Towards security defect prediction with AI

Carson D. Sestili
CERT - Software Engineering Institute
Carnegie Mellon Univerity
Pittsburgh, USA
cdsestili@cert.org

William S. Snavely
CERT - Software Engineering Institute
Carnegie Mellon Univerity
Pittsburgh, USA
wsnavely@cert.org

Nathan M. VanHoudnos
CERT - Software Engineering Institute
Carnegie Mellon Univerity
Pittsburgh, USA
nmvanhoudnos@cert.org

***Abstract*—Deep learning cannot yet predict security defects in C code.**

***Index Terms*—deep learning, static analysis, memory networks, question-answering**

I. Introduction

Predicting security defects in source code is a significant area of study. It is ideal to detect security defects during development, before the code is ever run to expose those defects. The current best methods to find security defects before running code are static analysis tools, a variety of which exist and model software in different ways that are all useful for different kinds of flaws. Developers of static analyzers carefully equip them with rules about program behavior, which are used to reason about the safety of the program if it were to run.

However, static analyzers are known to be insufficient at finding flaws. The Juliet Test Suite [1]–[3] is a collection of synthetic code containing intentional security defects across hundreds of CWEs, labeled at the line-of-code level. Even state-of-the art static analyzers perform poorly at finding the defects in Juliet, issuing too many false positives and also too many false negatives [4]–[7]. Finding an automated method that detects security defects in Juliet, with acceptably low false-positive and false-negative counts, is an open problem [3].

Artificial intelligence (AI) as a broad field, and machine learning as a tool for AI, have both seen great advances over the last decade (CLASSIC CITATIONS HERE). An attractive property of many data-driven AI systems that ingest large amounts of data and learn an accurate model of the underlying distribution is that the features that are most important to describe the model do not need to be explicitly extracted. These systems often discover "hidden" features that a human hand-crafting a model might never think to include. For recent problems where features are hard to describe but data is abundant, deep learning has proven to be an especially useful technique, both for extracting relevant features and for creating predictive models that use them (MORE RECENT CITATIONS HERE).

This environment naturally leads to investigating whether data-driven techniques can succeed in modeling source code. Some work has already been done in AI on code [8], mostly on problems about modeling gaps in the code from surrounding context; for example, the model of code2vec [9] is able to suggest the name of a method by reading the code in the method body. However, there is little work done yet in using AI to model security defects. The most significant barrier to this work is the lack of large enough quantities of data with accurately labeled security defects on which to train machine learning models. For example, the Juliet test suite is not ideal for training such models, even though it has labeled defects, because it has too few examples and is too complex.

Choi et al [10] are to our knowledge the first to propose a data-driven model to find security defects in code at the line-of-code level. This work shows that a deep learning model can in principle be used to model buffer overwrites. They show this by constructing a synthetic code dataset with labeled safe and unsafe buffer writes, which we call *CJOC-bAbI*, and training and testing on this codebase. However, their dataset does not contain real code–it is not syntactically valid C, and does not compile. Moreover, none of the functions in their dataset have any non-trivial control flow–there are no conditionals or loops, which would always be found in code developed for real applications. Although their work is a significant first step in demonstrating the utility of deep learning for code modeling, these weaknesses leave important questions unanswered.

To more thoroughly test the utility of AI for code understanding, a better dataset is needed, that can demonstrate whether a machine learning system can accurately model non-trivial control flow. If this dataset is free of other numerous sources of complexity usually found in source code, then researchers will be able to cleanly focus on developing models that are able to capture the relevant details of control flow.

The contributions of this work are as follows:

- We produce a code dataset that we call *s-bAbI*, containing syntactically-valid C programs with non-trivial control flow, and safe and unsafe buffer writes labeled at the line-of-code level. Although this code is simple as compared to real-world code, static analyzers do not perform with as high accuracy as expected, so we offer low-hanging fruit to improve them.
- We demonstrate the limits of the work by Choi et al, showing that their deep learning model needs too much training data to be reasonably used to model security defects, and that their approach necessarily overfits to synthetic datasets, not easily generalizing to real code.

Funding blinded for review.

- We point towards future approaches that may solve these problems; namely, using representations of code that can capture appropriate scope information and using deep learning methods that are able to perform arithematic operations.

II. Motivating Example

An example file from our dataset (`77056b8250.c`) with non-trivial control flow is shown in Figure 1.

```
 1 #include <stdlib.h>                // OTHER
 2 int main()                        // OTHER
 3 {                                 // OTHER
 4     int entity_4;                 // BODY
 5     char entity_8[11];            // BODY
 6     int entity_3;                 // BODY
 7     int entity_0;                 // BODY
 8     entity_0 = 9;                 // BODY
 9     entity_3 = rand();            // BODY
10     entity_4 = 42;                // BODY
11     if (entity_3 < entity_0){     // BODY
12     } else {                      // BODY
13     entity_3 = 69;                // BODY
14     }                             // BODY
15     while(entity_4 < entity_3){   // BODY
16     entity_4++;                   // BODY
17     }                             // BODY
18     int entity_9;                 // BODY
19     char entity_7[50];            // BODY
20     entity_8[entity_4] = 's';     // BUFWRITE_COND_UNSAFE
21     entity_9 = 75;                // BODY
22     entity_7[entity_9] = 'S';     // BUFWRITE_TAUT_UNSAFE
23     return 0;                     // BODY
24 }                                 // OTHER
```

Fig. 1. A sample function from our generated dataset.

