AFRL-RI-RS-TR-2019-068

DRAPER MINING AND UNDERSTANDING SOFTWARE ENCLAVES (MUSE)

THE CHARLES STARK DRAPER LABORATORY, INC.

MARCH 2019

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

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

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2019-068 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

# REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

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

4. TITLE AND SUBTITLE DRAPER MINING AND UNDERSTANDING SOFTWARE ENCLAVES (MUSE)	**5a. CONTRACT NUMBER** FA8750-15-C-0242
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 61101E
6. AUTHOR(S) Paul Ellingwood, Hugh Enxing, Lei Hamilton, Jacob Harer, Thomas Jost, Louis Kim, Leo Kosta, Marc McConley, Onur Ozdemir, Chris Reale, Rebecca Russell	**5d. PROJECT NUMBER** MUSE
	5e. TASK NUMBER CD
	5f. WORK UNIT NUMBER RA
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) The Charles Stark Draper Laboratory, Inc. 555 Technology Square Cambridge, MA 02139	**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-2019-068

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 research carried out by The Charles Stark Draper Laboratory, Inc. (Draper) team in the Defense Advanced Research Projects Agency (DARPA) Mining and Understanding Software Enclaves (MUSE) program under contract FA8750-15-C-0242. Our focus on the MUSE program was to develop big-data analytics using machine learning for automatic vulnerability classification and repair. Key technical advancements that we contributed to the MUSE program included: (1) fast and scalable machine learning-based classifiers to detect patterns in known types of software vulnerabilities; (2) a generative adversarial network (GAN) to advance the state of the art in automated repair of common types of software vulnerabilities; and (3) a data ingestion pipeline to scrape, build, and analyze millions of functions from open-source software to generate training data for learning-based algorithms.

15. SUBJECT TERMS
Cyber security, data analytics, deep learning, machine learning, software vulnerability detection, software vulnerability repair

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** STEVEN DRAGER
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	94	**19b. TELEPHONE NUMBER (*Include area code*)** N/A

Contents

LIST OF FIGURES

LIST OF TABLES

1. SUMMARY

This report describes research carried out by The Charles Stark Draper Laboratory, Inc. (Draper) team in the Defense Advanced Research Projects Agency (DARPA) Mining and Understanding Software Enclaves (MUSE) program under contract FA8750-15-C-0242. The Draper team included subcontractors Paradigm4 and Stanford University.

Our focus on the MUSE program was to develop big-data analytics using machine learning for automatic vulnerability classification and repair. This work encompasses three of the technical areas (TAs) for the MUSE program: TA2 (Artifact Generators), TA3 (Mining Engine), and TA4 (Analytics). Our work under TA2 involved mining open source code to generate a large quantity of data that we used to train the machine learning algorithms to classify and repair vulnerabilities. Our work under TA3 involved representing data in formats most conducive to training the algorithms. Our work under TA4 involved developing advanced algorithms that make optimal use of the available training data. We demonstrated the performance of our system through benchmarking and participation in program-wide evaluation events.

Key technical advancements that we contributed to the MUSE program included the following:

- Fast and scalable machine learning-based classifiers to detect patterns in known types of software vulnerabilities and demonstrated improved detection accuracy (precision and recall) over multiple open-source static analysis (SA) tools on a synthetic benchmark dataset.
- A generative adversarial network (GAN) to advance the state of the art in automated repair of common types of software vulnerabilities. The prior state of the art is sequence-to-sequence learning that requires labeled bad-good pairs of training data, the availability of which is extremely limited. The new GAN does not require labeled data pairs, and this greatly expands the amount of training data that can be made available for this approach.
- A data ingestion pipeline to scrape, build, and analyze millions of functions from open-source software to generate training data for learning-based algorithms.

The most promising direction for future work in this area would be a focus on expanding the availability of training data with truth labels, such as those from dynamic analysis tools or mined from security patches. We demonstrated that larger labeled training sets should provide improved vulnerability classification accuracy.

2. INTRODUCTION

Every year, thousands of new security vulnerabilities are reported and catalogued in the Common Vulnerabilities and Exposures (CVE) database [1]. Annual security-related hardware, software, and service expenses approach $100 billion worldwide [2]. Automation could substantially reduce the labor and increase the yield in the process of identifying and repairing such vulnerabilities in software. Draper's work under the DARPA MUSE program developed machine learning approaches for automatic identification and repair of software vulnerabilities.

2.1 Deep Learning Approach

Deep Learning is a subfield of machine learning in Artificial Intelligence (AI) concerned with algorithms inspired by the structure and function of the brain called biological neural networks. The term artificial neural networks (ANNs) (and more broadly AI) is used to describe Deep Learning, as ANNs loosely model the neurons in a biological brain, and the networks are supposed to imitate the workings of the human brain in processing data and creating patterns and representations for use in decision making. Draper chose Deep Learning as the technological foundation for work on the MUSE program because of its ability to learn high-level feature representations from low-level inputs. This is a critical capability for a system that can learn to identify patterns that recur in software with security vulnerabilities and generalize those patterns to identify new vulnerabilities in software that the system hasn't seen before. Deep Learning has also demonstrated success in related fields that have similar problems, such as image classification [3] [4] [5] [6] [7], image caption generation [8] [9], semantic natural language processing (NLP) [10] [11], and machine translation [12] [13]. Deep Learning has achieved the best performance in multiple machine learning competitions such as ImageNet [14], Medical Image Computing and Computer Assisted Intervention (MICCAI) [15], and Go [16].

2.2 Major Program Elements

Major elements of Draper's developments on the MUSE program included the following:

- Mining open source code to generate a large quantity of training data for the Deep Learning algorithms. This involved scraping code repositories to get information about code as it evolved over its revision history, building the code to enable training artifacts to be extracted, and applying training labels to the code that we derived from a variety of sources.
- Representing data in formats most conducive to training the algorithms. This involved defining a combination of source and build artifacts to use as features on which to train, extracting those features from the code, storing data in a variety of database formats best suited to retrieval for training, careful de-duplication of data to avoid overfitting, and representing data in vector formats for use by the Deep Learning algorithms.
- Developing advanced algorithms that make optimal use of the available training data. This involved developing and evaluating a variety of classification and repair algorithms.
- Evaluating performance of the system through benchmarking and participation in program-wide evaluation events.

The development occurred over the course of three program phases:

- During Phase 1, we developed and demonstrated the initial classification capability.
- During Phase 2, we refined the classification capability and advanced the state of the art in repair.
- During Phase 3, we refined the classification and repair capabilities further and evaluated system performance.

2.3 Subcontractors

The Draper team for the MUSE program included two subcontractors:

- Paradigm4 is the developer of an open-source data analysis and management system that addresses peta-scale operation through distributed, scalable and in-situ vector and matrix operation. During Phases 1 and 2 of the MUSE program, Paradigm4 developed a mathematical data analysis component called SciDB to provide support for efficient, distributed matrix and graph operations. SciDB was shown to provide substantial speedup for a central processing unit (CPU) based architecture but was not implemented for a graphics processing unit (GPU) based architecture. We did not continue this development in Phase 3 because the Draper DeepCode development moved to a GPU-based architecture, which would not have benefited from the CPU-based improvements that SciDB offered.
- Stanford University focused on Deep Learning, in which very large neural networks are constructed to learn from labeled and unlabeled data. During Phases 1 and 2 of the MUSE program, Stanford University performed initial work on sequence-to-sequence learning algorithms. We did not continue this development in Phase 3 because the Draper DeepCode development for repair moved away from the sequence-to-sequence approach and in the direction of developing a GAN model.

3. METHODS, ASSUMPTIONS, AND PROCEDURES

During the course of the MUSE program, Draper developed a data pipeline for training data generation and machine learning that encompassed TA2, TA3, and TA4 of the MUSE program. Section 3.1 describes the high-level system architecture for the data pipeline. Section 3.2 describes the data generation elements of the architecture (TA2 and TA3). Section 3.3 describes the machine learning elements of the architecture (TA4). We refer to the collection of Draper solutions for the MUSE program as the *DeepCode* system.

3.1 System Architecture

Figure 1 depicts the functional flow of data through the DeepCode system.

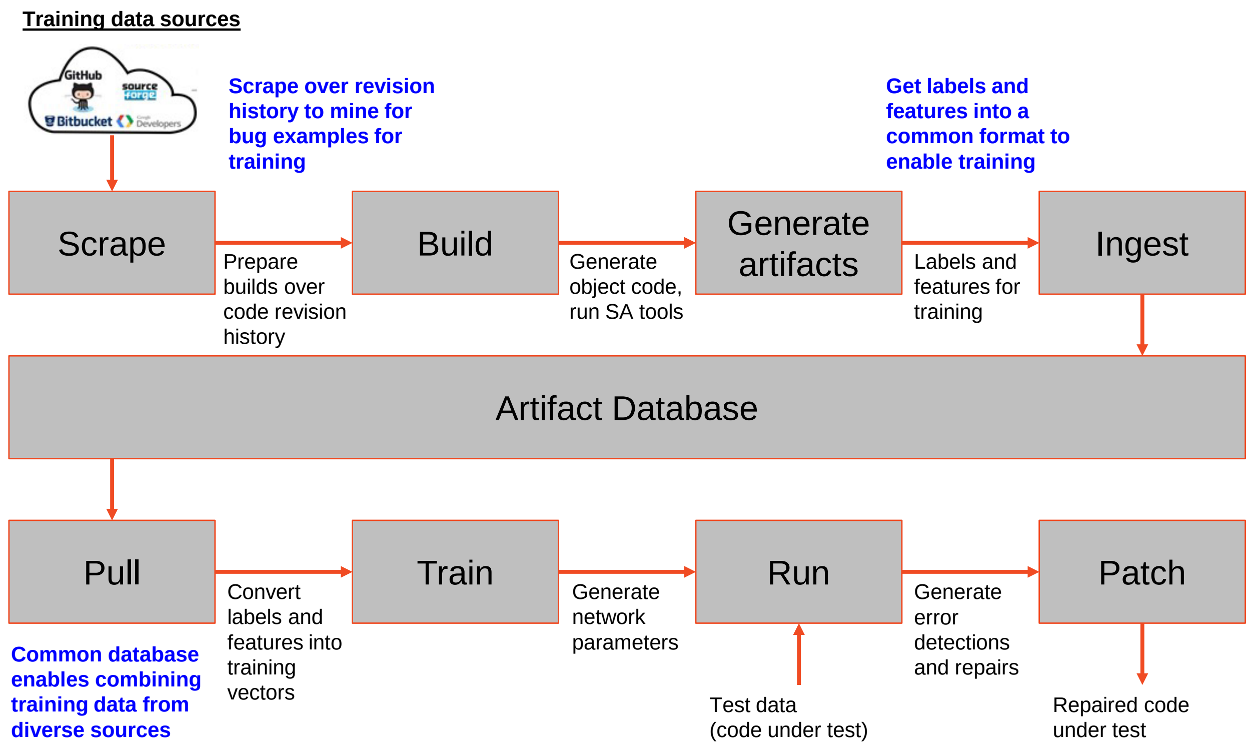

Training data sources
GitHub
source forge
Bitbucket
Developers
Scrape over revision history to mine for bug examples for training
Get labels and features into a common format to enable training
Scrape
Prepare builds over code revision history
Build
Generate object code, run SA tools
Generate artifacts
Labels and features for training
Ingest
Artifact Database
Pull
Convert labels and features into training vectors
Train
Generate network parameters
Run
Generate error detections and repairs
Patch
Common database enables combining training data from diverse sources
Test data (code under test)
Repaired code under test

Figure 1. Functional Flow of Data Through DeepCode System

Data flow through the system occurs in the following key steps:

1. Scrape: Starting with code from a variety of sources, we scrape over the revision history of the code repositories to mine for examples of software bugs that we can use to train the learning algorithms. This step prepares multiple builds over the code revision history.
2. Build: This step builds the scraped source code to generate object code and run SA tools.
3. Generate artifacts: This step generates labels and features to be used for training.
4. Ingest: This step collects labels and features into a common format to enable training.
5. Artifact database: A common artifact database allows us to combine training data from diverse sources.

6. Pull: This step collects artifacts from multiple sources stored in the database and converts the labels and features into training vectors. This step also performs functions such as de-duplication of training data.
7. Train: This step operates on the training vectors to generate network parameters (weights) for the learning algorithms.
8. Run: This step runs trained networks on code under test to generate error detections and repairs.
9. Patch: This step produces a repaired version of the code under test.

3.2 Training Data Generation (MUSE TA2 and TA3)

This section describes the training data generation elements of the DeepCode system. Section 3.2.1 describes the training corpus. Section 3.2.2 describes the system architecture that hosts the training data generation elements. Section 3.2.3 describes key elements of the processing chain used to generate the training data.

3.2.1 Training Corpus.

A key challenge in the application of machine learning is to get enough labeled data to train the algorithms. Learning algorithms should have hundreds of thousands of code examples that are *known* to have certain types of vulnerabilities, so that they can learn the patterns. Training requires that we have *truth data* about code with and without vulnerabilities. We refer to these truth data as “labels.” Requirements for the labels include the following:

- Volume: Training the networks requires a large number of labels.
- Quality: Labels need to be correct with low false positive rate (FPR).
- Effort: We need to be able to generate labels with reasonable time and labor.
- Diversity: Labels need to cover a wide range of types of errors. The labeled code should also be representative of the types of code to be classified and repaired.

Over the course of the MUSE program, we tried several label generation approaches as described below.

- Curated: Some code repositories, such as the source code for the Debian Linux distribution, provide precise information about which CVEs are corrected in which revisions of the software. We processed these and used this information to generate labels, but we were able to generate only a very small number of labels in this way.
- SA-derived: We can use SA to generate large numbers of labels quickly on arbitrary source code, but the quality of these labels is only as good as the accuracy of the SA tools.
- Manually labeled: It is possible to label code manually, but this is a labor-intensive process and as such produces very low yield. ManyBugs [17] is an example of a dataset for which a very limited amount of manually generated label data are available.

- Artificially injected: We attempted to inject known errors artificially into otherwise good code to generate synthetic code with known errors. This produces code with good labels but the diversity of errors represented in such data is very limited.

- Synthetic: There exists synthetic code with and without known vulnerabilities that provides benchmark data for SA. The best known datasets of this type are the Static Analysis Tool Exposition (SATE) IV dataset [18] and the Juliet Test Suite [19]. The National Institute of Standards and Technology (NIST) designed SATE IV to advance research in SA tools [20]. This dataset was later updated to the Juliet Test Suite for C/C++, a collection of test cases in the C/C++ language containing examples organized under 118 different Common Weakness Enumeration (CWE) [21] types as compiled by the MITRE Corporation. These datasets come with high-quality labels but are limited in volume. The vulnerability examples represented in these datasets are very diverse, but the code complexity is not representative of real-world code.

- Natural language: We tried various NLP techniques in an attempt to extract labels from commit messages in revision histories. These approaches did not correlate with the presence of vulnerabilities better than an *ad hoc* keyword search.

Figure 2 summarizes the performance of various types of training data relative to the labeling requirements listed above. This pictorially depicts a relative scale, with green representing the best performance relative to a given requirement, yellow in the middle, and red representing the worst performance.

Label Type	**Volume**	**Quality**	**Effort**	**Diversity**
Curated	🟥	🟩	🟨	🟥
SA-derived	🟩	🟨	🟩	🟨
Manually labeled	🟥	🟩	🟥	🟥
Artificially injected	🟨	🟩	🟨	🟥
Synthetic	🟨	🟩	🟩	🟨
Natural language	🟩	🟨	🟨	🟩

Figure 2. Labeling Performance for Various Types of Training Data

No single dataset satisfies all of the criteria for good training labels. Therefore, we combined data from several sources into a training corpus, as follows:

- Juliet Test Suite: We used this primarily for the high-quality vulnerability labeling that it provides.

- MUSE Corpus with SA-derived labels: This is a collection of GitHub projects originally provided by Leidos and later managed by Two Six Labs. The Draper team uses C and C++

code packages from this dataset. Since these datasets are not labeled in any way, we used SA to generate labels. We used this primarily to get a large volume of diverse open-source code with a large number of labels that we could generate automatically.

- Debian Linux distribution with SA-derived labels: We used this as an example of a well-curated code base to add to the training corpus.
- Debian Linux kernel with SA-derived labels: We used this primarily to expand the diversity of the training set by including kernel code.

3.2.2 Data Generation Architecture.

Upon corpus selection, the Draper team created a work flow that is based on Buildbot [22], an open source framework for automating software builds, to build each project in the corpus using a modified version of strace [23] to capture system calls associated with the build. A modified version of Clang, a C language family front end of LLVM (formerly known as Low Level Virtual Machine) [24], is used to perform a shadow build and capture build artifacts such as optimized intermediate representation (IR), control flow, use-def and def-use chains, loop features, abstract syntax tree (AST), SA, and more. These artifacts are then stored in graph form in TitanDB [25], a graph database accessed via TinkerPop [26]. TitanDB in turn resides on top of Cassandra [27], a distributed database for managing large amounts of structured data across many commodity servers.

From this data store, analytical data are migrated into Elasticsearch indexes for use by the machine learning parts of the system architecture. Because much of our dataset is not initially labeled in any way, three SA tools are used to generate a form of labeling. These three SA tools are the Clang Static Analyzer [28], Flawfinder [29], and Cppcheck [30]. While SA produces noisy labels, the use of multiple SA tools mitigates the effect of this noise on our labeling.

The Draper team constructed a cluster of about 40 low-end Dell R320 servers. We distributed the entire build infrastructure, graph storage, and analytic storage functions across these servers. Two servers are responsible for managing Cassandra and TitanDB, and 14 servers house the storage for the graph database. This graphical database is about 16 TB in size. It stores artifacts described above for thousands of C/C++ projects. The analytic store is a 7 node Elasticsearch cluster containing 12.7 billion documents taking up about 3.5 TB. These documents are JavaScript Object Notation (JSON) files containing detailed information about builds and associated analytics.

TitanDB is a graph database made up of nodes (or vertices) and edges. Figure 3 depicts the basic schema of the DeepCode TitanDB graph database structure. The schema starts with a build (as performed by Buildbot and shadowed by strace-Clang). Additional nodes in the graph include the package built, the modules that make up the package, the functions that make up each module, and the basic blocks that make up each function. Each node contains features representing the build artifacts that are relevant for network training.

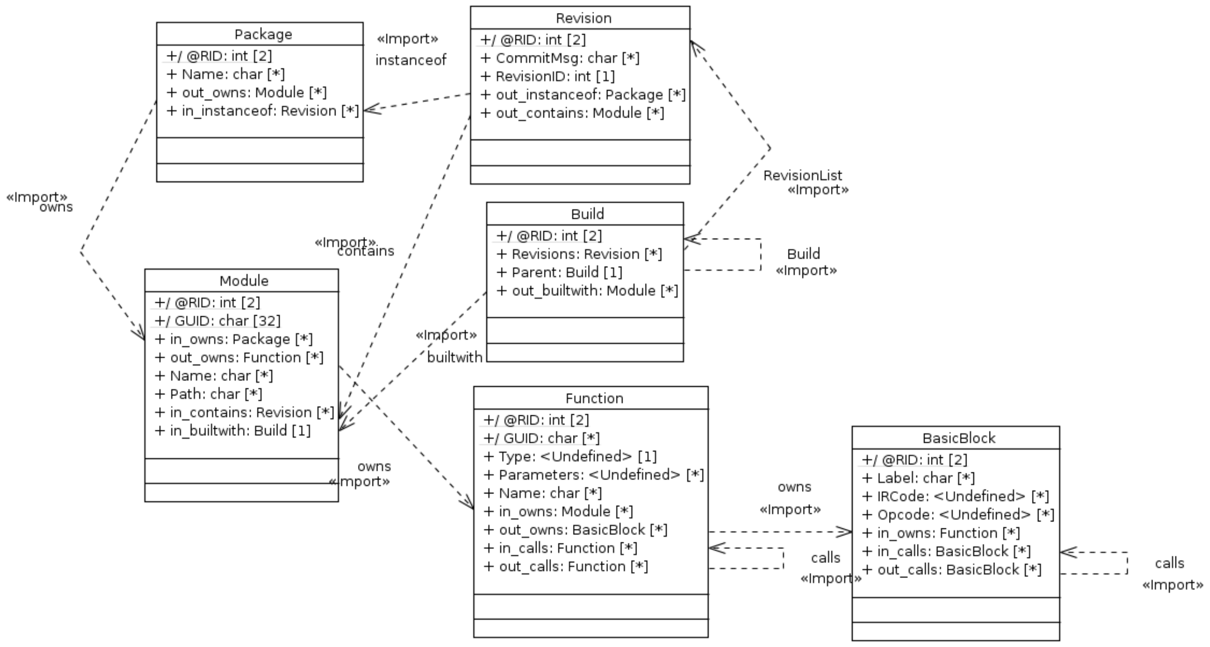


Figure 3. Graph Database Structure

Once a build has completed and generated all necessary artifacts, the ingest step ingests the information into TitanDB/Cassandra and Elasticsearch. Our Elasticsearch indexes include build step logs (Buildbot allows for several activities to be scheduled as steps in an overall build). The logs include details about the shadow build, the location of where build artifacts are stored and the results of other build steps. For example, the SA runs take place at different points in the build process. Clang SA is run while Clang is performing the build, while Cppcheck and Flawfinder take place upon completion of the build. All of these results are stored in an Elasticsearch index that contains all information associated with the build.

3.2.3 Data Generation System Components.

Figure 4 depicts the high-level system architecture for the DeepCode data pipeline. We describe each of the TA2 and TA3 elements of the system architecture in detail below.

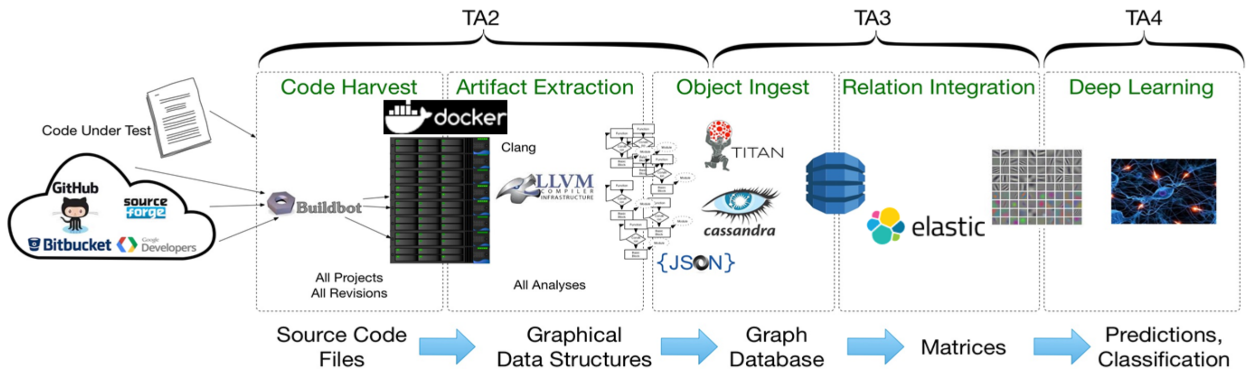


Figure 4. DeepCode Data Pipeline

Code Harvest

The Code Harvest element of the DeepCode data pipeline performs the scrape and build steps of the DeepCode functional flow. This can act either on a compressed file containing all source code associated with a project, or on a bare Git repository. The Draper team mirrored several Git repositories from GitHub on the aforementioned cluster of servers. In the scrape step, the Code Harvest element traverses these repositories, building each tag (or commit, depending on arguments supplied to the builder). The scrape and build steps are managed by a modified version of Buildbot consisting of a build master (located on one server) and several builders (located across several servers). Each builder can be a unique environment to get specific projects to build. For example, the Draper cluster consists of about 30 servers running CentOS, while 5 servers are running Ubuntu. Figure 5 depicts this build process.

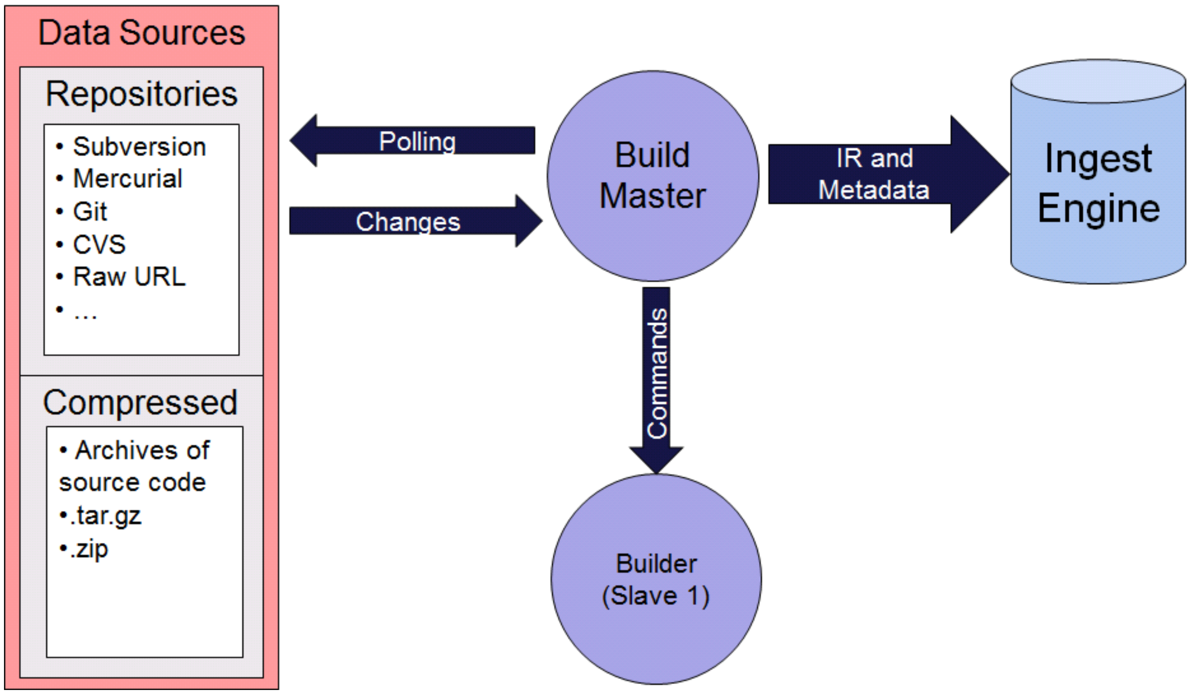


Figure 5. DeepCode Build Process

The Draper team has created several types of builders, not just in the sense of what environment in which to run, but also what type of build to perform, such as **cmake**, **make**, or **genericBash** (where various commands can be issued via a shell). Because the various Git repositories have various build methodologies, not all builds turn out to be successful. Because there are in fact two builds, there is a chance that the regular build succeeds while the shadow build fails, as the shadow build occurs in Clang in LLVM. In these cases, some modules may still succeed in building; but not all artifacts are extracted. Figure 6 shows an example from a **genericBash** build for which 218 modules were built, but only 216 were successfully extracted for storage. This illustrates that even if all modules build, they might not all have artifacts available to be extracted for use by the machine learning functions.

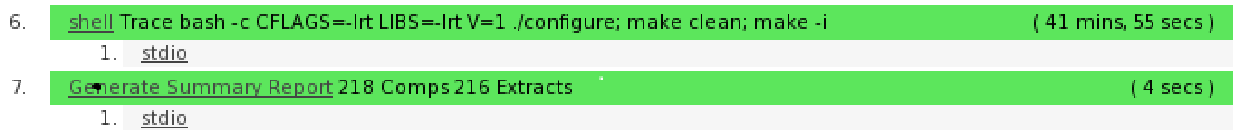

```
6.  shell Trace bash -c CFLAGS=-lrt LIBS=-lrt V=1 ./configure; make clean; make -i     ( 41 mins, 55 secs )
      1.  stdio
7.  Generate Summary Report 218 Comps 216 Extracts                                     ( 4 secs )
      1.  stdio
```

Figure 6. Sample Build Dashboard

Artifact Extraction

The Artifact Extraction generates a large JSON file that relates build artifacts to packages, builds, and revisions. This file is suitable for running through the Object Ingest element for ingest into the TitanDB server.

Object Ingest
The Object Ingest element is a Java wrapper for interacting with TitanDB. It is made up of Gremlin/TinkerPop code to create a TitanDB database, create a schema, and create and populate the vertices and edges based on JSON file generated by the Artifact Extraction element. Along with graph information concerning each build and package, metadata concerning the Buildbot build are stored in Elasticsearch. This creates a connection between the analytic store in Elasticsearch and the graph storage in TitanDB. These metadata consist of all output generated during each build step such as log output and the overall status of each step.

Relation Integration
The Relation Integration element collects build artifacts from various indices in the Elasticsearch analytic store into a summary index called a MUSE function index (MFI). The MFI contains all required information to train the Deep Learning element. This index is pulled using a Python script to generate a binary pickle file. This file is further processed to filter functions based on usability (control flow larger than zero, valid SA findings, and so on). The output of the filtered functions is then split further into three sets for training, validation, and testing. Finally, this output is written to H5 files used as inputs for the machine learning algorithms.

