Proceedings of WInnComm 2016, Copyright © 2016 Wireless Innovation Forum All Rights Reserved

BLACKSPOT: USING TENSOR DECOMPOSITIONS TO GUIDE INSPECTION OF SOURCE CODE

David Bruns-Smith (Reservoir Labs, New York, NY, USA; bruns-smith@reservoir.com)
James Ezick (Reservoir Labs, New York, NY, USA; ezick@reservoir.com)
Janice McMahon (Reservoir Labs, New York, NY, USA; mcmahon@reservoir.com)
Jonathan Springer (Reservoir Labs, New York, NY, USA; springer@reservoir.com)

ABSTRACT

In this paper we introduce Blackspot, an extension to R-Check SCA that uses unsupervised machine learning based on tensor decompositions to organize and highlight sections of source code for more systematic inspection. Using markers identified by R-Check SCA's Pitchfork rule language, multi-dimensional decompositions are used to cluster code so as to group similar structures for accelerated manual inspection and, when seeded with examples of known weaknesses, to prioritize code fragments for rigorous review based on similarity derived from latent features. We show how multi-dimensional analysis provides a precision advantage over matrix SVD-based approaches and enables both accelerated compliance testing and more directed discovery of potentially critical software weaknesses. Utilizing high-performance tensor decomposition techniques provided by Reservoir's ENSIGN Tensor Toolbox, Blackspot scales to millions of lines of code, making it practical for application to complex, large-scale cyber-physical systems. Using an open SCA radio waveform as a first example, we illustrate how Blackspot can be applied to guide inspection for SCA compliance testing and weakness discovery in the software radio domain.

1. INTRODUCTION

Software inspection naturally divides into two complementary classes: dynamic testing and static analysis. Dynamic testing involves the execution of the software being inspected, typically over a predetermined set of test cases or scripted list of use scenarios. While dynamic testing has the advantage of directly running the software being inspected, the scope and coverage of testing is limited to the execution paths exercised by the predefined set of test cases. In contrast, static analysis involves reasoning over the artifacts – usually source code – of the system being inspected. Because formal reasoning methods such as data-flow analysis and model checking are abstract, they are not limited to a finite set of execution traces and have the potential to make conclusive statements about all possible program paths. However, the analysis required for some properties can lead to reasoning challenges that are intractable over millions of lines of code. In other cases, the underlying logical problem can be undecidable [1]. These realities then lead to compromised forms of analysis that scale, but at the cost of decreased precision – either false positive or false negative results. In these cases – where a sufficient set of test cases cannot be constructed and the necessary abstract properties cannot be proven – there remains a need for human involvement in software testing.

In this paper, we introduce Blackspot as a new approach for using unsupervised machine learning based on tensor decompositions to help guide human code inspection. A tensor is a multi-dimensional array and a tensor decomposition is a generalization of the matrix Singular Value Decomposition (SVD). The SVD operation is the foundation of latent semantic analysis [2], a technique that has been applied to unsupervised topic discovery in text. SVD methods have also previously been applied to the discovery of known code weaknesses in modest-sized code bases [3][4].

In Blackspot, markers spanning multiple aspects of the source code are extracted using R-Check SCA's Pitchfork rule language. Tensor decompositions then guide source code inspection in two ways. First, components generated by the decomposition allow clustering of the code by structure. For tasks such as manual inspection required for compliance testing of certain SCA requirements [5][6], this allows sequencing of inspection tasks in a logical way leading to a reduced number of conceptual "context switches" between tasks. Ideally, this increased continuity should accelerate inspection and reduce the human tendency for error. Second, when combined with markers extracted from known forms of cyber weaknesses, decompositions provide guidance by allowing for the identification and ranking of "synonyms" for those weaknesses. When applied to weakness discovery, we show how the additional dimensions supported by tensor, rather than matrix, analysis provides a more precise result.

By utilizing lessons from machine learning rather than traditional formal methods from data-flow analysis and model checking, the goal is to sidestep undecidability limitations and the resulting necessity of false positives with an automated technique that provides a unique, somewhat

Proceedings of WInnComm 2016, Copyright © 2016 Wireless Innovation Forum All Rights Reserved

more human, perspective on source code. This perspective generates quantifiable multi-aspect equivalences that cluster code based on latent features that generally cannot be explicitly articulated as concise rules or patterns.

The remainder of this paper is organized as follows. Section 2 provides an overview of unsupervised machine learning based on tensor analysis as applied to source code. This includes the basics of tensors, tensor decompositions and clustering, tensor formation for code analysis, and the use of Pitchfork for extracting markers. Section 3 provides an application of Blackspot to organizing code for inspection, motivated by problems drawn from SCA compliance requirements. Section 4 illustrates how Blackspot extends prior work based on SVD methods and can be applied to detecting abstractions of source code weaknesses based on latent feature similarity. The paper concludes with a discussion of directions for future work in Section 5.