It is fairly easy to tell that the buffer write at line 22 is unsafe. Doing so requires resolving the value of the index variable `entity_9` and the length of the array `entity_7`, and observing that the index is greater than the length of the array. No knowledge of the control flow structure of the code granted by the `if` statement and the `while` loop is necessary to decide that this buffer write is unsafe. The existing CJOC-bAbI dataset frames all of its buffer overflow examples in a setting not much more complicated than this example, only requiring variable lookup and a single comparison.

It is more challenging to tell that the buffer write at line 20 is unsafe. To see this, first we note that `entity_3` can take any integer value on line 9. This means that the condition in the `if` statement on line 11 is sometimes true and sometimes false. Whenever it is false, the `else` branch executes, setting `entity_3` to 69. Since `entity_4` is originally 42 (from line 10), which is less than 69, the `while` loop on line 15 executes, setting the value of `entity_4` to 69. But the length of the array `entity_8` is 11 (from line 5), so the index is greater than the length of the array at the buffer write on line 20, and this access is unsafe.

To our knowledge, this is the first paper showing that a deep learning model has the ability to predict security defects, at the line-of-code level, in synthetic code with control flow elements like conditionals and loops. Our model successfully predicts that both of the buffer writes in the above example are unsafe.

III. Approach

We develop a synthetic code generator that addresses shortcomings in both the CJOC-bAbI dataset and the Juliet Test Suite that prevent these datasets from being useful for developing a data-driven approach to security defect detection. The CJOC-bAbI dataset is not made of syntactically valid C code, and none of the examples contain code with any non-trivial control flow elements like conditionals, loops, or variables with unknown values, so a data-driven technique trained and tested on this dataset provides a weak argument that a similar method will work on real code. On the other hand, the Juliet Test Suite dataset is far too small and too complex for current data-driven methods to learn to predict the labeled security defects. Our dataset presents a compromise, as it is complex enough to contain non-trivial control flow elements, but simple and large enough to successfully train data-driven approaches. In this section, we describe details of our dataset, give an overview of the static analyzers we test on this dataset, and describe the deep learning architecture we compare against these static analyzers by training and testing on this dataset.

A. s-bAbI: AI tasks for source code

We propose a synthetic C source code dataset generator, which we call *s-bAbI*. The purpose of this generator is to provide the simplest possible source code, labeled at the line-of-code level with safe and unsafe buffer writes, that contains control flow structures found in real-world code. The motivation is analogous to that of the original English-language bAbI dataset, but adapted to the purpose of code understanding, namely, that any automated system that claims to be able to detect buffer overflows at the line-of-code level should achieve high predictive performance on this dataset.

1) Code: Each file in the s-bAbI dataset is syntactically-correct C source code, consisting of a single `void main()` function. This is an improvement over the dataset developed by Choi et al (citation), which contains syntactically-invalid "C-like" code examples.

Every s-bAbI code example has at least one of the following three elements that are found in any non-trivial real-world code:

- Conditional statements (`if`/`else`).
- Loops (`for` and `while`).
- Variables with unknown values (set from `rand()`).

We acknowledge that these are not the only constructions that are found in non-trivial real-world code, e.g. function calls would be the next reasonable step to add complexity. However, even these three elements add enough complexity to identify weaknesses of both existing static analyzers and the deep learning model proposed by Choi et al, meeting the purpose of this dataset.

Every line of an s-bAbI instance that does not modify control flow contains exactly one of the following:

- Integer variable declaration
- Integer variable set to an integer literal

- Integer variable set to the result of `rand()`
- Integer variable increment via `++`
- Character array declaration with integer literal length
- Character array set at integer variable index (*buffer write*)

Each buffer write is either *safe* or *unsafe*, meaning that the character array is written to at a value less-than-or-equal-to its length, or greater than its length, respectively.

2) Labels: The generator that produces the s-bAbI dataset also produces exactly one label per line, from the following six labels:

- `BUFWRITE_COND_SAFE`
- `BUFWRITE_COND_UNSAFE`
- `BUFWRITE_TAUT_SAFE`
- `BUFWRITE_TAUT_UNSAFE`
- `BODY`
- `OTHER`

The *conditional* labels `BUFWRITE_COND_SAFE`, `BUFWRITE_COND_UNSAFE` refer to lines that contain exactly one buffer write, which is provably either safe or unsafe (buffer overflow), respectively, such that reasoning about the control flow is required to prove whether the write is safe or unsafe. We refer to this kind of vulnerability as "conditional" since the safety of these buffer writes conditionally depends on the program's control flow. For example, an integer index may be set to a value less than the array's length in the true branch of an `if` statement, and a value greater than or equal to the array's length in the false branch. Knowing whether the buffer write afterward is safe requires knowing which branch was taken. For instances where an integer index is first set to an unknown value through the use of `rand()`, the label is `BUFWRITE_COND_SAFE` if *every* initial value of this variable results in safe control flow, and `BUFWRITE_COND_UNSAFE` otherwise.

The *tautological* labels `BUFWRITE_TAUT_SAFE`, `BUFWRITE_TAUT_UNSAFE` refer to lines that contain exactly one buffer write, which is again provably either safe or unsafe, respectively, but whose safety can be determined without reasoning about control flow. We call this kind of vulnerability "tautological", abusing this term slightly, because their safety depends on no information about the program's control flow. In these instances, the integer index is always set exactly once in the main scope of the function, not inside any of the control flow structures.