3.3 Machine Learning (MUSE TA4)

Beyond the traditional tools (such as SA, dynamic analysis, and symbolic execution) that attempt to uncover common software vulnerabilities, there has been significant recent work on the use of machine learning for program analysis. The large amounts of open-source code now available open the opportunity to learn the patterns of software vulnerabilities directly from mined data. Allamanis et al. [31] provide a comprehensive review of learning from “Big Code.”

In the area of vulnerability detection, Hovsepyan et al. [32] used a support vector machine (SVM) on the bag of words (BOW) representation of a simple tokenization of Java source code to predict SA labels, though their work was limited to training and evaluating on a single software repository. Pang et al. [33] expanded on this work by including n-grams in the feature vectors used with the SVM classifier. Mou et al. [34] explored the potential of Deep Learning for program analysis by embedding the nodes of the AST representations of source code and training a tree-based convolutional neural network (CNN) for simple supervised classification problems. Li et al. [35] used a recurrent neural network (RNN) trained on code snippets related to library and application programming interface (API) function calls to detect two types of vulnerabilities related to the improper usage of those calls.

To our knowledge, no work has been done on using Deep Learning to learn features directly from source code and from build features extracted from source code in a large natural codebase to detect and repair a variety of vulnerabilities. The limited datasets (in both size and variety) used by most of the previous works limit the usefulness of the results and prevent them from taking full advantage of the power of Deep Learning. Section 3.3.1 describes the high-level system architecture of our machine learning elements. Section 3.3.2 describes the data processing procedures to generate features, labels, and to remove duplicate data samples in order to train machine learning models. Sections 3.3.3 and 3.3.4 describe vulnerability detection algorithms and repair algorithms in detail. Section 3.3.5 describes exploratory research on program synthesis approach as a way to generate labeled datasets.

3.3.1 Learning System Architecture.

The pull step of the DeepCode functional flow generates binary labels ("vulnerable" and "not vulnerable"), creates feature vectors (build features and source features), and performs data curation to remove duplicate data samples. Details of labeling, features, and duplicate removal follow in Section 3.3.2. After these training data are generated, the train step of the DeepCode functional flow separately trains the classification and repair networks. The run step of the DeepCode functional flow evaluates the trained networks on benchmark datasets.

Repair networks were trained only on source features while different variations of classifier models were trained – a model that uses build features only (which we call a "build feature-based classifier"), and a model that uses source features only (which we call a "source feature-based classifier"), and a model that uses combined feature sets (which we call a "combined classifier"). Details of these classifiers and repair networks are covered in Section 3.3.3 and 3.3.4. Figure 7 shows the flow of the training process.

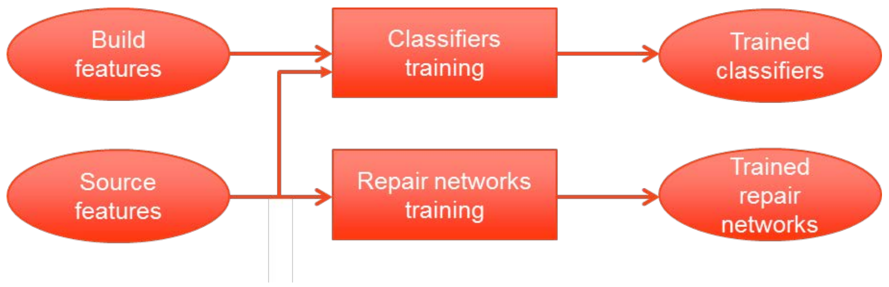


Figure 7. Overall Flow of the Training Process

Once the classifier models and repair networks were trained, we evaluated our model performance against the held-out portion of training data and against the benchmark dataset. Figure 8 shows the flow diagram of DeepCode's classification and repair processes.

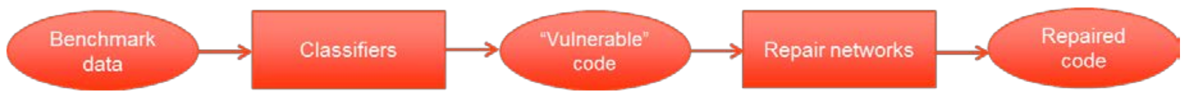


Figure 8. Flow Diagram of DeepCode's Classification and Repair Processes

3.3.2 Training.

Key components of training data generation include labeling, feature vector generation, and data curation to remove duplicate data samples and data that are otherwise unsuitable for use in training. We describe each of these components in detail below. Note that each data sample is a function-level example of C and C++ programs. We chose to analyze software packages at the function level because it is the lowest level of granularity capturing the overall flow of a subroutine.

Labeling
Labeling code vulnerabilities at the function level was a significant challenge. The bulk of our dataset was made up of mined open-source code (namely the MUSE Corpus, the Debian Linux distribution, and the Debian Linux kernel) without known ground truth. In order to generate labels, we pursued three approaches: dynamic analysis, commit-message/bug-report tagging, and SA.

Dynamic Analysis: While dynamic analysis is capable of exposing subtle flaws by executing functions with a wide range of possible inputs, it is extremely resource intensive. Performing a dynamic analysis using Draper's internal tool, Vader (version 1.0), on the roughly 400 functions in a single module of the LibTIFF 3.8.2 package from the ManyBugs dataset took nearly a day of effort. Therefore, this approach was not realistic for our extremely large dataset.

Commit Message Labeling: Commit-message labeling turned out to be very challenging, providing low-quality labels. In our tests, both humans and machine learning algorithms were poor at using commit messages to predict corresponding Travis continuous integration (CI) [36] build failures or fixes. Motivated by recent work by Zhou et al. [37], we also tried a simple keyword search looking for commit words like "buggy", "broken", "error", or "fixed" to label before-and-after pairs of functions, which yielded better results in terms of relevancy. However, this approach greatly reduced the number of candidate functions that we could label and still required significant manual inspection, making it inappropriate for our vast dataset.

SA Labeling: As a result, we decided to use three open-source SA tools, Clang [38] [28], Flawfinder [29], and Cppcheck [30], to generate labels. Each SA tool varies in its scope of search and detection. For example, Clang's scope is very broad but also picks up on syntax, programming style, and other findings which are not likely to result in a vulnerability. Flawfinder's scope is geared towards CWE [21] classes and does not focus on other aspects such as style. Cppcheck checks for memory leaks, mismatching allocation-deallocation, buffer overrun, and others with a goal of 0% false positives. Therefore, we incorporated multiple SA tools and pruned their outputs to exclude findings that are not typically associated with security vulnerabilities in an effort to create robust labels.

We had a team of dedicated security researchers map each SA tool's finding categories to the corresponding CWEs and identify which CWEs would likely result in potential security vulnerabilities. This process allowed us to generate binary labels of "vulnerable" and "not vulnerable", depending on the CWE. For example, Clang's "Out-of-bound array access" finding was mapped to "CWE-805: Buffer Access with Incorrect Length Value", an exploitable vulnerability that can lead to program crashes, so functions with this finding were labeled "vulnerable." On the other hand, Cppcheck's "Unused struct member" finding was mapped to "CWE-563: Assignment to Variable without Use", a poor code practice unlikely to cause a security vulnerability, so corresponding functions were labeled "not vulnerable" even though SA tools flagged them. Of the 390 total types of findings from the SA tools, 149 were determined to result in a potential security vulnerability. Roughly 5.1% of our curated, mined C/C++ functions triggered a vulnerability-related finding. Table 1 shows the statistics of frequent CWEs in these "vulnerable" functions; the "Frequency (%)" column in this table represents the distribution of each CWE among functions that had the CWEs listed.

Table 1. CWE Statistics of Vulnerabilities Detected in Our Dataset

CWE ID	CWE Description	Frequency (%)
120	Buffer Copy without Checking Size of Input ("Classic Buffer Overflow")	29.2%
119	Improper Restriction of Operations within the Bounds of a Memory Buffer	12.7%
469	Use of Pointer Subtraction to Determine Size	11.8%
476	NULL Pointer Dereference	10.7%
805	Buffer Access with Incorrect Length Value	5.4%
362	Concurrent Execution using Shared Resource with Improper Synchronization	3.6%
20, 234, 457, etc.	Improper Input Validation, Failure to Handle Missing Parameter, Use of Uninitialized Variable, etc.	26.5%

The Juliet Test Suite contains synthetic code examples with vulnerabilities from 118 different CWEs and was originally designed to explore the performance of static and dynamic analysis tools. While the Juliet Test Suite provides labeled examples of many types of vulnerabilities, it is made up of synthetic code snippets that do not sufficiently cover the space of natural code to provide an appropriate training set alone. The functions mined from the MUSE Corpus, Debian Linux distribution, and Debian Linux kernel appropriately provide a vast dataset of natural code to augment the Juliet Test Suite.

Program synthesis is another potential approach that could generate a large number of labeled C programs. Section 3.3.5 describes the approaches we took in this space.

Feature Generation
We extract two types of features as they offer different sources of information. The source features can provide statistical correlations in how code is written, while build features give inherent knowledge about the structure or semantics of the language in which the code was built.

Build Features: At the function level, the control flow graph (CFG) of the function is extracted. The CFG is a graph representation of the different paths a program can take during execution. Each node is a basic block – a unit of code with no branching behavior. The edges of the graph connect basic blocks that can flow into each other during execution and occur at control flow operations. At a high level, this representation is useful for vulnerability detection because models can potentially learn program execution topologies that are risky. The high-level view of the CFG is complemented by features extracted from each basic block within the CFG, so that models can also learn instruction-level behaviors that are associated with vulnerable code.

Within each basic block, we extract features which define the behavior of the basic block. The first of these features is the use-def matrix. This matrix tracks, within the basic block, the locations of instructions where variables are defined (def) and used (use). If a variable is defined at instruction i and used in instruction j, then both the (i, j) and (j, i) entries of the use-def matrix are set to 1. The second feature extracted for each basic block is the operation code vector (opvec). LLVM assigns operation codes to instructions in one of nine different categories: conditional, arrogate, binary, bit binary, conversion, memory address, termination, vector operation, and other. The opvec for a basic block is a vector that keeps counts of each of these possible classifications. Figure 9 shows the CFG and corresponding basic blocks of an example code snippet.

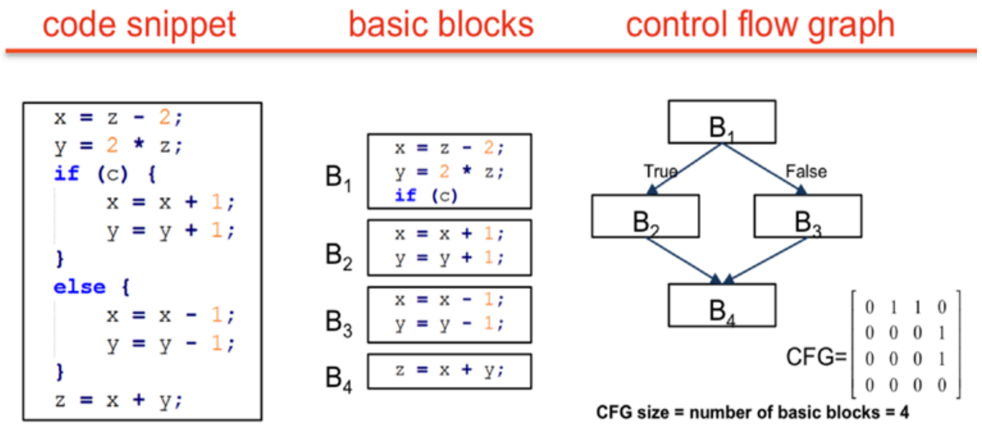


Figure 9. CFG and Basic Blocks

The size and content of the build features vary depending on the complexity of the given code snippet (function), as the CFG, the number of basic blocks, the sizes of use-def matrices, and the opvecs differ. In order to keep the input feature size manageable, and since it is unlikely for a single basic block to have a very large number of operations, we fix the size of the use-def matrix to be 15. Basic blocks with size over 15 are truncated, and basic blocks with size under 15 are padded with zeroes. Since the use-def matrix is symmetric and the main diagonal entries of the use-def matrix are equal to zero (a variable cannot be defined and used in the same instruction), we take the upper-triangular part of the use-def matrix and flatten it to a vector. As a result, a single use-def matrix can be represented as a vector of length $(15^2 - 15)/2 = 105$. Since an opvec is a vector of length 9, each basic block can be represented as a vector of length 114. The size of a build feature vector for a function of CFG size n can be represented as $n^2 + n * 114$.

Since most classifier algorithms require fixed size input vector, we took a few different approaches to convert variable-length build feature vectors to fixed size. The most straightforward way was to average basic block vectors into a single vector of length 114, then to compute the sparsity of the CFG matrix to represent the n-by-n matrix as a single value. We also added a constant variable representing the CFG size. As a result, we represented build feature vectors as fixed length vectors of length 116. We refer to this representation as the "simplified" build feature vector.

Another approach was to generate the sequence inputs that best capture the control flow of the basic blocks for the CNN and long-short term memory (LSTM). We first convert the CFG to a dominator tree, as a tree representation captures most of the control flow. (As evidence of this, we note that some compiler optimizations use dominator trees for memory usage analysis to find leaks and identify high memory usage.) After converting the CFG to a dominator tree, we still need to convert it to a sequence input. Here we use a topological sort. A topological sort is a linear ordering of the vertices of a graph such that for every directed edge e_{nd} from vertex n to vertex d, n comes before d in the ordering. This captures the ordering in which the basic blocks in a function must be performed, thus capturing the control flow of the function.

Figure 10 shows the overall flow from CFG to sequential representation of basic blocks. We first convert the CFG to a dominator tree; then traverse the tree with topological sorting to generate the sequence inputs. Some learning approaches require the input sequence lengths to be limited. For example, LSTM suffers from vanishing or exploding gradients if inputs are too long. Since only a small fraction of the functions we ingested have CFG size over 200, we fixed the number of basic blocks to be the first 200 basic blocks in topological ordering.

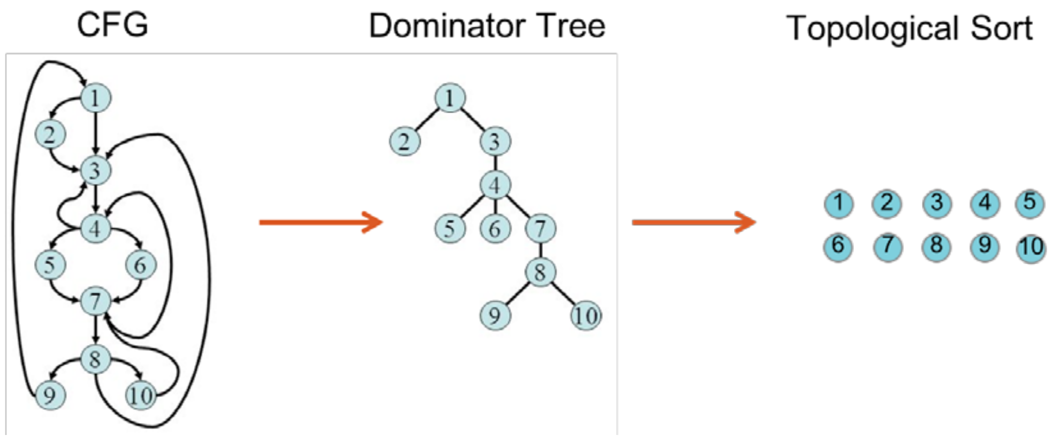


Figure 10. Topological Sorting of Dominator Tree Representation of CFG

Our "advanced" build features for each function are thus represented by an *N*-by-*K* matrix, where *N* = 200 is the number of basic blocks (we pad with zeroes for any function with fewer than 200 basic blocks) and *K* = 114 is the size of the basic block feature vector representation described above.

Source / Lexer Features: To generate useful features from the raw source code of each function, we created a custom C/C++ lexer designed to capture the relevant meaning of critical tokens while minimizing the total token vocabulary size. Standard lexers, designed for actually compiling code, capture far too much detail that can lead to overfitting in machine learning approaches. Our lexer was implemented via optimized regular expressions that allow large repositories to be lexed in seconds.

All base C/C++ keywords, operators, and separators are included in the lexer vocabulary. Code that does not affect compilation, such as comments, is stripped out. String, character, and float literals are lexed to type-specific placeholder tokens. Integer literals are tokenized digit-by-digit, as these values are frequently relevant to vulnerabilities. Common types and calls, especially ones that are likely relevant to vulnerabilities, are included. These common types and calls were discovered by looking at the most common identifiers that occur when lexing our entire dataset. All tokens that are not recognized by the lexer (such as internal variables) are mapped to generic indexed identifiers. For example, if the first variable to appear in the function is called **foo** and the second to appear is **bar**, all instances of **foo** and **bar** appearing in that function are lexed to **id1** and **id2**, respectively. This identifier indexing is needed for the source repair, as the lexed representation needs to be able to be inverted back into compilable code.

Table 2 represents our base lexer specification. The maximum base vocabulary size is the sum of the number of tokens in the base lexer specification (188 tokens in all) and the maximum number of unique identifiers in any function in the dataset. To bound the vocabulary size, we restrict attention to functions that have 500 or fewer tokens in the lexed representation. This results in a maximum base vocabulary size of 298.

Table 2. Base Lexer Specification

Operators and separators	!, !=, #, %, %=, &, &&, &=, (,), *, **, *=, +, ++, +=, ,, -, --, -=, ->, ., /, /=, :, ::, ;, <, <<, =, ==, >, >>, ?, [,], ^, {, |, |=, ||, }, ~
Integer literal digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Directives	#define, #elif, #else, #endif, #error, #if, #ifdef, #ifndef, #include, #line, #undef
Calls	calloc, cin, cout, defined, dprintk, endl, errmsg, fprintf, free, fscanf, fwscanf, len, malloc, memcmp, memcpy, memmove, memset, palloc, printf, printk, realloc, scanf, snprintf, sprintf, static_cast, strcat, strcmp, strcpy, strlen, strncmp, strncpy, swscanf
Keywords and common macros	assert, auto, break, case, catch, const, continue, default, delete, do, else, enum, errno, extern, false, for, goto, if, new, register, return, sizeof, static, std, stderr, struct, switch, this, throw, true, try, typedef, union, void, volatile, while, EOF, ERROR, FALSE, NUL, NULL, TRUE
Common types	bool, char, char_u, double, float, int, int32_t, l_int32, long, short, signed, size_t, ssize_t, string, u16, u32, u64, u8, uint, uint16, uint16_t, uint32, uint32_t, uint64, uint64_t, uint8, uint8_t, unsigned, BOOL, BYTE, DWORD, FILE, PyObject, QString, UBool, UErrorCode, UINT, UINT16, UINT32, UINT64, UINT8, U_FAILURE, UnicodeString, WORD
Other literals	characters, strings, floats

Unlike the repair algorithms, our classification algorithms do not require that the lexed representation be invertible. Thus, to reduce the vocabulary size as much as possible and reduce overfitting, we map many similar terms onto the same token. Learned embeddings of these individual tokens would likely distinguish them based on the kind of code they are commonly used in, so care was taken to build in the desired invariance. Making our lexed representation of code from different software repositories as standardized as possible empowers transfer learning across the full dataset.

For this generic representation, all identifiers are mapped onto the same identifier placeholder tokens instead of being indexed. This reduces the maximum vocabulary size down from 298 to 188. Additionally, we mapped some tokens (particularly types and function calls) with identical or nearly identical meanings onto the same token. This helps ensure that similar C and C++ code, or similar code using different common libraries, still have similar representations. For example, **u32**, **uint32_t**, **UINT32**, **uint32**, and **DWORD** are all lexed as the same generic token representing 32-bit unsigned data types. This generic representation is able to reduce the vocabulary size to only 156 tokens. Table 3 summarizes generic lexer representation mapping.

Table 3. Generic Lexer Representation Mapping

From	To
UINT	uint
u8, UINT8, uint8_t, char_u, BYTE	uint8
u16, UINT16, uint16_t, WORD	uint16
u32, UINT32, uint32_t DWORD	uint32
u64, UINT64, uint64_t	uint64
l_int32, int32_t	long
UBool	bool
BOOL	int
TRUE	0
FALSE	1
ssize_t	size_t
palloc	malloc
QString	UnicodeString
U_FAILURE, PyObject, errmsg	id
dprintk, printk	printf
UErrorCode	enum

Combined Feature Sets: Since the source and build features can capture different aspects of our input data, we created combined features sets to explore how these features can be used together for improved classification performance. The most straightforward way of combining the feature sets is by simply concatenating the "simplified" build feature vector and lexed feature vector of the source code. For each feature type, we also learn neural feature representations using CNN or

RNN. This allowed us to create many different combinations of source and build features by taking one feature vector from each column of options listed in Table 4.

Table 4. Feature Vector Combinations

Build Feature Vector Options	Source Feature Vector Options
• "Simplified" build feature vector • CNN learned build feature vector • RNN learned build feature vector	• CNN learned source feature vector • RNN learned source feature vector • BOW representation of lexed source feature vector

Data Curation – Filtering and Duplicate Removal
Data curation/preprocessing is a very important part of the machine learning process as having erroneous data samples, data samples with many missing feature values, or duplicate data samples can interfere with properly training good machine learning models. We perform filtering and duplicate removal as the main data curation processes.

During the filtering process, we filtered out functions that were missing source code caused by data extraction failure (since our Cassandra query stops at the first dependency, the source codes related to the functions in deeper dependency are not extracted), functions that were unreasonably long (lexed representation length over 500) or short (lexed representation length less than 2), and functions where our custom lexer failed to generate a lexed representation from the source code (when non-standard characters, commonly "@" and "$", are used in identifiers).

A subsequent critical step of our data preparation is the removal of potential duplicate functions. Open-source repositories often have functions duplicated across different packages. Such duplication can artificially inflate performance metrics and conceal overfitting, as training data can leak into test sets. Likewise, there are many functions that are near duplicates, containing trivial changes in source code that do not significantly affect the execution of the function. These near duplicates are challenging to remove, as they can often appear in very different code repositories and can look quite different at the raw source level.

To protect against these issues, we performed an extremely strict duplicate removal process. We removed any function with a duplicated lexed representation of its source code (source / lexer features) *or* a duplicate build feature vector. Two functions with identical instruction-level behaviors or functionality are likely to have both similar lexed representations and highly correlated vulnerability status. For our datasets, the functions remaining after the data curation processes (filtering and duplicate removal) were about 10% of the total number of functions (details are provided in Section 4.1.2). Our strict duplicate removal process filters out a significant amount of data and results in diminishing returns with each new dataset, as the likelihood of duplicates increases with the size of the dataset. Therefore, this approach provides *the most conservative performance* results, closely estimating how well our tool performs against code it has never seen before.

3.3.3 Classification.

We train separate classifiers that use varying amounts of information: build feature-based classifiers that operate on build features, source feature-based classifiers that work with sourcefeatures, and combined classifiers that utilize both build and source features. By pursuing all of these approaches, we allow the possibility of ultimately fusing multiple models to increase detection performance. Each of our classification models produces a continuous output between 0 and 1 that we can threshold to obtain a binary prediction indicating a bad ("vulnerable") or good ("not vulnerable") function. We tuned and selected models based on the highest validation Matthews Correlation Coefficient (MCC), a classification metric insensitive to class imbalance, which is described in further detail in Section 4.2.1. Each of our primary classifier approaches are described in detail below.

Build Feature-Based Classifiers

In addition to traditional machine learning algorithms such as random forest (RF), we explored two Deep Learning network architectures for "advanced" build feature representations: RNN (specifically the LSTM network) and CNN. Both networks are commonly used in several application domains including sentiment analysis [39] [40] [41], photo tagging and image classification [3] [4] [5] [6] [7] [42], machine translation and correction [43] [44] [45] [46], and autonomous vehicles [47].

RNN Using LSTM Network: Figure 11 illustrates our LSTM approach to build feature-based classification. The inputs are the "advanced" build features for each function represented in an $N \times K$ matrix, where N is the number of basic blocks in topological sorted ordering and K is the feature vector representation of the basic block. Recall from Section 3.3.2 that we use $N = 200$ and $K = 114$. This input matrix is fed into a multi-layer LSTM, and the hidden state from the last hidden layer outputs whether the function is vulnerable or not.

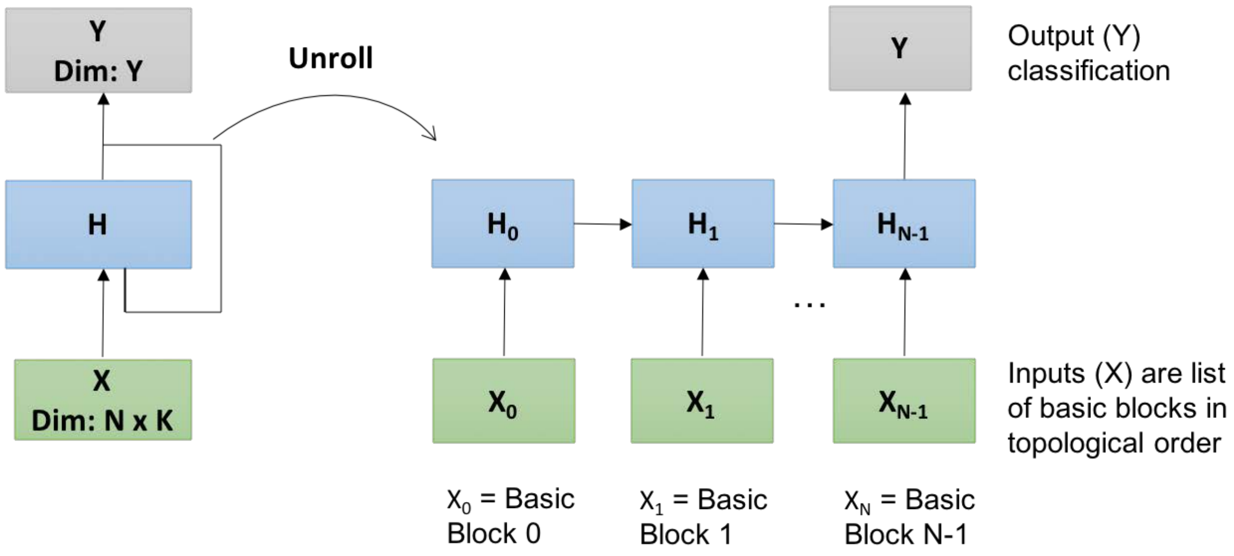


Figure 11. LSTM Network Used for Build Feature-Based Classification

Using RNNs allows for the capture of longer control flow dependencies in a function. RNNs take not only the current input into consideration but also what it has learned from the inputs previously through hidden states, and thus works well for sequence inputs. Vanilla RNNs often suffer from the vanishing gradient or exploding gradient problems, but LSTM enables RNNs to remember their inputs over a long period of time, because LSTM contain their information in a memory cell. Our "advanced" build feature representation is fed into a two-layer LSTM with hidden state size of 200, and the output from the last hidden layer at the length $N = 200$ is fed into a classification layer. The classifier uses a softmax (normalized exponential) output [48] to make predictions between 0 and 1 for the two classes (not vulnerable or vulnerable). Since it is a binary classification problem, logarithmic (cross-entropy) loss is used as the loss function, but penalized/weighted more heavily on the vulnerable class due to the imbalanced class problem with vulnerable functions being only a small percentage of the total dataset. To train the network, we use the Adam optimizer [49], a variant of the stochastic gradient descent algorithm, with a learning rate 1e-3, dropout 0.8, and batch size 128. In addition, we experimented with bi-directional LSTM instead of LSTM as well as average pooling all hidden layers instead of just output from last hidden layer – these achieve similar performance.

CNN: CNNs can capture correlation in the near neighbors by applying convolutional filters over data. They have shown lots of amazing results in computer vision, and they have also been applied in audio classification with good results [50]. We use n convolutional filters with shape $m \times K$. The filter size m determines the number of sequential basic blocks that are considered together and we found that a filter size of $m = 4$ effectively captures the correlation of the neighbor basic blocks. A total number of $n = 512$ filters was used to achieve best classification results. The convolutions were followed by batch normalization (which records data statistics to keep layer outputs appropriately normalized) and followed by the rectified linear unit (ReLU) non-linearity layer. After that, convolutional features are downsampled by applying a maximum filter along a given dimension via an operation known as *maxpooling*. This is further followed by two fully connected dense layers, where each layer's output is a linear combination of the values from the previous layer, plus a non-linearity, which applies dimension reduction. We also apply *dropout* to the fully connected layers, where some fraction of connections are randomly dropped out in each training step, thus preventing the network from overfitting. We used 50% dropout on the maxpooled feature representation connections to the first hidden layer when training. We found that using two hidden layers of size 64 and 16 (the number of hidden states per layer) before the final 2-dimensional output gave the best classification performance. Just like LSTM, a softmax function is applied to these outputs to generate interpretable "probabilities" of each function being vulnerable or not vulnerable.

Network Training: Both LSTM and CNN networks were trained with the Adam optimizer with batch size 128 and learning rate of 1e-3. We trained each network by minimizing the cross-entropy loss of the outputs with respect to the true vulnerability label. Since the dataset was strongly unbalanced, vulnerable functions were weighted more heavily in the loss function. This weight is one of the many hyper-parameters that needed to be tuned to get the best performance.

Source Feature-Based Classifiers

Since source code shares some commonalities with writing and work done for programming languages is more limited, we build off approaches developed for NLP [41]. We leverage feature-extraction approaches similar to those used for sentence sentiment classification with CNNs and RNNs for function-level source vulnerability classification.

