to help verify existing programs, suitably embedded in the ACL2 logic, by raising their level of abstraction via transformations that are inverses of the ones used in stepwise program refinement. The two kinds of transformations (for program synthesis and for program analysis) can be used together in an integrated way. This is, in fact, the *analysis-by-synthesis* approach that we have developed and used in this project. Picture the specification and the code linked by a sequence of APT transformations, including "top-down" transformations that transform the specification to be more like the code and "bottom-up" transformations that transform the code to be more like the specification. By chaining together the proofs of the individual steps, one can obtain a *derivation* formally connecting the code with the specification, of which it is proved to be a correct implementation. Such a derivation is often possible to construct for code found in the wild (assuming it is correct) even though not much code in the wild was actually derived from a specification using synthesis techniques. The derivation we create in such cases elucidates how and why the code is correct.

APT enables the user to focus on the creative parts of the program synthesis or analysis process, leaving the more mechanical parts to the automation provided by the tools. The user guides the process by choosing which transformation to apply at each point and by supplying key theorems

(e.g. applicability conditions of transformations), while APT takes care of the lower-level, error-prone details with speed and assurance. We are also working on incorporating into APT heuristics to provide further automated support, e.g. to explore and automatically choose transformations. Besides automation, APT's goals include robustness and practicality.

One of APT's important contributions is to realize the classic ideas of program transformation and stepwise program refinement in the state-of-the-art, industrial-strength theorem prover ACL2 (which is used at AMD, Centaur, Oracle, Rockwell Collins, and others). Thus, when developing correct-by-construction programs, users can seamlessly take advantage of the theorem prover's powerful reasoning tools, vast proof libraries, and vital user community.

APT currently includes about 40 transformations, all of which generate proofs that are checked by the ACL2 theorem prover. All of these transformations generate ACL2 proofs of correctness, have been tested, and have been used to develop various correct-by-construction programs as well as to verify numerous existing programs, particularly in this project.

APT builds on decades of experience in program synthesis. APT's novelty is that the transformations are tightly integrated with the widely used, industrial-strength theorem prover ACL2. This tight integration has several benefits: the extensive libraries of formal definitions and proofs, and the ACL2-based tools, developed by the ACL2 community over many years, are readily available for use in new correct-by-construction developments; and the powerful proof automation of ACL2 greatly reduces the effort to discharge the proof obligations (e.g., applicability conditions of transformations) that arise in correct-by-construction developments. Generating proofs in transformations is eased by the fact that we operate inside one logic and system, without the need for translators. Our transformations are readily available to the ACL2 community for experimentation, feedback, and collaboration.

3.9.1. Some Example Transformations.

The **tail recursion transformation** turns certain non-tail-recursive functions into tail-recursive versions. It provides several variants, which apply to non-tail-recursive functions that satisfy different properties (i.e., the applicability conditions of these transformation variants). For instance, the *monoid* variant turns a non-tail-recursive function of the form

$$f(x) = \textbf{if } a(x) \textbf{ then } b \textbf{ else } c(d(x), f(e(x)))$$

into a tail-recursive version with an accumulator y

$$g(x, y) = \textbf{if } a(x) \textbf{ then } y \textbf{ else } g(e(x), c(d(x), y)))$$

provided that the operator c that combines the result of the recursive call with (a function of) the argument is associative and has b as left and right identity (i.e., c and b form a monoid, hence the name of the transformation variant).

The transformation also generates a theorem of the form

$$f(x) = g(x, b)$$

that asserts the correctness of g with respect to f.

Tail recursion in logical and functional languages corresponds to iteration (i.e., loops) in imperative languages. Thus, the tail recursion transformation is important to formally bridge the gap between specifications that often use non-tail-recursive functions (which are often clearer

and easier to prove properties of, compared to their tail-recursive counterparts) and imperative code.