The `*COND*` buffer writes always occur in the main scope of the function, and are always reachable. The `*TAUT*` labels are allowed to occur inside the scope of both reachable and unreachable control flow structures, so they denote the safety of these lines under the assumption that they can be reached. Since multiple buffer writes can occur in a file, they are labeled as if the program would reach these lines even in the case of an earlier unsafe write (that is, as if the program would not crash after trying to unsafely write to a buffer). The motivation behind this labeling scheme is that it is ideal for tools to be able to identify *all* potentially unsafe lines in a function, not only those which can provably be reached, so that the developer can be alerted of all bugs simultaneously.

The `BODY` label refers to a line in the body of the function that does not contain a buffer write. The `OTHER` label refers to a line outside the body of the function.

B. Static Analysis Engines

A variety of static analysis techniques are applicable to the buffer overflow detection problem. As a simple motivating example, suppose we wish to determine, for a given array index operation in a program, if it is possible for the index to be outside the bounds of the array. In the general case, for a Turing-complete programming language, answering this question is undecidable. Analyses therefore make trade-offs between soundness and completeness [11]. A sound analysis will never accept an incorrect program—it yields no false negatives. A complete analysis will always accept a correct program—it yields no false positives. Due to the noted theoretical limitations, a sound program analysis cannot be complete, nor a complete analysis sound.

The theory of abstract interpretation provides a framework for sound static analyses [12]. Under this approach, one develops sound abstractions of program semantics, which are used to compute over-approximations of program behavior. For example, for each array access in a program, we could over-approximate the domains of the array size and index, and soundly identify bounds violations—at the risk of yielding false positives.

Other analysis approaches sacrifice soundness with the goal of yielding fewer false positives and/or improving analysis performance. Loop analysis is one domain where this trade-off can be made. For example, a static analysis might only compute the effects of a finite number of loop iterations, instead of soundly computing the effect of the loop. For the array bounds problem, this heuristic certainly could yield false negatives, for instance if a loop contains an array index that is initially safe, but which goes out of bounds after some number of iterations. Saxena discusses loop analysis in more detail [13].

A 2004 study of static analysis tools applied to buffer overflows provides an overview of techniques that remains relevant, touching on abstract interpretation, symbolic execution, and model checking [14]. This study also discusses some of the practical problems tools face, such as aliasing and interprocedural analysis. A large body of research has focused on refining and extending these techniques, for example improving the precision and efficiency of sound analyzers ([15]–[18]), and building better symbolic execution engines [19].

Our work uses four static analysis tools for C. Three of these tools are open source: Frama-C, the Clang static analyzer, and Cppcheck. We also used a commercial static analysis tool (anonymized). Frama-C provides "a collection of scalable, interoperable, and sound software analyses" for ISO C99 source code [20]. We used the value analysis plugin to Frama-C to look for buffer overflows. This plugin uses abstract interpretation to compute information about integers and pointers in C programs, and issues warnings about possible out-of-bounds accesses. See [20] for more information

about the abstract domains employed by Frama-C to model these program entities. The Clang static analyzer is based on symbolic execution, and, by default, makes use of unsound heuristics such as loop unrolling to contend with state space explosion[1]. We believe Cppcheck also makes use of unsound heuristics, though little has been published about the specific approach of this tool. The commercial tool we used is well-known to be unsound.

C. Memory Network

We test the effectiveness of a data-driven, deep learning-enabled approach to predicting buffer overflows using a *memory network* architecture. Weston et al (citation) and Sukhbaatar et al (citation) developed and refined memory networks to answer questions about English-language stories in the bAbI dataset, as described in greater detail in Section V. Choi et al (citation) performed the first application of memory networks to the task of buffer overflow prediction in synthetic code. We use the same architecture as Choi et al, except for five key differences, noted below with asterisks.

1) Data Preprocessing: We prepare a C file for processing by the following procedure.

1) Split the file into lines of code, and tokens within each line of code, using the `libclang` (citation) utility.
2) (*) Prepend each line with a special token `<line i>` indicating its line number. Whereas Choi et al do not add line numbers, we do, as a simple preprocessing step that does not require any additional parameters to be optimized in the model. Line numbers provide sequential information to the memory network, which has no way of representing line order.
3) Using a consistent mapping from tokens to integers $\{1, 2, \ldots, V-1\}$ where $V-1$ is the number of unique tokens, convert each token to an integer.
4) Save the integer tokens in an array, using zero padding on the right (dimension 1, to fill the rest of each line) and bottom (dimension 0, to fill the missing lines), obtaining an array of shape $N_F \times N \times J$, where N_F is the number of files, N is the maximum number of lines in a file, and J is the maximum number of tokens in a line. The elements of the array are the V integers $\{0, \ldots, V-1\}$.

2) Network Architecture: We describe the forward pass of the memory network.