Figure 12 illustrates our convolutional neural representation-learning approach to source code classification. This approach combines the neural feature representations of lexed function source code with RF, a powerful ensemble classifier. Input source code is lexed into a token sequence of variable length $\boldsymbol{l}$, embedded into a $\boldsymbol{l} \times \boldsymbol{k}$ representation, filtered by $\boldsymbol{n}$ convolutions of size $\boldsymbol{m} \times \boldsymbol{k}$, and maxpooled along the sequence length to a feature vector of fixed size $\boldsymbol{n}$. The embedding and convolutional filters are learned by weighted cross entropy loss from fully connected classification layers. The learned $\boldsymbol{n}$-dimensional feature vector is used as input to an RF classifier, which improves performance compared to the neural network classifier alone.

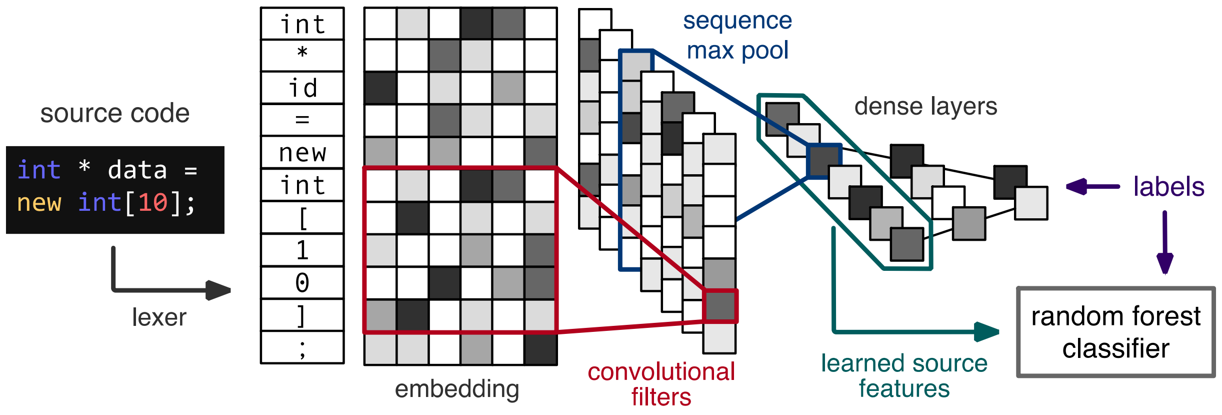


Figure 12. Convolutional Neural Representation Learning for Source Classification

Our neural representation-learning approach to source feature-based classification contains the following key steps:

1. *Embedding:* The tokens making up the lexed functions are first embedded into a fixed k-dimensional representation (limited to range [-1, 1]) that is learned during classification training via backpropagation to a linear transformation of a one-hot embedding. We also tried a fixed one-hot embedding, but this approach overfit more, resulting in lower MCC. As our vocabulary size is much smaller than those of natural languages, we were able to use a much smaller embedding than is typical in NLP applications. Our experiments found that $k = 13$ performed the best for supervised embedding sizes, balancing the expressiveness of the embedding against overfitting, while typical NLP applications use embedding sizes between 300 and 500.

    a. *Embedding initialization*: Several unsupervised word2vec approaches [51] trained on a much larger unlabeled dataset were explored for seeding this embedding. A word2vec model learns a vector representation of “words” (in our case, tokens) by predicting which word occurs based on surrounding words. Thus, words that are used in similar ways occur near each other by Euclidean distance in the representation. This learned embedding did seem to take on significant meaning for our lexer tokens and is shown by a t-distributed stochastic neighbor embedding (t-SNE) visualization in Figure 13. Unfortunately, seeding the embedding used by the classifier with this learned embedding yielded minimal improvement in classification performance over randomly-initialized directly-learned embeddings. It is likely that the most important latent variables for vulnerability detection are significantly different from the most important ones for function reconstruction and since we were limited to a relatively small embedding dimension size, there ended up not being much overlap.

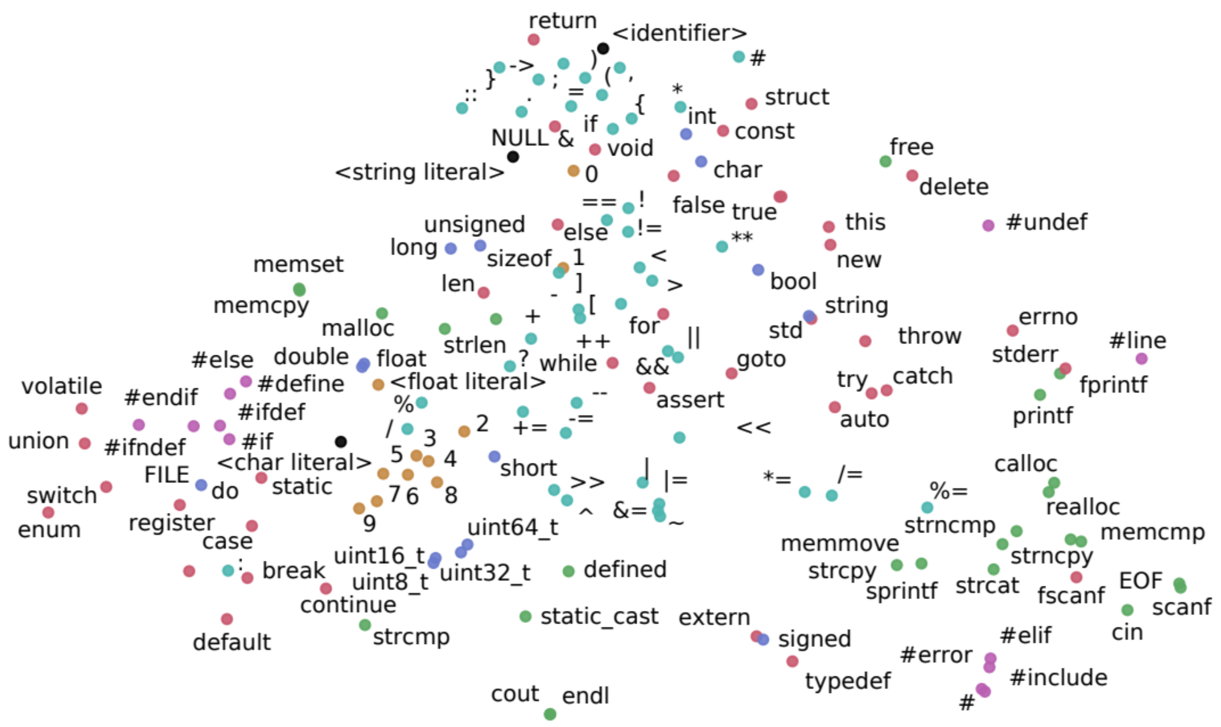


Figure 13. t-SNE Visualization of a 10-Dimensional word2vec Embedding of Lexer Tokens

b. *Embedding regularization*: Even though our dataset is very large, overfitting was a major problem for our most powerful network architectures. When training neural networks on images, it is common to perform data augmentation (such as random cropping, rotations, or color adjustments) to prevent overfitting, but this is not possible with our data modality. Instead, we found that adding a small amount of random Gaussian noise $\mathcal{N}(\mu = 0, \sigma^2 = 0.01)$ to each embedded representation substantially improved resistance to overfitting and was much more effective than other more common regularization techniques such as weight decay.

2. *Feature extraction*: We explored both CNNs and RNNs for feature extraction from the embedded source representations.

a. *Convolutional feature extraction*: Neural network convolutions are a powerful way of learning effective convolutional filter operations over data and have been extremely successful in computer vision. We took inspiration from Kim's work [41], which showed that learned convolutions could be very effective feature extractors for natural language classification problems. We use n convolutional filters with shape $m \times k$, so each filter spans the full space of the token embedding. The filter size m determines the number of sequential tokens that are considered together and we found that a fairly large filter size of $m = 9$ maximized MCC during hyperparameter tuning. This size effectively represents the smallest token length of critical code snippets that can be considered separately. A total number of $n = 512$ filters, corresponding to the number of code snippet classes detected, maximized MCC during hyperparameter tuning. As with the build feature-based CNN, the convolutions were followed by batch normalization and ReLU non-linearity.

   b. *Recurrent feature extraction*: We also explored using RNNs for feature extraction to allow longer token-dependencies to be captured. RNNs process sequences step by step, using information from previous steps, and thus operate like learned finite state machines and can handle arbitrarily long sequences. Our embedded representation is fed into a multi-layer RNN and the output at each step in the length l sequence is concatenated. We used two-layer Gated Recurrent Unit RNNs with hidden state size of 256, though LSTM RNNs performed equally well. Both of these RNN architectures have a notion of both state and memory that make them more effective for long sequences.

3. *Pooling*: As the length of C/C++ functions found in the wild can vary dramatically, both the convolutional and recurrent features are maxpooled (keeping the largest values) along the sequence length l in order to generate a fixed-size (n or n', respectively) representation, as was done for the build feature-based RNN and CNN. In this architecture, the feature extraction layers should learn to identify different signals of vulnerability and thus the presence of any of these along the sequence is important.

4. *Dense classification layers*: The feature extraction layers are followed by a fully connected classifier, where each value in a layer is a linear combination of the values in the previous layer, plus a non-linearity. We once again used 50% dropout on the maxpooled feature representation connections to the first hidden layer when training as well as two hidden layers of sizes 64 and 16 before the final softmax output to generate interpretable "probabilities" of each function being vulnerable or not vulnerable.

5. *Network training*: As with the build feature-based networks, the source feature-based networks were trained using the Adam optimizer and a class-weighted cross-entropy loss function. For data batching convenience, we trained only on functions with token length $10 \leq \ell \leq 500$ zero-padded to the maximum length of 500. Both the convolutional and recurrent networks were trained with batch size 128 (which allowed every minimization step to include some vulnerable functions) and learning rates of 5e-4 and 1e-4, respectively.

6. *Ensemble learning on neural representations*: While the neural network approaches automatically build their own features, their classification performance on our full dataset was suboptimal. We found that using the neural features (outputs from the sequence-maxpooled convolution layer in the CNN and sequence-maxpooled output states in the RNN) as inputs to a powerful ensemble classifier such as RF or extremely randomized trees yielded the best results on our full dataset. These classifiers are both based on decision trees, which learn the best combinations of univariate decision boundaries to separate our two data classes. The ensemble variations of these take advantage of the observation that populations of randomized weak classifiers are usually superior to a single strong classifier. Optimizing the neural features and ensemble classifier separately also makes it more convenient to retrain a classifier quickly on new sets of features or combinations of features. Our most effective ensemble classifiers used an RF with 300 trees and a minimum of 7 data samples per split.

7. *Error localization using backpropagation and class activation mapping*: The source feature-based classification has an advantage over the build feature-based classifications in that the data format is inherently more understandable to humans. However, discovering why neural networks make the decisions they do from these features is a challenge. We explored several ways of determining which tokens in an input function were most important to a classification result, inspired by approaches used in the computer vision domain. First, we tried guided backpropagation [52], where the derivatives with respect to the outputs are backpropagated all of the way to the initial one-hot vector inputs. While this provides a very fine-grained way of visualizing the impact each token has on the output, the visualization of the magnitude of these derivatives tends to be difficult to interpret. More successful was using an approach similar to class activation mapping [53], where the gradients are merely backpropagated to the output of the convolutional layers (right before the maxpooling operation.) Then, each feature earns a weight based on how much it contributes to the final result and the contribution of each feature is summed for every position along the sequence, weighted by these amounts. This is then deconvolved onto the initial sequence positions. Section 4.2.4 shows some examples of error localization using the class activation mapping. This technique produces a rough "heat map" of the main output class, as shown in the figures in Section 4.2.4. While this is "blurrier" than backpropagation all of the way to the input vectors (*i.e.*, the position is smeared via a convolutional filter), it results in somewhat more interpretable visualizations. Visualizations such as this can help localize the origin of a vulnerability and can make the tool more helpful to developers.

Combined Classifier(s)

Since the source and build features can capture different aspects of our input data, it is interesting to explore how these features can be used together for more effective and robust classifiers. If our learned classifiers can be interpreted as detecting signals of vulnerabilities in their respective feature sets, the classifier having access to another view of the same data could significantly improve its ability to detect vulnerabilities.

The most straightforward way of learning on both sets of features is by simply concatenating unordered feature representations. For the build features, this is the "simplified" feature vector. For the source features, this is the learned neural feature representation, from either the CNN or

RNN classifier. Both of these sets of features were used effectively individually with the RF classifier and thus it is reasonable to use the RF classifier trained on the combined set of features.

While the classifier has access to more information, it is not guaranteed that the RF classifier trained on the combined feature set is significantly better than one trained on the individual feature representations. If both feature sets contain overwhelmingly redundant information, improvement is likely to be minimal, and the risk of overfitting the data increases. Likewise, if classification with one feature set is much better than classification from the other, the benefit is not likely to outweigh the cost of increasing the dimensionality of the classification problem.

Ensemble Approaches

Ensembling is the approach of combining the outputs of multiple different classifiers to boost the performance. The goal is to obtain improved performance over individual classifiers. This is best achieved when individual classifiers are statistically independent, *i.e.*, they make independent errors. To combine the strengths of different classification approaches we developed, we investigated various ensembling approaches:

- Convex combination of normalized (probability) scores (also known as linear opinion pool): This is the simplest and the most effective ensembling approach for our problem, where the final (ensembling) score is computed as the weighted average of individual classifier scores:

$$P_f = w_1 P_1 + \cdots + w_N P_N, \tag{1}$$

  where $\sum_i w_i = 1$, P_f is the final classification score, P_i is the score of classifier i, and N is the number of classifiers to be ensembled.

- Logistic regression: This is similar to the convex combination approach except that the weights do not have to sum to 1, and the final weighted score is passed through a logistic function to ensure that $0 \leq P_f \leq 1$:

$$P_f = 1/\left[1 + e^{-(w_1 P_1 + \cdots + w_N P_N)}\right] \tag{2}$$

- Copula-based ensembling: This approach is inspired by [54], where we develop a generative model by using copula functions to fit a joint probability distribution function to the different classifier probabilities under each class. This is achieved by using a maximum likelihood approach combined with kernel density estimation. Then we invoke Bayes' Theorem to compute the posterior probability of each class. Specifics of this approach are outlined in [54]. Although this approach may improve the performance over other ensembling approaches, it works best when there are only two classifiers to be ensembled. When there are more than two classifiers, this approach is computationally very expensive to implement which requires several approximations such as vine copulas.

In all of the ensembling approaches that we adopted, we held out 2% of total samples as an ensemble set to train the ensemble models. More specifically, we train each individual classifier using the training set (78% of total samples). Then we compute classifier scores on the ensemble set samples and use these probabilities along with the ground truth labels to optimize as follows:

- For convex combination, we find the optimal weights using a simple grid search.
- For logistic regression, we find the optimal weights using standard gradient descent.
- For copula-based ensembling, we find the best fitted copula functions using a maximum likelihood approach.

3.3.4 Repair.

We describe repair approaches that take “vulnerable” source code and output the repaired (“not vulnerable”) source code. We first describe the state-of-the-art method known as the sequence-to-sequence approach, then provide details of our new GAN approach, that addresses the main constraint of the sequence-to-sequence approach. We also discuss a variation of the GAN approach called cycle-GAN.

Sequence-to-Sequence

The problem of repair of source code shares many similarities to the problem of grammar correction in NLP, in which a grammatically incorrect sentence is translated into a correct one. In our case, bad (“vulnerable”) source code takes the place of an incorrect sentence and is repaired into good (“not vulnerable”) source code.

Sequence-to-sequence systems have recently achieved the state-of-the-art performance on language translation and correction tasks [43] [44] [45] [46]. These models use an encoder-decoder approach to transform an input sequence $\boldsymbol{x} = (x_0, \dots, x_T)$ into an output sequence $\boldsymbol{y} = (y_0, \dots, y_{T'})$, *e.g.*, translating a sequence of words forming a sentence in English to one in German.

However, by far the most common method of training sequence-to-sequence systems is to use labeled pairs of examples to compare the likelihood of network output to a desired version, necessitating a one-to-one mapping between input and desired output data. During Phase 2 of the MUSE program it became clear that the number of labeled paired examples we could obtain in a reasonable time frame was so limited as to make this approach intractable for code vulnerability repair. This led us to develop the new GAN approach.

Generative Adversarial Network (GAN)

Our GAN approach to repair allows us to train without paired examples. GANs are generative models that were originally developed to generate realistic images, y, from random noise vectors, z [55]. GANs find a mapping $G: z \to y$ by framing the learning problem as a two player minimax game between a generator $G(\cdot)$ and a discriminator $D(\cdot)$, where the generator learns to generate realistic looking data samples by minimizing the performance of a discriminator whose goal is to maximize its own performance on discriminating between generated and real samples.

We employ a traditional sequence-to-sequence model as the generator and replace the typical negative likelihood loss with the gradient stemming from the loss of an adversarial discriminator. The discriminator is trained to distinguish between outputs generated by the sequence-to-sequence model and real examples of desired output, and so its loss serves as a proxy for the discrepancy between the generated and real distributions.

This problem has three main difficulties. First, sampling from the output of sequence-to-sequence systems, in order to produce discrete outputs, is non-differentiable. We address this problem by using a discriminator which operates directly on the expected (soft) outputs of the sequence-to-sequence system during training. Second, adversarial training does not guarantee that the generated code corresponds to the input bad code (*i.e.*, the generator is trained to match distributions, not samples). To enforce the generator to generate useful repairs, (*i.e.*, generated code is a repaired version of input bad code), we condition our sequence-to-sequence generator on the input x by incorporating two novel generator loss functions. Third, the domains we consider are not bijective, *i.e.*, a bad code can have more than one repair or a good code can be broken in more than one way. The regularizers we use still work in this case.

GANs were first introduced by Goodfellow et al. [56] to learn a generative model of natural images. Since then, many variants of GANs have been created and applied to the image domain [57] [58] [59] [60] [61]. GANs have generally focused on images due to the abundance of data and their continuous nature. Applying GANs to discrete data (*e.g.*, text) poses technically challenging issues not present in the continuous case (*e.g.*, propagating gradients through discrete values). One successful approach is that of Yu et al. [62], which treats the output of the discriminator as a reward in a reinforcement learning setting. This allows the sampling of outputs from the generator since gradients do not need to be passed back through the discriminator. However, since a reward is provided for the entire sequence, gradients computed for the generator do not provide information on which parts of the output sequence the discriminator thinks is incorrect, resulting in long convergence times. Several other approaches have had success with directly applying an adversarial discriminator to the output of a sequence generator with likelihood output. Zhang et al. [63] replace the traditional GAN loss in the discriminator with a maximum mean discrepancy (MMD) metric in order to stabilize GAN training. Both Press et al. [64] and Rajeswar et al. [65] are able to generate fairly realistic looking sentences of modest length using Wasserstein GAN [58], which is the approach we adopt in our work.

Work has also been done on how to condition a GAN's generator on an input sequence x instead of a random variable. This can easily be performed when paired data are available, by providing the discriminator with both x and y, thereby formulating the problem as in the conditional approach of Mirza and Osindero [66] [67]. This approach, however, is clearly more difficult when pairs are not available. One approach is to enforce conditionality through the use of dual generator pairs which translate between domains in opposite directions. Gomez et al. apply the cycle GAN [68] approach to cipher cracking [69]. They train two generators, one to take raw text and produce ciphered text, and the other to undo the cipher. Having two generators allows Gomez et al. to encrypt raw data using the first generator, then decrypt the data with the other, ensuring conditionality by adding a loss function which compares this doubly translated output with the original raw input. Lample et al. [70] adopt a somewhat similar approach for neural machine translation. They translate using two encoder/decoder pairs which convert from a given language to a latent representation and back respectively. They then use an adversarial loss to ensure that the latent representations are the same between both languages, thus allowing translation by encoding from one language and then decoding into the second. For conditionality they adopt a similar approach to Gomez et al. by fully translating a sentence from one language to another, translating it back, and then comparing the original sentence to the produced double translation.

The approaches of both Gomez et al. and Lample et al. rely on the ability to transform a sentence across domains in both directions. This makes sense in many translation spaces as there are a finite number of reasonable ways to transform a sentence in one language to a correct one in the other. This allows for a network which finds a single mapping from every point in one domain to a single point in the other domain, to still cover the majority of translations. Unfortunately, in a sequence correction task such as our problem, one domain contains all correct sequences, while the other contains everything not in the correct domain. Therefore, the mapping from correct to incorrect is not one-to-one, it is one to many. A single mapping discovered by a network would fail to elaborate the space of all bad functions, thus enforcing conditionality only on the relatively small set of bad functions it covers. As such we enforce conditionality using a self-regularization term on the generator, similar in nature to the one used by Shrivastava et al. [71] in which they generate realistic looking images from simulated ones.

We should note that our problem here is different from the original GAN problem in that our goal is to find a mapping between two discrete valued domains, namely between a given bad code (or source) domain X and a good code (or target) domain Y by using unpaired training samples $\{x_i\}_{i=1}^{N}$ and $\{y_i\}_{i=1}^{M}$, where $x_i \in X$ and $y_i \in Y$.

The original GAN loss of Goodfellow et al. [55] is expressed as

$$L_{GAN}(D,G) = E_{y\sim P(y)}[\log D(y)] + E_{x\sim P(x)}[\log(1 - D(G(x)))] \tag{3}$$

where the optimal generator is $G^* = \arg min_G max_D\, L_{GAN}(D,G)$. It is well known that this loss can be unstable when the support of the distributions of generated and real samples do not overlap [57]. This causes the discriminator to provide zero gradients. Further, this standard loss function can lead to mode collapse, where the resulting samples come from a single mode of the real data distribution. To alleviate these problems, Arjovsky et al. [58] proposed the Wasserstein Generative Adversarial Network (WGAN) loss which instead uses the Wasserstein-1 or Earth Movers (EM) distance between generated and real data samples in the discriminator. EM distance is relatively straightforward to estimate using the following easily computable loss function:

$$L_{WGAN}(D,G) = E_{y\sim P(y)}[D(y)] - E_{x\sim P(x)}[D(G(x))] \tag{4}$$

We use WGAN in our model as it leads to more stable training.

In the context of source code repair, or more generally sequence correction, we need to constrain our generated samples $G(x)$ to be corrected versions of x. Therefore, we have the following two requirements: (1) correct sequences should remain unchanged when passed through the generator; and (2) repaired sequences should be close to the original corresponding incorrect input sequences.

We explore two regularizers to address these requirements. As our first regularizer, in addition to GAN training, we train our generator as an auto encoder (AE) on data sampled from correct sequences. This directly enforces item (1), while indirectly enforcing item (2) since the AE loss encourages subsequences which are correct to remain unchanged. The AE regularizer is given as

$$L_{AUTO}(G) = E_{x\sim P(x)}[-x \log G(x)] \tag{5}$$

As our second regularizer, we enforce that the frequency of each token in the generated output remains close to the frequency of the input tokens. This enforces item (2) with the exception that it may allow changes in the order of the sequence, *e.g.*, arbitrary reordering does not increase this loss. However, the GAN loss alleviates this issue since arbitrary reordering produces incorrect sequences which differ heavily from $P(\boldsymbol{y})$. Our second regularizer is given as

$$L_{FREQ}(G) = E_{x\sim P(x)}[\textstyle\sum_{i=0}^{n} \|\mathrm{freq}(x,i) - \mathrm{freq}(G(x),i)\|_2^2] \tag{6}$$

where n is the size of the vocabulary and $\mathrm{freq}(x,i)$ is the frequency of the i-th token in x.

A block diagram for the GAN architecture we use is shown in Figure 14. The network is provided with samples of vulnerable code (x) and good code (y). Note, these samples are independent and do not have to be drawn from pairs of vulnerable and correpondingly repaired code. The vulnerable code samples are provided as input to the generator, which uses an encoder and decoder approach to generate an output sequence $G(x)$. Note that the generator architecture here is identical to a traditional sequence-to-sequence model. The generated code ($G(x)$) and good code (y) are then fed into the discriminator. One variant of our GAN approach uses curriculum learning, in which we clip the generated and good sequences (y and gx, respectively) to a specific length determined by the current curriculum. This behavior is controlled by a curriculum controller module, shown as orange as the "Cur Controller" block.

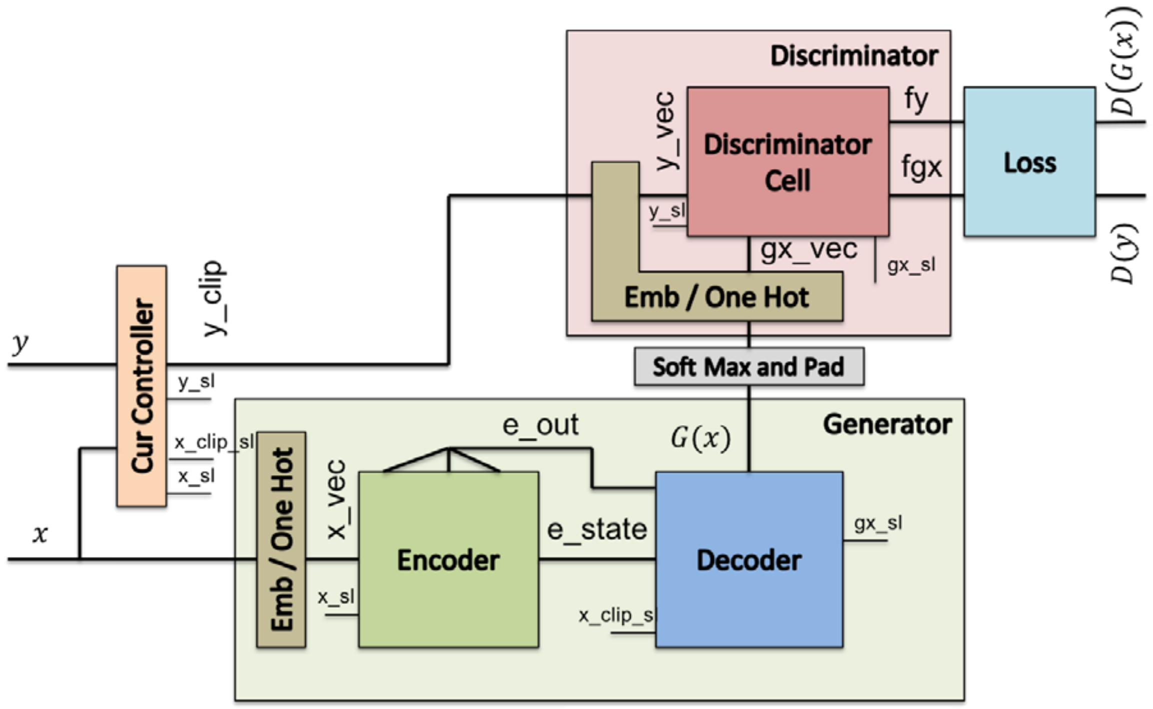


Figure 14. Block Diagram of GAN Architecture

Both the encoder and decoder of the generator consist of multiple RNN layers with LSTM hidden states. These RNN layers form a “cell” which for both encoder and decoder takes an input at step t and its own state from the previous step $t-1$ to produce an output for step t, and are shown in Figure 15. Tokenized vulnerable code is used as the input to the encoder with input x_t being the t’th token. The decoder then generates an output sequence $G(x)$ one token at a time, with the input to decoder step t being the generated token from step $t-1$. In addition to RNN layers our decoder also uses a scaled dot product attention mechanism [72] to look back across all encoder outputs, allowing easier passing of information from the encoder to decoder.

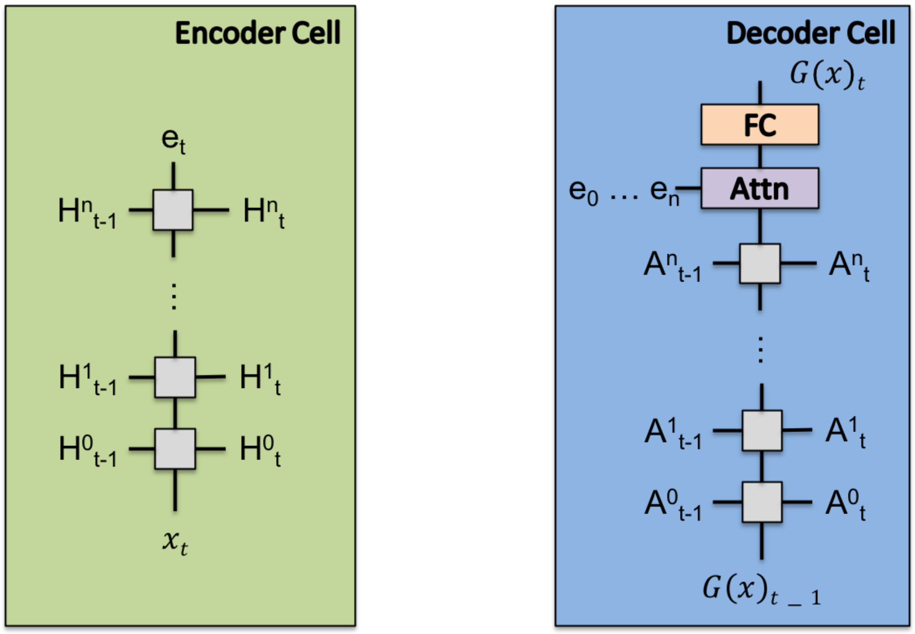


Figure 15. Block Diagram of Encoder (Left) and Decoder (Right) RNN Cells

The discriminator in our network is a simple sequence classification network consisting of a convolution layer, a temporal max pooling layer and two fully connected layers. This is shown in Figure 16. Inputs are passed through the convolution layer, which finds patterns across multiple time steps. The outputs from the convolution are then passed through a temporal max pool in order to convert them to a fixed length vector. Finally, these outputs are processed by the fully connected layers to produce a single output, which is used in the computation of the WGAN loss described above ($D(y)$ and $D(G(x))$ for good and generated inputs respectively).