The **finite differencing transformation** caches partial results to speed up repeated computations. As a very simple example, consider a function that recursively computes the sum of the squares of the numbers up to some limit. The transformation can automatically generate a version of the function that incrementalizes the squaring of the argument, computing the square of each value from the square of the preceding value, and avoiding multiplications (which could be useful on a platform where multiplication is more expensive than addition and subtraction). This is just a simple demonstrative example.

The **data isomorphism transformation** changes the representation of data (e.g., the arguments of a function) into isomorphic representations. This transformation can carry out a large class of data type refinements.

3.9.2. Publication and Availability.

More details on the simplification transformation can be found in the paper "A Versatile, Sound Tool for Simplifying Definitions", by Alessandro Coglio, Matt Kaufmann, and Eric Smith, published at the 14th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2017), May 2017, EPTCS (Electronic Proceedings in Theoretical Computer Science) Volume 249, https://arxiv.org/abs/1705.01228v1

Some APT transformations are available, open-source, in the ACL2 community books. We are actively working on open-sourcing the rest of the transformations to the ACL2 community books as well.

3.10. ACL2 in Java, and ACL2 to Java

The analysis-by-synthesis approach that we developed and employed in this project may involve not only the (safe) re-use of existing code, but also the generation of new code that interoperates with the existing code. The new code is generated from low-level, executable, formal specifications, which in turn are derived from high-level, declarative (possibly non-executable, or inefficiently executable), formal specifications via a sequence of transformation steps.

The ACL2 theorem prover's tight integration with the underlying Lisp platform enables the executable subset of the ACL2 logical language to run readily and efficiently as Lisp, without the need for explicit code generation facilities. Nonetheless, some situations may call for running ACL2 code in other programming languages: specifically, when the ACL2 code must interoperate with external code in those programming languages in a more integrated and efficient way than afforded by inter-language communication via foreign function interfaces or by inter-process communication with the ACL2/Lisp runtime. Using Lisp implementations written in the target programming languages involves not only porting ACL2 to them, but also including much more runtime code than necessary for the target applications. Compilers from Lisp to the target programming languages may need changes or wrappers, because executable ACL2 is not quite a subset of Lisp; furthermore, the ability to compile non-ACL2 Lisp code is an unnecessary complication as far as ACL2 compilation is concerned, making potential verification harder.

Indeed, for the analysis-by-synthesis approach, it is desirable to generate code from ACL2 in the more widely used programming languages that the existing found code is likely to be written in. In particular, in this project, we focused on Java code. Therefore, we built a Java code generation facility for ACL2. This facility consists of two parts:

- **ATJ** is a Java code generator for ACL2. ATJ translates executable, side-effect-free, non-stobj-accessing ACL2 functions, without their guards, into Java. It does so in a simple way, by turning the functions into deeply embedded Java representations that are executed by an ACL2 evaluator written in Java.
- **AIJ** is a deep embedding in Java of an executable, side-effect-free, non-stobj-accessing subset of the ACL2 language without guards. AIJ consists of (i) a Java representation of the ACL2 values, terms, and environment, (ii) a Java implementation of the ACL2 primitive functions, and (iii) an ACL2 evaluator written in Java. AIJ executes the deeply embedded Java representations of ACL2 functions generated by ATJ. AIJ is of independent interest and can be used without ATJ.

The ACL2 language subset supported by ATJ and AIJ includes all the values, all the primitive functions, and many functions with raw Lisp code.

3.10.1. AIJ: The Deep Embedding.

AIJ is a Java package whose public classes and methods provide an API to (i) build and unbuild representations of ACL2 values, (ii) build representations of ACL2 terms and of an ACL2 environment, and (iii) evaluate calls of ACL2 primitive and defined functions, without checking guards. By construction, the ACL2 code represented and evaluated by AIJ is executable, has no side effects, does not access stobjs, and has no guards.

AIJ consists of a few thousand lines of Java code (including blank and comment lines), thoroughly documented with Javadoc comments. The implementation takes advantage of object-oriented features like encapsulation, polymorphism, and dynamic dispatch.