2. UNSUPERVISED LEARNING BASED ON TENSOR DECOMPISTIONS APPLIED TO SOURCE CODE

Blackspot uses the unsupervised discovery of latent features in source code to cluster similar code structures for systematic inspection. One previously-applied approach for automatically detecting latent patterns in source code has been to embed functions within a vector space. Specifically, there is one index (vector space dimension) for each symbol of interest, such as a variable type or a function call, within the code. Let *S* be the set of all symbols within the source code. Then, each function is embedded within an |*S*|-dimensional vector space, where in each dimension the function is assigned a value of 1 if the corresponding symbol appears within the function and 0 otherwise. Principal Component Analysis (PCA) can then be performed to find patterns within the usage of symbols across functions. This is accomplished by forming a matrix with one column per function vector, computing the Singular Value Decomposition (SVD) of the matrix, and then using cosine similarity to compare vectors for different functions projected into the latent feature space. This technique has been applied to open software projects to aid discovery of previously undetected weaknesses [3]. However, this approach is severely limited in that it can only analyze a single aspect of functions – in this case the occurrence of symbols.

Analyzing source code accurately depends not just on the occurrence of symbols, but the structures in which those symbols occur, and the context in which those symbols are being used. In a later refinement of the prior work [4], each function is written as a composition of Abstract Syntax Tree (AST) subtrees, combining structural information with symbolic information. While this improves the quality of the analysis, the manner of combining the two types of information is necessarily imprecise owing to the limitations of matrix analysis. To illustrate, consider the simple case where two functions have precisely the same structure, but do not share any of the same symbols. One would expect source code analysis to recognize this similarity. In the SVD-based approach, the symbols are different, so the AST subtrees would appear different, and the similarity would not be recognized. What is needed to accurately perform this analysis are tensors – arrays that allow symbol, structure, context, and potentially other orthogonal factors to be represented as wholly-separate modes.

Representing Source Code as Tensors

With Blackspot, we extend the basic approach from a two-dimensional array (matrix) formulation to a multi-dimensional array (tensor) formulation in order to compare diverse aspects of the source code simultaneously. This allows us to analyze information such as structure and symbol occurrence separately and distinctly – to detect, for example, cases where different symbols are used within similar structure or similar structure occurs with different symbols.

For the rest of this paper we will assume the language of tensor analysis. A tensor is a multi-dimensional array. In this parlance, a vector is a one-dimensional tensor and a matrix is a two-dimensional tensor. (The overloading of the word dimension is unfortunate. Each index within a vector in a vector space is called a dimension – we will refer to them as *indices*. When we use *dimension* it is to refer to a *mode* of a tensor.) Each mode of the tensor has an index set – for Blackspot this will include the set of functions, the set of symbols, the set of AST subtrees, and the set of symbol kinds. The size of each mode is equal to the number of indices. In a *d*-dimensional tensor, for every *d*-tuple of indices there is a corresponding entry in the tensor. In the special case of a two-dimensional tensor (a matrix), the index sets are the rows and columns and for every combination of row and column there is an entry within the matrix. A *d*-dimensional tensor is said to be *rank-1* if it can be written as the outer product of *d* vectors. That is, for tensor T and vectors $v_1, \ldots, v_d$:

$$T = v_1 \otimes v_2 \otimes \ldots \otimes v_d$$
$$T(i_1, i_2, \ldots, i_d) = v_1(i_1) * v_2(i_2) * \ldots * v_d(i_d)$$

In general, the *rank* of a tensor is the smallest number of rank-1 tensors whose sum equals the original tensor.

So to extend the previous approach, instead of building a matrix where every column is a function and every row is a symbol, we build a tensor. Functions and symbols are dimensions in the tensor just as in the matrix case. However, the tensor will also include structure and kind information as additional tensor modes.

Proceedings of WInnComm 2016, Copyright © 2016 Wireless Innovation Forum All Rights Reserved