Input:

- A program code X $[N \times J]$, consisting of N lines $X_1, \ldots, X_N$, where each line X_i is a list of integer tokens $w_i^1, \ldots, w_i^J$
- A query line q $[1 \times J]$, equal to one of the lines X_i encoding a buffer write

Embedding: We fix an embedding dimension d and establish two learnable embedding matrices E_{val} and E_{addr}, both of dimension $V \times d$. Letting A represent both E_{val} and E_{addr}, we encode each integer token twice, letting Aw_i^j $[1 \times d]$ be the w_i^j-th row of A. We use the position encoding of Sukhbaatar et al (citation) to encode lines: for $i = 1, \ldots, N$, define m_i $[1 \times d]$ by

$$m_i = \sum_{j=1}^{J} l_j \cdot Aw_i^j$$

where $\cdot$ denotes elementwise multiplication and l_j $[1 \times d]$ is defined by its k-th element as

$$l_j^k = (1 - j/J) - (k/d)(1 - 2j/J)$$

(*) We apply Dropout (citation) with parameter 0.3 to each line, so that

$$m_i \ [1 \times d] = \text{Dropout}_{0.3}(m_i)$$

We store the lines m_i encoded by E_{val} in a matrix M_{val} $[N \times d]$, and store the lines encoded by E_{addr} in a matrix M_{addr}. We embed the query line q by E_{addr} and store the result in u^1 $[1 \times d]$.

Memory search: For each "hop number" $h = 1, \ldots, H$ in a fixed number of "hops" H:

$$\begin{aligned}
p \ [N \times 1] &= \text{softmax}(M_{\text{addr}} u^T) \\
o \ [1 \times d] &= \sum_{i=1}^{N} p_i (M_{\text{val}})_i \\
(*) \ r \ [1 \times d] &= R_h o \\
(*) \ s \ [1 \times d] &= \text{Norm}_h(r) \\
u^{h+1} \ [1 \times d] &= u^h + s
\end{aligned}$$

where R_h $[d \times d]$ is an internal learnable weight matrix, and Norm_h is a batch normalization layer (citation) with parameters θ_h.

Classification:

$$\widehat{y} \ [2 \times 1] = \text{softmax}(W(u^H)^T)$$

where W $[2 \times d]$ is a learnable weight matrix.

The forward pass of the network is effectively an iterative inner-product search (citation) matching the current query line u^h, which changes with each processing hop, against each line m_i of the stored memory, which remains fixed.

We use the standard cross-entropy loss function to evaluate goodness of fit. To train, we use gradient descent to minimize the loss across the training set as a function of the learnable parameters $E_{\text{val}}, E_{\text{addr}}, R_h, \theta_h, W$. Like Choi et al, we use the Adam (citation) learning rate optimizer. (*) Although Choi et al claim they use a learning rate of 1e-2, we only found acceptable results with a learning rate of 1e-3. We train each network for 30 epochs. To compensate for class imbalance, we use a random sampling technique that always creates batches with input programs and accompanying query lines such that the number of query lines with each label are equal.

IV. Evaluation

Research questions

- RQ1: How accurate are static analyzers and memory networks on predicting buffer overflows as a function of code complexity?

[1] http://lists.llvm.org/pipermail/cfe-dev/2017-February/052818.html

- RQ2: How does the performance of static analyzers compare with memory networks as a function of training set size?
- RQ3: What considerations are unique to memory networks as compared to other tools that predict buffer overflows?

A. Methods

The s-bAbI dataset contains four different labels for lines with buffer writes: two kinds of unsafe writes, and two kinds of safe writes, as described above. We trained our memory networks with these four labels, so that e.g. if the network predicts that a label is `BUFWRITE_TAUT_SAFE` when the true label is `BUFWRITE_COND_SAFE`, the loss function penalizes this as a wrong answer. In contrast, static analyzers give warnings when they encounter lines that they find unsafe, and are not designed to communicate any reasoning process related to control flow. We will say that a static analyzer gives a positive result on a buffer write line whenever it generates a buffer overflow warning; we say that the tool gives a negative result when it does not issue a buffer overflow warning on that line. Research Questions 1 and 2 are about comparing memory networks to static analyzers. To fairly compare memory networks to static analyzers, for these questions, we collapse both ground-truth labels and network predictions to two classes, unsafe (positive) and safe (negative). Research Question 3 is only about memory networks. In that discussion, we use the four classes without collapsing.

As described in Section III-A2, we intentionally create labels under the assumption that the program does not crash when it attempts to perform an unsafe buffer write. However, some static analyzers have a *sound* model of software, described below, that assumes the program stops executing after an unsafe buffer write attempt. For these tools, we created a "sound subset" of the test dataset, attempting to minimize the number of buffer writes queried that occur in program execution after the first unsafe buffer write. Since we interpret the tool's output as "safe" whenever it does not issue a warning, this minimizes the number of times where we inappropriately interpret "the program never gets there" as safety.

We created the sound subset by querying only the unsafe buffer write with the smallest line number, and all (safe) buffer writes with smaller line numbers, in each function. From the set of all 76549 buffer writes in the full test set, the sound subset retains 51468 (67%).

This is not a perfect solution, as sometimes the first unsafe buffer write reached in a program's control flow is not the same as the unsafe buffer write with the lowest line number. However, we believe that this solution provides a fair enough testing ground for the sound tools.

B. RQ1

In Table I, we compare the performance of the memory network and several static analysis tools (open-source tools `clang_sa`, `cppcheck`, `frama-c`, and a commercial tool)

TABLE I
F1 (%) PER TASK.

tool	CJOC-bAbI	s-bAbI	s-bAbI (sound)	Juliet
Mem Net	59.6,71.3,77.9	90.9,91.7,92.5	89.7,90.6,91.7	—
`clang_sa`	—	72.9	84.9	8.0
comm. tool	—	93.0	91.9	40.6
`cppcheck`	—	80.4	76.9	2.2
`frama-c`	—	85.2	98.6	16.3

on the labeled buffer overflow datasets discussed. We use the F1 metric, the harmonic mean of precision and recall, as a single-number estimate of the model's predictive performance (on a scale from 0 to 1 where higher is better). The memory network's F1 scores are listed in the minimum, median, and maximum of 10 independent runs, differing only by randomness during training in the initialization of network parameters and in the random order in which training examples are shown to the network.