AIJ represents ACL2 values as immutable objects of a number of Java classes organized in a hierarchy. Each class corresponds to a set of values. The subset relationships match the inheritance relationships. The sets of values that are unions of other sets of values correspond to abstract classes; the other sets correspond to concrete classes. The information about the represented ACL2 values is stored in private fields of the non-abstract classes. The value classes provide public static factory methods to build objects of these classes (there are no public Java constructors, thus encapsulating the details of object creation and re-use, which is essentially transparent to external code because these objects are immutable). The value classes provide public instance getter methods to unbuild (i.e., extract information from) objects of these classes.

AIJ represents ACL2 terms in a manner similar to ACL2 values, as immutable objects of a number of Java classes organized in a hierarchy. The superclasses are abstract, while the subclasses are concrete. The information about the represented ACL2 terms is stored in private fields of the non-abstract classes. The term classes provide public static factory methods to build objects of these classes, but no public Java constructors, similarly to the classes for ACL2 values. The term classes currently provide no public instance getter methods to unbuild (i.e., extract information from) objects of these classes, as they are not needed for the purpose of Java code generation from ACL2.

ACL2 terms are evaluated in an environment that includes function definitions, package definitions, etc. AIJ stores information about part of this environment in a Java class. Since there is just one environment at a time in ACL2, this class has no instances and only static fields and methods.

Since the ACL2 primitive functions have no definitions, AIJ cannot evaluate their calls via their bodies as described below. AIJ implements these functions "natively" in Java, in a package-private class. Each primitive function (except *if*, whose calls are evaluated non-strictly as described below), is implemented by a private static method in this class.

AIJ recursively evaluates ACL2 terms via methods in the term classes. The methods take as argument a Java map that binds values to variables, and return the result of evaluating the target object with respect to the supplied binding. Function calls are evaluated by first recursively evaluating all the arguments, and then either (i) calling the method that natively implement the primitive function (if the called function is primitive) or (ii) recursively evaluating the defining body of the function with respect to a binding that associates the actual arguments to the formal arguments (if the called function is not primitive). However, if the called function is *if*, it is evaluated non-strictly: the first argument is evaluated, and then either the second or the third one is evaluated, based on the value of the first argument.

3.10.2. ATJ: The Code Generator.

ATJ is an ACL2 tool that provides an event macro to generate Java code from specified ACL2 functions. The generated Java code provides a public API to (i) build an AIJ representation of the ACL2 functions and other parts of the ACL2 environment and (ii) evaluate calls of the functions on ACL2 values via AIJ.

ATJ consists of a few thousand lines of ACL2 code (including blank lines, implementation-level documentation, and comments), accompanied by a few hundred lines of user-level documentation in XDOC. The implementation is thoroughly documented as well.

ATJ generates a single Java file containing a single class. The file has the same name as the class; it is (over)written in the current working directory, unless the user specifies a directory. ATJ directly generates Java concrete syntax, via formatted printing to the ACL2 output channel associated to the file, without going through a Java abstract syntax and pretty printer.

As part of building an AIJ representation of the ACL2 environment, the Java code generated by ATJ builds AIJ representations of ACL2 values and terms: function definitions include terms as bodies, and constant terms include values. It does so via the factory methods discussed earlier. In principle, ATJ could turn each ACL2 value or term into a single Java expression with an "isomorphic" structure. However, values and terms of even modest size (e.g., function bodies) would lead to large expressions, which are not common in Java. Thus, ATJ breaks them down into sub-expressions assigned to local variables. In general, ATJ turns each ACL2 value or term into (i) zero or more Java statements that incrementally build parts of it and (ii) one Java expression that builds the whole of it from the parts. ATJ does so recursively: the expression for a sub-value or sub-term is assigned to a new local variable that is used in the expression for the containing super-value or super-term. To generate new local variable names, ATJ keeps track of three numeric indices (for values, terms, and lambda expressions – recall that the latter are mutually recursive with terms) as it recursively traverses values and terms. The appropriate index is appended to '*value*', '*term*', or '*lambda*' and then incremented.