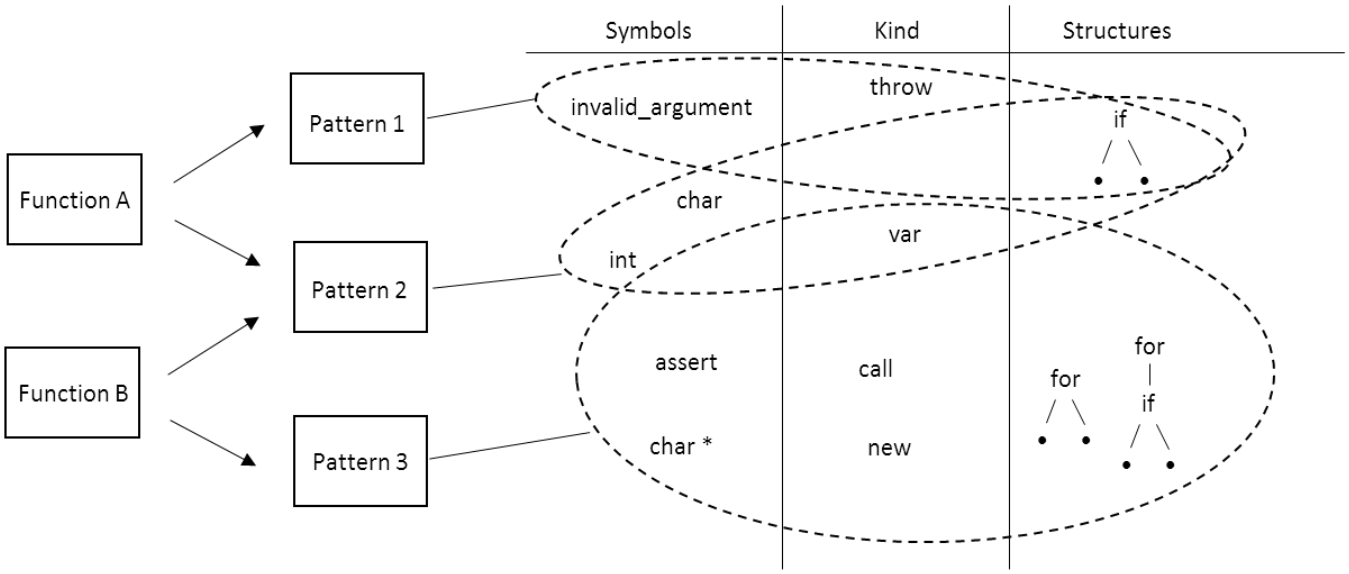


Figure 1. Tensor decompositions show how functions are broken down into patterns composed of symbols, structures, and kinds. This figure shows a hypothetical example of two functions composed of three patterns.

Additional Dimensions for Tensor Analysis

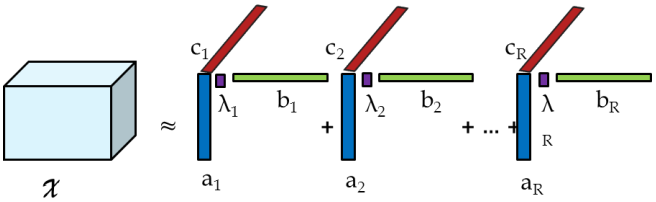
In Blackspot, the *structure* mode contains subtrees of the AST. In order to emphasize code that is structurally similar regardless of surface level differences, we introduce a layer of abstraction. Loops of various kinds are all grouped under the name *LOOP*. Likewise, conditionals are grouped together as *IF*. Furthermore, for the purposes of storing results as strings, we introduce a notation for writing these tree structures as follows: (1) if a symbol occurs at the top level of a function, record its structure as *, (2) if a symbol occurs within a loop, record its structure as *LOOP()*, (3) if a symbol occurs within a conditional, record its structure as *IF()*, and (4) if one structure is a child of another within the subtree, then place that structure within the parentheses of the other structure e.g., *LOOP(IF()IF())*. A symbol occurs inside a structure if that symbol occurs as a leaf anywhere within the subtree. For building the tensor used in this analysis, we look (recursively) at all subtrees of the AST with tree depth ≤ 2.

The symbol *kind* mode describes what kind of symbol appears. We have divided the context of symbols into five kinds: *VAR*, *CALL*, *THROW*, *NEW*, and *DEL*. The symbol `char *` can occur as a normal variable where it would be assigned the kind *VAR*. On the other hand, the symbol `char *` might be explicitly allocated with `malloc` or `new` and in this case it would be identified with the *NEW* kind. Likewise, *invalid_argument* (and other exceptions) would have the kind *THROW* and `getPort()` (and other functions occurring as symbols) would get the kind *CALL*.

In the resulting four-mode tensor, there is an entry for every *(function, symbol, structure, kind)* combination, just as there is an entry for every *(function, symbol)* combination in the two-dimensional matrix case. In the tensor, the entry is a 1 if that symbol of that particular symbol kind occurs as leaf of that AST subtree within that function, and 0 otherwise.

Tensor Decompositions

The two-dimensional approach uses the SVD of the matrix to extract latent patterns. With additional modes, the tensor is decomposed using a higher-order generalization of the SVD called a CANDECOMP/PARAFAC (CP) decomposition [7]. An SVD decomposes a matrix M into $M = UDV^T$ where U and V are called factor matrices and D is a diagonal matrix of size $R \times R$ that contains the R singular values. The CP decomposition takes a d-mode tensor T and decomposes it into a predetermined number of *components*. Each component is a rank-1 tensor and consists of a weight λ and vector of *eigenscores* for each mode in the tensor. The tensor is thus approximated by the weighted sum of the components.



