Packet Processing Algorithm Identification using Program Embeddings

S. VenkataKeerthy
IIT Hyderabad
India
cs17m20p100001@iith.ac.in

Yashas Andaluri
IIT Hyderabad
India
cs17b21m000001@iith.ac.in

Sayan Dey
IIT Hyderabad
India
cs22mtech02005@iith.ac.in

Rinku Shah
IIIT Delhi
India
rinku@iiitd.ac.in

Praveen Tammana
IIT Hyderabad
India
praveent@cse.iith.ac.in

Ramakrishna Upadrasta
IIT Hyderabad
India
ramakrishna@cse.iith.ac.in

ABSTRACT

To keep up with the network speeds, many recent works propose to offload network functions to SmartNICs. The process involves identifying packet-processing algorithms in a network function program then offloading them to appropriate accelerators available on SmartNICs. This process is often done manually for each architecture and is error-prone and laborious. In this work, we propose an automated solution to identify algorithms in network function programs. We model our approach as a classification problem of Machine Learning (ML) and propose using sophisticated program embeddings for representing the network function programs. We also identify the limited availability of datasets and propose a way of extrapolating them by systematically generating equivalent programs using (existing) compiler transformations in popular compiler infrastructures. Our approach relies on modeling programs as embeddings, uses ML models trained on such extrapolated datasets, and shows superior results over the recent works.

CCS CONCEPTS

• **Hardware** → **Emerging languages and compilers**; • **Networks** → **Programmable networks**; *In-network processing*; • **Computing methodologies** → *Machine learning*.

KEYWORDS

Network Function program identification, SmartNICs, Machine Learning, Program Embeddings

ACM Reference Format:
S. VenkataKeerthy, Yashas Andaluri, Sayan Dey, Rinku Shah, Praveen Tammana, and Ramakrishna Upadrasta. 2022. Packet Processing Algorithm Identification using Program Embeddings. In *6th Asia-Pacific Workshop on Networking (APNet 2022), July 1–2, 2022, Fuzhou, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3542637.3542649

APNet 2022, July 1–2, 2022, Fuzhou, China

ACM ISBN 978-1-4503-9748-3/22/07…$15.00
https://doi.org/10.1145/3542637.3542649

1 INTRODUCTION AND MOTIVATION

With the consistent growth in the data-rates in high-speed networks (e.g., data centers), there is an increasing gap between CPU packet-processing speeds and network speeds. As a consequence, more CPU resources are allocated for packet-processing, which can otherwise be allocated to tenant VMs and generate revenue. To handle this, many recent works [9, 13, 20, 31, 36, 41] proposed offloading Network Functions (NF) to SmartNICs that process packets at network speeds.

SmartNICs are recently emerging Domain Specific Architecture (DSAs) with many programmable SoC cores, memory systems, high-speed packet I/O, and a wide range of specialized accelerators. Other examples of such recent DSAs include NVIDIA Bluefield DPUs [28], Netronome SmartNICs [27], Cavium LiquidIO [6], etc. To improve their programmability, many SmartNIC vendors provide toolchains (like SDKs). This eases the burden of the application developers: who can either write new or port the existing NF programs written for general-purpose processors to SmartNICs.

However, while porting applications to SmartNICs, to understand and improve performance, the developers need to perform manual analysis and multiple rounds of hand-tuning.

There have been recent research efforts [22, 34] that attempt-to-port/aid-in-porting the network functions and distributed applications written for general-purpose processors (*e.g.*, x86) to SmartNICs so as to improve performance. Clara [34] showed up to 13.8x performance improvements by applying different porting strategies on a trial-and-error basis. iPipe [22] offloaded compute-intensive IPsec gateway network function that requires AES-256-CTR encryption and SHA-1 authentication. They observed near-linear performance of 8.6 Gbps and 22.9 Gbps for 1 KB packet size on the 10 and 25 GbE Cavium LiquidIOII [24] SmartNIC cards, respectively.

A significant portion of these performance gains can be achieved by offloading packet-processing algorithms to SmartNIC accelerators that often use hardwired logic for efficient implementations.

Tab. 1 shows examples of a few SmartNICs along with the supported on-chip accelerators that are widely used by network functions and are highly compute-intensive. Hash accelerators include operations such as CRC32 and CSUM16; crypto accelerators include hardware implementations of public-key, private-key, and cryptographic hash algorithms; and match-algorithms include regular expression match, wild-card match, and longest prefix match.

Table 1: Accelerator support for SmartNICs and use cases.