The Java code generated by ATJ builds an AIJ definition of each ACL2 package known when ATJ is invoked. The names of the known packages are the keys of the alist returned by the built-in function *known-package-alist*, in reverse chronological order.

The Java code generated by ATJ builds an AIJ definition of each (non-primitive) ACL2 function specified via one or more function symbols supplied to ATJ. Each function symbol implicitly specifies not only itself, but also all the functions called directly or indirectly by it, ensuring the "closure" of the generated Java code under ACL2 calls.

ATJ uses a worklist algorithm, initialized with the list of functions supplied by the user, to calculate a list of their closure under calls. Each iteration removes the first function *fn* from the worklist, adds it to the result list, and extends the worklist with all the functions directly called by *fn* that are not already in the result list. Here 'directly called by' means 'occurring in the *unnormalized-body* property of'. If *fn* has no *unnormalized-body* property, it must be primitive, otherwise ATJ stops with an error – this happens if *fn* is a constrained, not defined, function. If *fn* has raw Lisp code, it must be in a whitelist of functions that are known to have no side effects. If *fn* has input or output stobjs, ATJ stops with an error – this may only happen if *fn* is not primitive.

3.10.3. Publication and Availability.

More details on AIJ and ATJ can be found in the paper "A Simple Java Code Generator for ACL2 Based on a Deep Embedding of ACL2 in Java", by Alessandro Coglio, published at the 15th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2018), November 2018, EPTCS (Electronic Proceedings in Theoretical Computer Science) Volume 280, https://cgi.cse.unsw.edu.au/~eptcs/paper.cgi?ACL22018.1

Both AIJ and ATJ are open-source, in the ACL2 community books.

Approved for Public Release; Distribution Unlimited.

4. RESULTS AND DISCUSSION

During the MUSE project, we solved a variety of challenge problems. A typical problem involved the desire to implement particular functionality, using code from the corpus to do much of the work and to save time over writing new code by hand. We used our corpus tools to search the corpus for relevant code and analyzed the results using our formal tools to ensure correctness. When the desired functionality was not available in the corpus as a single program, we used our formal tools to synthesize larger systems that included the code from one or more corpus projects. We then proved, again using our formal tools, that the found / synthesized code was correct. The proofs performed about the corpus code can be broken down into two groups: When the loops were unrollable, we usually applied Axe to perform highly automated equivalence checking against a specification. When the loops were not unrollable, we usually applied APT to the lifted code to construct an analysis-by-synthesis derivation formally linking the code to its specification. All told, we created several dozen derivations of corpus code. The rest of this section describes some of our major accomplishments, in the form of examples.

4.1. Bresenham Example

Bresenham's line drawing algorithm calculates a discrete best-fit line between two integer-coordinate points, using only integer addition. This algorithm is useful to draw a line on a screen. The algorithm was first published in a paper in 1965; the paper describes the algorithm and proves its correctness. The algorithm is relatively short and seemingly simple, but at first sight it is not at all obvious why the algorithm works and how it was devised.

Bresenham's line drawing algorithm was chosen, by the Evaluation Team, as one of the Benchmark Problems in Phase 1 of the MUSE project. Our team used the analysis-by-synthesis approach to find Java code in the corpus that implements this algorithm and to formally prove it functionally correct.

4.1.1. Specification.

We wrote a high-level, non-executable specification, in ACL2, of the input/output behavior of the algorithm. Essentially, the input is a pair of points with integer coordinates, and the output is a list of discrete points, also with integer coordinates, that is a best fit to the "continuous" line between the two points. (We use the indeterminate article 'a' because the best fit is not necessarily unique; sometimes the ideal point falls right in the middle of two discrete points, which are therefore equally acceptable.) This specification is expressed via a precondition and a postcondition; it makes use of quantifiers and is therefore not immediately executable.