1) CJOC-bAbI: The CJOC-bAbI column shows our attempt to replicate the results of Choi et al on their own dataset, released on Github (reference). Specifically, we use their training set of 10,000 functions and the "level 4" testing set of 1,000 functions, representing the most challenging test cases. The memory network achieves moderately good performance on the CJOC-bAbI dataset, with a median F1 of 71.3%. However, this is significantly less than the F1 of 82% that they report in their paper. Since they "averaged the scored of the ten best cases with the smallest training error," but do not specify how many cases they chose the ten best out of, we believe that they may have generated many more than ten experiments, artificially inflating their performance.

The static analyzers' performance is not displayed for the CJOC-bAbI dataset because although their dataset contains "C-like" code, there are several issues present that prevent any example from being valid C. Therefore, it is not possible to run static analyzers on the dataset without significantly altering the data.

2) s-bAbI: We display performance of the memory network and each static analyzer on the full s-bAbI testing set, as well as on the sound subset. The full testing set has 38,400 files and 76,549 buffer writes, while the sound subset has 51,468 buffer writes. The tools are able to run without altering the data because every s-bAbI example is valid C. In this section, memory network results are shown for the largest training set, with 153,600 example files, and a total of 307,650 labeled buffer writes among all the files. Results that evaluate the memory network's performance as a function of training set size are shown in section IV-C.

Although the memory network has high performance on the s-bAbI datasets, with median F1 of 91.7% and 90.6%, respectively, it does not decisively outperform all of the static analyzers on either dataset. One strength of the memory network is that its recall is higher than that of all static analyzers on the full testing set, and on all but `frama-c` in the sound subset, indicating that it is able to correctly warn

on a greater proportion of the unsafe lines, as seen in Table II. Since the memory network functions by finding patterns that look like buffer overflows, it has the opportunity to pick up on many more *potential* buffer overflows than static analyzers, which may fail to find a proof that a given line contains a buffer overflow. This intuition is supported by the fact that `frama-c` has much greater recall than the memory network on the sound subset, where it is able to find proofs of buffer overflow a significantly greater proportion of the time.

TABLE II
RECALL (%) PER TASK.

tool	s-bAbI	s-bAbI (sound)
Mem Net	86.7, 88.2, 90.6	84.5, 86.0, 88.7
`clang_sa`	57.3	73.9
comm. tool	86.8	85.0
`cppcheck`	68.8	64.2
`frama-c`	74.2	97.2

However, the pattern-finding function of the memory network causes it to also generate too many false positives, so its precision is less than that of the static analyzers, as seen in Table III.

TABLE III
PRECISION (%) PER TASK.

tool	s-bAbI	s-bAbI (sound)
Mem Net	93.9, 95.4, 96.0	94.0, 95.5, 96.4
`clang_sa`	99.9	99.9
comm. tool	100	100
`cppcheck`	96.7	95.9
`frama-c`	100	100

We emphasize that since the memory network was trained only on our synthetic training dataset, its predictive capability is only high on a dataset with a similar distribution. The testing dataset was generated from the same dataset as the training distribution, so the good performance meets expectations. We would not expect the memory network trained on this dataset to be able to predict buffer overflows in any code that does not come from the s-bAbI data generator.

The static analyzers show fairly good performance on the s-bAbI datasets, although surprisingly not as high as we would hope, since these are fairly simple test cases without the complexity of function calls, and with a maximum of three control flow nodes. Moreover, there is no clear best static analyzer, as shown by the difference in performance between the full s-bAbI test set and the sound subset. For example, since `frama-c` uses a sound model of program execution, it performs significantly better on the sound subset.

3) Juliet: The memory networks failed to converge in training on the Juliet buffer overflow dataset. In every training run, after only a few epochs, the network's weights would be incorrectly tuned so that it would either always predict "safe" or always predict "unsafe." This is because the Juliet dataset is too small and too complex for the memory network architecture to succeed in learning. Although the Juliet dataset is, like CJOC-bAbI and s-bAbI, a synthetic dataset, it contains a relatively small number (5,906) of C files, containing 4096 lines with buffer overflow. The Juliet dataset is also complex: there are 116 unique functional variants of buffer overflow vulnerabilities presented. Therefore for each functional variant of buffer overflow vulnerability presented, there are only a small number of files that show examples of that variety. There is not enough data present for the memory network to successfully learn the distribution of the dataset.

We also see that the static analyzers have very low performance on the Juliet dataset, confirming previous findings (citation).

4) Discussion: Although memory networks are competitive with static analyzers on the s-bAbI dataset, and have the advantage of greater recall, they achieve this success only when being trained on a large, simple, synthetic code dataset. This method does not easily generalize when applied to a small, complex dataset like Juliet, as there is simply not enough data to allow the memory network to learn any of the code's relevant structure. Although this work shows that a deep learning method has the ability to learn structure on code with non-trivial control flow, advancing the initial work by Choi et al, significant effort is needed to find a method that will generate to code of real-world complexity.