Such a decomposition can only be exact if the number of components is greater than or equal to the rank, R, of the tensor T. In many cases the rank R of a decomposition is less than the rank of the tensor T, resulting in an approximate decomposition. However, this kind of low-rank approximation is often desirable as it clusters together similar

Proceedings of WInnComm 2016, Copyright © 2016 Wireless Innovation Forum All Rights Reserved

```
CORBA::Object_ptr
AudioCapture::getPort(const
CORBA::Char* name)
  throw (CORBA::SystemException,
     CF::PortSupplier::UnknownPort)
{
  std::string temp(name);

  if (temp != "AudioCaptureOut") {
    throw
CF::PortSupplier::UnknownPort();
  }

  return m_pcmout->getPort();
}
```

AudioCapture::getPort

Pattern	Uniqueness Score
39	0.2222
57	0.0502
93	0.0172

Pattern 39

Structure	Score
IF()	1.0

Symbols	Score
CF::PortSupplier::UnknownPort	0.5
operator!=	0.5

Kind	Score
CALL	1.0

Pattern 57

Structure	Score
IF()	1.0

Symbols	Score
tk_error	1.0

Kind	Score
THROW	1.0

Pattern 93

Structure	Score
*	1.0

Symbols	Score
tk_class	0.98
tk_typeref_tk_class	0.02

Kind	Score
VAR	1.0

Figure 2. These tables show the results of the tensor decomposition for the AudioCapture::getPort() function. The table below the code shows the breakdown of the function into patterns. The Uniqueness Score is a value between 0 and 1 indicating how unique the pattern is to this particular function. The tables to the right show three of these patterns and their composition in terms of structure, symbol, and symbol kind. The score indicates what fraction of the total symbols, structures, or kinds within the pattern that particular index represents. Note that "tk_error" and "tk_class" are Pitchfork type symbols that refer to the type of "UnknownPort" and "std::string," respectively.

patterns of behavior from the original data. From this view, each component represents a latent pattern or concept found within the source input, and in this sense tensor decompositions are an unsupervised learning technique.

As one might expect, tensor decompositions are significantly more computationally intensive than traditional matrix decomposition operations. Tensors are large and operating on them requires leveraging the considerable sparsity common to most practical applications. This bottleneck has been a limiting factor to the more widespread application of tensor methods. In Blackspot, we apply ENSIGN [8], a high-performance toolbox specifically engineered to scale to large, sparse tensor problems.

Each one of the components produced by the decomposition is a latent pattern discovered within the original data. The component weight reflects the number of tensor entries captured by the pattern. As discussed above, a component is made up of one vector from each mode where there is an entry (eigenscore) in the vector for each tensor mode index. So in this case, there is one vector with an entry for each function, another vector with an entry for each symbol, and so on. An eigenscore indicates whether or not a particular mode index contributes to the pattern. If in one of the vectors an entry has a value of 1.0, that means it is the only index that contributes to the pattern captured by the component. If an entry has a score of 0.0 then it is unrelated to the pattern. Likewise, if two entries have a score of 0.5 in a single vector, the pattern is comprised equally of those two indices.

Typically, one analyzes each component as a different pattern of activity between the *d* modes of the tensor. In Blackspot, we apply an "asymmetric" function-centric analysis. Each component is interpreted as a pattern composed of symbols, kinds and syntactic structures (AST subtrees). As Figure 1 illustrates, each function, in turn, is then composed from a combination of these patterns. Applying a tensor decomposition thus produces a two-fold result: (1) a breakdown of each function into constituent patterns, and (2) a breakdown of patterns into constituent structures, symbols, and kinds.

Proceedings of WInnComm 2016, Copyright © 2016 Wireless Innovation Forum All Rights Reserved

Pattern 10. Uniqueness = 0.05

Structure	Score
IF(IF())	0.79
IF()	0.21

Symbols	Score
tk_error	1.0

Kind	Score
THROW	0.9
NEW	0.1

Pattern 11. Uniqueness = 0.11

Structure	Score
IF()	0.88
IF(IF())	0.12

Symbols	Score
char *	1.0

Kind	Score
THROW	1.0

Pattern 37. Uniqueness = 1.0

Structure	Score
IF(LOOP()LOOP()IF())	1.0

Symbols	Score
tk_class	0.4
char	0.2
tk_class *	0.2
int	0.2

Kind	Score
VAR	0.8
THROW	0.2

Figure 3. These three tables show three different patterns from the FM3TR code base decomposition, all of which include the THROW symbol kind.