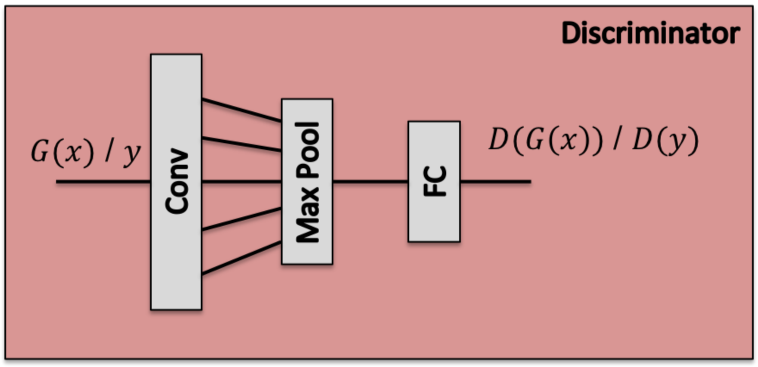


Figure 16. Block Diagram of Decoder

We have two different regularized loss models given as

$$L(D,G) = L_{WGAN}(D,G) + \lambda L_{AUTO}(G) \tag{7}$$
$$L(D,G) = L_{WGAN}(D,G) + \lambda L_{FREQ}(G) \tag{8}$$

We also experimented with the unregularized base loss model where we set $\lambda = 0$.

We rely heavily on pre-training to give our GAN a good starting point. Our generators are pre-trained as de-noising AEs on the desired data [73]. Specifically, we train the generator with the loss function:

$$L_{AUTO\_PRE}(G) = E_{y \sim P(y)}[-y \log G(\hat{y})], \tag{9}$$

where $\hat{y}$ is the noisy version of the input created by dropping tokens in y with probability 0.2 and randomly inserting and deleting n tokens, where n is 0.03 times the sequence length. These numbers were selected based on hyperparameter tuning.

Likelihood-based methods for training sequence-to-sequence networks often utilize teacher forcing during training, where the input to the decoder is forced to be the desired value regardless of what was generated at the previous time step [74]. This allows stable training of very long sequence lengths even at the start of training. Adversarial methods cannot use teacher forcing since the desired sequence is unknown, and must therefore always pass a sample of s_{t-1} as the input to time t. This can lead to unstable training since errors early in the output propagate forward, potentially creating meaningless phrases in the latter parts of the sequence. To avoid this problem, we adopt a curriculum learning strategy in which we incrementally increase the length of produced sequences throughout training. Instead of selecting subsets of the data for curriculum training, we clip all sequences to have a predefined maximum length for each curriculum step. Although this approach relies on the discriminator being able to handle incomplete sentences, it does not degrade the performance as long as the discriminator is briefly retrained after each curriculum update.

Hyperparameter Tuning

We first train our generator as a denoising AE, for which we use the Adam optimizer with a learning rate of 1e-4. The same pre-trained network is used to initialize the generator for all GAN and sequence-to-sequence networks.

GAN networks are trained using the root mean square propagation (RMSProp) optimization algorithm. Learning rates are initialized to 5e-4 for the discriminator and 1e-5 for the generator. We train the discriminator 15 times for every generator update. Sequence-to-sequence models are trained using the Adam optimizer with a learning rate of 1e-4. We experimented extensively with varying the learning rate but found that increasing the discriminator learning rate caused its accuracy to decrease. Increasing the generator learning rate causes it to update too quickly for the discriminator, meaning the discriminator does not remain close to optimal and therefore gradients through it are not reliable. To ensure that the discriminator starts close to optimal, we initialize it by training it alone for the first 10 epochs. The generator's learning rate is decayed by a factor of 0.9 every 10 epochs. In systems with curriculum learning, this decay is only done after the curriculum is completed.

GAN training uses the original clipped version of Wasserstein GAN with a clipping threshold of 0.05. We also experimented heavily with this threshold, and found that a lower threshold led to low discriminator accuracy, and a higher threshold led to the discriminator providing poor gradients to the generator.

Our curriculum clips each sequence to a given length. We step up the curriculum length either when the discriminator accuracy falls below 55% or after 40 epochs, whichever comes first.

For the sorting and grammar experiments, the curriculum starts at 5 and is increased by 2 each step. For the Juliet Test Suite experiment, the curriculum starts at 75 and is increased by 5 each step.

Cycle-GAN

Another method we experimented with to ensure that the generator outputs correct versions of the specific input functions (rather than arbitrary correct functions) is cycle-consistency (also known as cycle-GAN) [75]. In this method, we train two generators: the original generator that is used to fix vulnerable code and an additional generator that *introduces* vulnerabilities into code. Cycle-consistency is the property that if a correct function is fed through the new generator (to introduce a bug) and then through the original generator (to repair the bug), the output should be equivalent to the initial data. We enforce this through a cycle-consistency loss function which measures how different a data sample is from the output after feeding it through the two generators sequentially. Specifically, we minimize the following,

$$L_{CYCLE} = \|E(y) - E(G_1(G_2(y,z)))\|_2^2 \tag{10}$$

Here G_1 is the repair GAN, G_2 is the new GAN that creates vulnerabilities, y is a correct data sample, z is a random vector that allows G_2 operate stochastically (since the mapping from correct to vulnerable code is one to many), and E is a function that encodes samples in a space in which the distance between two samples corresponds approximately to how different they are. In our experiments, we set E to be the encoder part of the encoder/decoder AE used to initialize the repair GAN.

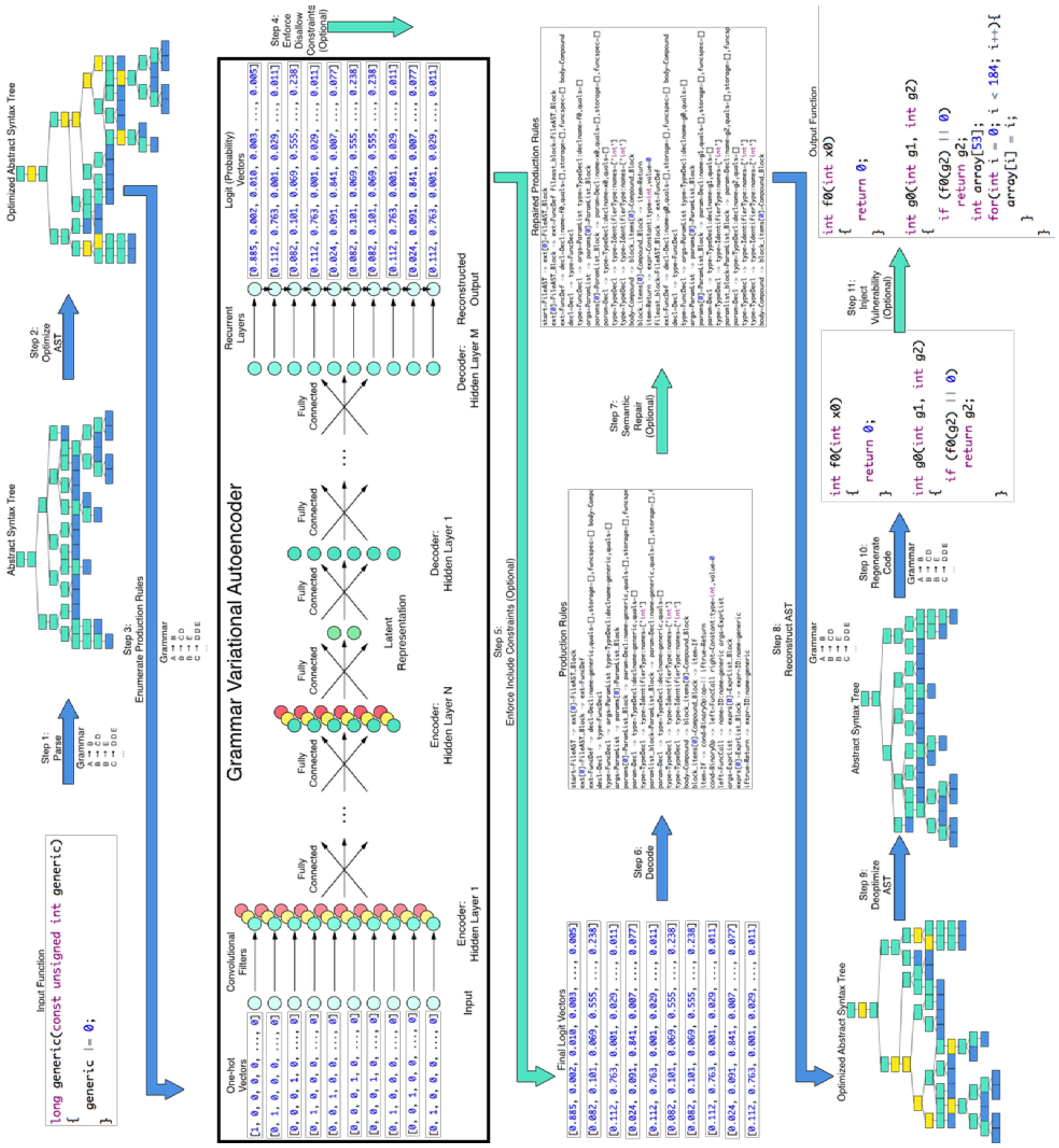
3.3.5 Program Synthesis.

The limited availability of labeled datasets is one of the main challenges for machine learning for vulnerability detection and repair. Since substantial time and investment are needed to compile a corpus of labeled functions at the scale required for machine learning, we explored an alternative approach to generate labeled datasets. We created a model that allows us to generate an arbitrary number of C functions that are known to obey some set of user-provided constraints, such as either containing or not containing a security vulnerability.

To build this model, we implemented a grammar variational auto encoder (VAE) similar to that designed by Kusner et al. [76]. The grammar VAE is a generative model that guarantees the syntactic validity of the output if the input can be expressed as a parse tree according to a specified grammar. It does not, in general, guarantee semantic validity, but it has been shown that grammar VAEs are particularly good at learning semantics because they do not need to learn any syntax.

We define syntactic and semantic validity as follows: a function string is *syntactically valid* if a parse operation succeeds in producing a corresponding AST without error, and a function string is *semantically valid* if it is syntactically valid and a compile operation succeeds in producing a corresponding binary file without error (we allow warnings in order to reduce the problem size). We use pycparser [77] for parsing and the GNU C Compiler (GCC) [78] for compilation. The grammar VAE attempts to learn a latent representation of input sequences and to reconstruct inputs from their latent vectors.

Figure 17 depicts the grammar VAE subsystem. The input sequence accepted by the grammar VAE, shown as the output of Step 3 in Figure 17, must correspond to a sequence of production rules in some grammar. In our grammar VAE, each function is: parsed into an AST (Step 1), which is then optimized to reduce the number of production rules in the vocabulary (Step 2); transformed into a sequence of production rules through a preorder traversal of the AST and changed into a sequence of one-hot vectors, where each index represents the production rule's index in the vocabulary (Step 3); and fed into the grammar VAE for encoding. The encoder produces a latent vector that represents the function. The decoder performs the inverse operation and outputs a sequence of production rules that correspond with a function similar to (ideally the same as) the input function. The entire decode process encompasses Steps 4 through 7, and includes the first two optional constraint enforcement points. We use 50 production rules as the shape of both input and output layers, which corresponds to functions approximately 2 to 10 lines in length. From the production rules, we can reconstruct the AST (Step 8), deoptimize the AST to remove any custom production rules that cannot be handled by pycparser (Step 9), and then use the pycparser API to generate the C code from the AST (Step 10). If the user chooses to inject vulnerabilities, this will occur in Step 11, after the code has been regenerated. The latent space forms the basis of the generative model: new functions that are similar to those on which the network was trained can be found by sampling arbitrary vectors from the latent space and passing them through the decoder.



```
long generic(const unsigned int generic)
{
    generic |= 0;
}
```

```
int f0(int x0)
{
    return 0;
}

int g0(int g1, int g2)
{
    if (f0(g2) || 0)
        return g2;
}
```

```
int f0(int x0)
{
    return 0;
}

int g0(int g1, int g2)
{
    if (f0(g2) || 0)
        return g2;
    int array[53];
    for(int i = 0; i < 184; i++)[
        array[i] = i;
    }
}
```

Figure 17. Program Synthesis System Overview

Early in the development of this model, we encountered a problem. The grammar VAE of Kusner et al. [76] was designed to handle a context-free grammar, but the language accepted by our parser is context-sensitive. In other words, the meaning of a symbol in the grammar (a node in the AST) cannot be determined independently, but must take into account the symbols around it. This poses a problem because we cannot reconstruct the AST properly after the decode step without incorporating context into the grammar, and thus there is no guarantee that functions output by this network are syntactically valid. Such a guarantee is important because every invalid function must be discarded from the final dataset, and this can be computationally expensive. To address this problem, we add one step of backward-looking context (*i.e.*, each rule has information about its predecessor rule) into the production rules when we get the rules from the traversal of the AST. This solves our first problem, but it causes an explosive increase in our vocabulary size. We reduce the vocabulary size by only taking the production rules found in 100 randomly selected functions in the dataset. After taking steps to make rules more generic by replacing literal values and identifier names, we find 234 rules, which is more than three times the number in the grammar used by Kusner et al. [76]. We do not capture most of the language of syntactically valid C functions with this small vocabulary. However, we find it sufficient to describe many of the basic constructs in which we are interested.

There is one additional problem. Because our training dataset, contains few functions that use only these production rules, our training dataset is now too small. Our solution to this was to create a dataset of randomly generated functions that only use the production rules found in the vocabulary. We generate functions by selecting production rules at random (masking out all invalid transitions based on the grammar, just like in the decoder) until we have a complete function. If we reach 50 production rules without completing the function, then we retry. We can now create a training dataset that is sufficiently large for our purposes. The randomly generated dataset is not ideal for training the grammar VAE, but the functions that our model outputs are still useful for training the DeepCode classifiers for two main reasons. First, we only used production rules found in our training dataset, so output functions share many syntactic properties with the original functions. Second, our output functions provide more training examples of the most common vulnerabilities found in our training dataset, and thus may improve our classifier performance.

Table 5. Selected Constraints

Constraint	Category
Return Type Modifier	Disallow
Argument Number Modifier	Disallow
Argument Type Modifier	Disallow
No Loops	Disallow
Contains Loop	Include
Semantically Valid	Repair
Contains Buffer Overflow	Repair

The final major aspect of our model is the ability to impose constraints on generated functions. Table 5 shows some of the constraints that we consider. Constraints fall into 3 categories: disallow, include, and repair. “Disallow” constraints can be expressed as masking operations on the vocabulary. For example, if we want to force a function to return a certain type, we can simply mark as invalid any rules that result in a different return type. “Include” constraints are more difficult to enforce. First, during the decode step we disallow any rules that result in the function ending before the desired property is included. Second, if the function does not have the desired property, we sort the production rules in the function completion operation to prioritize the rules that produce the desired property. (For example, if we want the function to contain a loop, then all the rules resulting in a loop are prioritized.) "Repair" constraints are handled entirely after the function decode step. This is the most complicated and diverse class of constraints, and each repair constraint requires its own logic that acts directly on either the production rules (as in the semantic repair constraint) or the reconstructed source code (as in the vulnerability injection constraint).

The entire decode and constraint enforcement operation is deterministic. Although randomness would allow us to achieve constraint enforcement by repeatedly decoding the latent vector until the output function meets all of the user-specified constraints, this is not a satisfactory solution. This approach could work if the user supplies only a small number of constraints to enforce, but it may take many iterations of decoding to find a satisfactory function if many constraints are given simultaneously. Furthermore, it may be the case that no such latent vector exists that produces the function in question, so the model would spend an unbounded amount of time searching. Our solution scales well, demonstrates rigor, and overcomes the aforementioned problem.

4. RESULTS AND DISCUSSION

This section provides the results of evaluations of the developments described in Section 3. Section 4.1 provides details of datasets that we generated and used for the development of classification and repair tools. Section 4.2 provides evaluation results for the newly developed classification technology. Section 4.3 provides evaluation results for the newly developed repair technology. Section 4.4 presents results of the program synthesis investigation. Section 4.5 summarizes results from program evaluations and hackathons.

4.1 Data

4.1.1 Data Generation and Ingestion.

Table 6 presents metrics on training data extracted in order to train the machine learning algorithms. The definitions of the data presented in the columns in Table 6 are as follows:

- "Total Functions" is the total number of functions built and extracted from the dataset.
- "CFG > 0x0" is the total number of these functions that have a non-zero CFG size. (Many functions that are defined by macros in header files are reported as having a CFG size of zero.)
- "Functions Pulled" is the total number of these functions kept during the pull step.
- "Valid Build" is the total number of these functions for which a valid build identifier is available in the database to link back to source code.
- "Found Source" is the total number of these functions for which source code is available in the database.
- "Labeling" indicates whether only SA labels are available or whether truth labels are also available.

Table 6. Summary of Training Data

Project	Total Functions	CFG > 0x0	Functions Pulled	Valid Build	Found Source	Labeling
SATE IV	417,311	83,922	83,922	83,222	82,765	SA + truth
MUSE Corpus	11,426,565	3,013,190	2,847,950	2,596,967	1,142,150	SA
Juliet Test Suite	651,812	121,904	121,904	121,353	100,863	SA + truth
Debian Packages	41,075,788	10,736,998	10,736,998	6,926,830	3,346,313	SA
Debian Kernel	778,644	246,684	246,684	244,582	191,754	SA

Table 7 presents metrics on benchmark data extracted in order to evaluate performance of the machine learning algorithms. ManyBugs is a special dataset that has labeled data. Table 7 also presents metrics on data generation for two challenge problem datasets: PureOS [79] and LibTIFF version 3.8.2.

Table 7. Summary of Benchmark and Challenge Problem Data

Project	Total Functions	CFG > 0x0	Functions Pulled	Valid Build	Found Source	Labeling
ManyBugs	478,311	109,379	109,379	90,849	67,653	SA + truth
PureOS	861,003	269,593	269,593	266,348	203,477	SA
LibTIFF 3.8.2	2,059	694	694	594	545	SA

4.1.2 Data Extraction.

As described in section 3.3, data extraction during the pull step of the DeepCode functional flow performed necessary processing to clean the data queried from Elasticsearch and to transform to formats required by our machine learning models. This included creating binary labels (“not vulnerable”/“good” and “vulnerable”/“bad”) and feature vectors, and filtering out the data samples that were missing features, were unreasonably long or short, or were considered duplicates. We generated two datasets, which we refer to as Dataset 1 and Dataset 2. Dataset 1 is the initial dataset with labels generated using Clang static analyzer only. Subsequently, Dataset 2 was generated during Phase 3 of the program to improve some of the shortcomings of Dataset 1 including using multiple static analyzers to generate labels and incorporating additional kernel codes. Details of the datasets are explained below.

Table 8 summarizes the initial dataset that we used for the development. The “Total” row represents the total number of functions extracted during the pull process. The “Filtered” row represents the total number of functions filtered out during the data curation process. The "No source extracted" row represents the number of functions filtered out because our database does not contain source code for them. The "Lexer size out of bounds" row represents the number of functions filtered out because the functions were unreasonably long or short, or our custom lexer failed to generate a lexed representation from the source code. The "Duplicates" row represents the number of functions removed because they were considered duplicates by having identical build or source features. The "Kept" row represents the number of functions remaining after the filtering process, with the "Not vulnerable" and "Vulnerable" rows representing the number functions with and without vulnerabilities, respectively. The numbers for the combined corpus do not always add up to the sum of the numbers for the individual corpora because of duplicate rejection across corpora.

Table 8. Summary of Dataset 1

CORPUS NAME:	SATE IV	MUSE	DEBIAN	COMBINED
Total	83,222	2,805,962	6,731,561	9,620,835
Filtered (% of total)	75,827 (91.1%)	2,472,686 (88.1%)	6,121,012 (90.9%)	8,724,364 (90.7%)
▪ No source extracted	457	1,668,515	4,420,999	6,089,971
▪ Lexer size out of bounds	5,020	195,265	371,464	571,749
▪ Duplicates	70,350	608,906	1,328,549	2,062,644
Kept (% of total)	7,395 (8.9%)	333,276 (11.9%)	610,639 (9.1%)	**896,471 (9.3%)**
▪ Not vulnerable (% of kept)	3,808 (51.5%)	321,045 (96.3%)	590,565 (96.7%)	**862,419 (96.2%)**
▪ Vulnerable (% of kept)	3,587 (48.5%)	12,231 (3.7%)	20,074 (3.3%)	**34,052 (3.8%)**

One metric that stands out is that about 90% of the available functions are filtered out. Most of these are because no source was extracted, with the next highest contributor being rejection of duplicate functions. Source is not extracted for most function objects because many functions are defined in header files, and the data extraction process does not traverse dependencies to pull all source code. Because these header files are included by many source files, each such function produces a large number of duplicate function objects in the database. Therefore, the amount of data discarded due to this filtering step is not nearly as great as the metrics make it appear.

Note that the functions kept were randomly split into 78% training, 10% validation, 2% ensemble, and 10% test sets for model development. The training dataset was used to train the models. The validate dataset was used for hyperparameter tuning of the trained models. The ensemble dataset was used to develop ensemble models. The test dataset was used to evaluate the performance of the fully tuned models.

During Phase 3 of the program, we expanded the dataset with the following updates:

- Updated the SATE IV data to the newer Juliet Test Suite Version 1.3, which includes more functions and vulnerabilities
- Added about 3 million additional functions from Debian Linux distribution packages
- Added the Debian Linux kernel, consisting of over 200,000 functions
- Used two additional SA tools, Cppcheck and Flawfinder, in order to generate more robust labels

Table 9 summarizes the expanded dataset, where the labels for the functions from MUSE and Debian corpora were generated using the findings from the three SA tools.

Table 9. Summary of Dataset 2

CORPUS NAME:	JULIET	MUSE	DEBIAN	COMBINED
Total	121,353	2,806,469	9,532,081	12,459,903
Filtered (% of total)	109,272 (90.1%)	2,426,088 (86.5%)	8,576,398 (90.0%)	11,185,661 (89.8%)
▪ No source extracted	20,490	1,671,685	6,132,024	7,824,199
▪ Lexer size out of bounds	91	92,275	244,559	336,925
▪ Duplicates	88,771	662,128	2,199,815	3,024,537
Kept (% of total)	12,001 (9.9%)	380,381 (13.5%)	955,683 (10.0%)	**1,274,242 (10.2%)**
▪ Not vulnerable (% of kept)	6,559 (54.6%)	364,306 (95.8%)	907,186 (94.9%)	**1,207,396 (94.8%)**
▪ Vulnerable (% of kept)	5,442 (45.4%)	16,075 (4.2%)	48,497 (5.1%)	**66,846 (5.2%)**

Dataset 2 offers several potential improvements over Dataset 1. The additional function examples improve the ability of our models to learn the patterns that help them distinguish vulnerable functions. Inclusion of the Debian kernel functions (as distinct from ordinary package functions) exposes our models to learn from the wider variety of code samples, allowing our models to generalize to other kernel code. Generating labels using the three SA tools instead of one creates more reliable and robust labels.

4.2 Classification

This section presents evaluation results for classification. Section 4.2.1 presents the approach we took to quantifying classification accuracy. Section 4.2.2 presents the accuracy results based on validation using held-out examples from the training corpus. Section 4.2.3 presents results based on evaluation of independent datasets. Section 4.2.4 showcases selected classification examples.

4.2.1 Approach to Quantifying Accuracy.

We chose to evaluate our classifiers with several different metrics that can capture different aspects of classifier quality. Each of our classification approaches produces a score as an output and binary classifications are made based on these scores surpassing some threshold T between 0 and 1.

Given a labeled evaluation dataset, we produce for each classifier the count of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN), where $\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}$ must equal the total number of functions in the dataset for all threshold values T. Area under curve (AUC) metrics evaluate the quality of a classifier over the full range of T by integrating over a tradeoff curve, and thus capture the classification quality for a range of applications.

Receiver Operating Characteristic (ROC) Curve
The ROC curve is produced by plotting the true positive rate (TPR), $\mathrm{TP}/(\mathrm{TP} + \mathrm{FN})$, versus the FPR, $\mathrm{FP}/(\mathrm{FP} + \mathrm{TN})$, while varying the discrimination threshold of the classifier. We include the ROC AUC (which ranges from 0 to 1) as one of our metrics since it is a standard metric for binary classifiers. The ROC curve is good for quickly seeing how a classifier compares to a random classifier, as a random classifier always falls along the diagonal of the plot (0.5 AUC.) This metric is primarily useful for evaluation datasets with relatively little class imbalance, such as the Juliet Test Suite. However, for evaluation datasets with strong class imbalances, like our main test dataset, this metric should be used with caution since TN is very large compared to all of the other counts, making the FPR very small for most plausible threshold values. In effect, for our class-imbalanced dataset, the ROC AUC is weighted heavily towards scenarios in which a very large number of FP is acceptable.

Precision-Recall (PR) Curve
While the ROC curve provides accepted metrics for classification accuracy analysis, in the case of software vulnerability detection the metric produces optimistic estimates because the data are highly imbalanced (the number of functions with vulnerabilities is significantly lower than the number of functions without vulnerabilities). Therefore, we also evaluated other metrics that give a more accurate assessment of how classification error statistics would translate into a user's experience using the classifiers to detect vulnerabilities in their code.

The PR curve is more appropriate for imbalanced evaluation data, since it plots the precision, $\mathrm{TP}/(\mathrm{TP} + \mathrm{FP})$, against the TPR (which is equivalent to recall). In effect, it shows the direct tradeoff between FN and FP as the threshold is varied. A random classifier PR curve looks like a horizontal line at a precision equal to the fraction of the evaluation data that is positive. For example, for a dataset in which 5% of the functions are vulnerable, a random classifier PR curve would be a horizontal line at a precision of 5% (and AUC for the random classifier would therefore be 0.05 in this example). The PR AUC is probably the most useful general-purpose metric for classifier selection on our dataset, though its value still depends on the class imbalance of the evaluation dataset.

Matthews Correlation Coefficient
The MCC is our primary fixed-threshold metric. It is calculated by:

$$\mathrm{MCC} = (\mathrm{TP} \times \mathrm{TN} - \mathrm{FP} \times \mathrm{FN}) \big/ \sqrt{(\mathrm{TP} + \mathrm{FP})(\mathrm{TP} + \mathrm{FN})(\mathrm{TN} + \mathrm{FP})(\mathrm{TN} + \mathrm{FN})} \qquad (11)$$

MCC is effectively a correlation coefficient between our predictions and the true values. Thus, the MCC can range from -1 (perfect anticorrelation) to 1 (perfect correlation), and a random classifier will have an MCC of 0. The MCC metric has a major advantage over the more common accuracy $(\mathrm{TP} + \mathrm{TN})/(\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN})$ and F_1 score (the harmonic mean of precision and recall) metrics in that it is invariant to class imbalance. Thus, the MCC can be directly compared between evaluating on a balanced dataset and an imbalanced dataset. For most of our classification models, model parameters and checkpoints were determined based on the highest MCC. The threshold for the test MCC for each classifier is determined by finding the threshold that maximizes the MCC on the validation data. This is the “optimal threshold” in the sense that it maximizes the correlation between the validation predictions and the labels.

4.2.2 Training Results.

As described previously in Section 3.3.3, we investigated various different approaches for each classifier type. For build feature-based classifiers, we investigated RF using simplified features, CNN with advanced features, one-layer LSTM with advanced features, and two-layer LSTM with advanced features. For source feature-based classifiers, we investigated RF and extra-trees using the BOW representation of lexed source, CNN and RNN each with lexed source and embedded lexed source, and RF with CNN and RNN learned-feature representation of lexed source.

Two-layer LSTM performed best among the build feature-based classifiers, RF with CNN learned-feature representation of lexed source performed best among the source feature-based classifiers, and RF with simplified features and CNN learned-feature representation of lexed source performed best among the combined classifiers.

Table 10 compares the performance metrics of the best-performing classifiers on Dataset 1, and Figure 18 shows the ROC curve and PR curve comparisons. The source feature-based classifier offers an advantage over the build feature-based classifier in all three metrics, and the combined model performs better than either model does individually, illustrating that the build features provide useful information that source code itself does not provide. Finally, ensemble model using the linear opinion pool approach to ensemble all three models further improves the performance. This analysis illustrates the accuracy with which the classifiers predict the training labels. Since we do not have ground truth for most of the functions in this dataset, this mostly evaluates the ability of the classifiers to predict SA labels.