	Hash	AES, HMAC, SHA, etc.	RSA, ECC, DH, DSA, etc.	IPsec, TLS enc, MACsec, etc.	LPM	Ternary	regex	DPI	QoS	AI/ML	Decompression
SmartNIC Vs. Accelerators											
Netronome Agilio CX [27]	✓	✓	✓	✓	✓	✓		✓	✓		
Nvidia Bluefield 3 DPU [28]		✓	✓	✓			✓	✓	✓	✓	✓
Cavium LiquidIO [6]				✓				✓	✓		
Marvell OCTEON 10 [24]		✓		✓						✓	
Pensando DSC-100 [33]	✓	✓		✓	✓						✓
Network functions Vs. Accelerators											
L3 router	✓				✓						
Stateful firewall	✓					✓	✓		✓		
VPN gateway	✓	✓	✓	✓							
Intrusion Detection System (IDS)							✓	✓		✓	
5G network functions (e.g., UPF)	✓			✓	✓	✓	✓	✓	✓		
Distributed data stores	✓	✓								✓	✓

However, for offloading a packet-processing algorithm to an accelerator, the application developer should identify the correct region of NF program written for general-purposed cores and replace it with the appropriate calls to the accelerator. In Tab. 1, we also show the accelerators that can be leveraged by network functions for performance gains. However, the process of understanding and identifying the code is a highly laborious and tedious task, given the fact that the same code can be expressed in a multitude of ways and parameters, more so when it could have been optimized for performance on that particular architecture.

Our goal is to provide an automated workflow that simplifies cross-platform porting process for developers. Towards this goal, one question we investigate in this work is: *"Can we automatically identify regions in network functions that can be potentially replaced with calls to appropriate accelerators?"*

As a first step towards automating this process, the regions of code that perform a specific function should be identified and labeled. Such labeling could serve as suggestion to the application developer who can then replace it.

We propose to look at this problem as a ML classification approach.

There are mainly two challenges in realizing the above:

(1) How can we represent packet processing algorithms and programs as input to the machine learning model?
(2) How can we create a realistic packet processing program dataset that begins from practical applications and has wide applicability on various varieties of DSAs?

For (1), one approach is to treat programs as a token of natural languages and apply natural language processing techniques to obtain the representations [1]. Another approach [16, 23, 34] is to collect features (e.g., compute instructions, memory instructions) using domain expertise for representing programs. However, these approaches use either syntactic or semantic information of the program, but not both. Also, we tend to miss out on program-specific information while treating programs as natural languages.

To address these issues, different approaches for the automatic representation of programs, called program embeddings have been proposed [4, 37, 39]. Such representations are learned using machine learning approaches to represent programs as continuous, distributed vectors in an n-dimensional space. As opposed to the feature vectors that involve engineering features by domain experts, these distributed vector representations automatically learn to represent the relevant features of the underlying input. Such a learned distributed vector (or embedding) is a real-valued vector whose individual dimensions are not associated with a *meaning*; rather values of all the dimensions put together describe the input. In our work, we propose to use such program embeddings (more details in Sec. 2) to represent NF algorithms followed by training classification models using the program embeddings.

Challenge (2) is related to the availability of datasets for training models. The various machine learning methods are often data-hungry, i.e., they use thousands of data points, if not more, for training. There are no datasets for our problem that are readily available. And obtaining such an amount of program snippets of a particular algorithm in a network function itself is hard, and its availability is also not guaranteed.

In this work, we first manually extract code for multiple NF algorithms from the standard libraries (*e.g.*, OpenSSL [30], CryptoPP [8], etc.) and propose a way of extrapolating these functions to create the dataset.

In summary, our key contributions are the following:

- We use IR2Vec, a sophisticated program-level embedding algorithm/infrastructure, to represent programs from network domain for algorithm identification.
- Modeling the algorithm identification problem using a simple and scalable machine learning approach.
- Realistic dataset collection and systematic generation of semantically equivalent programs from the collected dataset using compiler transformations.

2 BACKGROUND

2.1 LLVM IR

LLVM IR is the Intermediate Representation (IR) of the LLVM compiler toolchain. LLVM IR expresses program statements as a simple set of atomic instructions similar to the assembly language but at a higher level of abstraction. Such a representation is Turing complete and is independent of the source language and the target architecture [21]. The instructions are typed and follow Static Single Assignment (SSA) [12] representations. This makes LLVM

IR more amenable for program analysis and optimizations. An example IR is shown in Fig. 1. Language independence nature of IR and ease of extracting control flow structures and other program analysis information like use-def, live variables [18], etc., makes it a suitable choice of processing programs for our problem.