4.1.2. Code Search.

With the specification in hand, we searched the corpus for implementations of Bresenham's line drawing algorithm. We started with a very simple search for the text 'bresenham', which sufficed to find a good number of potential candidates.

We proceeded to examine the candidates. Some were quickly excluded because they just mentioned the word 'bresenham' but did not actually contain any discernible implementation of the algorithm. Others were excluded because they actually implemented Bresenham's circle

drawing algorithm – an algorithm based on similar principles, but not fitting our specification for a straight line. Additional ones were excluded after some manual examination, which revealed that they did not actually implement Bresenham's algorithm, but rather something perhaps related to it. Eventually, we picked one of a few candidates that were left as the one to verify.

4.1.3. Invariant Generation.

We ran the CodeHawk static analyzer on the code that we chose (as described above). The purpose was to infer an invariant for the loop of the algorithm (the invariant was used as described below). Given the relative complexity of the algorithm, inferring a loop is not immediate. Helpfully, CodeHawk automatically generated and proved an invariant, which we translated into the ACL2 language.

4.1.4. Code Lifting.

In order to verify the code against the specification, which is written in ACL2, the code had to be suitably represented in the ACL2 language as well. We did that via the Axe lifter, which turns Java bytecode into a representation in ACL2 as a collection of functions.

We supplied, to the Axe lifter, the loop invariant inferred by CodeHawk (see above). This was necessary for the Axe lifter to prove the termination of the loop (under the preconditions of the specification).

4.1.5. Top-Down Derivation Steps.

With the code, found in the corpus, now represented in the ACL2 logical language, we proceeded to construct a derivation that links the code to the specification described above. We did that by first performing several top-down derivation steps from the high-level specification towards the code.

First, we used the APT narrowing transformation to resolve the under-specification inherent in the best-fit requirements. We did that by choosing a specific rounding function for the ideal point (one that rounds numbers of the form $n.5$, where n is an integer, up), and by narrowing the postcondition $[abs(y_{\text{integer}} - y_{\text{ideal}}) \leq 1/2]$ to $[y_{\text{integer}} = round(y_{\text{ideal}})]$, where y_{integer} is the integer, approximating ordinate and y_{ideal} is the ideal ordinate.

Second, we turned the bounded universal quantification in the postcondition into a recursive function. (We did that step manually because the APT transformation to do that had not been implemented yet. However, it was already clear then how to implement that transformation.)

Third, we used the APT finite differencing transformation to cache the value of integer ordinate y_{integer} in the loop. This way, at each loop iteration, this ordinate either stayed the same or got incremented by 1 (Bresenham's basic algorithm assumes that the line is in the first octant of the plane, since lines in the other octants can be drawn symmetrically to the first octant). In this way, the non-integer operations to calculate the integer ordinate at each iteration got replaced with integer operations (do nothing or add 1). We note that this finite differencing step also involves algebraic simplification steps, carried out using the APT simplification transformation.

Fourth, we use again the APT finite differencing transformation to cache the value of the test used to decide whether the integer ordinate stays the same or gets incremented by one. We also

used the APT simplification transformation to carry out some algebraic simplification steps that resulted in integer operations being exclusively used in the loop iterations.

The result of the above transformation is an executable implementation of Bresenham's algorithm in ACL2. Interestingly, this top-down derivation provides a rational reconstruction of how Bresenham's algorithm works and why.

4.1.6. Bottom-Up Derivation Steps.

While the result of the top-down derivation steps described above is an executable ACL2 implementation of Bresenham's algorithm, there is still some distance between that and the result of lifting the Java code into the ACL2 logic. In particular, the lifted code uses bit vectors and modular arithmetic operations to represent the Java integer type and operations. Thus, we carried out a few bottom-up steps to turn this lifted code into a form that is close to the executable ACL2 implementation of Bresenham's algorithm obtained via the top-down steps.