FM3TR Example

As a first illustrative example, we applied Blackspot to the familiar FM3TR waveform – approximately 100,000 lines of C++. The tensor built from the FM3TR code base has four modes – as discussed above – with the following sizes: 353 functions, 331 symbols, 5 symbol kinds and 24 structures. There are a total of 1,515 non-zero tensor entries, so the input tensor is very sparse (99.9891% sparsity). The tensor was decomposed into 100 components using the Alternating Poisson Regression (APR) algorithm for calculating a CP decomposition [9]. On a single-core tablet PC, the decomposition took 1.1 seconds – this tensor is extremely small by the standards of typical ENSIGN use cases.

Figure 2 shows an example of a single function from the FM3TR source code and how that function is broken down into patterns (components) by the tensor decomposition. In the figure, we refer to the eigenscore associated with the function in each component as its *uniqueness score*. A uniqueness score of 1.0 means this pattern only occurs within this function. A score of 0.1 would mean $1/10^{th}$ of the occurrences of this pattern are in this function. In Figure 3, the function is represented in five patterns – three of which are expanded. Within the patterns, the score for each structure, symbol, and kind represents what fraction that structure, symbol, or kind contributes to this pattern. For example, a score of 1.0 for structure = *IF()* means that this is the only structure included in this pattern. Likewise a score of 1.0 for kind = *VAR* means all the symbols within this pattern are variables. A score of 0.5 for `char` and 0.5 for `int` means the symbols in this pattern are split evenly between `char` and `int` types. In this way, the decomposition provides a concise, abstract summary of the composition of the code that can be queried.

Using Pitchfork to Extract Markers

Blackspot uses Pitchfork [10] to extract the markers from source code that are used to form the tensors that are then decomposed for analysis. Pitchfork is a part of R-Check SCA, Reservoir's platform for static SCA compliance testing [11], and enables user-configurable matching of source code constructs through a scriptable pattern language. To make it convenient to match distinct, but structurally similar programs, Pitchfork augments the base C/C++ syntax with a metalanguage, providing regular expression syntax, pattern-match variables, wildcards, and other helpful features. In native operation, users write rules similar in syntax to C/C++ code fragments, which the R-Check SCA analysis engine then accepts and identifies in the code. Actions associated with each scripted rule dictate how a matched (or absence of matched) pattern is reflected in a report back to the user.

For Blackspot, a fixed set of Pitchfork rules are used to extract features of interest from the program (populating the modes of the tensor), while eliding all of the uninteresting detail of a program. For example, the specific name of a local variable in a program is not relevant to code similarity, but the patterns and interplay of control structure and system calls are crucial. In essence, Pitchfork is capturing whole fragments of the AST from the source code. For unsupervised learning, the choice of which observable markers to record has a significant impact on the ability of algorithm to identify meaningful latent features. The ability of Pitchfork, combined with the depth of analysis provided by R-Check SCA, to extract a rich and flexible set of observable code constructs is key to the success of the approach.

Proceedings of WInnComm 2016, Copyright © 2016 Wireless Innovation Forum All Rights Reserved

AudioPlayback::Init

Function	Cosine Similarity
AudioCapture::Init	0.968
galois::operator-	0.968
CF__PropertySet_impl::index	0.956
CVSDEncoder::ConvertToCVSD	0.956
galois::GaloisField::gen_inverse	0.956
galois::operator>>	0.922
RsBlockDecoder::Init	0.589
FileInput_MAC_LLC::Init	0.576
CVSDEncoder::CVSDEncoder	0.557
RfChannelEmulator::Init	0.469

Figure 4. This table shows a list of the functions with the top 10 cosine similarities to AudioPlayback::Init.

3. ORGANIZING CODE FOR INSPECTION

By themselves, the patterns captured as the components of a tensor decomposition can aid code inspection by providing an organized profile of which sections of the source code include which features. This capability can be applied to accelerate manual inspection for such tasks as SCA compliance testing. For example, the SCA includes several requirements that dictate the conditions under which a specific type of exception must be thrown. These requirements naturally fall into the gap described in Section 1 – dynamic testing cannot exercise all of the possible exception conditions and static analysis cannot prove that the execution of the exception is limited to the proper conditions.

From the decomposition, one can query the list of patterns that involve the *THROW* kind and see the structures in which exceptions occur along with the symbols of particular interest. Figure 3 shows three different patterns from the same FM3TR code base involving the *THROW* kind. The first, Pattern 10, appears mostly in nested conditionals and, to some extent, in single-level conditionals. The symbols are all of the `tk_error` type and the pattern represents a correlation between allocation (*NEW*) and exceptions. The pattern has a low uniqueness score meaning it occurs in many functions throughout the code base. The fact that allocation and exceptions over the same type are highly correlated in Pattern 10 is a point of interest for systematic inspection of those functions. The person inspecting the code might also check that no function is missing from this list associated with the `tk_error` type.