2.2 IR2Vec

We use IR2Vec embeddings [39] to represent the NF programs for our problem. IR2Vec embeddings are continuous, distributed vector representations in n-dimensions, learned to represent a given LLVM IR. Such program embeddings are application-independent similar to the commonly used natural language embeddings like word2vec [25] and gloVe [32]. IR2Vec learns the syntactic correlations between different instructions in IR from a program corpus and uses a representation learning technique to generate a vocabulary. This vocabulary contains embeddings for each opcode, type, and operands that are present in the IR. These learned embeddings mainly capture the syntactic information of a given program and can now be used to represent programs in any language that LLVM supports. Starting from these learned representations, IR2Vec encodes semantic information using various program analysis information like use-def, liveness, and reaching definitions to form the final representation at the function level.

IR2Vec provides a generic approach to generate embeddings at various levels of the program. Specifically, the framework provides a systematic approach for representing instructions as instruction embeddings, which can be combined using different approaches (like linear combination, machine learning models, etc.) to represent functions and programs. And these embeddings are shown to exhibit better scalability - that can work with diverse machine learning models, does not encounter *Out Of Vocabulary* (OOV) words. Such representations are application-independent and encode both syntax and semantic information of the programs. It has been shown to perform better in other related tasks in comparison with other approaches like inst2vec [4], prograML [10], and those that follow token-based [11] representations. Hence, we choose to use IR2Vec embeddings for representing functions in our problem.

3 PROPOSED METHODOLOGY

Overview. For a given high-level program that implements an algorithm (that are part of a network function), we generate LLVM IR. As shown in Fig. 1, from the LLVM IR, we first extract the call graph and then obtain the function level IR2Vec representations. Next, we create a machine learning model to learn the distinction between the functions corresponding to different functionalities. Using this trained model, we evaluate our methodology.

Program Representation. LLVM IR of a program can potentially contain infinite variables/constants. We abstract out such symbolic variables/constants with generic information. This step is analogous to the commonly followed preprocessing steps in natural language processing. IR2Vec uses this preprocessed representation to generate program embeddings, as described in Sec. 2.

To generate the embeddings at function level, we extract the program's call graph to determine the appropriate caller and callee information. And the function level embeddings are obtained as the sum of all instruction embeddings corresponding to the instructions in the function, as proposed by IR2Vec. If there is a call instruction in the function, we obtain the embeddings of the callee function corresponding to the function call. The embeddings of the callee is now used to represent the call instruction while forming function level embeddings. This process of generating function level embeddings is done in a bottom-up, depth-first manner, starting from the nodes of the call graph that do not have any function call and iterating up to generate the representation of the callers, whose embeddings are to be determined.

This process of forming representations of call instructions based on the called functions serves as contextual information, making our approach interprocedural. Also, such an approach is important, as almost all the standard libraries are written in a modular manner, where the core functions are reused. For instance, OpenSSL contains a core `AES_encrypt()` method where the encryption logic using AES is present. This method is invoked by the methods that perform different modes of AES encryption like CBC - `AES_cbc_encrypt()`, ECB - `AES_ecb_encrypt()`, etc. In such cases, without having the insights on the core `AES_encrypt()` method, it would not be possible to identify the functionality.

Training. We model the problem of algorithm identification as a classification problem - where we classify the functions in a NF program as one of the known classes of algorithms that can be accelerated. Each algorithm constitutes a class, and the functions that implement the algorithm result in the data points of that class.

We use a simple three-layered fully connected deep neural network as the classifier. Function level IR2Vec representations, generated as described above is used as the input to the model. These representations are directly obtained using the trained vocabulary [39] without a separate learning specific to our approach.

The embeddings are further normalized following the standard min-max normalization. Batch normalization [19] with ReLU as the activation function is used, and a dropout of 25% is used as a regularizer between the layers. The last layer of the network is a softmax layer, with the number of neural units equal to the number of classes under consideration. Given the function embeddings, we train the model using the Stochastic Gradient Descent (SGD) algorithm, with categorical cross-entropy as the loss function to predict the desired class in a supervised manner.

4 EXPERIMENTATION

We set the following evaluation objectives for our approach to identifying Packet Processing Algorithms:

(a) Evaluate if our approach using IR2Vec can qualitatively represent the programs, and help in improving the accuracy of algorithm identification.
(b) Evaluate whether our approach to classify Network Function algorithms can be generalized for more classes.