We used the APT simplification transformation, with a specific set of rewrite rules, to turn the modular arithmetic operations into non-modular arithmetic operations. Note that this is correct only if the modular operations do not wrap around. In the application of the rewrite rules by the APT simplification transformation, this condition manifests as proof obligations. These were all automatically proved by ACL2, and thus the arithmetic operations were successfully transformed.

Then we used the APT data isomorphism transformation to change the representation of the values manipulated by the lifted code from bit vectors to ACL2 integers (in suitable ranges). This introduced conversion functions, which we removed via the APT simplification transformation with another specific set of rewrite rules; this also involved some proof obligations to show that the conversions can be eliminated because the values of the expressions are in suitable ranges, which again ACL2 proved automatically.

The result of these bottom-up steps is almost the same as the result of the top-down steps. ACL2 easily proved their equivalence, thus completing the formal proof chain from the high-level specification to the code.

4.2. Micro Air Vehicle Link (MAVLink) Example

In Phase 2 of the MUSE program, the Kestrel team solved a Challenge Problem (CP) regarding the MAVLink protocol.

4.2.1. Problem Statement.

Kestrel's MAVLink CP was to create a formally verified implementation, using code from the corpus, of the MAVLink message parsing and creation needed by the client of the MUSE Phase 2 CP ecosystem: The client generates MAVLink messages in response to graphical user interface (GUI) events from the user (they then get wrapped in hypertext transfer protocol (HTTP) messages and sent by the code Kestrel created for the HTTP challenge problem). When MAVLink responses come back (after being extracted from the HTTP messages by the code for the HTTP challenge problem), the client parses the MAVLink responses and shows the results in the GUI.

4.2.2. MAVLink Specification.

Kestrel wrote a formal specification for MAVLink message creation and parsing. The specification uses the XML description (common.XML) of the MAVLink message types. Kestrel downloaded it from the official MAVLink GitHub page. It also formalizes the MAVLink field types, packet layout, serialization and deserialization, field reordering (required by the protocol to increase alignment, e.g., with 2-byte and 4-byte boundaries), and the cyclic redundancy check (CRC) checksum calculation (including the inclusion of an extra byte that depends on the message descriptor).

For message creation, Kestrel specified a function for each message type. Each of these functions takes arguments representing all of the fields of its message type (and all of the standard MAVLink fields) and returns an array of bytes representing the MAVLink message. For message parsing, Kestrel specified a single function that takes a MAVLink message (of unknown type) as an array of bytes, performs various well-formedness checks, and returns a structured representation of its contents.

4.2.3. Corpus Code.

Kestrel found a corpus project, called "ghelle" after the GitHub username of its creator, that included the ability to generate code to process MAVLink messages. Of course, this code did not come with any formal specifications or proofs. It also provided a somewhat different API than was desired for the challenge problem. Kestrel ran ghelle's generator to obtain code for creating and parsing various kinds of MAVLink messages.

4.2.4. Glue Code Generation.

Kestrel generated glue code to wrap ghelle's code to provide the desired APIs: one glue code function for creation of each type of message and a single function (with a switch statement on the message type) for parsing a message of unknown type. The generated code relies on ghelle's code to do most of the actual work in message processing.

Note: 12 types of messages are not dealt with by ghelle's code. In addition, ghelle's code used 'incorrect' data types for the fields of 5 additional message types. We believe both of these issues are due to the ghelle project using an older version of common.XML. These message types were excluded when the code and proof generators were run. (Also excluded were 13 message types with IDs greater than 255, since these are not legal in MAVLink 1.0.) This leaves 119 types of messages whose code was synthesized / verified.