Pattern 11 is similar, except that it is restricted to string (`char *`) exceptions and these exceptions are not tightly correlated with allocations. Pattern 11 also occurs more heavily in single-level conditional statements rather than in nested conditionals. Now, if an inspector has to check a condition for all string exceptions, then searching for all functions containing Pattern 11 will provide an organized list of code to inspect.

Cluster 1

Function
CharInPort_impl::~CharInPort_impl
CharOutPort_impl::~CharOutPort_impl
ShortInPort_impl::~ShortInPort_impl
ShortOutPort_impl::~ShortOutPort_impl
AudioCapture::~AudioCapture
AudioPlayback::~AudioPlayback
CVSDEncoder::~CVSDEncoder
CVSDDecoder::~CVSDDecoder
MacReceive::~MacReceiver
MacXmit::~MacXmit

…

Cluster 25

Function
BaseAssemblyController:getPort
Packetizer5to7::getPort
Packetizer7to5::getPort
CVSDEncoder::getPort
CVSDDecoder::getPort
FileInput_MAC_LLC::getPort
FileOutput_MAC_LLC::getPort
FM3TR_WaveformPacketizer::getPort
FM3TR_WaveformReceiver::getPort
MacReceive::getPort
MacXmit::getPort

…

Figure 5. These tables show two clusters that result from applying a simple clustering algorithm that uses cosine similarity as the distance metric with a distance cutoff of 0.25. The functions in the first cluster are all deconstructors, whereas the functions in the second cluster are all "getPort" functions.

Pattern 37 is interesting because it is very specific. It has a uniqueness score of 1.0 meaning it only occurs in a single function, found to be `CF_PropertySet_impl:query`. Its uniqueness is high because it occurs within a very specific structure, with correlated use of both variables and exceptions for `int`, `char`, `tk_class`, and pointers to `tk_class`. This is a pattern including exceptions not seen anywhere else in the code base and so it should be checked separately.

MskDemodulator::SymbolSynch

Pattern	Uniqueness Score
6	0.584
87	0.487
84	0.286
74	0.222
67	0.021
16	0.016

MskDemodulator::SymbolSynch

Function	Cosine Similarity	Shared Patterns
MskDemodulator::Run	0.482	6, 67
FileOutput_MAC_LLC::RewritePacket	0.321	84
ShortInPort_impl::ShortInPort_impl	0.263	74
MskModulator::Modulate	0.263	87, 67, 16
CVSDDecoder::CVSDDecoder	0.217	74
RsBlockDecoder::RsBlockDecoder	0.216	74, 67
FileInput_MAC_LLC::Init	0.129	74
FileOutput_MAC_LLC::Run	0.121	84
MskDemodulator::Init	0.116	74, 67

Figure 6. This figure describes the SymbolSynch function in which Cppcheck discovered a memory weakness. The table on the right-hand side shows the patterns that make up SymbolSynch whereas the left-hand table shows those functions with the top cosine similarities to SymbolSynch. Additionally, for each function, the table shows which patterns that function shares with SymbolSynch.By focusing on functions with shared patterns, less inspection in necessary to identify similar weaknesses.

From any set of patterns, one can then query the list of the specific functions that contain these patterns to produce a list of where exceptions of specific types occur in the code, sorted by pattern. In this way, code for inspection can be organized so as to minimize conceptual context switches – an approach that is more systematic than simple file or symbol sequence alone. This approach also makes it straightforward to isolate similar code for deeper, focused inspection when a violation is found.

4. DETECTING ANOMALIES AND CODE WEAKNESSES

While using the breakdown of functions into discrete patterns is useful for organized code inspection, we can also use the inclusion or exclusion of a function across patterns as a tool for function comparison. Given a decomposition of a tensor into *R* components, we can form a vector of length *R* for each function comprised of the eigenscore for that function in each component. This vector acts as a sort of signature for the function – its occurrence or not in each latent pattern found by the decomposition. From these vectors, we can compare each pair of functions by calculating a distance metric such as cosine similarity between the vectors. Then, for any given function, we can rank the other functions by their cosine similarity to get a list of the most similar functions. As an example, Figure 4 provides a ranked list of similar functions for the `AudioPlayback::Init` function.

So, for any function of interest, we can find all of those functions with similar structures, symbols and kinds, substantially improving ease of inspection. By applying a simple threshold-based clustering algorithm using the cosine similarity as the distance metric, we can separate the functions into distinct similarity clusters. Some of these clusters of functions are unsurprising. For example, Figure 5 shows a cluster of deconstructors and a cluster of `getPort` functions from various classes.