Table 10. Summary of Classification Results on Dataset 1

MODEL	ROC AUC	PR AUC	MCC	THRESHOLD
Build feature-based	0.801	0.361	0.413	0.530
Source feature-based	0.863	0.473	0.482	0.298
Combined	0.865	0.491	0.506	0.255
Ensemble	0.877	0.505	0.515	0.260

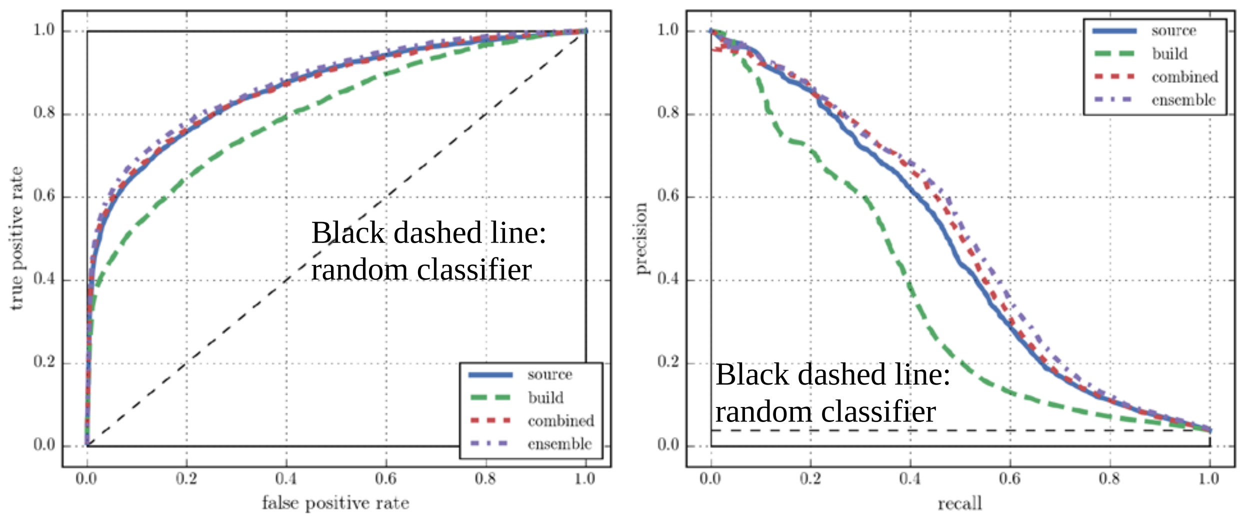


Figure 18. Classifier Comparison on Dataset 1

Table 11 and Figure 19 show the performance comparison of the three types of classification models on Dataset 2. The source feature-based classifier offers an advantage over the build feature-based classifier in all three metrics, and the combined model performs better than either model does individually, illustrating that the build features provide useful information that source code itself does not provide. From Figure 19 we observe that there is no point on the curve where the source feature-based or build feature-based classifier performs better than the combined classifier, indicating that the ensemble approach will not improve performance. Therefore, the ensemble classifier was omitted for Dataset 2.

Table 11. Summary of Classification Results on Dataset 2

MODEL	ROC AUC	PR AUC	MCC	THRESHOLD
Build feature-based	0.768	0.263	0.274	0.256
Source feature-based	0.895	0.490	0.455	0.231
Combined	0.896	0.512	0.474	0.201

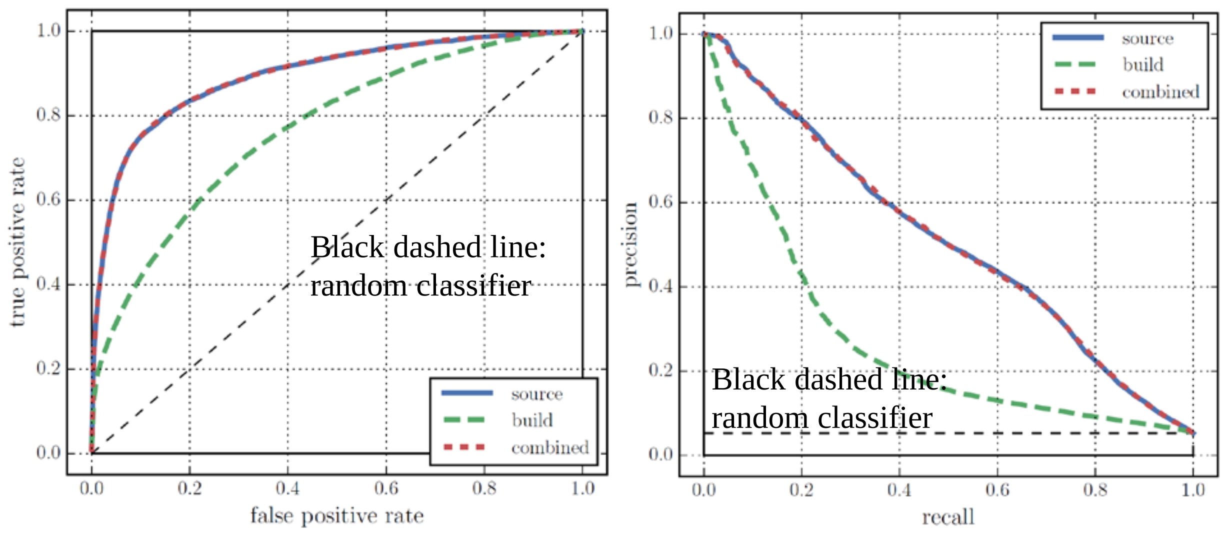


Figure 19. Classifier Comparison on Dataset 2

The build feature-based classifier gives lower prediction accuracy on Dataset 2 than it does on Dataset 1. We conclude that the build features are less effective for predicting SA labels from the two SA tools that were added for Dataset 2 (Flawfinder and Cppcheck) than for those from the original SA tool from Dataset 1 (Clang). Unlike Clang, Flawfinder and Cppcheck use analysis rules that operate directly on a program's source code that may be difficult to detect from build artifacts. As our source features come directly from the lexed source code, the classifiers with access to these should more easily learn to recognize the patterns that result in Flawfinder and Cppcheck findings.

4.2.3 Evaluation Results.

Evaluation of classification accuracy is complicated by a lack of available benchmark datasets that represent real-world software (as opposed to the synthetic examples of the Juliet Test Suite), include examples of software bugs that are typical of software vulnerabilities (as opposed to algorithm bugs that would produce incorrect output but would not expose vulnerabilities), and provide truth data for evaluation.

For our accuracy assessment challenge problem, we ran the DeepCode classifiers on code from the Juliet Test Suite and demonstrated better accuracy than three SA tools. Section 4.5.4 presents those results.

To evaluate accuracy against more realistic software, we first ran the Vader dynamic analysis tool against functions in a specific version of the LibTIFF package that were deemed by a team of Draper cybersecurity experts to be most likely to contain security vulnerabilities. This exercise uncovered vulnerabilities in three functions. This is not a large enough sample size for a statistical analysis, but we were able to run the DeepCode classifiers on the same LibTIFF package and determine classification scores. We then determined where the three known vulnerable functions ranked relative to the other functions in LibTIFF according to classification scores on a percentile basis, with the 100th percentile indicating the function scored as being most likely to be vulnerable and the 0th percentile indicating the function scored as being least likely to be vulnerable.

Figure 20 shows the results for three DeepCode classifier versions. All three functions known to be vulnerable scored in the upper 40% of results. We would expect a random result to be distributed with a mean about the 50th percentile, so these results are better than random, representing a good comparison relative to an independent vulnerability assessment. It should further be noted that dynamic analysis requires orders of magnitude more processing time than the DeepCode classifiers.

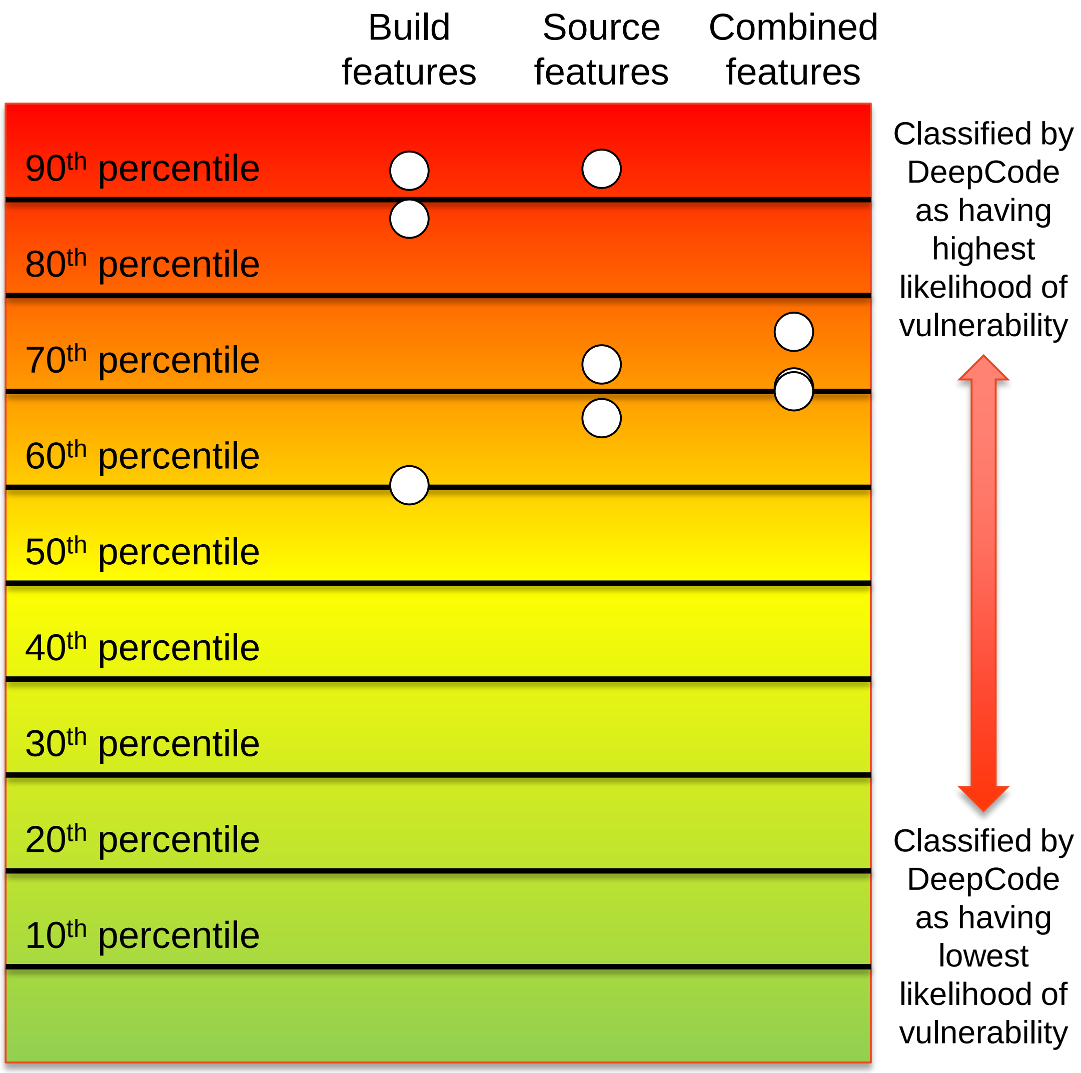
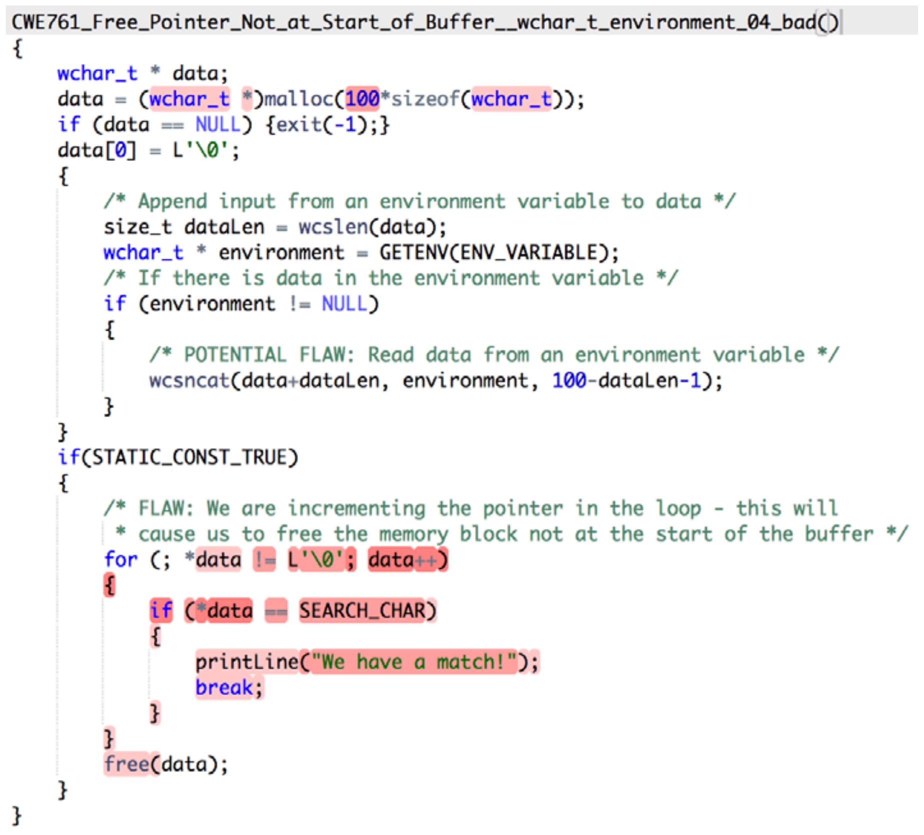


Figure 20. Classification Results on LibTIFF

4.2.4 Examples.

In this section, we provide selected examples of error localization using the class activation mapping technique described in Section 3.3.3.

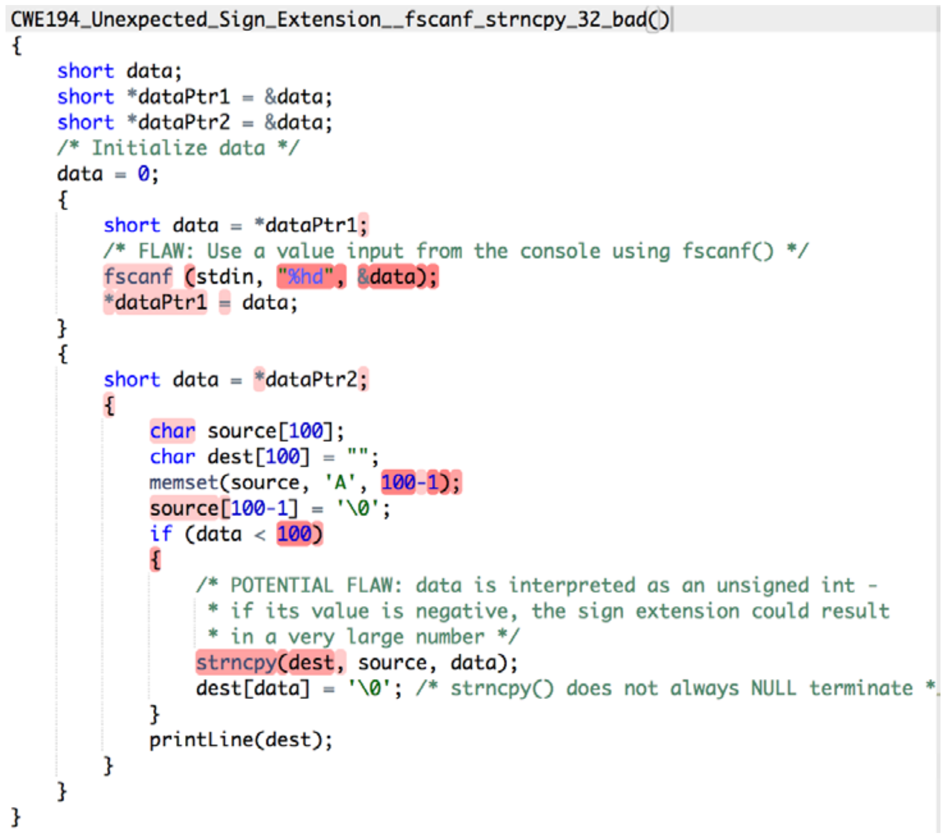
Figure 21 shows the error localization highlighting the relevant code block with a potential vulnerability, in which incrementing the pointer in the **for** loop cause memory to be freed using a pointer that is not at the start of the buffer.



```
CWE761_Free_Pointer_Not_at_Start_of_Buffer__wchar_t_environment_04_bad()
{
    wchar_t * data;
    data = (wchar_t *)malloc(100*sizeof(wchar_t));
    if (data == NULL) {exit(-1);}
    data[0] = L'\0';
    {
        /* Append input from an environment variable to data */
        size_t dataLen = wcslen(data);
        wchar_t * environment = GETENV(ENV_VARIABLE);
        /* If there is data in the environment variable */
        if (environment != NULL)
        {
            /* POTENTIAL FLAW: Read data from an environment variable */
            wcsncat(data+dataLen, environment, 100-dataLen-1);
        }
    }
    if(STATIC_CONST_TRUE)
    {
        /* FLAW: We are incrementing the pointer in the loop - this will
         * cause us to free the memory block not at the start of the buffer */
        for (; *data != L'\0'; data++)
        {
            if (*data == SEARCH_CHAR)
            {
                printLine("We have a match!");
                break;
            }
        }
        free(data);
    }
}
```

Figure 21. Free of Pointer Not at Start of Buffer

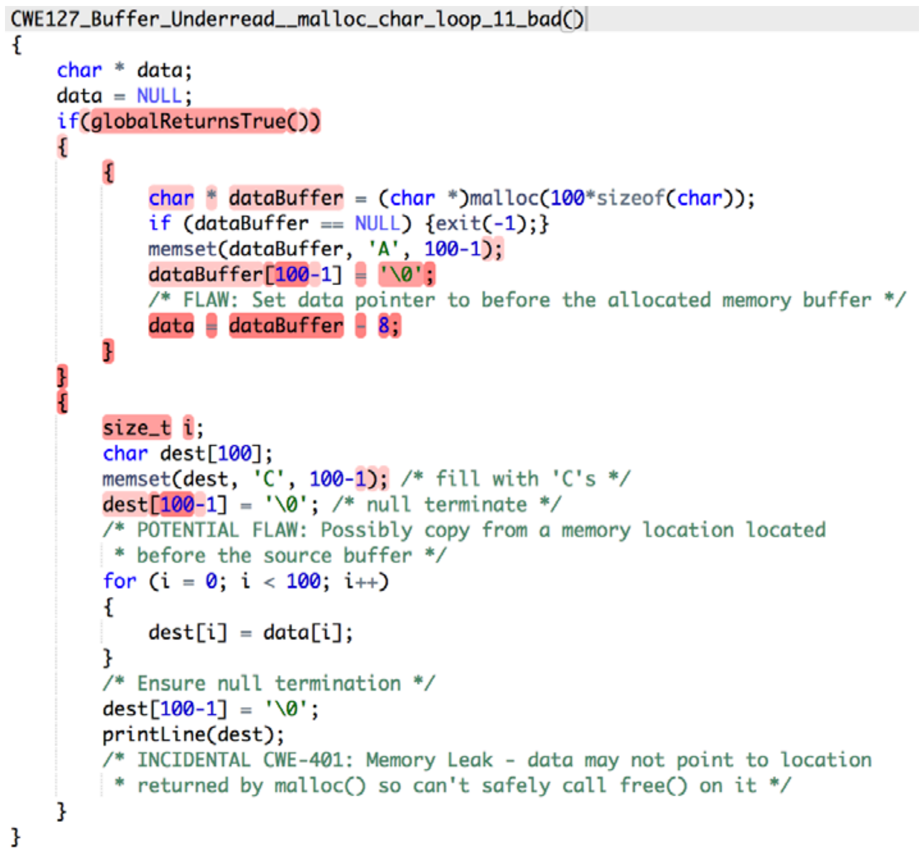
Figure 22 shows the error localization highlighting the **strncpy** function use with a potential vulnerability in which unexpected sign extension of the variable **data** exceeds the boundaries of the **dest** array for the **strncpy** operation. The error localization also highlighted another relevant code block with a potential weakness where the **fscanf** function is used instead of **scanf** to read data from console input.



```
CWE194_Unexpected_Sign_Extension__fscanf_strncpy_32_bad()
{
    short data;
    short *dataPtr1 = &data;
    short *dataPtr2 = &data;
    /* Initialize data */
    data = 0;
    {
        short data = *dataPtr1;
        /* FLAW: Use a value input from the console using fscanf() */
        fscanf (stdin, "%hd", &data);
        *dataPtr1 = data;
    }
    {
        short data = *dataPtr2;
        {
            char source[100];
            char dest[100] = "";
            memset(source, 'A', 100-1);
            source[100-1] = '\0';
            if (data < 100)
            {
                /* POTENTIAL FLAW: data is interpreted as an unsigned int -
                 * if its value is negative, the sign extension could result
                 * in a very large number */
                strncpy(dest, source, data);
                dest[data] = '\0'; /* strncpy() does not always NULL terminate *
            }
            printLine(dest);
        }
    }
}
```

Figure 22. Unexpected Sign Extension

Figure 23 shows the error localization highlighting the relevant code block with a potential vulnerability where the pointer references a memory location prior to the targeted buffer.



```
CWE127_Buffer_Underread__malloc_char_loop_11_bad()
{
    char * data;
    data = NULL;
    if(globalReturnsTrue())
    {
        {
            char * dataBuffer = (char *)malloc(100*sizeof(char));
            if (dataBuffer == NULL) {exit(-1);}
            memset(dataBuffer, 'A', 100-1);
            dataBuffer[100-1] = '\0';
            /* FLAW: Set data pointer to before the allocated memory buffer */
            data = dataBuffer - 8;
        }
    }
    {
        size_t i;
        char dest[100];
        memset(dest, 'C', 100-1); /* fill with 'C's */
        dest[100-1] = '\0'; /* null terminate */
        /* POTENTIAL FLAW: Possibly copy from a memory location located
         * before the source buffer */
        for (i = 0; i < 100; i++)
        {
            dest[i] = data[i];
        }
        /* Ensure null termination */
        dest[100-1] = '\0';
        printLine(dest);
        /* INCIDENTAL CWE-401: Memory Leak - data may not point to location
         * returned by malloc() so can't safely call free() on it */
    }
}
```

Figure 23. Buffer Under-Read

4.3 Repair

This section presents evaluation results for repair. Section 4.3.1 presents the approach we took to quantifying repair accuracy. Section 4.3.2 presents accuracy results for several repair experiments. Section 4.3.3 showcases selected repair examples.

4.3.1 Approach to Quantifying Accuracy.

While our GAN approach does not require paired examples to train, we focus our experiments on datasets with paired examples so that we can meaningfully evaluate the performance of our approach. These datasets also allow direct comparison to sequence-to-sequence networks, and we can use their performance as a benchmark for comparison with our GAN approach.

We use the Bilingual Evaluation Understudy (BLEU) score [80], which is one of the most commonly used evaluation metrics for machine translation problems, as the main evaluation metric for the repair performance. The BLEU score compares n-grams (we use n of 4, which we refer to as BLEU-4) of the repaired function's sequence tokens with the n-grams of the sequence tokens of the desired good version and count the number of matches. This metric is more robust to simple insertion and deletion changes that may be over-penalized by "Sequence Accuracy" that compares the entire sequence of tokens.

In addition to BLEU score, we use sequence accuracy, order accuracy, or grammar accuracy depending on the experiments and contexts of the datasets. The next section includes details of these additional metrics when discussing each of the experiments.

4.3.2 Repair Accuracy Results.

We evaluated our GAN approach using three experiments: repair of sequences of sorted numbers, repair of sentences in a context-free grammar, and repair of vulnerabilities in C and C++ code. The first two of these involve hand-curated datasets and are intended to highlight the benefits of our GAN approach to address the domain mapping problem. The third evaluates repair performance in the problem domain of interest.

Sorting Experiment

To show the necessity of enforcing accurate domain mapping, we conduct an experiment for which the repair task is to sort the input into ascending order. We generate sequences of 20 randomly selected integers (without replacement) between 0 and 50 in ascending order. We then inject errors by swapping n selected tokens which are next to each other, where n is a (rounded) Gaussian random variable with mean 8 and standard deviation 4. The task is to sort the sequence back into its original ascending order given the error-injected sequence. This scheme of data generation allows us to maintain pairs of good (before error injection) and bad (after error injection) data, and to compute the "sequence accuracy" with which our GAN is able to restore the good sequences from the bad. To assess our domain mapping approach and evaluate the usefulness of our self-regularizer loss functions, we also compute the percentage of sequences which have valid orderings but not necessarily valid domain mappings, which we refer to as "order accuracy".

We use identical networks for the generator in our GAN model and the sequence-to-sequence baseline. The generator RNNs contain 3 layers of 512 hidden states each. The discriminator convolutional layer has 3 filter sizes (3, 7, and 11 hidden states) and 300 filters for each size, leading to a total of 900 filters. The fully connected layer for the discriminator output consists of two layers (the first with 512 hidden states and the second with a single hidden state). Networks are trained for 200 epochs. The curriculum starts at 5 and is increased by 2 each step.

Table 12 presents the sorting repair experiment results. In Table 12, “Cur” refers to experiments using curriculum learning, while “Auto”, “Freq”, and “Cycle” are those using L_{AUTO}, L_{FREQ}, and L_{CYCLE}, respectively. The base GAN easily learns to generate sequences with valid ordering, without necessarily paying attention to the input sequence. This leads to high order accuracy, but low sequence accuracy. However, adding Auto or Freq loss regularizers significantly improves the sequence accuracy, which shows that these losses effectively enforce the correct mapping between the source and target domains.

Table 12. Sorting Repair Experiment Results

Model	**Configuration**	**Sequence Accuracy**	**Order Accuracy**
Sequence-to-sequence	Base	99.7	99.8
	Base + Cur	99.7	99.8
GAN	Base	82.8	96.9
	Base + Auto	98.9	99.6
	Base + Freq	99.3	99.7
	Base + Cur	81.5	98.0
	Base + Cur + Auto	96.2	98.0
	Base + Cur + Freq	98.2	99.1
	Base + Cur + Cycle	91.0	97.8

Grammar Experiment

For our second experiment, we generate data from a simple context-free grammar similar to that used by Rajeswar et al. [65]. Our good data are selected randomly from the set of all sequences which satisfy the grammar and are less than length 20. We then inject errors into each sequence, where the number of errors is chosen as a Gaussian random variable (zero thresholded and rounded) with mean 5 and standard deviation 2. Each error is then randomly chosen to be either a deletion of a random token, insertion of a random token, or swap of two random tokens.

The network is tasked with generating the original sequence from the error injected one. This task better models real data than the sorting task above, because each generated token must follow the grammar and is therefore conditioned on all previous tokens.

We use identical networks for the generator in our GAN model and the sequence-to-sequence baseline. The generator RNNs contain 3 layers of 512 hidden states each. The discriminator convolutional layer has 3 filter sizes (3, 7, and 11 hidden states) and 300 filters for each size, leading to a total of 900 filters. The fully connected layer for the discriminator output consists of two layers (the first with 512 hidden states and the second with a single hidden state). Networks are trained for 400 epochs. The curriculum starts at 5 and is increased by 2 each step.

Table 13 presents the grammar repair experiment results. These results show that our GAN approach is able to achieve high grammar accuracy, in terms of generating correct sequences that fit the context-free grammar. Notably, all of our methods preform reasonably well on this task, which shows that the GAN approach is able to correctly map a bad distribution to a good distribution.

Table 13. Grammar Repair Experiment Results

Model	Configuration	Grammar Accuracy
Sequence-to-sequence	Base	99.3
	Base + Cur	98.9
GAN	Base	98.0
	Base + Auto	96.5
	Base + Freq	97.5
	Base + Cur	98.9
	Base + Cur + Auto	97.8
	Base + Cur + Freq	96.3
	Base + Cur + Cycle	98.3

Juliet Test Suite

We tested our GAN model on the Juliet Test Suite. We use identical networks for the generator in our GAN model and the sequence-to-sequence baseline. The generator RNNs contain 4 layers of 512 hidden states each. The discriminator convolutional layer has 3 filter sizes (3, 7, and 11 hidden states) and 300 filters for each size, leading to a total of 900 filters. The fully connected layer for the discriminator output consists of two layers (the first with 512 hidden states and the second with a single hidden state). Networks are trained for 1000 epochs. The curriculum starts at 75 and is increased by 5 each step.

Table 14 presents the Juliet Test Suite experiment results. Our GAN approach achieves progressively better results when we add (a) curriculum training, and (b) either L_{AUTO} or L_{FREQ} regularization loss. The Base+Cur+Freq model proves to be the best among different GAN models, and performs reasonably close to the sequence-to-sequence baseline (which is the upper performance bound in this experiment because it is trained using paired training examples). Code examples where our GAN makes correct repairs are provided in Section 4.2.3. Non-curriculum tests with L_{AUTO} or L_{FREQ} regularization losses were not included in one-line testing because they were already shown in multi-line testing to have poorer performance than the curriculum-based approaches.