This work also confirms previous findings that static analyzers have low performance at predicting security vulnerabilities in the Juliet dataset. A developer wishing to improve the performance of a static analyzer might be overwhelmed by the complexity of the Juliet dataset and the number of false positives and false negatives generated by their tool. The s-bAbI dataset is considerably less complex than Juliet, but is still valid C code with control flow structures found in real-world code, presenting low-hanging fruit for developers to easily improve their tools.

C. RQ2

For memory networks to achieve the high level of performance at predicting buffer overflow vulnerabilities in the s-bAbI dataset, as described in Section IV-A, they need to be trained on a large dataset, with 153,600 example files, and a total of 307,650 labeled buffer writes among all the files. For a data-driven technique to be applied to real code, access to such a large labeled code dataset is a significant limiting factor. To show the effect of training set size on network performance, we ran training runs with identical network parameters on training datasets of increasing size, evaluating each network on the same full testing set and sound subset as in Section IV-B2. The training set size ranges from 9,600 to 153,600 files, doubling in each experiment. We ran ten training experiments for each training set size. The results are shown in Figure 2.

Larger training set sizes are correlated with greater overall performance, shown by the median F1 rising with each increase of the training set size, although we see diminishing gains as the training set gets larger. On the full validation set, the median F1s were 84.8% for the smallest training set,

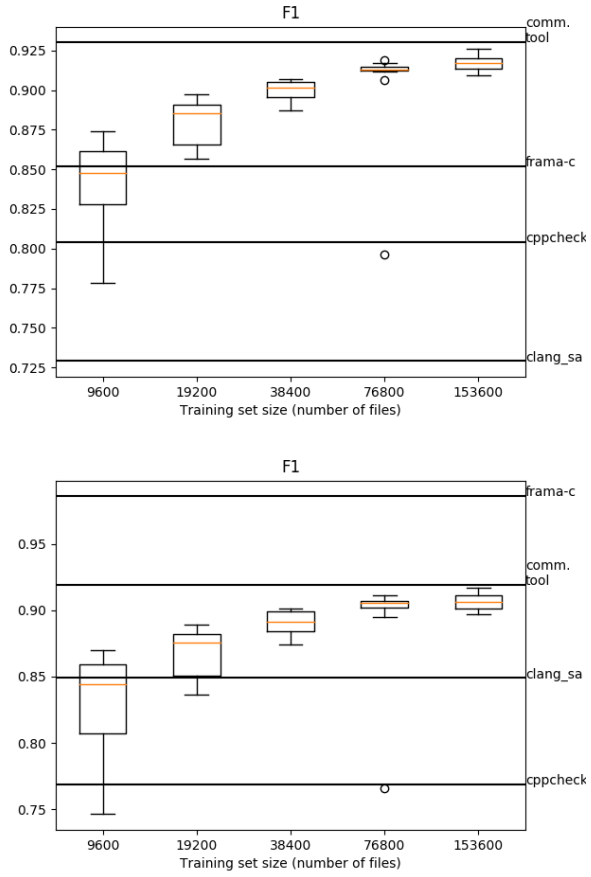

Fig. 2. F1-score as a function of training set size (top) as evaluated on the entire validation subset (bottom) as evaluated on the sound subset.

followed by 88.5%, 90.1%, 91.3%, and finally 91.7% for the largest training set.

Larger training set sizes are also correlated with lower variability between runs. The networks trained on the smallest training set are highly variable in predictive performance, with minimum and maximum F1 scores on the full testing set of 77.8% and 87.4%, respectively. Increasing the training set size tends to reduce the variability, with the networks trained on the largest training set having minimum and maximum F1 of 90.9% and 92.5%, respectively. We caution that there are some exceptions; the networks trained on the training set of size 76,800 mostly had very small variance in F1, in the range 90.6%–91.9%, but one run had an F1 of 79.6%. We intentionally do not exclude this as an outlier, to show that even with very large training sets, unfavorable gradient descent conditions can lead to poor performance.

Memory networks trained in the purely-supervised method of this work need a large amount of labeled training data to show competitive performance with static analyzers. Since this amount of labeled training data about security vulnerabilities is difficult to find and expensive to create for real-world code, this simple approach is not a viable method for finding security vulnerabilities with the data sources currently available. A training method that is more conservative with the need for labeled data, such as a semi-supervised or active learning method, as well as a data representation that more accurately represents the structure of source code, such as the bag of AST paths in (code2vec citation), or both, will be necessary to create a viable data-driven method of security vulnerability detection.

D. RQ3

Here we consider some properties of the trained memory networks that help illuminate how they work differently than static analyzers. These tests are conducted on the full testing set, since the memory networks do not use a sound model of program behavior and do not need to be restricted to the sound subset. As described in Section IV-A, we trained our networks on four labels, which capture not only the safety of a line (`SAFE` or `UNSAFE`), but also the scope of reasoning required to prove the safety (`COND` or `TAUT`). For the purpose of these tests, since we are no longer comparing to static analyzers, we keep all four labels as they are, without collapsing.

1) Recognizing vulnerability type: In addition to distinguishing safe from unsafe, the memory networks are able to distinguish conditional buffer writes from tautological, as shown in the confusion matrix in Table IV. Across the ten networks that were trained on the largest training set, we took the median value of each cell in their confusion matrices. Rows indicate the true label. Columns indicate the predicted label.