4.2.5. Formal Proof Generation.

For both message parsing and creation, Kestrel wrote small tools that automatically generate the proofs (as sequences of events submitted to the theorem prover to invoke code unrolling and equivalence checking). The generators spare the user from having to write 119 proofs, one for each message type. Each proof shows that the code matches the spec for that message type. The proof for parsing is a bit more involved than for message creation, due to the need to handle the various kinds of ill-formed messages coming in.

These generators run automatically in less than 1 second. ACL2 then takes about an hour to check the message creation proofs and another hour for the message parsing proofs.

Kestrel analyzed the rewrite rules used in the proofs about message creation and found that the set of rules used rapidly 'converges'. After the first few proofs, new rules were rarely needed to verify additional message types.

4.2.6. Size of the Synthesized Artifacts.

A quick estimate of the size of the MAVLink challenge problem solution is that it contains 13,795 lines, including corpus code and generated glue code. (Disclaimer: Some of the corpus code counted here may not actually be called, but we think that most of it is.)

4.3. Bitcoin Public Key To Address Example

In Phase 3 of the MUSE project, we set out to synthesize an implementation of an important operation in the Bitcoin cryptocurrency: the computation of Bitcoin address from a public key. This operation, depicted in Figure 12, is fairly complicated, involving 3 calls of the SHA-256 hash function, a call of the RIPEMD-160 hash function, and the use of Base58 encoding.

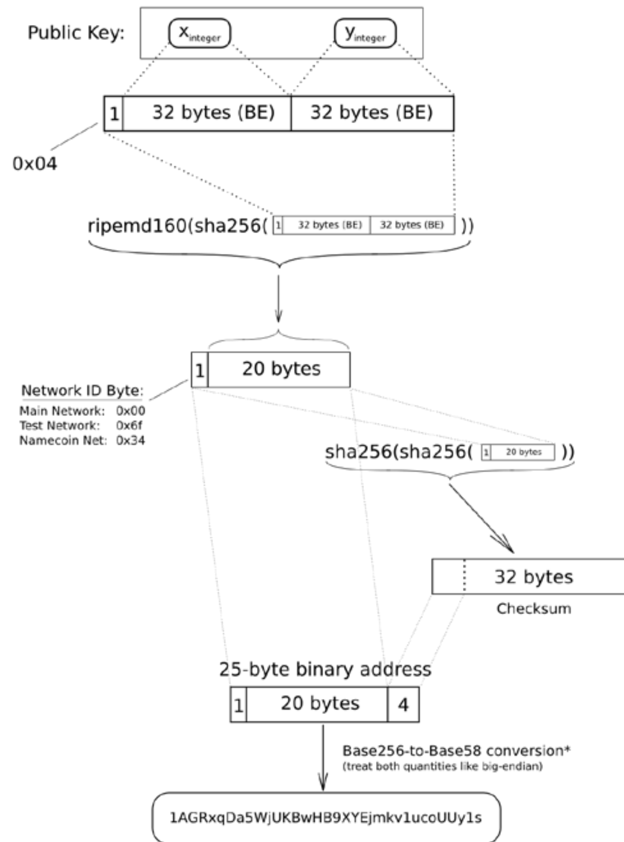

Figure 12. Bitcoin's Public Key to Address Computation

Our goal was to use code from the corpus to obtain a verified implementation of this operation. We began by specifying its correct behavior in the ACL2 logic, including writing formal specifications for the SHA-256 and RIPEMD-160 hash functions and for Base58 encoding. We then identified suitable components from two projects on Github: we took implementations of SHA-256 and RIPEMD-160 from the bouncycastle project, and we took the implementation of Base58 encoding from the bitcoinj project. From these components, we wrote a bit of "glue" code to create an implementation of the public-key-to-address functionality. Almost all of the work in the implementation is done by the corpus code.

It remained to verify the implementation that used the components from the corpus, proving it equivalent to our formal specification in ACL2. This we did by lifting the code into logic using the Axe Lifter, applying the Axe equivalence checker to verify the cryptographic hash functions, creating an APT derivation to connect the Base58 code to its specification, and using Axe to combine these various proofs into a proof of the whole operation.