Table 14. Juliet Test Suite Repair Experiment Results

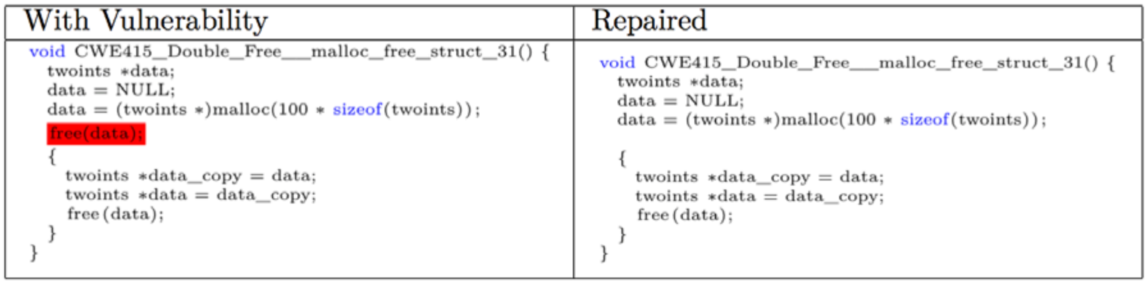
Model	**Configuration**	**One-Line**	**Multi-Line**
Sequence-to-sequence	Base	.997	.963
	Base + Cur	.997	.964
GAN	Base	.873	.842
	Base + Auto		.857
	Base + Freq		.862
	Base + Cur	.904	.883
	Base + Cur + Auto	.956	.899
	Base + Cur + Freq	.962	.903
	Base + Cur + Cycle	.918	.831

We note that the sequence-to-sequence baseline has higher accuracy metrics than the GAN approach for these experiments. These are experiments involving simple repairs of very simple synthetic code. We relied on such data for repair accuracy assessment because we needed paired examples with ground truth data in order to do the analysis for the accuracy assessment. We expect the real power of the GAN approach to come with repair problems involving more complex code, but we lacked the benchmark dataset necessary for such an evaluation.

4.3.3 Repair Examples.

The figures in this section show selected GAN repair examples.

Figure 24 shows an error where the memory is used after it is freed. Our GAN repairs it correctly by removing the piece of code that frees the memory.



With Vulnerability

```
void CWE415_Double_Free__malloc_free_struct_31() {
  twoints *data;
  data = NULL;
  data = (twoints *)malloc(100 * sizeof(twoints));
  free(data);
  {
    twoints *data_copy = data;
    twoints *data = data_copy;
    free(data);
  }
}
```

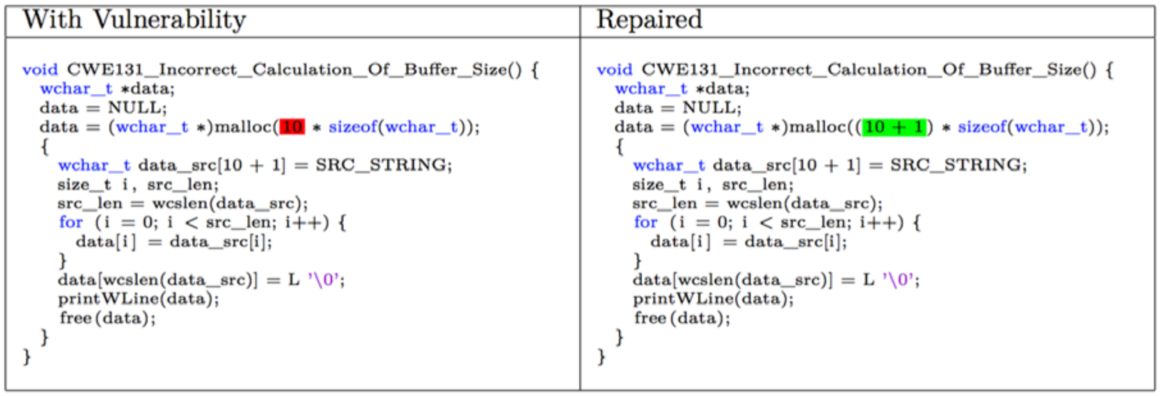
Repaired

```
void CWE415_Double_Free__malloc_free_struct_31() {
  twoints *data;
  data = NULL;
  data = (twoints *)malloc(100 * sizeof(twoints));

  {
    twoints *data_copy = data;
    twoints *data = data_copy;
    free(data);
  }
}
```

Figure 24. Memory Use After Free

Figure 25 shows a function that has a buffer allocated which is too small for the resulting data write. Our GAN repairs it by increasing the amount of memory allocated to the buffer.



With Vulnerability

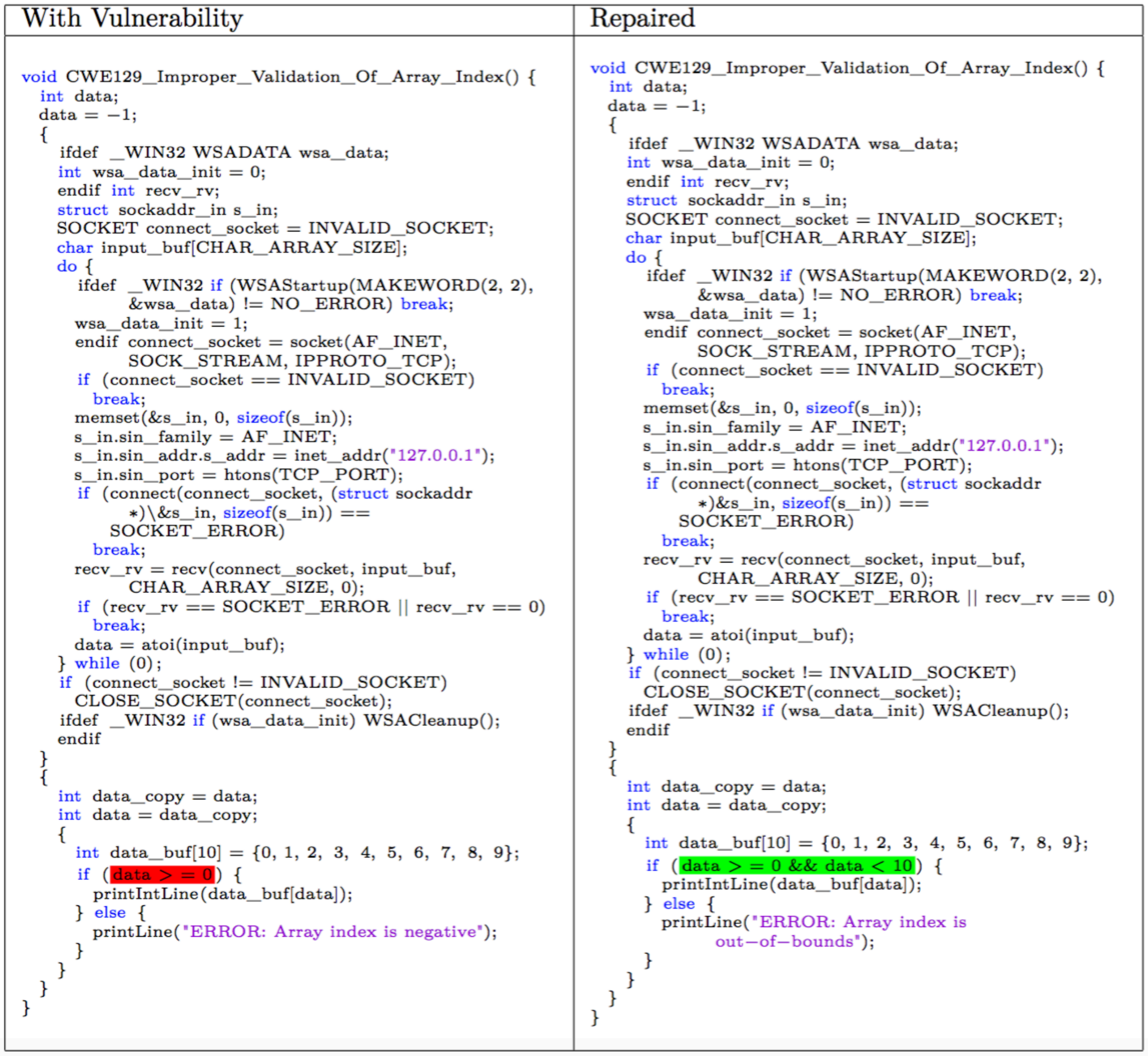
```
void CWE131_Incorrect_Calculation_Of_Buffer_Size() {
  wchar_t *data;
  data = NULL;
  data = (wchar_t *)malloc(10 * sizeof(wchar_t));
  {
    wchar_t data_src[10 + 1] = SRC_STRING;
    size_t i, src_len;
    src_len = wcslen(data_src);
    for (i = 0; i < src_len; i++) {
      data[i] = data_src[i];
    }
    data[wcslen(data_src)] = L '\0';
    printWLine(data);
    free(data);
  }
}
```

Repaired

```
void CWE131_Incorrect_Calculation_Of_Buffer_Size() {
  wchar_t *data;
  data = NULL;
  data = (wchar_t *)malloc((10 + 1) * sizeof(wchar_t));
  {
    wchar_t data_src[10 + 1] = SRC_STRING;
    size_t i, src_len;
    src_len = wcslen(data_src);
    for (i = 0; i < src_len; i++) {
      data[i] = data_src[i];
    }
    data[wcslen(data_src)] = L '\0';
    printWLine(data);
    free(data);
  }
}
```

Figure 25. Buffer Allocation Error

Figure 26 shows a function that reads the index of an array access from a socket and returns the memory at the index. The vulnerable function only checks the lower bound on the array size. Our GAN repairs it by adding an additional check on the upper bound.



With Vulnerability

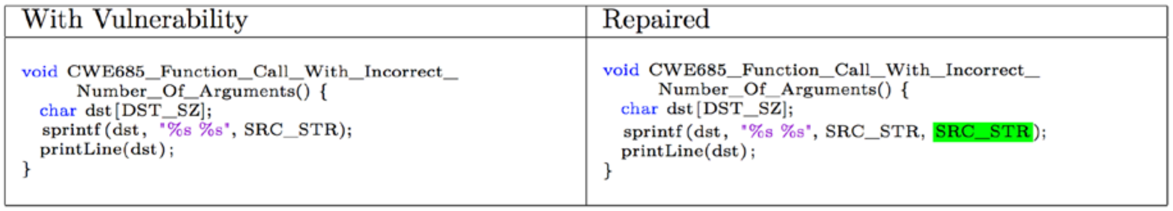
```
void CWE129_Improper_Validation_Of_Array_Index() {
  int data;
  data = −1;
  {
    ifdef _WIN32 WSADATA wsa_data;
    int wsa_data_init = 0;
    endif int recv_rv;
    struct sockaddr_in s_in;
    SOCKET connect_socket = INVALID_SOCKET;
    char input_buf[CHAR_ARRAY_SIZE];
    do {
      ifdef _WIN32 if (WSAStartup(MAKEWORD(2, 2),
              &wsa_data) != NO_ERROR) break;
      wsa_data_init = 1;
      endif connect_socket = socket(AF_INET,
              SOCK_STREAM, IPPROTO_TCP);
      if (connect_socket == INVALID_SOCKET)
        break;
      memset(&s_in, 0, sizeof(s_in));
      s_in.sin_family = AF_INET;
      s_in.sin_addr.s_addr = inet_addr("127.0.0.1");
      s_in.sin_port = htons(TCP_PORT);
      if (connect(connect_socket, (struct sockaddr
              *)\&s_in, sizeof(s_in)) ==
           SOCKET_ERROR)
        break;
      recv_rv = recv(connect_socket, input_buf,
              CHAR_ARRAY_SIZE, 0);
      if (recv_rv == SOCKET_ERROR || recv_rv == 0)
        break;
      data = atoi(input_buf);
    } while (0);
    if (connect_socket != INVALID_SOCKET)
      CLOSE_SOCKET(connect_socket);
    ifdef _WIN32 if (wsa_data_init) WSACleanup();
    endif
  }
  {
    int data_copy = data;
    int data = data_copy;
    {
      int data_buf[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
      if (data > = 0) {
        printIntLine(data_buf[data]);
      } else {
        printLine("ERROR: Array index is negative");
      }
    }
  }
}
```

Repaired

```
void CWE129_Improper_Validation_Of_Array_Index() {
  int data;
  data = −1;
  {
    ifdef _WIN32 WSADATA wsa_data;
    int wsa_data_init = 0;
    endif int recv_rv;
    struct sockaddr_in s_in;
    SOCKET connect_socket = INVALID_SOCKET;
    char input_buf[CHAR_ARRAY_SIZE];
    do {
      ifdef _WIN32 if (WSAStartup(MAKEWORD(2, 2),
              &wsa_data) != NO_ERROR) break;
      wsa_data_init = 1;
      endif connect_socket = socket(AF_INET,
              SOCK_STREAM, IPPROTO_TCP);
      if (connect_socket == INVALID_SOCKET)
        break;
      memset(&s_in, 0, sizeof(s_in));
      s_in.sin_family = AF_INET;
      s_in.sin_addr.s_addr = inet_addr("127.0.0.1");
      s_in.sin_port = htons(TCP_PORT);
      if (connect(connect_socket, (struct sockaddr
              *)&s_in, sizeof(s_in)) ==
           SOCKET_ERROR)
        break;
      recv_rv = recv(connect_socket, input_buf,
              CHAR_ARRAY_SIZE, 0);
      if (recv_rv == SOCKET_ERROR || recv_rv == 0)
        break;
      data = atoi(input_buf);
    } while (0);
    if (connect_socket != INVALID_SOCKET)
      CLOSE_SOCKET(connect_socket);
    ifdef _WIN32 if (wsa_data_init) WSACleanup();
    endif
  }
  {
    int data_copy = data;
    int data = data_copy;
    {
      int data_buf[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
      if (data > = 0 && data < 10) {
        printIntLine(data_buf[data]);
      } else {
        printLine("ERROR: Array index is
              out−of−bounds");
      }
    }
  }
}
```

Figure 26. Socket Array Access Error

Figure 27 shows a function that calls sprint to print out two strings, but only provides the first string to print. Our GAN repairs it by providing a second string.



With Vulnerability

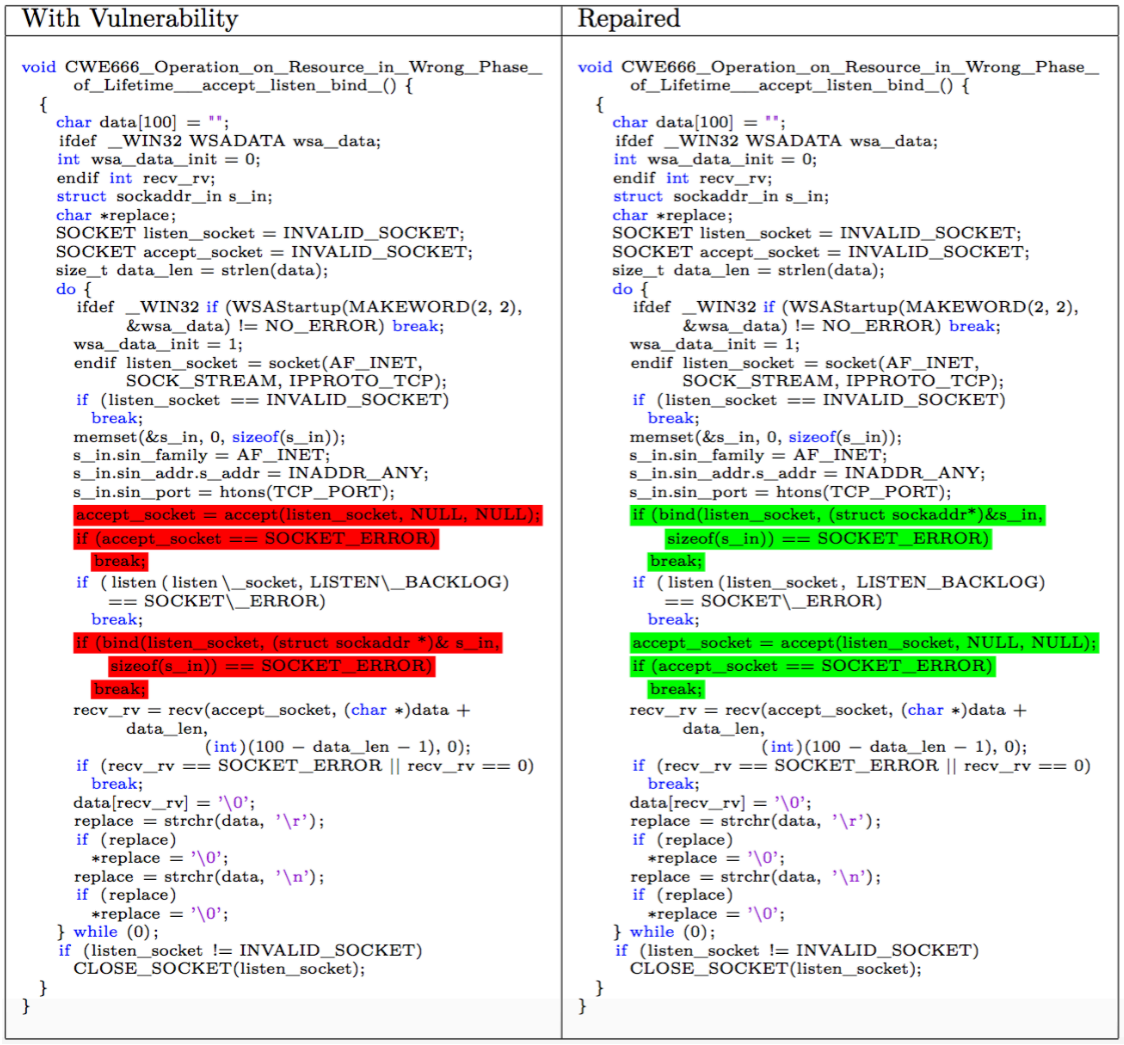
```
void CWE685_Function_Call_With_Incorrect_
        Number_Of_Arguments() {
  char dst[DST_SZ];
  sprintf(dst, "%s %s", SRC_STR);
  printLine(dst);
}
```

Repaired

```
void CWE685_Function_Call_With_Incorrect_
        Number_Of_Arguments() {
  char dst[DST_SZ];
  sprintf(dst, "%s %s", SRC_STR, SRC_STR);
  printLine(dst);
}
```

Figure 27. Format Print Error

Figure 28 shows a function that attempts to accept a socket and use it before it has bound it. Our GAN approach repairs the function by reordering the bind, listen, and accept into the correct order.



With Vulnerability

```
void CWE666_Operation_on_Resource_in_Wrong_Phase_
       of_Lifetime___accept_listen_bind_() {
  {
    char data[100] = "";
    ifdef _WIN32 WSADATA wsa_data;
    int wsa_data_init = 0;
    endif int recv_rv;
    struct sockaddr_in s_in;
    char *replace;
    SOCKET listen_socket = INVALID_SOCKET;
    SOCKET accept_socket = INVALID_SOCKET;
    size_t data_len = strlen(data);
    do {
      ifdef _WIN32 if (WSAStartup(MAKEWORD(2, 2),
            &wsa_data) != NO_ERROR) break;
      wsa_data_init = 1;
      endif listen_socket = socket(AF_INET,
            SOCK_STREAM, IPPROTO_TCP);
      if (listen_socket == INVALID_SOCKET)
        break;
      memset(&s_in, 0, sizeof(s_in));
      s_in.sin_family = AF_INET;
      s_in.sin_addr.s_addr = INADDR_ANY;
      s_in.sin_port = htons(TCP_PORT);
      accept_socket = accept(listen_socket, NULL, NULL);
      if (accept_socket == SOCKET_ERROR)
        break;
      if (listen(listen\_socket, LISTEN\_BACKLOG)
          == SOCKET\_ERROR)
        break;
      if (bind(listen_socket, (struct sockaddr *)& s_in,
          sizeof(s_in)) == SOCKET_ERROR)
        break;
      recv_rv = recv(accept_socket, (char *)data +
            data_len,
                  (int)(100 - data_len - 1), 0);
      if (recv_rv == SOCKET_ERROR || recv_rv == 0)
        break;
      data[recv_rv] = '\0';
      replace = strchr(data, '\r');
      if (replace)
        *replace = '\0';
      replace = strchr(data, '\n');
      if (replace)
        *replace = '\0';
    } while (0);
    if (listen_socket != INVALID_SOCKET)
      CLOSE_SOCKET(listen_socket);
  }
}
```

Repaired

```
void CWE666_Operation_on_Resource_in_Wrong_Phase_
       of_Lifetime___accept_listen_bind_() {
  {
    char data[100] = "";
    ifdef _WIN32 WSADATA wsa_data;
    int wsa_data_init = 0;
    endif int recv_rv;
    struct sockaddr_in s_in;
    char *replace;
    SOCKET listen_socket = INVALID_SOCKET;
    SOCKET accept_socket = INVALID_SOCKET;
    size_t data_len = strlen(data);
    do {
      ifdef _WIN32 if (WSAStartup(MAKEWORD(2, 2),
            &wsa_data) != NO_ERROR) break;
      wsa_data_init = 1;
      endif listen_socket = socket(AF_INET,
            SOCK_STREAM, IPPROTO_TCP);
      if (listen_socket == INVALID_SOCKET)
        break;
      memset(&s_in, 0, sizeof(s_in));
      s_in.sin_family = AF_INET;
      s_in.sin_addr.s_addr = INADDR_ANY;
      s_in.sin_port = htons(TCP_PORT);
      if (bind(listen_socket, (struct sockaddr*)&s_in,
          sizeof(s_in)) == SOCKET_ERROR)
        break;
      if (listen(listen_socket, LISTEN_BACKLOG)
          == SOCKET\_ERROR)
        break;
      accept_socket = accept(listen_socket, NULL, NULL);
      if (accept_socket == SOCKET_ERROR)
        break;
      recv_rv = recv(accept_socket, (char *)data +
            data_len,
                  (int)(100 - data_len - 1), 0);
      if (recv_rv == SOCKET_ERROR || recv_rv == 0)
        break;
      data[recv_rv] = '\0';
      replace = strchr(data, '\r');
      if (replace)
        *replace = '\0';
      replace = strchr(data, '\n');
      if (replace)
        *replace = '\0';
    } while (0);
    if (listen_socket != INVALID_SOCKET)
      CLOSE_SOCKET(listen_socket);
  }
}
```

Figure 28. Socket Binding Error

4.4 Program Synthesis

Here we describe some of the preliminary results of our work in program synthesis.

Our first experiment tested our ability to reconstruct functions from the sequences of production rules output by our grammar VAE. We reconstructed over 500,000 functions and found that all of them were syntactically valid. This is an improvement over the work of Kusner et al. [76] in two ways: (1) we guarantee syntactic validity even when the neural network outputs only a partial sequence (the neural network can only predict the first 50 production rules in the sequence, but this does not always correspond with a complete function); and (2) we successfully demonstrate that the grammar VAE architecture can handle a context-sensitive grammar, rather than a simple context-free grammar.

In our second experiment, we trained the grammar VAE on functions from our training corpus so that the model could learn a latent space representative of the kinds of functions found in our classifier and repair datasets. We found that our network architecture is comparable to that of Kusner et al. [76], but our vocabulary is more than three times as large. We tuned the model to minimize validation loss (a combination of cross-entropy loss and Kullback-Leibler divergence). We reviewed 1,000 randomly selected functions from the training dataset and verified that all 1,000 of these were parsed correctly.

In our third experiment, we demonstrate our ability to impose constraints on generated functions in a reliable and efficient manner. Table 15 shows these results. Our experiments have shown that we can enforce constraints with a 100% success rate without significant overhead. We can enforce the semantic validity constraint with a 99.97% success rate.

Table 15. Constraint Enforcement Success Rate

Constraint	Success Rate
Return Type Modifier	100%
Argument Number Modifier	100%
Argument Type Modifier	100%
No Loops	100%
Contains Loop	100%
Semantically Valid	99.97%

We anticipate one potential problem in our approach. While this approach can guarantee that a function has a vulnerability, it cannot guarantee that a function does not have a vulnerability. This could result in noisy labels for the negative (non-buggy) functions, but we anticipate that the number of mislabeled functions would be small.

4.5 Program Evaluations

Over the course of the three MUSE program phases, Draper supported program-wide evaluation events and hackathons hosted during the demonstration workshops. The sections that follow describe Draper's results from these events.

4.5.1 Phase 1 Hackathon.

Draper participated in a hackathon sponsored by DARPA at the end of Phase 1 of the MUSE program, concurrent with the Phase 1 Demonstration Workshop in February 2016. Draper's goal for this hackathon was to uncover vulnerabilities in the SATE IV dataset.

During the course of the hackathon, the Draper DeepCode classifier successfully identified the Heartbleed Bug [81] associated with the OpenSSL cryptographic software library. Figure 29 shows the example of the vulnerability identified in this case. The **memcpy** at line 1487 is an implicit loop with a loop bound identified by a variable **(payload)**. Neither this variable nor any of its ancestors is involved in a compare instruction. We verified that the Draper DeepCode classifier correctly assigned this function an "untaint" value of 0, indicating a loop variable that was not untainted (compared before use).

```
1464         n2s(p, payload);
1465         pl = p;
1466
1467         if (s->msg_callback)
1468                 s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
1469                         &s->s3->rrec.data[0], s->s3->rrec.length,
1470                         s, s->msg_callback_arg);
1471
1472         if (hbtype == TLS1_HB_REQUEST)
1473                 {
1474                 unsigned char *buffer, *bp;
1475                 int r;
1476
1477                 /* Allocate memory for the response, size is 1 byte
1478                  * message type, plus 2 bytes payload length, plus
1479                  * payload, plus padding
1480                  */
1481                 buffer = OPENSSL_malloc(1 + 2 + payload + padding);
1482                 bp = buffer;
1483
1484                 /* Enter response type, length and copy payload */
1485                 *bp++ = TLS1_HB_RESPONSE;
1486                 s2n(payload, bp);
1487                 memcpy(bp, pl, payload);
```

Figure 29. Tainted Loop Bound Associated with Heartbleed Bug

Figure 30 shows the corrected version of this code with the correct untaint operation. Here, the **payload** variable is involved in the assignment to **write_length** at line 1479. Then **write_length** is involved in the comparison at line 1484. We verified that the Draper DeepCode classifier correctly assigned this function an "untaint" value of 1, indicating a loop variable that was untainted (compared before use).

```
1479    unsigned int write_length = 1 /* heartbeat type */ +
1480                                2 /* heartbeat length */ +
1481                                payload + padding;
1482    int r;
1483
1484    if (write_length > SSL3_RT_MAX_PLAIN_LENGTH)
1485            return 0;
1486
1487    /* Allocate memory for the response, size is 1 byte
1488     * message type, plus 2 bytes payload length, plus
1489     * payload, plus padding
1490     */
1491    buffer = OPENSSL_malloc(write_length);
1492    bp = buffer;
1493
1494    /* Enter response type, length and copy payload */
1495    *bp++ = TLS1_HB_RESPONSE;
1496    s2n(payload, bp);
1497    memcpy(bp, pl, payload);
```

Figure 30. Untainted Loop Bound Associated with Corrected Code

4.5.2 Phase 2 Evaluation.

During Phase 2, Draper provided a remote interface and documentation for Leidos to conduct an independent evaluation of the DeepCode classification and repair tools. Leidos conducted this evaluation and reported at the Phase 2 Demonstration Workshop that Draper had met our evaluation criteria.

We evaluated classification accuracy using data held out from training (that is, not used in the training set) from the SATE IV dataset. We used SATE IV for the evaluation because this dataset comes with truth data to enable accuracy analysis. Table 16 indicates the accuracy with which DeepCode identified errors associated with different CWEs. Blank rows in Table 16 correspond to CWEs which were present in the training set but did not have enough statistics to be present in the test set.

Table 16. Classification Test Results on SATE IV by CWE

CWE	**Accuracy**	**%**
CWE15_External_Control_of_System_or_Configuration_Setting		
CWE114_Process_Control	2/2	100%
CWE121_Stack_Based_Buffer_Overflow	88/90	98%
CWE122_Heap_Based_Buffer_Overflow	115/115	100%
CWE123_Write_What_Where_Condition	2/2	100%
CWE124_Buffer_Underwrite	70/70	100%
CWE126_Buffer_Overread	1/1	100%
CWE127_Buffer_Underread	75/75	100%
CWE129_Improper_Validation_Of_Array_Index	22/22	100%
CWE131_Incorrect_Calculation_Of_Buffer_Size	24/24	100%
CWE134_Uncontrolled_Format_String	16/20	100%
CWE135_Incorrect_Calculation_Of_Multibyte_String_Length	1/1	100%
CWE170_Improper_Null_Termination	8/8	100%
CWE187_Partial_Comparison	15/15	100%
CWE190_Integer_Overflow	31/31	100%
CWE191_Integer_Underflow	14/14	100%
CWE193_Off_by_One_Error	5/6	83%
CWE194_Unexpected_Sign_Extension	9/9	100%
CWE195_Signed_To_Unsigned_Conversion	15/15	100%
CWE196_Unsigned_To_Signed_Conversion_Error	2/2	100%
CWE197_Numeric_Truncation_Error	9/9	100%
CWE242_Use_of_Inherently_Dangerous_Function	2/2	100%
CWE252_Unchecked_Return_Value		
CWE253_Incorrect_Check_of_Function_Return_Value	34/35	97%
CWE369_Divide_By_Zero	8/9	89%
CWE374_Passing_Mutable_Objects_to_Untrusted_Method	2/2	100%
CWE390_Error_Without_Action	36/38	95%
CWE401_Memory_Leak	38/42	90%
CWE415_Double_Free	49/50	98%
CWE416_Use_After_Free	45/50	90%
CWE457_Use_of_Uninitialized_Variable	29/29	100%
CWE459_Incomplete_Cleanup		
CWE467_Use_of_sizeof_on_Pointer_Type	5/6	83%
CWE468_Incorrect_Pointer_Scaling		
CWE469_Use_Of_Pointer_Subtraction_To_Determine_Size	7/10	70%
CWE476_NULL_Pointer_Dereference	4/6	67%
CWE480_Use_of_Incorrect_Operator		
CWE481_Assigning_instead_of_Comparing	2/2	100%
CWE482_Comparing_instead_of_Assigning		
CWE562_Return_Of_Stack_Variable_Address		
CWE587_Assignment_Of_Fixed_Address_To_Pointer		
CWE588_Attempt_To_Access_Child_Of_A_Non_Structure_Pointer		
CWE590_Free_Of_Invalid_Pointer_Not_On_The_Heap	43/45	96%
CWE606_Unchecked_Loop_Condition		
CWE680_Integer_Overflow_To_Buffer_Overflow	7/7	100%
CWE690_NULL_Deref_from_Return		
CWE761_Free_Pointer_Not_At_Start_Of_Buffer	2/2	100%
Overall	837/866	97%

To evaluate repair we constructed several datasets as follows:

- Synthesize extremely simple fix examples from templates, focusing specifically on buffer overflows.
- Inject simple buffer overflow examples into real functions.
- Gather bad-good function pairs from the SATE IV dataset.