4.1 Dataset Collection

Algorithm identification is an undecidable problem [35]. Hence, there is a wide potential for using ML based solutions. Identification of suitable accelerators, manually, for hundreds of functions in a large program can become very tedious. Hence, ML based solutions have a wide potential to arrive at an approximate identification. However, as mentioned earlier, an important challenge for using

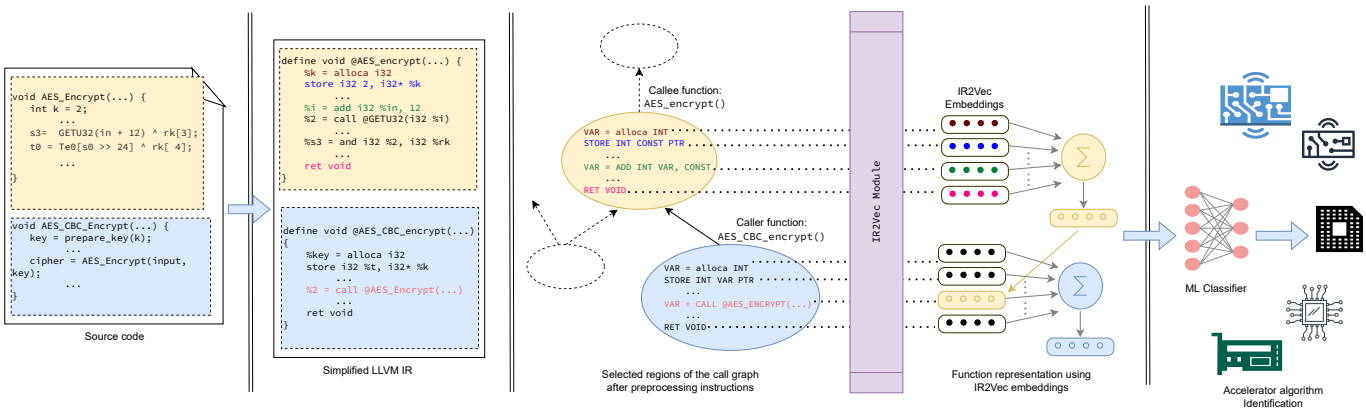

Figure 1: Overview of the proposed approach

ML approaches for identifying the algorithms is the non-availability of dataset. One way to address this challenge is to manually write different programs corresponding to each algorithm of our interest. However, this is much time-consuming and laborious. On the other hand, there are very few standard sources of implementations of algorithms of a network function, inhibiting the usage of ML based solutions. For instance, in the case of cryptography algorithms, there are very few libraries like OpenSSL, CryptoPP, etc. with each of them providing functions for encryption and decryption using various modes of operation wherever applicable. We use two datasets for evaluation - (i) CRC codes obtained from Clara [34], (ii) another dataset created out of cryptography algorithms by collecting several implementations as described below.

CRC Dataset. A recent work, Clara [34] proposes a tool that provides offload insights useful to the developer during cross-platform porting. As part of this tool, they propose a methodology to identify if the given code performs a network function. For this, they propose a dataset containing two classes: (i) implementation of CRC algorithms, and (ii) implementation of non-network functions scraped from the online judge platforms. In particular, it contains 22 CRC programs and 2,785 non-CRC programs in the training set and 33 CRC and 1,971 non-CRC programs in the test set. As it can be seen, this dataset is of two classes, and *skewed* with very few CRC programs when compared to the other class.

Cryptography Dataset. In real-world use cases, a single packet processing application (or NF) comprises of multiple compute-intensive algorithms and can leverage multiple SmartNIC accelerators for speedups (Tab. 1). For example, a VPN gateway is composed of 4 algorithms that can map to the accelerators for error detection, symmetric/asymmetric crypto, and secure end-to-end tunnels. Hence, there is an increased scope of extending the binary classification dataset proposed by Clara to support multiple classes.

Hence, we further extend the dataset to contain multiple NF algorithms and to evaluate the generalizability of our approach. We systematically collected programs that implement cryptography algorithms in popular cryptography libraries like OpenSSL (version 1.1 and 3) [30], CryptoPP [8], Botan [5], Nettle [26], Wolfcrypt [40] and MbedTLS [3].

Table 2: Dataset Description

	OpenSSL (v1.1)	OpenSSL (v3)	Crypto-PP	Botan	Nettle	wolf-Crypt	Mbed-TLS	Total
AES	8	7	8	6	2	6	5	42
DES	13	13	8	4	4	2	4	48
RSA	5	7	4	5	0	7	3	31
Total	26	27	20	15	6	15	12	121