While unsurprising, even these clusters turn out to be useful because functions that aren't clustered as expected may have anomalous properties to be inspected. For example there are several deconstructors that are not part of Cluster 1:

- `~Packetizer5to7,`
- `~Packetizer7to5,`
- `~FM3TR_WaveformPacketizer`
- `~FM3TR_WaveformReceiver`

A user can use this information to check why these deconstructors behave differently to make sure there are not, for example, any structural errors or memory leaks. Likewise, with the `getPort` functions, they are not all contained in Cluster 25:

- `AudioCapture::getPort`
- `AudioPlayback::getPort`
- `CF_PortSupplier_impl::getPort`

In particular, `CF PortSupplier impl::getPort` is clustered together with functions completely unrelated to ports.

One particularly powerful use of cosine similarity across the function-pattern vectors is weakness extrapolation. If one

Proceedings of WInnComm 2016, Copyright © 2016 Wireless Innovation Forum All Rights Reserved

MskDemodulator::SymbolSynch

Function	Cosine Similarity (SVD)
MskModulator::Modulate	0.286
ShortInPort_impl::ShortInPort_impl	0.279
MskDemodulator::Demodulate	0.247
CVSDDecoder::CVSDDecoder	0.227
MskDemodulator::Run	0.218
RsBlockDecoder::RsBlockDecoder	0.190
MskModulator::MskModulator	0.183
RfChannelEmulator::RfChannelEmulator	0.180
MskModulator::~MskModulator	0.178
MskDemodulator::~MskDemodulator	0.164

…

Figure 7. This figure shows the cosine similarities with SymbolSynch as reported by the 2D SVD approach. Note that the vulnerable function the tensor method discovered, FileInput_MAC_LLC::Init, does not appear in this list.

can identify one weakness in a code base, it is possible to look at the top functions in cosine similarity to the function containing the weakness in order to try to automatically detect similar weaknesses. In this section, we show how our tensor methods successfully detected a weakness that neither the open-source Cppcheck [12] static analysis tool nor the prior SVD-based approach could identify.

The experiment requires a known weakness from which to extrapolate, so the starting point was to run a static analysis tool, Cppcheck, over the FM3TR code base. This analysis only reported one serious weakness: a memory leak within the `MskDemodulator::SymbolSynch` function. Inspection of other functions with high cosine similarity discovered a new instance of the same weakness. Figure 6 lists the functions with the highest cosine similarity and indicates shared patterns.

From this list, one could simply inspect each function in turn. However, it's possible to further pare down which functions should be inspected by considering what patterns they share with `SymbolSynch`. Since the weakness is a memory leak, the relevant functions must contain symbols of the *NEW* kind. Pattern 74 is the only one of these patterns that includes *NEW* kind symbols, so functions without Pattern 74 either do not use allocation or they use allocation in a different way. Thus, even though `MskDemodulator::Run` has the highest cosine similarity, it doesn't include Pattern 74 so it doesn't share this particular weakness. In this case, the high similarity is due to the specific symbols unique to these functions. For example, they both call the functions `CMPLX_INT16::real` and `CMPLX_INT16::imag` (contained in Pattern 6), which is not called by many other functions.

Furthermore, among those similar functions that do include Pattern 74, many of them are constructors. In the case of the memory leak within `SymbolSynch`, there exists a *NEW* without any corresponding *DEL*. That same pattern is appearing within these constructors, but in this case we know it is not a memory leak because a constructor should not free the memory it's allocating. So, without even looking at the code, there are only two likely candidates remaining for the same weakness. The first, `FileInput_MAC_LLC::Init`, was the very first section of source code we inspected in applying this process, and was found to have an almost identical memory leak to `MskDemodulator::SymbolSynch`.

This was a weakness that Cppcheck 1.59 was unable to detect. In addition, we tried to repeat this experiment using the SVD approach and also could not detect this second weakness. Just as described in Section 2, the function vectors were composed into a matrix, the resulting matrix was decomposed with an SVD, and then cosine similarities were calculated between the function-pattern vectors in the factor matrix. Figure 7 shows the high cosine similarities with `SymbolSynch` using the SVD approach.

The function `FileInput_MAC_LLC::Init` does not appear anywhere near the top in the list. Our hypothesis for why the matrix approach is incapable of producing the same results is as follows: ultimately, trying to extrapolate between different functions is a multi-dimensional problem. It is not sufficient to look at just symbols, just structures, or even a one-dimensional (vector concatenation) combination of structures and symbols. For detecting weaknesses, it is important to maintain distinctions between structure, symbol, and even the context of the symbol. Further note that in the SVD case, without a symbol kind dimension, it would be necessary to look through every function in the list because there is no way to identify only those sharing a similar use of allocator symbols.

Proceedings of WInnComm 2016, Copyright © 2016 Wireless Innovation Forum All Rights Reserved

In this example, we use one weakness already identified in the source code to identify a similar weakness. However, the technique can also be extended to detecting instances of known weaknesses. To do this, the source code base can be merged with examples of known weaknesses, such as examples taken from the Common Weakness Enumeration [13] database or past project-specific examples from a bug database. By seeding the code with known weaknesses, similarities can be discovered using the same process described in this section. The power of this approach is that it is based on latent factors rather than a specific rule instance. While not providing a precise result, it detects synonyms in code in a way that would be very difficult for a strictly rule-driven engine to replicate.