During Phase 2 we were in the early stages of developing the repair capability; therefore, we focused on single-line repairs for simplicity in the Phase 2 evaluation. Table 17 summarizes results on test sets for the different datasets described above. "Localization %" is the percentage of examples in which the correct line to fix was identified. "Repair %" is the percentage of examples in which the new line was generated entirely correctly. The validation and test sets were a 50/50 split between bad-good pairs and good-good pairs. Therefore, some of the accuracy in Table 17 is due to the network correctly recognizing a good function that does not need to be repaired, while part comes from fixes being generated successfully.

Table 17. Repair Accuracy Summary

Evaluation Dataset	**Localization %**	**Repair %**
Basic templates	99.99%	99.99%
Injected templates	96.9%	75.9%
SATE IV (overflows only)	100%	100%
SATE IV (expanded bug set)	98.5%	96.7%

We can think of these datasets along two axes – repair complexity and number of bug variants. While the SATE IV dataset has relatively simple functions as examples, we were able to test across a wide variety of bug variants. The injected templates were more complex because they were injected into real code, but they covered a smaller breadth of bugs.

Figure 31 depicts accuracy relative to these two metrics.

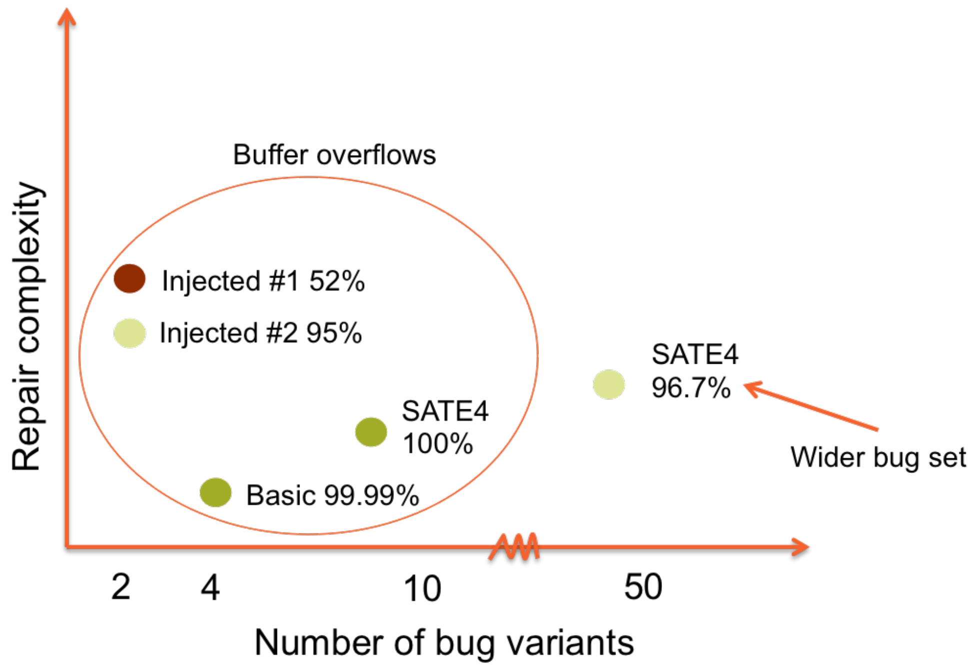


Figure 31. Repair Accuracy Dependence

One way of visualizing what the network has learned is to do a principal component analysis (PCA) of the encoding layer to see how it is separating examples. Figure 32 shows an example from our simplest set of buffer overflow templates. We can see three clusters, corresponding to the three templates from which we generated our dataset. The reason the network was able to perform so well is that it essentially memorized the templates.

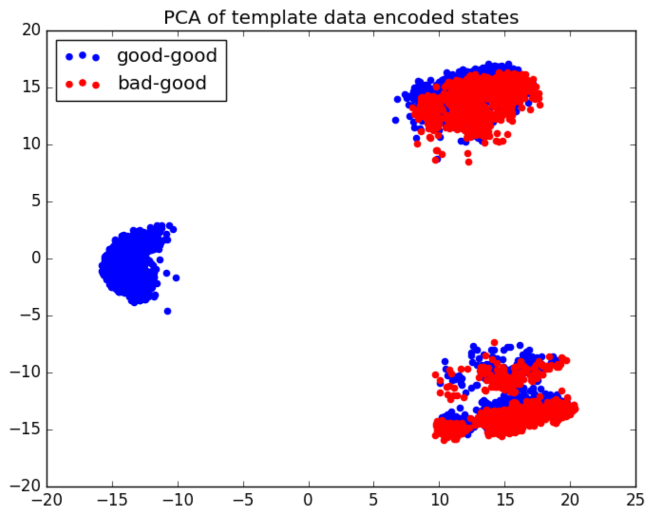


Figure 32. Repair PCA for Buffer Overflow Templates

On the more realistic dataset of templates injected into real functions, the network still shows an ability to separate the templates. However, the clustering is not as tight as seen previously because of the additional variation in the data. Figure 33 shows the PCA mapping of that network's encoding.

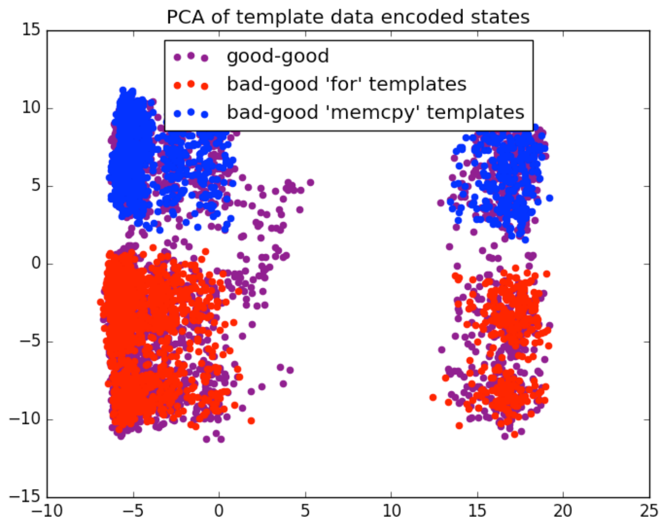


Figure 33. Repair PCA for Injected Templates

In this dataset, the aggregate performance was strongly dependent on the different templates we injected. One template, a bug where **memcpy** was used to copy too large a source array into a smaller destination array, showed very good repair performance, close to 95%. The other template, where the integer limit on a **for** loop was larger than an array that was indexed within the loop, showed worse performance, with only 52% accuracy. This is because in the second case, the network had to memorize the exact size of the array in order to generate a correct repair. While the network was able to correctly identify the line to be repaired most of the time, it did not sufficiently remember the integer value needed for a completely correct fix. In the **memcpy** case, only the correct ordering of the source and destination was necessary to generate the repair.

4.5.3 Phase 2 Hackathon.

Draper participated in a hackathon sponsored by DARPA at the end of Phase 2 of the MUSE program, concurrent with the Phase 2 Demonstration Workshop in May 2017. Draper's goal for this hackathon was to uncover vulnerabilities that had been injected into the Ardupilot C/C++ code by the system evaluator (Leidos). Since we had a build-feature-based classifier prepared for the Phase 2 hackathon, we analyzed the subset of functions from Ardupilot from which we were able to extract build features during the course of the hackathon.

We applied 3 different classifier models that we had previously trained in preparation for the hackathon: AE, RF, and RNN. We also combined individual models together to create new classification scores. We tested these models on 1,239 functions from Ardupilot that our pipeline ingested. Since we did not know ground truth for this test, we could not compute accuracy metrics. Instead, we reviewed functions that the classifiers identified as having a high probability of error and looked for issues in those functions.

Of the 1,239 functions analyzed, 127 of these had an ensemble score greater than 80%. Of these, we labeled 36 as especially suspect. Many of these were not necessarily vulnerabilities in and of themselves, but we identified some edge cases that could be triggered. Figure 34 shows the breakdown of the high-scoring examples by library, and we provide some detailed examples later in this section.

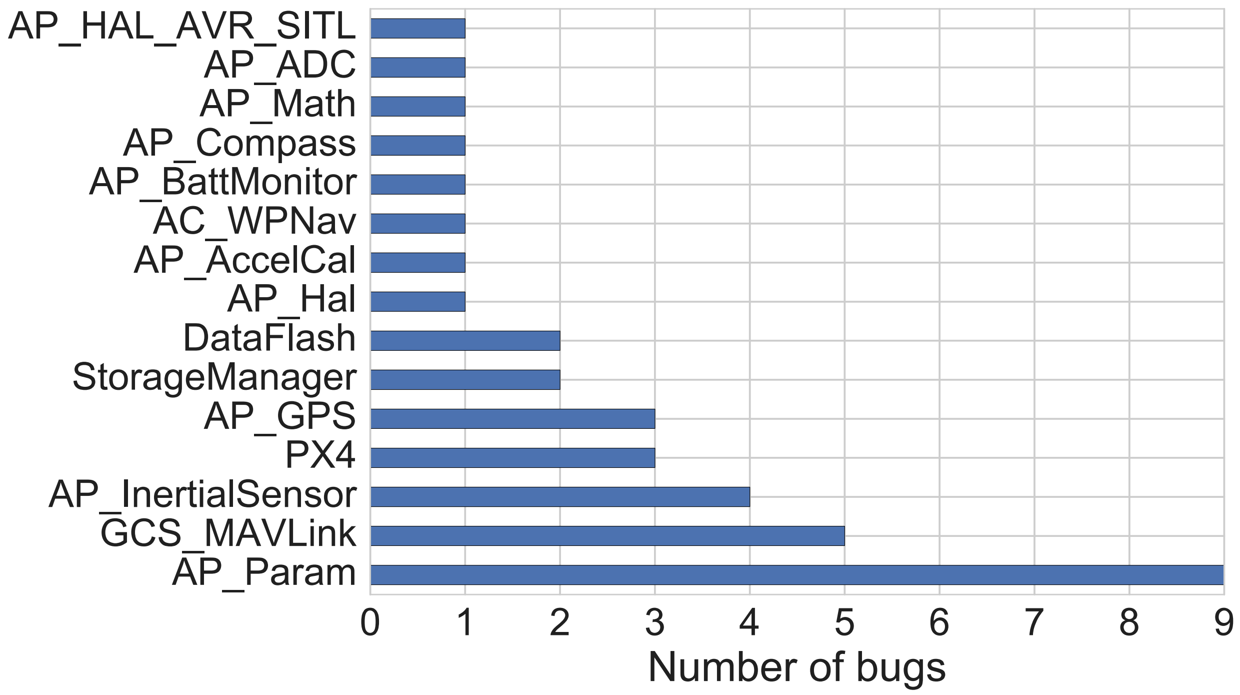


Figure 34. Error Detection Breakdown by Library

We also tried 3 different repair models that we had previously trained on the following data:

1. Hand-crafted bug templates injected into real code
2. SATE IV data only
3. SATE IV templates injected into real code

We found that the bugs we detected could not be patched with the types of patches in the training sets, but it was still useful to keep track of repair attempts as an additional measure of bug localization. Functions that had a repair attempted sometimes had higher classification scores as well. Out of the 36 verified high-scoring functions, 5 also had a repair attempt.

Figure 35 shows histograms of classification score and repair attempts for the three types of classifier (AE, RF, and RNN) and an ensemble classifier.

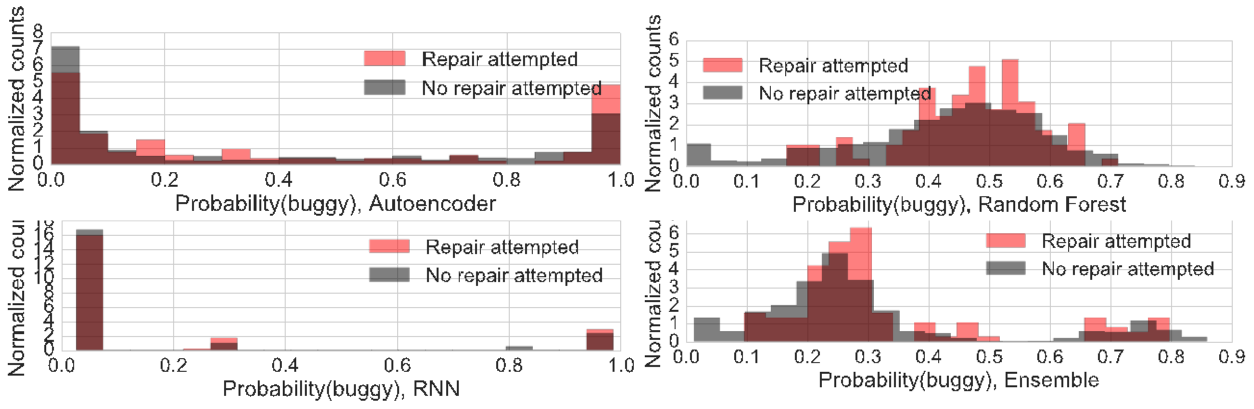


Figure 35. Classification Attempts and Repair Histograms

Figure 36 shows an example of a function tagged by the DeepCode classifiers. Note that the **for** loop highlighted in orange has an incorrect conditional statement ("length" should be "ofs < length").

```
void StorageManager::erase(void){
    uint8_t blk[16];
    memset(blk, 0, sizeof(blk));
    for (uint8_t i=0; i<STORAGE_NUM_AREAS; i++) {
        const StorageManager::StorageArea &area = StorageManager::layout[i];
        uint16_t length = pgm_read_word(&area.length);
        uint16_t offset = pgm_read_word(&area.offset);
        for (uint8_t ofs=0; length; ofs += sizeof(blk)) {
            uint8_t n = 16;
            if (ofs + n > length) {
                n = length - ofs;
            }
            hal.storage->write_block(offset + ofs, blk, n);
        }
    }
}
```

Figure 36. Error Example: For Loop Conditional Statement

Figure 37 shows an example of a function tagged by the DeepCode classifiers that may or may not be erroneous depending on how the function is used and its relationship to other software in the ecosystem. Note that in the operations highlighted in orange, a pointer is set and dereferenced without checking the size of the array to which the pointer points.

```
uint32_t AP_GPS_NMEA::_parse_decimal_100()
{
    char *p = _term;
    uint32_t ret = 100UL * atol(p);
    while (isdigit(*p))
        ++p;
    if (*p == '.') {
        if (isdigit(p[1])) {
            ret += 10 * DIGIT_TO_VAL(p[1]);
            if (isdigit(p[2]))
                ret += DIGIT_TO_VAL(p[2]);
        }
    }
    return ret;
}
```

Figure 37. Error Example: Pointer De-Reference

Figure 38 shows an example of a function tagged by the DeepCode classifiers in which an element of an array is referenced based on an input parameter to the function, without checking whether the parameter is within the bounds of the array. It is possible that this is checked before the function is called, but if the function is used without such a check, this could lead to errors.

```
void AP_Compass_Backend::correct_field(Vector3f &mag, uint8_t i){
   Compass::mag_state &state = _compass._state[i];

   if (state.diagonals.get().is_zero()) {
      state.diagonals.set(Vector3f(1.0f,1.0f,1.0f));
   }

   const Vector3f &offsets = state.offset.get();
   const Vector3f &diagonals = state.diagonals.get();
   const Vector3f &offdiagonals = state.offdiagonals.get();
   const Vector3f &mot = state.motor_compensation.get();

   /*
    * note that _motor_offset[] is kept even if compensation is not
    * being applied so it can be logged correctly
    */
   mag += offsets; …
```

Figure 38. Error Example: Array Index

Figure 39 shows an example of a double pointer that is de-referenced and used in a loop without validity checks. Again, it is possible that everything is checked before the function is called, or that the input is constructed in a way that guarantees this to work, but if the function is used differently, this could lead to errors.

```
static char *dequote_value(const char *varname, char *varval)
{
  const char **dqnam;
  char *dqval = varval;
  int len;

  if (dqval)
    {
      /* Check if the variable name is in the list of strings to be
dequoated */

      for (dqnam = dequote_list; *dqnam; dqnam++)
        {
          if (strcmp(*dqnam, varname) == 0)
            {
              break;
            }
}
```

Figure 39. Error Example: Double Pointer De-Reference

By running the DeepCode classifiers on the sample code from this hackathon, we were able to identify situations in the code where many assumptions were made about global buffers, pointers, and the like. While not necessarily vulnerabilities in and of themselves, they could pose potential pitfalls for a novice writing code in this environment, which could possibly lead to vulnerabilities being introduced.

At the time of the Phase 2 hackathon, we had only build-feature-based classifiers available that required the code to build in order to analyze it. Difficulties building the code in the environment of the DeepCode data pipeline made the analysis less efficient and comprehensive than it could have been. We took this as a lesson learned that led us to develop source-feature-based classifiers in Phase 3.

We also found that we needed to expand the training data substantially for our repair network. We also took this as a lesson learned that led us to develop the GAN for repair in Phase 3.

4.5.4 Phase 3 Challenge Problem.

For Phase 3 we had two challenge problems: one to support an analysis goal and one to support a demonstration goal. This section describes the challenge problems and our results relative to these challenge problems.

Challenge Problem 1 (Analysis Goal)

Challenge problem 1 was to demonstrate our ability to identify and repair C and C++ functions in one or more open-source benchmark datasets that have security vulnerabilities. The main challenge for this goal lay in finding a dataset with sufficient truth data to serve as a benchmark for evaluation. We investigated ManyBugs [17], Google OSS-Fuzz [82], Codeflaws [83], and the Juliet Test Suite [19] as possible benchmark datasets. ManyBugs provides differences before and after fixes in open-source software with enough information to determine the functions that changed, but with only a small number of security vulnerabilities for testing. Google OSS-Fuzz provides truth data for about 2000 security vulnerabilities but lacks resolution at the function level, which we needed for evaluation. Codeflaws provides truth data for over 7000 software bugs, but most of these are algorithmic issues that would not translate to security vulnerabilities. The Juliet Test Suite has the best truth data about vulnerabilities, so we used the Juliet Test Suite for this evaluation. Since some code from the Juliet Test Suite was also in the training corpus, we were very careful to hold out examples for the evaluation that we did not include in the training data.

The current state of practice consists of SA, which provides a fast analysis of software to look for known issues, and dynamic analysis, which is more flexible but takes orders of magnitude more time. Learning-based algorithms show promise to generalize better while taking similar (or even less) time than SA. In addressing our Phase 3 accuracy assessment challenge problem, we compared DeepCode vulnerability detection against a number of SA tools using held-out functions from the Juliet Test Suite. Figure 40 demonstrates that our machine learning approaches outperformed the SA tools in this comparison. We also determined that our pre-trained source-feature-based classifier takes about an order of magnitude less run time than the Clang SA because the source-feature-based classifier does not have to build the software first. DeepCode classifiers that use build features take about the same amount of time as the Clang SA because they need to run Clang to do the builds first.

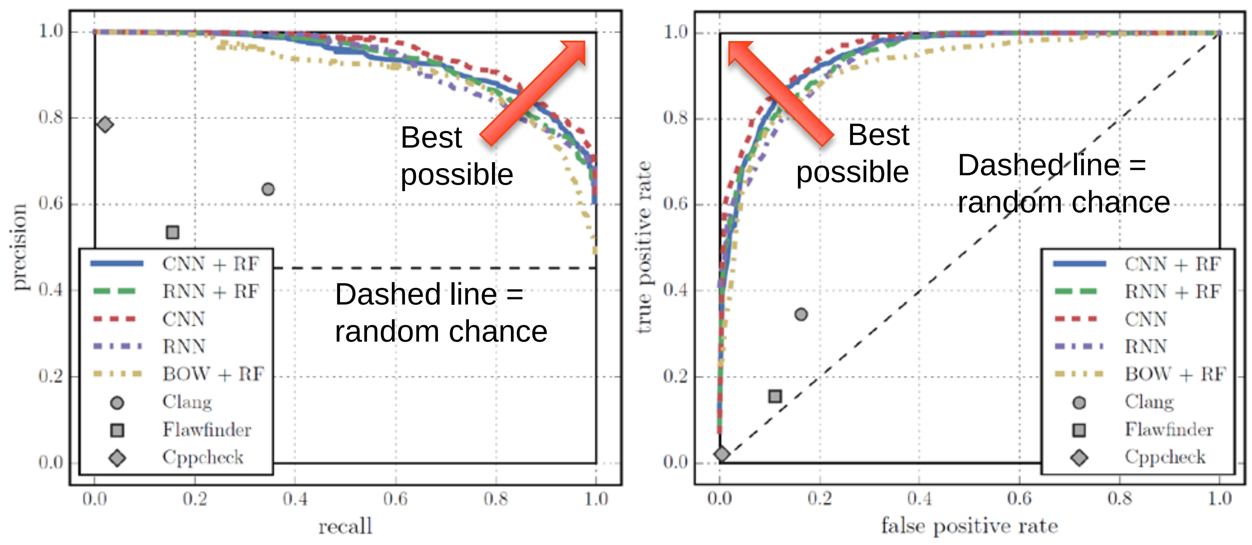


Figure 40. DeepCode Classifier Comparisons with SA on Juliet Test Suite

Challenge Problem 2 (Demonstration Goal)

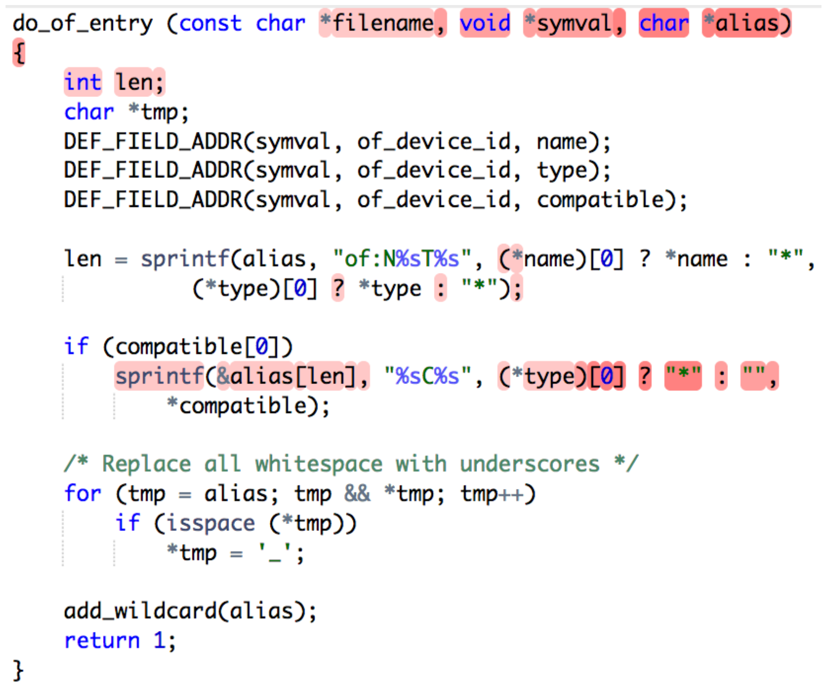
Challenge problem 2 was to identify and repair previously undiscovered security vulnerabilities in one or more open-source C / C++ software packages in widespread use. We chose the PureOS kernel as the code-under-test for this challenge problem due to its importance as a privacy-centric Linux distribution. Our initial attempts to apply DeepCode classifiers to this kernel software showed much lower accuracy than we obtained against other application software. Our hypothesis for this result was that the training corpus did not contain examples similar enough to kernel software from which the classifiers could learn patterns. As a result, we augmented the training corpus with examples from the Debian Linux kernel.

After augmenting the training corpus with Debian kernel code examples, we ran again the DeepCode classifier against the PureOS kernel code. We selected a threshold of 0.80 and identified 35 functions from the PureOS kernel code with classification scores above threshold (indicating that DeepCode ranked these 35 functions as having the highest likelihood of having vulnerabilities). We then asked an internal team of cybersecurity experts at Draper to inspect these 35 functions manually to determine how many were vulnerable. The cybersecurity expert team found that 17 of the 35 functions (about half) had vulnerabilities. By contrast, we would not expect 35 randomly selected functions to include vulnerabilities in nearly half of those functions. Table 18 provides the details of the DeepCode rankings and the manual inspection findings.

Table 18. Classification Demonstration Challenge Problem Results

File	Score	Manual inspection finding	Line number
scripts/pnmtologo.c	0.94	none	
scripts/kallsyms.c	0.92	none	
drivers/infiniband/core/cma.c	0.89	memcpy	1676
kernel/debug/kdb/kdb_main.c	0.89	sprintf	2615
drivers/gpu/drm/radeon/mkregtable.c	0.88	none	
drivers/net/wireless/brcm80211/brcmsmac/main.c	0.88	none	
init/do_mounts.c	0.87	none	
scripts/asn1_compiler.c	0.86	atoi usage … ; ' 'buffer' allocated but never free'd	557;576
drivers/net/usb/sr9800.c	0.85	none	
drivers/net/ethernet/mellanox/mlx4/en_rx.c	0.84	none	
scripts/mod/file2alias.c	0.84	sprintf	693
arch/x86/boot/compressed/mkpiggy.c	0.84	p: ilen is signed but treated as unsigned in print @L37@L67@L83 || fopen doesn't check for symlink @L51 || atoi doesn't have error handle, input must be str that can be converted to signed int, is deprecated, use strtol	
drivers/staging/lustre/lustre/obdclass/cl_lock.c	0.84	none	
drivers/connector/cn_proc.c	0.84	cn_proc_event_id is not declared in func scope	181
drivers/connector/cn_proc.c	0.84	none	
drivers/connector/cn_proc.c	0.84	cn_proc_event_id is not declared in func scope	265
drivers/connector/cn_proc.c	0.84	cn_proc_event_id is not declared in func scope	292
drivers/ata/ahci.c	0.83	none	
drivers/message/fusion/mptbase.c	0.83	none	
drivers/connector/cn_proc.c	0.83	none	
drivers/mfd/rtsx_pcr.c	0.83	memcpy	
mm/memcontrol.c	0.82	memcpy	3713
arch/x86/tools/relocs_common.c	0.82	none	
arch/x86/pci/i386.c	0.82	the value of 'dev' is not checked for null	314
sound/pci/mixart/mixart.c	0.82	strcpy, sprintf	990
sound/pci/mixart/mixart.c	0.82	strcpy, sprintf	1023
drivers/gpu/drm/i2c/adv7511.c	0.81	none	
net/mac80211/wpa.c	0.81	memcpy	559
scripts/mod/mk_elfconfig.c	0.81	none	
scripts/mod/file2alias.c	0.81	sprintf	796
Documentation/spi/spidev_fdx.c	0.80	none	
sound/usb/mixer_quirks.c	0.80	none	
drivers/gpu/drm/radeon/mkregtable.c	0.80	none	
drivers/net/wireless/brcm80211/brcmsmac/main.c	0.80	memcpy	1908
scripts/mod/modpost.c	0.80	sprintf possible overwriting other struct member, or even outside of struct data	2464

Figure 41 shows a selected example of one of the vulnerabilities identified by DeepCode. This is an example of CWE-120 (buffer copy without checking size of input). In this example, we also see the result of using backpropagation to isolate the location of the bug. In this case, backpropagation automatically highlighted the portions of the code that contributed to the vulnerability finding. This highlights backpropagation as a valuable tool for a user to isolate issues quickly that DeepCode finds.


```
do_of_entry (const char *filename, void *symval, char *alias)
{
    int len;
    char *tmp;
    DEF_FIELD_ADDR(symval, of_device_id, name);
    DEF_FIELD_ADDR(symval, of_device_id, type);
    DEF_FIELD_ADDR(symval, of_device_id, compatible);

    len = sprintf(alias, "of:N%sT%s", (*name)[0] ? *name : "*",
              (*type)[0] ? *type : "*");

    if (compatible[0])
        sprintf(&alias[len], "%sC%s", (*type)[0] ? "*" : "",
            *compatible);

    /* Replace all whitespace with underscores */
    for (tmp = alias; tmp && *tmp; tmp++)
        if (isspace (*tmp))
            *tmp = '_';

    add_wildcard(alias);
    return 1;
}
```

Figure 41. Classification Demonstration Challenge Problem Example

For the repair portion of the demonstration challenge problem, we attempted repairs on the errors we identified in the LibTIFF and PureOS software. The results of these repair attempts did not work out as well as we hoped. Some repair attempts produced compilable code, but others did not, and the repair attempts that compiled still didn’t make sense upon manual inspection. We reviewed the results and identified reasons for this performance. In these more realistic code examples (unlike the synthetic SATE IV examples), a one-to-many relationship exists between a given vulnerable function and several possible repairs. GAN training breaks down when the generator produces several possible repairs with similar probability.