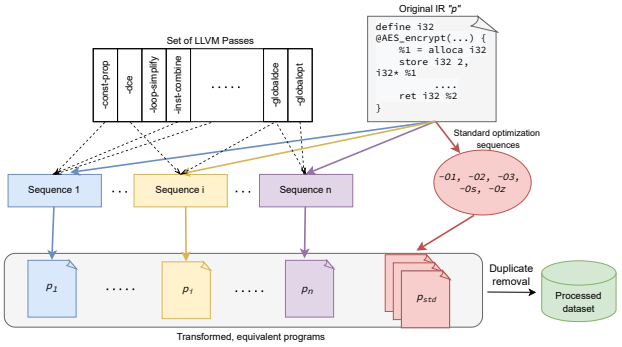

Figure 2: Our approach for extrapolating data points

In Tab. 2, we show the libraries chosen and the number of functions collected for each algorithm. As an initial step, we chose to collect methods for encryption and decryption corresponding to AES, DES, RSA algorithms. We consider the functions of encryption/decryption algorithms that use different modes of operation—like CBC, OFB, ECB, CFB, CTR, CCM, GCM—as belonging to the same class.

In total, we obtained about 121 functions performing AES, DES, and RSA encryption/decryption from these standard libraries and implementations. It can be seen that despite our efforts, the number of variations of these algorithms available is small; they are *too few in number* to be suitable for training a machine learning model.

Dataset expansion. Generally in modern compilers, program transformations are implemented as optimization passes. Some common transformations include optimizations like constant propagation, dead code elimination, and loop-unrolling. In addition to

these individual transformations, compilers also provide standard optimization sequences like `-O1`, `-O2`, `-O3`, `-Os`, and `-Oz`. These standard optimization sequences contain a sequence of transformation passes in a predetermined order arrived by expert compiler engineers. As an example, in LLVM12, the `-O2` sequence contains a sequence of 99 individual transformation passes, while the same number is 102 in `-O3` sequence.

We propose expanding the dataset of the extracted methods by applying various compiler transformations. These transformations, while preserving the semantics, change the code structure by changing the size, execution time, memory characteristics and power efficiency. This process results in several implementations of the same algorithm that are syntactically different (but semantically equivalent), thereby adding sufficient diversity to our dataset . These transformations include: (a) standard optimization sequences provided by the LLVM compiler toolchain, (b) new transformation sequences that we generate by concatenating individual transformations.

An overview of our approach is shown in Fig. 2. We first generate the base LLVM IR file for each library. For (a), we use the 5 standard optimization sequences readily available from LLVM to generate 5 different (semantically equivalent) versions of the same input program. In addition, for (b), we randomly choose different permutations of varying lengths among the available transformation passes. More specifically, we obtain 300 different pass sequences, with a random number of passes in each sequence, obtaining 306 variations (in total) for each function. Overall, we are able to generate about $37K$ (306×121) data points by following this process. It can be seen that this process can yield a large-enough set of data points with sufficient variation. Because the LLVM compiler is widely used for performing architecture-independent/dependent optimizations, these variations have a potential to have representatives that are closer to the optimized codes that run efficiently on various DSAs as well.

It can be noted that not all transformations would have an impact on the input program and yield new transformed programs. A simple example would be loop transformations having a little or no impact on the program that do not contain any loops. Hence the above process could potentially lead to duplicate data points.

There are two ways to handle duplicates: (i) duplicates can be allowed, considering that they are the result of a valid transformation/use case; or (ii) such duplicates can be identified and removed from the dataset, as they might have no impact depending on the application (ML algorithm).

For (ii), it is not possible to use a duplicate elimination based on IR-level comparison; it will not be a (program-theoretically) sound [7] method. So, IR2Vec embedding based comparison can be used to identify the duplicates. Such a comparison is guaranteed to be sound, and it is very simple to implement, involving just vector operations.

In this work, we choose not to remove the duplicates.

4.2 Detection of CRC algorithm

For our first objective, we compare our approach based on IR2Vec embeddings for algorithm identification with the methodology of Clara [34]. For this experiment, we use the CRC dataset of Clara after using our dataset expansion approach to remove the skewedness. We use the non-CRC class of this dataset to serve as data points for functions that do not have a suitable accelerator.

We obtained IR2Vec embeddings for this dataset, as explained above. These embeddings are given as input to various classification models like Gradient Boosting Decision Trees (GBDT), Decision Trees (DT), K- nearest neighbors (KNN), Multilayer perceptron (MLP), TPOT [29] (AutoML), and Support Vector Machine (SVM) classifiers. We use sklearn module with the standard set of parameters for these models, like Clara. In addition to these, we studied the results from the three-layered neural network (DNN) described earlier. We used precision, recall, and F1 score as the metrics for evaluation.