5. DIRECTIONS FOR FUTURE WORK

Blackspot represents a unique capability in the realm of static analysis – one that is intended to augment rather than replace traditional tools based on dataflow analysis or model checking. In that spirit, this paper provides a framework for using tensor decompositions to analyze source code, but the specific methodology described in the preceding sections is extremely flexible. There is no limitation to using a tensor formed from *function x structure x symbol x kind* as the modes. We can add any details that are found to be relevant to detecting source code weaknesses and that can be collected from the code base. This includes both directly observable features and derived values such as cyclomatic complexity [14] for individual functions or empirical "moments" (statistical observations) over feature counts. Future work will include building more extensive libraries of Pitchfork rules and applying other tools and techniques for collecting and assembling this additional information, making the analysis even more powerful.

The ENSIGN tensor toolbox used to perform the decompositions uses specialized sparse data structures and exposed parallelism to scale to tens of millions of non-zero entries. This is more than sufficient to support deep analysis of multi-million line code bases and extend the applicability of Blackspot through a broad range of software-radio and cyber-physical systems.

In order to transition Blackspot to use in a development or testing environment, more automation of the decomposition post-processing is required. For example, Blackspot analysis can be specifically tied to certain SCA requirements that still depend on manual inspection of code. A test engineer, selecting a requirement for testing, could get a guided, "mechanical turk" style tour through an enumerated list of code inspection tasks. The tasks would be ordered so as to minimize mental context switches and each task would be accompanied by relevant instructions based on the requirement and what is known about the function. We believe that such a directed workflow would significantly accelerate testing and reduce the tendency for human error. These claims, however, need to be validated in future work.

The same guided inspection process could also be used to organize waveform porting tasks. In this way, Blackspot would be applicable to the Wireless Innovation Forum's Most Wanted Innovations #1: *Techniques for Efficient Porting of Waveform Applications Between Embedded Heterogeneous Platforms* [15].

To be a robust tool for weakness discovery, Blackspot needs access to a complete library of example code weaknesses distilled into tensor entries. In addition, while this paper has illustrated the power of unsupervised learning, a practical tool for weakness detection should incorporate supervised (classification) techniques to make concrete assertions about the likelihood of a weakness occurrence. Tensor decompositions can be used to train models such as neural nets that can perform classification tasks [16]. We intend to explore these extensions in future work.

6. REFERENCES

[1] J. Ezick and J. Springer, "The Benefits of Static Compliance Testing for SCA Next," Proceedings of the SDR'11 WInnComm Technical Conference, November 2011.

[2] S. Deerwester, S. T. Dumais and R. Harshman, "Indexing by Latent Semantic Analysis," Journal of the American Society for Information Science, Vol. 41, No. 6, 1990.

[3] F. Yamaguchi, F. Lindner and K. Rieck, "Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities using Machine Learning," WOOT'11, August 2011.

[4] F. Yamaguchi, M. Lottmann and K. Rieck, "Generalized Vulnerability Extrapolation using Abstract Syntax Trees," ACSAC'12, December 2012.

[5] JPEO JTRS Test and Evaluation Laboratory (JTEL) SCA 2.2.2 Application Requirements List v2.2 Release Notes, July 8, 2010.

[6] JPEO JTRS Test and Evaluation Laboratory (JTEL) SCA 2.2.2 OE Requirements List v2.2 Release Notes, November 4, 2010.

[7] T. Kolda and B. Bader, "Tensor Decompositions and Applications," SIAM Review, Vol 51, No. 3, 2009.

[8] M. Baskaran, B. Meister, N. Vasilache and R. Lethin, "Efficient and Scalable Computations with Sparse Tensors," HPEC'12, September 2012.

[9] E. C. Chi and T. G. Kolda, "On Tensors, Sparsity, and Nonnegative Factorizations," SIAM Journal on Matrix Analysis and Applications Vol. 33, No. 4, 2012.

[10] J. Springer, S. Bernier, J. Ezick and J.P. Zamora Zapata, "Accelerating SCA Compliance Testing with Advanced Development Tools," WInnComm'15, March 2015.

[11] https://www.reservoir.com/rchecksca.

[12] http://cppcheck.sourceforge.net.

[13] https://cwe.mitre.org.

[14] T. McCabe, Sr., "A Complexity Measure," IEEE Transactions on Software Engineering, December 1976.

[15] http://www.wirelessinnovation.org.

[16] M. Janzamin,, H. Sedghi, and A. Anandkumar, "Beating the Perils of Non-convexity: Guaranteed Training of Neural Networks using Tensor Methods," CoRR abs/1506.08473, August 2015.