TABLE IV
MEDIAN CONFUSION MATRIX.

	C_S	C_U	T_S	T_U
C_S	11378.5	1324.5	0	0
C_U	4230	21465.5	0	0
T_S	0	0	18226	672
T_U	0	0	947	18303

A block-diagonal structure is evident, showing that the memory network is decisively able to distinguish between conditional and tautological buffer writes. Since the integer indices for conditional buffer writes always appear in the conditions of the `if` statements and the loop guards of the `for` and `while` loops, and the indices for the tautological buffer writes never appear in these places, we believe that the network is making the conditional-or-not decision based on the co-occurrence of variable names in lines with these control flow keywords.

2) Failure to generalize: Test set with integers mapped to one integer

TABLE V
MEDIAN CONFUSION MATRIX ON UNSEEN INTEGER SET.

	C_S	C_U	T_S	T_U
C_S	2956	9741	0	0
C_U	3083	22604	1	0
T_S	0	0	16859	2038
T_U	0	1	17066	2182

E. Threats to validity

2-3 paragraphs.

Briefly acknowledge shortcomings.

The sound subset is not perfect, but is a good approximation to highlight the strengths of the sound tools.

V. Related Work

Our works builds on three distinct threads in the literature: research on question answering tasks, the emerging literature of artificial intelligence for software engineering, and the development of data sets to assess software security tools.

A. Question Answering Tasks

Question answering is a classic problem in text retrieval. For example, the NIST sponsored Text REterival Conference (TREC) has had a Question Answering track since 1999 [21]. Current question answering systems are able to rival human performance for a subset of tasks focusing on answering realistic reading comprehension questions [22], [23].

Our work is closely related to that of Weston's group at Facebook, where they took an alternative approach: for a simple neural network, find the limits of the questions it can answer by generating synthetic, labeled data which are sufficiently complex to break the network, and then improve the neural network until the tasks are able to be cleared [24], [25]. More specifically, the memory network architecture first proposed by Weston et al. [24] to answer reading comprehension questions about short stories in the English language, trained and tested on the original bAbI dataset [25]. Weston et al's training scheme involved supplying the network with a story, broken into sentences, a single-sentence question about the story. The ground-truth data that they used during training included both a single-word correct answer to the question, and the set of sentences in the story that were relevant to answer the question. Sukhbaatar et al. [26] refined the memory network architecture so that the set of relevant sentences did not need to be supplied during training, an improvement that they give the name *end-to-end training*. We use this approach in our work, although in the domain of source code instead of natural language.

B. Artificial Intelligence for Software Engineering

There is much recent interest in applying artificial intelligence and machine learning techniques to a variety of software engineering tasks; see [8] for a comprehensive review.

Our work builds on Choi et al. [10], which used a variant of the Sukhbaatar et al. [26] architecture to predict buffer overflow vulnerabilities in synthetic source code. They trained and tested on a synthetic code dataset that they created, which we refer to as *c-bAbI*. Their work, however, has several limitations: first, *c-bAbI* was not valid C, making it difficult to compare against existing static analysis tools. Second, *c-bAbI* only generated basic blocks, and the absence of loops, conditionals, and variables of unknown value make their test cases far too simple.

Our work is a natural extension of Choi et al. [10] where we address the major shortcomings of their paper: we generate valid C code instead of 'C-like' code; we incrementally increase the complexity of the code generated to include conditionals and loops; and we study how well the combination of the representation used to input the code into the neural network and the network architecture interact during training.

C. Software Security Data Sets

The development and testing of software security data sets is well established in the literature. We note, however, that much of the existing work on software security data sets focuses primarily on finding realistic defects in realistic code, a high, and important threshold to cross. For example, the NIST Software Assurance Metrics And Tool Evaluation (SAMATE) project [27], which has coordinated the release of several data sets (Juliet [3], SARD [2], IARPA Stone Soup [28] and hosted competitions for static analysis tools [29]–[32], focuses on realistic test cases. Similarly, LAVA [33] and many other efforts ([4], [14], [34]–[37]) have focused on the realism of the defect in realistic code settings.

Although the goal of these projects is to find realistic defects in realistic code, their performance is not yet high enough; i.e. Table I and [38].

Our work, in contrast, is more modest: we attempt to find the minimal code complexity necessary to break the state of the art AI system [10]. In doing so, we find that a memory network can learn how to identify buffer overflows in synthetic code, but the it appears as though the memory network needs a nearly exhaustive amount of training data in order to do so.

VI. Conclusions and Future Work

In this study, we investigate the limits of the current state of the art AI system for detecting buffer overflows and compare it with current static analysis engines. To do so, we developed a code generator, *s-bAbI*, capable of producing an arbitrarily number of samples of controlled complexity. We found that the static analysis engines we chose have good precision, but poor recall. We found that the state of the art AI system, a memory network modeled after Choi et al. [10], can achieve similar performance to the static analysis engines, but requires and exhaustive amount of training data in order to do so.

VII. Acknowledgements

 This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute. DM18-0973

References

[1] T. Boland and P. E. Black, "Juliet 1.1 C/C++ and Java Test Suite," *Computer*, no. 10, pp. 88–90, Oct. 2012.

[2] P. E. Black, "A Software Assurance Reference Dataset: Thousands of Programs With Known Bugs," *Journal of research of the National Institute of Standards and Technology*, vol. 123, Apr. 2018.