We generated the IR2Vec embeddings for this dataset using LLVM V12.0. Precision, recall, and F1 score obtained by our approach is shown in Tab. 3. Clara's approach for algorithm identification involves extracting opcodes of the programs, applying pattern extraction algorithms, and representing them using one-hot encodings, followed by performing binary classification. We feel that this process of extracting different features from the code that can be used to identify the underlying algorithm is non-trivial. For instance, programs that perform different functions on different architectures would necessitate identifying different features. Hence, coming up with such a feature set needs domain expertise. Also, on using the IR generated with LLVM v12.0 for comparison, Clara (which uses LLVM v6.0) generates an empty feature set making it infeasible to compare the results. Hence, we had to tune Clara's implementation and experimentally set support and confidence thresholds to 0.05 and 0.001. We also set the maximum length of sequences to 4. We observe that on tuning the support and confidence thresholds of the feature extraction algorithm, Clara generated new set of features. We use this new setup for experimentation for the purpose of comparison. We obtain 0.983, 0.488, and 0.899 as the precision, recall, and F1 scores from the best performing DT model described by Clara for this experiment. In comparison, using our approach we obtain 1.0, 0.998, and 0.998 as the precision, recall, and F1 scores from our dense model.

It can also be seen that the precision/recall values across all models corresponding to our approach are consistently better than the earlier approach of Clara's algorithm identification.

4.3 Detection of cryptography algorithms

As earlier, we generate the embeddings for our collected cryptography dataset; these are given as input to the classification phase along with the CRC dataset used in Sec. 4.2. Even for this, we perform experiments using various machine learning models and obtain the weighted precision, recall, and F1 score.

In order to compare our results with Clara, we have modified it to predict multiple classes. We provide the LLVM IR from our generated dataset to Clara's framework. Even for this dataset, we observe that the default Clara setup generates an empty feature set, and hence could not be used for experimentation as such. So, we modify Clara's MSPE implementation as explained earlier in Sec. 4.2 for further experimentation and analysis.

Table 3: Precision, Recall and F1 score for algorithm identification using Clara's approach and our approach

Model	CRC						Cryptography					
	Clara			IR2Vec			Clara			IR2Vec		
	Precision	Recall	F1 score	Precision	Recall	F1 score	Precision	Recall	F1 score	Precision	Recall	F1 score
GBDT	**0.992**	0.435	0.605	0.974	**0.997**	**0.985**	0.664	0.594	0.627	**0.949**	**0.947**	**0.948**
DT	0.983	0.488	0.652	**0.994**	**0.995**	**0.994**	0.661	0.622	0.641	**0.969**	**0.968**	**0.968**
MLP	0.983	0.437	0.605	**0.997**	**0.999**	**0.998**	0.666	0.590	0.626	**0.959**	**0.958**	**0.958**
SVM	0.992	0.484	0.651	**0.999**	**0.995**	**0.997**	0.646	0.604	0.624	**0.898**	**0.894**	**0.896**
kNN	0.980	0.486	0.650	**0.999**	**0.999**	**0.999**	0.596	0.630	0.613	**0.976**	**0.974**	**0.975**
AutoML	0.980	0.487	0.651	**0.999**	**0.999**	**0.999**	0.661	0.621	0.640	**0.979**	**0.978**	**0.978**
DNN	-	-	-	1	0.998	0.999	-	-	-	0.977	0.976	0.976

From the best performing DT model of Clara, we obtain precision, recall, and F1 values as 0.661, 0.622, and 0.641 respectively. Using our approach, we obtain 0.979, 0.978, and 0.978 as the precision, recall, and F1 scores from the AutoML model. Metrics from other ML models using our dataset is shown in Tab. 3. We obtain an F1 score of 0.99 and 0.97 using the DNN model described in Sec. 3 on CRC and cryptography datasets respectively. It can be seen that our approach consistently results in superior performance irrespective of the underlying ML model. Also, our approach using IR2Vec embeddings achieves superior results and generalizes better on multiclass dataset in comparison with Clara. This could be attributed to the ability of IR2Vec embeddings as it captures semantic and syntactic information of the program. However, in comparison, Clara uses a set of opcodes obtained by pattern extraction algorithms for identifying algorithms.

5 RELATED WORK

Clara [34] demonstrates performance gains by offloading software middlebox (NFs) programs to SmartNICs. Clara generates offloading insights by analyzing unported NF programs and suggests effective porting strategies based on the insights. As part of it, Clara automatically identifies packet-processing algorithms (*e.g.*, CRC, LPM) that can be accelerated. However, Clara's approach to represent programs requires domain expertise, especially for selecting relevant program features based on the underlying dataset and application. In contrast, this work proposes to use program embedding, an alternative and sophisticated approach that automatically learn to represent the relevant program features. Such an approach encodes both syntactic and semantic information while representing programs and is application-independent and more scalable. Gallium [42], Flightplan [38], and Lyra [17] enable mapping of code blocks of an NF program across multiple devices (*e.g.*, Server, SmartNICs, FPGAs, programmable switch ASICs). In contrast, this work enables the mapping of code blocks across accelerators.