The typical sequence-to-sequence network outputs the probability of an output sequence given a particular input sequence. This allows us to obtain the most probable repair candidates using an approach like beam search. We pass these probability outputs as one input to the GAN discriminator and real samples as another. GAN training then attempts to minimize the distance between the distribution of generator outputs and the real samples. Specifically, we use a WGAN, which attempts to minimize the Wasserstein-1 or EM distance. Unfortunately, it turns out that minimizing the Wasserstein-1 distance between these probabilistic generator outputs and the real samples does not minimize the distance between samples from the generator output and the real samples (which is what we want). This does occur for cross-entropy loss (used by sequence-to-sequence), but not for many other distance metrics. This means our method may produce some good samples, but will likely struggle when there are several possible repairs with relatively equal probability, which is more likely for real data than for synthetic data like SATE IV.

5. CONCLUSIONS

During the course of the DARPA MUSE program, Draper advanced the state of the art in automatic detection and repair of security vulnerabilities in software. In the sections that follow, we summarize these contributions, discuss the readiness of the technology for transition, and provide recommendations for future work in this area.

5.1 Classification

We have developed a fast and scalable vulnerability detection tool based on machine learning for the detection of bugs that can lead to security vulnerabilities in C/C++ code. We have shown that machine learning is able to effectively learn to detect vulnerability labels at the function level. In particular, Deep Learning is a powerful way of generating high-level features for vulnerability detection from both source code and build artifacts. When learning directly on lexed source code, Deep Learning approaches were able to learn the vulnerability labels from three different SA tools well. Deep Learning on build features was similarly effective for build feature-based vulnerability labels, but weaker on the labels from the two source-based SA tools. The most effective approach for detecting build-based SA labels was achieved by training the classifier on a combined set of simplified build features and source features derived through neural network classification training.

We also evaluated the ability of the DeepCode classifiers to find true vulnerabilities in software (as distinct from predicting SA labels). This was more difficult to quantify due to the limited availability of benchmark datasets that provide ground truth for evaluation. We were able to quantify accuracy with which we predicted ground truth labels for the Juliet Test Suite, and we showed that the DeepCode classifiers were more accurate than the three SA tools that we also evaluated. We also showed that vulnerabilities identified by the Vader dynamic analysis tool also scored highly for likelihood of vulnerability when analyzed by the DeepCode classifiers.

5.2 Repair

We have developed a GAN-based approach to train a system for software vulnerability repair. We demonstrated that our new approach is an effective technique for repairing software vulnerabilities, performing close to the state-of-the-art sequence-to-sequence approaches that require labeled pairs.

One of the main challenges of code repair is the lack of paired training data (*i.e.*, having training data consisting of corresponding vulnerable and repaired functions). In our comparisons between the GAN-based and sequence-to-sequence approaches, the sequence-to-sequence approach achieved higher accuracy metrics (BLEU scores), but this was for an evaluation on synthetic data, for which we already had labeled pairs available for training.

Given the training challenge and the results of our work, GANs (and generative models in general) represent the most promising path towards automatic code repair due to two important attributes. First, they harness the power of neural networks, which have been shown to provide state-of-the-art results on a variety of machine learning problems. Second, they can be trained on exclusively unpaired training data. This is crucial for real-world scenarios where paired training data is scarce if not non-existent.

5.3 Program Synthesis

Our program synthesis work provided a proof of concept that satisfactory performance could be obtained for generation of training examples. Kusner et al. [76] previously showed that the grammar VAE is a powerful generative model for grammar-based constructs that can achieve excellent performance on the sequence learning task. We showed that it could be applied to the domain of program synthesis.

5.4 Transition

Table 19 lists products and potential products of Draper's developments under the DARPA MUSE program and summarizes the readiness of each product for transition.

Table 19. Readiness for Transition of DeepCode Products

Product	**Readiness for Transition**
Portable error detection engine based on pre-trained networks	Mature. Docker image enables portability. More / better training data will improve accuracy. Would benefit from additional user interface design effort.
Full pipeline for "power users" that have substantial compute resources and datasets that they would like to train on	Mature. Docker image enables portability. More development needed to transition the training engine.
Automatic code repair engine	Docker image possible with modest additional development. Ongoing research needed to improve performance.
Tool to prioritize the effort required to find security vulnerabilities through code reviews	Mature. Docker image enables portability. FP are less of a concern for this use case because it still saves time and money over exhaustive code reviews.
Continuous integration (CI) tool to look for new errors as code is updated	Not currently developed, but would be possible with additional user interface and integration work. Would require careful management of FP.
Ongoing machine learning research for automatic error detection and repair	Best return would come from ongoing research into more / better labeled training data.

One of the products that is closest to readiness for transition is a tool to prioritize the effort required to find security vulnerabilities through code reviews. We see this as a particularly attractive application because this is a relatively mature application for DeepCode and because it can produce high value to users with the accuracy levels that we have already demonstrated under the MUSE program.

Figure 42 demonstrates how our classification accuracy metrics presented earlier in this report would translate into a user's experience using DeepCode as a code review prioritization tool. This analysis assumed a fictitious 10,000-lines-of-code program with typical industry-standard error discovery rates. The DeepCode classifiers provide a score that would allow a user to prioritize reviewing the highest-scoring findings first to find many vulnerabilities in a short time, thereby substantially reducing labor hours vs. code review alone. In this example, about 80% of the vulnerabilities that would normally be found through code review could be found in about 20% of the time that would be required for an exhaustive review of the code.

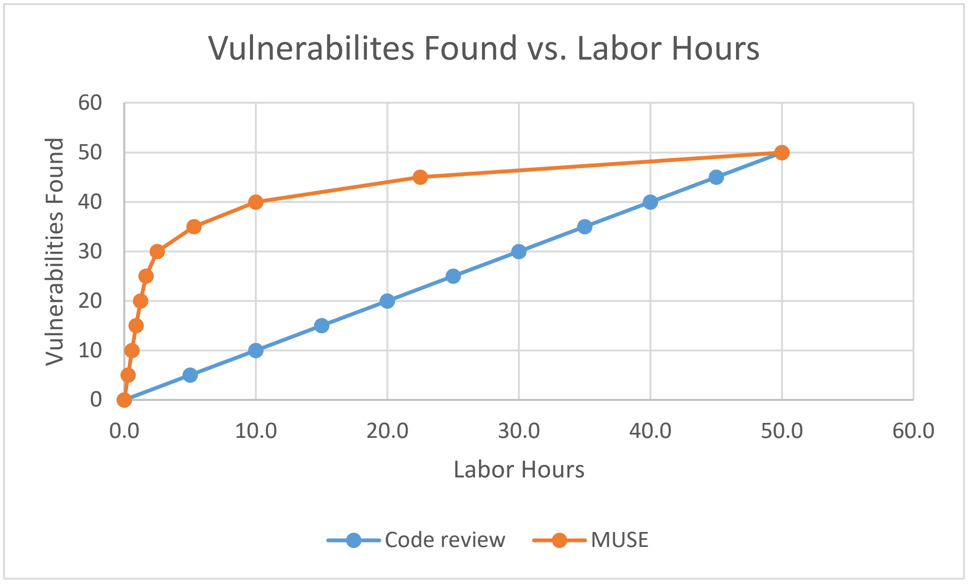


Figure 42. Vulnerabilities Found vs. Labor Hours

5.5 Recommendations for Future Work

5.5.1 Labeling.

Future work should focus on improved labels, such as those from dynamic analysis tools or mined from security patches. This would allow scores produced from the machine learning models to be more complementary with SA tools.

Figure 43 shows the current DeepCode classifier accuracy trend relative to the size of the training corpus. This shows that accuracy is still increasing as a function of training corpus size, meaning that we expect that there are significant accuracy improvements still available to be achieved through substantial increases in the amount of training data.

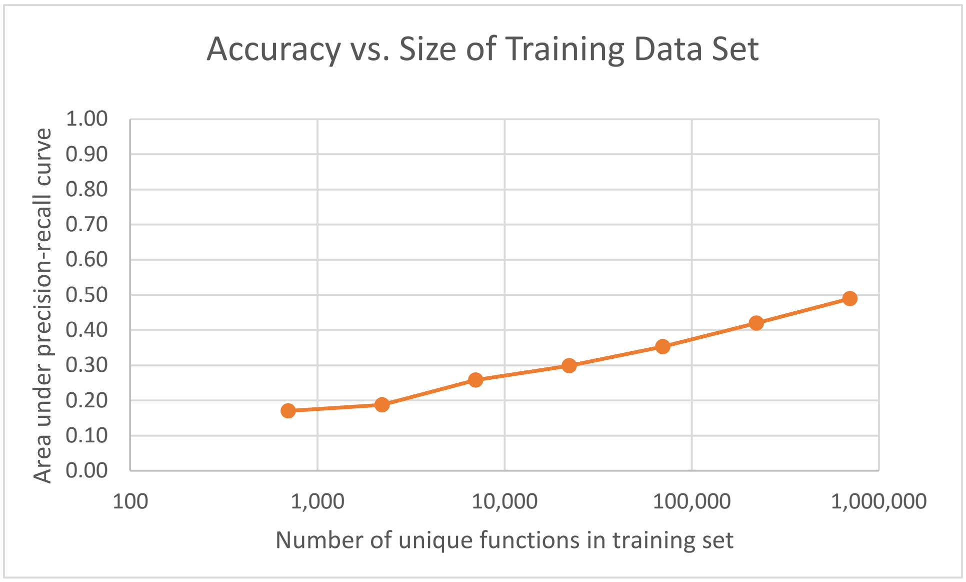


Figure 43. Classification Accuracy vs. Size of Training Dataset

5.5.2 Classification.

We posed the vulnerability detection as a binary classification problem indicating whether a given function contains at least one of the several vulnerabilities or not. Future work could develop multi-label classification approaches to specify the type of vulnerability to provide users more detailed code review experience.

5.5.3 Repair GAN.

Our initial GAN attempts have been very successful, but current performance is limited by two main issues. The first issue is that the discriminator has difficulty distinguishing valid repairs from invalid ones, which it needs to do in order to provide good instruction for the generator. The second issue is that the real data are samples from a distribution, whereas the desired generator output is an estimate of the distribution itself. This drives errors in GAN training when there are several possible repairs with relatively equal probability.

Both of these issues may be addressed by using an appropriate embedding on the input to the discriminator. This would serve as a continuous relaxation for the discrete samples in the real data, as well as allow for easier approximation of these samples by the generator. To this end we are experimenting with techniques from graph embedding in order to generate linear embeddings for our data which preserve interesting features. For example, one such embedding minimizes the distance in the embedded space between tokens which often appear adjacent to one another in the code.

As a second approach towards solving the difference in distribution problem, we are looking at the inclusion of noise at the input of the generator’s decoder, similar to how traditional GANs work. This removes the necessity for the discriminator to estimate the probability distribution and instead allows it to approximate samples from the distribution, thus providing the discriminator with far more similarity between generated and real inputs. However, this approach does have the issue that outputs from the generator are not dependent upon this noise vector, and as such it is difficult to determine which generated repairs are the most probable. We believe we may be able to rectify this by using transfer learning to train a new GAN which does not require noise from one that does.

5.5.4 Program Synthesis.

One area for future work in program synthesis is to expand the semantic content of generated functions. Examples of additional types of training data we might wish to generate include functions with nested loops, recursive functions, functions with return value constraints (say, only positive integers), or functions that exhibit simple semantic qualities such as not having statements after a return. We have designed an extensible framework in which constraints could be implemented with relative ease.

Opportunities to improve our semantic repair constraint include more meaningful variable names, better capture of the original user intent, and warnings during semantic repair. Even semantically valid code can produce warnings at compile time, and repaired code quality would be significantly improved by also resolving issues that produce warnings.

Our next suggested area for future work is to improve the vocabulary. The expressiveness of the current vocabulary is limited by the need to include context to the production rules to determine whether certain rules are valid additions to the function’s current rule sequence during the decode step. It may be possible to reduce or eliminate this need for context by developing an inference algorithm to perform these functions. This is complicated by the desire for a guarantee that all decoded functions are syntactically valid, but it might be possible to relax that restriction (allowing function decoding to produce invalid functions some small fraction of the time) and develop an approach that eliminates the need for context. This would significantly decrease the number of production rules needed to produce the same range of functions. This would enable adding more rules to the vocabulary by using more functions as input (allowing greater variety in generated functions) or including more semantic information, such as variable names and constant values (allowing the VAE to learn more semantic information).

The grammar VAE guarantees that all output sequences are syntactically valid by virtue of the associated context-free grammar. However, it makes no such guarantee when using a context-sensitive grammar. Extensive testing of our model supports the conclusion that all output sequences are syntactically valid, but there is no guarantee based on formal theory. The same is true for the semantically valid constraint. Future implementations of a grammar VAE in a context-sensitive environment would be improved by finding ways to make guarantees about syntactic and semantic validity.

6. REFERENCES

[1] The MITRE Corporation, "CVE - Common Vulnerabilities and Exposures (CVE)," 29 May 2018. [Online]. Available: https://cve.mitre.org/. [Accessed 8 June 2018].

[2] Business Wire, "Worldwide Revenue for Security Technology Forecast to Surpass $100 Billion in 2020, According to the New IDC Worldwide Semiannual Security Spending Guide," 12 October 2016. [Online]. Available: https://www.businesswire.com/news/home/20161012005102/en/Worldwide-Revenue-Security-Technology-Forecast-Surpass-100. [Accessed 8 June 2018].

[3] A. Krizhevsky, I. Sutskever and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, 2012.

[4] M. D. Zeiler and R. Fergus, "Visualizing and Understanding Convolutional Networks," 28 November 2013. [Online]. Available: https://arxiv.org/abs/1311.2901. [Accessed 11 June 2018].

[5] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," 10 April 2015. [Online]. Available: https://arxiv.org/abs/1409.1556. [Accessed 11 June 2018].

[6] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich, "Going Deeper with Convolutions," 2015. [Online]. Available: https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Szegedy_Going_Deeper_With_2015_CVPR_paper.pdf. [Accessed 11 June 2018].

[7] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," 10 December 2015. [Online]. Available: https://arxiv.org/abs/1512.03385. [Accessed 11 June 2018].

[8] K. Xu, J. L. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhutdinov, R. S. Zemel and Y. Bengio, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attentino," in *Proceedings of the 32nd International Conference on Machine Learning*, Lille, France, 2016.

[9] O. Vinyals, A. Toshev, S. Bengio and D. Erhan, "Show and Tell: A Neural Image Caption Generator," 2015. [Online]. Available: https://www.cv-foundation.org/openaccess/content_cvpr_2015/app/2A_101.pdf. [Accessed 11 June 2018].

[10] R. Collobert and J. Weston, "A unified architecture for natural language processing: deep neural networks with multitask learning," in *ICML '08 Proceedings of the 25th international conference on Machine learning* , Helsinki, 2008.

[11] R. Socher, C. C.-Y. Lin, A. Y. Ng and C. D. Manning, "Parsing Natural Scenes and Natural Language with Recursive Neural Networks," in *Proceedings of the 28th International Conference on Machine Learning*, Bellevue, WA, 2011.

[12] D. Bahdanau, K. Cho and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," 19 May 2016. [Online]. Available: https://arxiv.org/abs/1409.0473. [Accessed 11 June 2018].

[13] I. Sutskever, O. Vinyals and Q. V. Le, "Sequence to Sequence Learning with Neural

Networks," in *Advances in Neural Information Processing Systems 27*, 2014.

[14] Stanford Vision Lab, "ImageNet Large Scale Visual Recognition Challenge (ILSVRC)," 2015. [Online]. Available: http://www.image-net.org/challenges/LSVRC/. [Accessed 11 June 2018].

[15] The Medical Image Computing and Computer Assisted Intervention Society, "EndoVis," 2018. [Online]. Available: https://endovis.grand-challenge.org/. [Accessed 11 June 2018].

[16] DeepMind Technologies Limited, "AlphaGo," 2018. [Online]. Available: https://deepmind.com/research/alphago/. [Accessed 11 June 2018].

[17] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest and W. Weimer, "The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, 2015.

[18] National Institute of Standards and Technology, "SATE IV," [Online]. Available: https://samate.nist.gov/SATE4.html. [Accessed 12 June 2018].

[19] National Institute of Standards and Technology, "Software Assurance Reference Dataset," November 2017. [Online]. Available: https://samate.nist.gov/SRD/testsuite.php. [Accessed 12 June 2018].

[20] "National Institute of Standards and Technology," [Online]. Available: https://www.nist.gov/. [Accessed 14 July 2018].

[21] The MITRE Corporation, "Common Weakness Enumeration," 3 April 2018. [Online]. Available: https://cwe.mitre.org/data/index.html. [Accessed 20 July 2018].

[22] "Buildbot main page," [Online]. Available: https://buildbot.net/. [Accessed 14 July 2018].

[23] "strace," [Online]. Available: https://strace.io/. [Accessed 20 July 2018].

[24] "The LLVM Compiler Infrastructure," [Online]. Available: http://llvm.org/. [Accessed 20 July 2018].

[25] "TITAN Distributed Graph Database," [Online]. Available: http://titan.thinkaurelius.com/. [Accessed 20 July 2018].

[26] K. Yim , "TinkerPop3 Documentation," [Online]. Available: http://tinkerpop.apache.org/docs/3.3.3/reference/. [Accessed 20 July 2018].

[27] The Apache Software Foundation, "Apache Cassandra," 2016. [Online]. Available: http://cassandra.apache.org/. [Accessed 30 July 2018].

[28] "Clang Static Analyzer," [Online]. Available: https://clang-analyzer.llvm.org/. [Accessed 20 July 2018].

[29] D. A. Wheeler, "Flawfinder," [Online]. Available: https://www.dwheeler.com/flawfinder/. [Accessed 20 July 2018].

[30] "Cppcheck - A tool for static C/C++ code analysis," [Online]. Available: http://cppcheck.sourceforge.net/. [Accessed 20 July 2018].

[31] M. Allamanis, E. T. Barr, P. Devanbu and C. Sutton, "A Survey of Machine Learning for Big Code and Naturalness," *arXiv*, 2017.

[32] A. Hovsepyan, R. Scandarlato, W. Joosen and J. Walden, "Software vulnerability prediction using text analysis techniques," in *4th International Workshop on Security Measurements and Metrics*, 2012.

[33] Y. Pang, X. Xue and A. S. Namin, "Predicting vulnerable software components through n-

gram analysis and statistical feature selection," in *14th International Conference on Machine Learning and Applications (ICMLA),* 2015.

[34] L. Mou, G. Li, Z. Jin, L. Zhang and T. Wang, "TBCNN: A Tree-Based Convolutional Neural Network for Programming Language Processing," *CoRR*, vol. abs/1409.5718, 2014.

[35] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng and Y. Zhong, *CoRR,* vol. abs/1801.01681, 2018.

[36] "Travis CI," [Online]. Available: https://travis-ci.org/. [Accessed 15 November 2017].

[37] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *11th Joint Meeting on Foundations of Software Engineering*, 2017.

[38] Z. Xu, T. Kremenek and J. Zhang, "A memory model for static analysis of C programs," in *4th International Conference Leveraging Applications of Formal Methods, Verification, and Validation*, 2010.

[39] X. Zhang, J. Zhao and Y. LeCun, "Character-level Convolutional Networks for Text Classification," in *Advances in Neural Information Processing Systems*, 2015.

[40] C. N. d. Santos and M. Gatti, "Deep convolutional neural networks for sentiment analysis of short texts," in *International Conference on Computational Linguistics*, 2014.

[41] Y. Kim, " Convolutional Neural Networks for Sentence Classification," in *Empirical Methods in Natural Language Processing*, 2014.

[42] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus and Y. LeCun, "OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks," 2014. [Online]. Available: https://arxiv.org/abs/1312.6229.

[43] J. Ji, Q. Wang, K. Toutanova, Y. Gong, S. Truong and J. Gao, "A Nested Attention Neural Hybrid Model for Grammatical Error Correction," *Annual Meeting of the Association for Computational Linguistics (ACL),* pp. 753-762, 2017.

[44] Z. Yuan and T. Briscoe, "Grammatical error correction using neural machine translation," *North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL HLT),* 2016.

[45] A. Schmaltz, Y. Kim, A. M. Rush and S. M. Shieber, "Adapting Sequence Models for Sentence Correction," *Empirical Methods in Natural Language Processing (EMNLP),* 2017.

[46] Z. Xie, A. Avati, N. Arivazhagan, D. Jurafsky and A. Y. Ng, "Neural Language Correction with Character-Based Attention," *arXiv:1603.09727,* 3 2016.

[47] C. Chen, A. Seff, A. Kornhauser and J. Xiao, "DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving," in *The IEEE International Conference on Computer Vission (ICCV),* 2015.

[48] C. M. Bishop, Pattern Recognition and Machine Learning, New York: Springer, 2006.

[49] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *3rd International Conference for Learning Representations*, San Diego, 2015.

[50] S. Hershey, S. Chaudhuri, D. P. W. Ellis, J. F. Gemmeke, A. Jansen, C. Moore, M. Plakal, D. Platt, R. Saurous, B. Seybold, M. Slaney, R. Weiss and K. Wilson, "CNN Architectures for Large-Scale Audio Classification," in *International Conference on Acoustics, Speech and Signal Processing (ICASSP),* 2017.

[51] T. Mikolov, I. Sutskever, K. Chen and G. Corrado, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems,* 2013.

[52] J. T. Springenberg, A. Dosovitskiy, T. Brox and M. Riedmiller, "Striving for simplicity: The all convolutional net," in *International Conference on Learning Representations*, 2015.

[53] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva and A. Torralba, "Learning deep features for discriminative localization," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[54] O. Ozdemir, T. Allen, S. Choi, T. Wimalajeewa and P. K. Varshney, Copula base classifier fusion under statistical dependence, in press.: IEEE Trans. Pattern Analysis and Machine Intelligence, 2017.

[55] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville and Y. Bengio, "Generative Adversarial Networks," *Neural Information Processing Systems (NIPS),* 6 2014.

[56] I. J. Goodfellow, O. Vinyals and A. M. Saxe, "Qualitatively characterizing neural network optimization problems," *International Conference on Learning Representations (ICLR),* 2015.

[57] M. Arjovsky and L. Bottou, "Towards Principled Methods for Training Generative Adversarial Networks," *International Conference on Learning Representations (ICLR),* 2017.

[58] M. Arjovsky, S. Chintala and L. Bottou, "Wasserstein Generative Adversarial Networks," *International Conference on Machine Learning (ICML),* 2017.

[59] X. Chen, Y. Duan, R. Houthooft, J. Schulman, I. Sutskever and P. Abbeel, "Infogan: Interpretable representation learning by information maximizing generative adversarial nets," *Neural Information Processing Systems (NIPS),* 2016.

[60] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta and A. A. Bharath, "Generative Adversarial Networks: An Overview," *IEEE Signal Processing Magazine,* vol. 35, pp. 53-65, 2018.

[61] A. Radford, L. Metz and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *International Conference on Learning Representations (ICLR),* 2016.

[62] L. Yu, W. Zhang, J. Wang and Y. Yu, "SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient," *Association for the Advancement of Artifical Intelligence (AAAI),* 2017.

[63] Y. Zhang, Z. Gan, K. Fan, Z. Chen, R. Henao, D. Shen and L. Carin, "Adversarial Feature Matching for Text Generation.," *International Conference on Machine Learning (ICML),* 2017.

[64] O. Press, A. Bar, B. Bogin, J. Berant and L. Wolf, "Language Generation with Recurrent Generative Adversarial Networks without Pre-training," *1st Workshop on Subword and Character Level Models in NLP (SCLeM),* 2017.

[65] S. Rajeswar, S. Subramanian, F. Dutil, C. Pal and A. Courville, "Adversarial Generation of Natural Language," *2nd Workshop on Representation Learning for NLP (RepL4NLP),* 2017.

[66] M. Mirza and S. Osindero, "Conditional Generative Adversarial Nets," *arXiv:1411.1784,*

11 2014.

[67] Z. Yang, W. Chen, F. Wang and B. Xu, "Improving Neural Machine Translation with Conditional Sequence Generative Adversarial Nets.," *North American Chapter of the Association for Computational Linguistics (NAACL),* 2018.

[68] J.-Y. Zhu, T. Park, P. Isola and A. A. Efros, "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks.," *International Conference on Computer Vision (ICCV),* 2017.

[69] A. N. Gomez, S. Huang, I. Zhang, B. M. Li, M. Osama and L. Kaiser, "Unsupervised Cipher Cracking Using Discrete GANs," *International Conference on Learning Representations (ICLR),* 2018.

[70] G. Lample, L. Denoyer and M. Ranzato, "Unsupervised Machine Translation Using Monolingual Corpora Only," *International Conference on Learning Representations (ICLR),* 2018.

[71] A. Shrivastava, T. Pfister, O. Tuzel, J. Susskind, W. Wang and R. Webb, "Learning from Simulated and Unsupervised Images through Adversarial Training.," *Computer Vision and Pattern Recognition (CVPR),* 2017.

[72] M.-T. Luong, H. Pham and C. D. Manning, "Effective Approaches to Attention-based Neural Machine Translation," *Empirical Methods in Natural Language Processing (EMNLP),* 2015.

[73] P. Vincent, H. Larochelle, Y. Bengio and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders.," *International Conference on Machine Learning (ICML),* 2008.

[74] R. J. Williams and D. Zipser, "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks.," *Neural Computation,* 1989.

[75] J.-Y. Zhu, T. Park, P. Isola and A. A. Efros, "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks," in *International Conference on Computer Vision and Pattern Recognition (CVPR),* Honolulu, HI, 2017.

[76] M. J. Kusner, B. Paige and J. Hernandez-Lobato, "Grammar Variational Autoencoder," *arXiv,* 2017.

[77] GitHub, Inc., "eliben / pycparser," 2018. [Online]. Available: https://github.com/eliben/pycparser. [Accessed 5 August 2018].

[78] Free Software Foundation, Inc., "GCC, the GNU C Compiler," 30 July 2018. [Online]. Available: https://gcc.gnu.org. [Accessed 5 August 2018].

[79] Purism, "PureOS," [Online]. Available: https://pureos.net/. [Accessed 20 July 2018].

[80] K. Papineni, S. Roukos, T. Ward and W.-J. Zhu, "BLEU: a Moethod for Automatic Evaluation of Machine Translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL),* 2002.

[81] Synopsys, Inc., "Heartbleed Bug," 2017. [Online]. Available: http://heartbleed.com/. [Accessed 13 July 2018].

[82] Google, "OSS-Fuzz," GitHub, Inc., 2018. [Online]. Available: https://github.com/google/oss-fuzz. [Accessed 13 July 2018].

[83] S. H. Tan, J. Yi, S. Mechtaev and A. Roychoudhury, "Codeflaws: A Programming

Competition Benchmark for Evaluating Automated Program Repair Tools," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017.

LIST OF SYMBOLS, ABBREVIATIONS AND ACRONYMS

AE	auto encoder
AI	artificial intelligence
ANN	artificial neural network
API	application programming interface
AST	abstract syntax tree
AUC	area under curve
BLEU	Bilingual Evaluation Understudy
BOW	bag of words
CFG	control flow graph
CI	continuous integration
CNN	convolutional neural network
CPU	central processing unit
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DARPA	Defense Advanced Research Projects Agency
Draper	The Charles Stark Draper Laboratory, Inc.
EM	earth movers
FN	false negatives
FP	false positives
FPR	false positive rate
GAN	generative adversarial network
GCC	GNU C Compiler
GPU	graphics processing unit
IR	intermediate representation
JSON	JavaScript Object Notation
LLVM	formerly known as Low Level Virtual Machine
LSTM	long-short term memory

MCC	Matthews Correlation Coefficient
MFI	MUSE function index
MICCAI	Medical Image Computing and Computer Assisted Intervention
MMD	maximum mean discrepancy
MUSE	Mining and Understanding Software Enclaves
NIST	National Institute of Standards and Technology
NLP	natural language processing
PCA	principal component analysis
PR	precision-recall
ReLU	rectified linear unit
RF	random forest
RMSProp	root mean square propagation
RNN	recurrent neural network
ROC	receiver operating characteristic
SA	static analysis
SATE	Static Analysis Tool Exposition
SVM	support vector machine
t-SNE	t-distributed stochastic neighbor embedding
TA	technical area
TN	true negatives
TP	true positives
TPR	true positive rate
VAE	variational auto encoder
WGAN	Wasserstein Generative Adversarial Network