[3] ——, "Juliet 1.3 test suite: changes from 1.2," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep., Jun. 2018.

[4] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger, "Comparing Bug Finding Tools with Reviews and Tests," in *Testing of Communicating Systems*. Springer Berlin Heidelberg, 2005, pp. 40–55.

[5] P. Emanuelsson and U. Nilsson, "A Comparative Study of Industrial Static Analysis Tools," *Electronic notes in theoretical computer science*, vol. 217, pp. 5–21, Jul. 2008.

[6] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why Don'T Software Developers Use Static Analysis Tools to Find Bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 672–681.

[7] K. Goseva-Popstojanova and A. Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," *Information and Software Technology*, vol. 68, pp. 18–33, Dec. 2015.

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

[9] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning Distributed Representations of Code," Mar. 2018.

[10] M. Choi, S. Jeong, H. Oh, and J. Choo, "End-to-end prediction of buffer overruns from raw source code via neural memory networks," *CoRR*, vol. abs/1703.02458, 2017. [Online]. Available: http://arxiv.org/abs/1703.02458

[11] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security & Privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[12] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1977, pp. 238–252.

[13] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended symbolic execution on binary programs," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 225–236.

[14] M. Zitser, R. Lippmann, and T. Leek, "Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 6, pp. 97–106, Oct. 2004.

[15] F. Logozzo and M. Fähndrich, "Pentagons: a weakly relational abstract domain for the efficient validation of array accesses," in *Proceedings of the 2008 ACM symposium on Applied computing*. ACM, 2008, pp. 184–188.

[16] H. Nazaré, I. Maffra, W. Santos, L. Barbosa, L. Gonnord, and F. M. Quintão Pereira, "Validation of memory accesses through symbolic analyses," in *ACM SIGPLAN Notices*, vol. 49, no. 10. ACM, 2014, pp. 791–809.

[17] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival, "Why does astrée scale up?" *Formal Methods in System Design*, vol. 35, no. 3, pp. 229–264, 2009.

[18] É. Payet and F. Spoto, "Checking array bounds by abstract interpretation and symbolic expressions," in *International Joint Conference on Automated Reasoning*. Springer, 2018, pp. 706–722.

[19] R. Baldoni, E. Coppa, D. C. Delia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, p. 50, 2018.

[20] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Formal Aspects of Computing*, vol. 27, no. 3, pp. 573–609, 2015.

[21] E. M. Voorhees and D. M. Tice, "Building a Question Answering Test Collection," in *Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '00. New York, NY, USA: ACM, 2000, pp. 200–207.

[22] "The Stanford Question Answering Dataset," https://rajpurkar.github.io/SQuAD-explorer/, accessed: 2018-8-21.

[23] A. W. Yu, D. Dohan, M.-T. Luong, R. Zhao, K. Chen, M. Norouzi, and Q. V. Le, "QANet: Combining Local Convolution with Global Self-Attention for Reading Comprehension," Apr. 2018.

[24] J. Weston, S. Chopra, and A. Bordes, "Memory networks," *CoRR*, vol. abs/1410.3916, 2014. [Online]. Available: http://arxiv.org/abs/1410.3916

[25] J. Weston, A. Bordes, S. Chopra, A. M. Rush, B. van Merriënboer, A. Joulin, and T. Mikolov, "Towards AI-Complete Question Answering: A Set of Prerequisite Toy Tasks," Feb. 2015.

[26] S. Sukhbaatar, A. Szlam, J. Weston, and R. Fergus, "Weakly supervised memory networks," *CoRR*, vol. abs/1503.08895, 2015. [Online]. Available: http://arxiv.org/abs/1503.08895

[27] P. E. Black, "SAMATE's contribution to information assurance," *NIST Special Publication*, vol. 500, no. 264, p. 2, 2006.

[28] "STONESOUP," https://www.iarpa.gov/index.php/research-programs/stonesoup, accessed: 2018-8-20.

[29] V. Okun, R. Gaucher, and P. E. Black, "Static analysis tool exposition (SATE) 2008," *NIST Special Publication*, vol. 500, p. 279, 2009.

[30] V. Okun, A. Delaitre, and P. E. Black, "The second static analysis tool exposition (SATE) 2009," *NIST Special Publication*, pp. 500–287, 2010.

[31] ——, "Report on the third static analysis tool exposition (SATE 2010)," *NIST Special Publication*, pp. 500–283, 2011.

[32] ——, "Report on the static analysis tool exposition (sate) iv," *NIST Special Publication*, vol. 500, p. 297, 2013.

[33] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-Scale Automated Vulnerability Addition," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 110–121.

[34] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating Static Analysis Defect Warnings on Production Software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '07. New York, NY, USA: ACM, 2007, pp. 1–8.

[35] S. Shiraishi, V. Mohan, and H. Marimuthu, "Test suites for benchmarks of static analysis tools," in *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Nov. 2015, pp. 12–15.

[36] K. Kratkiewicz and R. Lippmann, "Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools," *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[37] I. Pashchenko, S. Dashevskyi, and F. Massacci, "Delta-bench: Differential Benchmark for Static Analysis Security Testing Tools," in *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 163–168.

[38] D. Oliveira, E. Fong, T. S. C. Vadim Okun, D. Cupif, A. Delaitre, and C. D, "Improving Software Assurance through Static Analysis Tool Expositions," *Journal of Cyber Security and Information Systems*, vol. 5, no. 3, Nov. 2017.