Prior works have leveraged accelerators on SmartNICs and demonstrated significant performance gains for network functions. QoS accelerator is used by OpenBNG [20] and VFP [14]; packet-rewrite accelerator by HyperNAT [13] and [9]; symmetric crypto (AES) accelerator by [9]; flow-cache accelerator by SmartWatch [31]; AI/ML accelerators by [15]; crypto engines, compression engine, and pattern-matching engines by ipipe [22]. These works provide strong motivation for mapping packet-processing applications to accelerators. Our work is a first step to enable automatic mapping of code blocks to appropriate accelerators.

Recently, different ways of representing programs have been proposed. They either follow token based representations [11], Abstract Syntax Tree based representations [2], or IR based representations [4, 37, 39]. Naturally, IR-based embeddings are programming language independent and are easier to encode semantic information via compiler analysis. These embeddings are often learned in a supervised manner, specific for a task [2, 11]. We use IR2Vec, which is learned in an unsupervised manner, making it application-independent.

6 CONCLUSIONS AND FUTURE WORK

We proposed a novel way to classify packet processing algorithms as a means to map them to accelerators. Our methodology relies on (i) realistic collection of programs that could potentially be mapped to accelerators, (ii) systematic generation of large dataset using (existing) compiler transformation, (iii) using program embeddings based techniques, and (iv) ML techniques for better results.

We plan to expand the dataset with more data points and more classes like SHA, HMAC, ECC, etc.

Another potential future work is to reorient our approach such that, given a network function with n algorithms, we would be able to map them to the respective accelerator.

ACKNOWLEDGMENTS

We thank the authors of Clara for sharing the datasets to replicate their work. We also thank the anonymous reviewers for the feedback and insightful comments.

This research is supported by NMICPS TiHAN IIT Hyderabad, an NSM research grant (MeitY/R&D/HPC/2(1)/2014) and a Google PhD fellowship.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2Vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL, Article 40 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290353

[3] arm. 2009. SSL Library mbed TLS/PolarSSL. https://tls.mbed.org/. [Online; accessed 16-March-2022].

[4] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. In *Proceedings of the 32Nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., USA, 3589–3601. http://dl.acm.org/citation.cfm?id=3327144.3327276

[5] Botan. 2000. Botan: Crypto and TLS for Modern C++. https://botan.randombit.net/. [Online; accessed 16-March-2022].

[6] Cavium. 2013. Cavium LiquidIO® Server Adapter Family. https://datasheet.octopart.com/CN6130-110SV-G-Cavium-Networks-datasheet-26366670.pdf. [Online; accessed 16-March-2022].

[7] P. Cousot and R. Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, Los Angeles, California, 238–252.

[8] Crypto++. 1995. Crypto++® Library 8.6. https://www.cryptopp.com/. [Online; accessed 16-March-2022].

[9] Tianyi Cui, Wei Zhang, Kaiyuan Zhang, and Arvind Krishnamurthy. 2021. *Offloading Load Balancers onto SmartNICs*. Association for Computing Machinery, New York, NY, USA, 56–62. https://doi.org/10.1145/3476886.3477505

[10] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 2244–2253.

[11] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 219–232.

[12] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

[13] Shaoke Fang, Qingsong Liu, and Wenfei Wu. 2021. HyperNAT: Scaling Up Network Address Translation with SmartNICs for Clouds. *2021 IEEE Global Communications Conference (GLOBECOM)* (Dec 2021). https://doi.org/10.1109/globecom46510.2021.9685551

[14] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 315–328. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/firestone

[15] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

[16] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher K. I. Williams, and Michael O'Boyle. 2011. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *International Journal of Parallel Programming* 39, 3 (01 Jun 2011), 296–327. https://doi.org/10.1007/s10766-010-0161-2

[17] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 435–450. https://doi.org/10.1145/3387514.3405879

[18] Matthew S. Hecht. 1977. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY, USA.