5. CONCLUSIONS

The DARPA MUSE project set out to investigate the hypothesis that large online corpora of open source code can be helpful for a variety of programming and software maintenance tasks. Our team's research effort focused on the use of code found in such corpora for the purpose of program synthesis. The idea of incorporating code found on the internet into a program being synthesized carries risks in that the found code may contain bugs or security vulnerabilities. Our effort thus focused on *high-assurance* code reuse, in which the found code, and often the entire synthesized artifact, is proven correct. The examples described in the previous section, which we solved using tools developed and improved during the MUSE project, served as a confirmation of the MUSE hypothesis. In particular, our tools for corpus processing, indexing and analysis, together with our database and machine learning tools for code search (including similarity search) allowed us to find code in the MUSE corpus that implements a variety of different kinds of desired functionality. Our formal proof tools, including the APT and Axe toolkits based on the ACL2 theorem prover, allowed us to reuse the found code *safely*, providing proofs of its functional correctness and providing high assurance in the synthesized programs.

6. REFERENCES

1. **A Formalization of the ABNF Notation and a Verified Parser of ABNF Grammars.** Alessandro Coglio. *10th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2018)*, July 2018. LNCS 11294, pages 177–195. https://link.springer.com/chapter/10.1007/978-3-030-03592-1_10

2. **A Simple Java Code Generator for ACL2 Based on a Deep Embedding of ACL2 in Java.** Alessandro Coglio. *15th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2018)*, November 2018, pages 1–17. https://cgi.cse.unsw.edu.au/~eptcs/content.cgi?ACL22018

3. **Axe: An Automated Formal Equivalence Checking Tool for Programs.** Eric Smith. Ph.D. Thesis. Stanford University. 2011. http://www.kestrel.edu/home/people/eric.smith/dissertation.pdf

4. **A Versatile, Sound Tool for Simplifying Definitions.** Alessandro Coglio, Matt Kaufmann, and Eric Smith. May 2017. *14th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2017),* May 2017, pages 61–77. https://arxiv.org/abs/1705.01228v1

5. **DefunT: A Tool for Automating Termination Proofs by Using the Community Books.** Matt Kaufmann. *15th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2018)*, November 2018, pages 161–163. https://cgi.cse.unsw.edu.au/~eptcs/paper.cgi?ACL22018.12

6. **Introduction to Information Retrieval.** Christopher D. Manning and Prabhakar Raghavan and Hinrich Schutze. Cambridge University Press. 2008.

7. **Second-Order Functions and Theorems in ACL2.** Alessandro Coglio. *13th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2015)*, October 2015, pages 17–33. http://dx.doi.org/10.4204/EPTCS.192.3

7. LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ACL2	A Computational Logic for Applicative Common Lisp
AIJ	ACL2 in Java
API	Application Programming Interface
APT	Automated Program Transformations
ASCII	American Standard Code for Information Interchange
ATJ	ACL2 to Java
BSD	Berkeley Software Distribution
CF	Collection Frequency
CP	Challenge Problem
CRC	Cyclic Redundancy Check
CP-850	Code Page 850
DAG	Directed Acyclic Graphs
DARPA	Defense Advanced Research Projects Agency
DF	Document Frequency
GUI	Graphical User Interface
IDF	Inverse Document Frequency
ISO	International Organization for Standardization
JSON	Javascript Object Notation
JVM	Java Virtual Machine
KT	Kestrel Technology
MAVLink	Micro Air Vehicle Link
MUSE	Mining and Understanding Software Enclaves
NMF	Non-negative Matrix Factorization
SMT	Satisfiability Modulo Theories
SQL	Structured Query Language
STP	Simple Theorem Prover
TF	Term Frequency
TF-IDF	Term Frequency - Inverse Document Frequency
UCI	University of California at Irvine
UUID	Universally Unique Identifier
XML	Extensible Markup Language