[19] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 37)*, Francis Bach and David Blei (Eds.). PMLR, Lille, France, 448–456. http://proceedings.mlr.press/v37/ioffe15.html

[20] Ralf Kundel, Leonhard Nobach, Jeremias Blendin, Wilfried Maas, Andreas Zimber, Hans-Joerg Kolbe, Georg Schyguda, Vladimir Gurevich, Rhaban Hark, Boris Koldehofe, and Ralf Steinmetz. 2021. OpenBNG: Central Office Network Functions on Programmable Data Plane Hardware. *Int. J. Netw. Manag.* 31, 1 (jan 2021), 25 pages. https://doi.org/10.1002/nem.2134

[21] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[22] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading Distributed Applications onto SmartNICs Using IPipe. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 318–333. https://doi.org/10.1145/3341302.3342079

[23] Alberto Magni, Christophe Dubach, and Michael F. P. O'Boyle. 2013. A large-scale cross-architecture evaluation of thread-coarsening. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, William Gropp and Satoshi Matsuoka (Eds.). ACM, 11:1–11:11. https://doi.org/10.1145/2503210.2503268

[24] Marvell. 2021. Marvell Octeon LiquidIO SmartNICs and DPUs. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-octeon-10-dpu-platform-product-brief.pdf. [Online; accessed 16-March-2022].

[25] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[26] Niels Möller et al. 2001. Nettle: a low-level cryptographic library. https://www.lysator.liu.se/~nisse/nettle/nettle.html. [Online; accessed 16-March-2022].

[27] Netronome. 2017. Netronome Agilio SmartNICs. https://www.netronome.com/media/documents/PB_NFP-4000-7-20.pdf. [Online; accessed 16-March-2022].

[28] Nvidia. 2022. Nvidia Bluefield Data Processing Units. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf. [Online; accessed 16-March-2022].

[29] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. 2016. Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016* (Denver, Colorado, USA) *(GECCO '16)*. ACM, New York, NY, USA, 485–492. https://doi.org/10.1145/2908812.2908918

[30] OpenSSL. 1998. The Open Source Toolkit for SSL/TLS. http://openssl.org/. [Online; accessed 16-March-2022].

[31] Sourav Panda, Yixiao Feng, Sameer G Kulkarni, K. K. Ramakrishnan, Nick Duffield, and Laxmi N. Bhuyan. 2021. SmartWatch: Accurate Traffic Analysis and Flow-State Tracking for Intrusion Prevention Using SmartNICs. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies* (Virtual Event, Germany) *(CoNEXT '21)*. Association for Computing Machinery, New York, NY, USA, 60–75. https://doi.org/10.1145/3485983.3494861

[32] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.

[33] Pensando. 2020. Pensando DSC-100 Distributed Services Card. https://pensando.io/wp-content/uploads/2020/03/DSC-100-ProductBrief-v06.pdf. [Online; accessed 16-March-2022].

[34] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. 2021. Automated SmartNIC Offloading Insights for Network Functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 772–787. https://doi.org/10.1145/3477132.3483583

[35] H. G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. http://www.jstor.org/stable/1990888

[36] Rinku Shah, Vikas Kumar, Mythili Vutukuru, and Purushottam Kulkarni. 2020. TurboEPC: Leveraging Dataplane Programmability to Accelerate the Mobile Packet Core. In *Proceedings of the Symposium on SDN Research* (San Jose, CA, USA) *(SOSR '20)*. Association for Computing Machinery, New York, NY, USA, 83–95. https://doi.org/10.1145/3373360.3380839

[37] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2Vec: Value-Flow-Based Precise Code Embedding. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 233 (nov 2020), 27 pages. https://doi.org/10.1145/3428301

[38] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. 2021. Flightplan: Dataplane Disaggregation and Placement for P4 Programs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 571–592. https://www.usenix.org/conference/nsdi21/presentation/sultana

[39] S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and Y. N. Srikant. 2020. IR2Vec: LLVM IR Based Scalable Program Embeddings. *ACM Trans. Archit. Code Optim.* 17, 4, Article 32 (Dec. 2020), 27 pages. https://doi.org/10.1145/3418463

[40] wolfSSL. 2006. wolfCrypt Embedded Crypto Engine. https://www.wolfssl.com/products/wolfcrypt-2/. [Online; accessed 16-March-2022].

[41] Jinli Yan, Lu Tang, Junnan Li, Xiangrui Yang, Wei Quan, Hongyi Chen, and Zhigang Sun. 2019. UniSec: A Unified Security Framework with SmartNIC Acceleration in Public Cloud. In *Proceedings of the ACM Turing Celebration Conference - China* (Chengdu, China) *(ACM TURC '19)*. Association for Computing Machinery, New York, NY, USA, Article 9, 6 pages. https://doi.org/10.1145/3321408.3323087

[42] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. 2020. Gallium: Automated Software Middlebox Offloading to Programmable Switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 283–295. https://doi.org/10.1145/3387514.3405869