information

MDPI

Article

CryptoKnight: Generating and Modelling Compiled Cryptographic Primitives

Gregory Hill [1] and Xavier Bellekens [2,*]

[1] School of Informatics, The University of Edinburgh, Edinburgh EH8 9YL, UK; gregorydhill@outlook.com
[2] Division of Cyber Security, Abertay University, Dundee DD1 1HG, UK
* Correspondence: x.bellekens@abertay.ac.uk; Tel.: +44-(0)-1382-30-8482

Received: 12 July 2018; Accepted: 6 September 2018; Published: 10 September 2018

Abstract: Cryptovirological augmentations present an immediate, incomparable threat. Over the last decade, the substantial proliferation of crypto-ransomware has had widespread consequences for consumers and organisations alike. Established preventive measures perform well, however, the problem has not ceased. Reverse engineering potentially malicious software is a cumbersome task due to platform eccentricities and obfuscated transmutation mechanisms, hence requiring smarter, more efficient detection strategies. The following manuscript presents a novel approach for the classification of cryptographic primitives in compiled binary executables using deep learning. The model blueprint, a Dynamic Convolutional Neural Network (DCNN), is fittingly configured to learn from variable-length control flow diagnostics output from a dynamic trace. To rival the size and variability of equivalent datasets, and to adequately train our model without risking adverse exposure, a methodology for the procedural generation of synthetic cryptographic binaries is defined, using core primitives from *OpenSSL* with multivariate obfuscation, to draw a vastly scalable distribution. The library, CryptoKnight, rendered an algorithmic pool of AES, RC4, Blowfish, MD5 and RSA to synthesise combinable variants which automatically fed into its core model. Converging at 96% accuracy, CryptoKnight was successfully able to classify the sample pool with minimal loss and correctly identified the algorithm in a real-world crypto-ransomware application.

Keywords: cryptography; deep learning; reverse engineering

1. Introduction

The idea of cryptovirology was first introduced by Young and Yung [1] to describe the offensive nature of cryptography for extortion-based security threats. It comprises a set of revolutionary attacks that combine strong cryptographic techniques with unique viral technology; designed to infect, encrypt and lock-down available hosts, this category of malware has had disastrous consequences for many [2,3]. For those who can afford to reclaim their private data, the financial loss is typically quite substantial, despite the fact that there is no guaranteed recovery. Ultimately, without a backup, there is little that can be done. Preventative frameworks have been proven to effectively halt unusual activity [4,5] by closely monitoring the file system's Input/Output (I/O), but administrators are not always likely to follow best practices [6] and this overhead is quite substantial for the average user. In any case, malware authors will continually locate unique exploits to further their advantage.

The cryptovirological landscape has evolved in recent years, and a definite growth has been noted in the overall number of targeted attacks and variants [7]. A long-term study of 1359 ransomware samples [4] observed between 2006 and 2014 found a distinct number of variants with cryptographic capabilities. During analysis, these instances were found to use both standard and customised cryptography with generational enhancements, specifically in terms of key creation and management. Infamous variants are known to have employed well-established documented

algorithms [8,9], but many use other techniques. In some instances, malware variants have employed custom 'cryptography', lesser-known algorithms (such as the Soviet/Russian symmetric block cipher GOST [10]), common substitution-ciphers, or exclusively, encoding mechanisms. Tailored cryptosystems particularly limit the scale of effective analysis [8], and due to the large threat surface [7], more efficient tools are required.

The field of malware analysis seeks to determine the potential impact of malicious software by examining it in a controlled environment. Investigators find flaws otherwise unknown to current identification technologies—sourcing keys and blocking further infection [2]. When reverse engineering a potentially malicious executable, several issues should be addressed. Possible problems include the accuracy of analysis, quality of the application's obfuscation, and the lifetime of findings. The analysis of a binary can typically be considered from two viewpoints: static or dynamic. Static analysis is performed in a non-runtime environment, therefore examination is relatively safe; however potential morphism restricts the accuracy of results [11,12]. Alternatively, dynamic analysis [13] sequentially assesses a binary throughout its execution, which can provide significantly more accurate results and contend with obfuscatory measures [14,15], but if not properly handled, samples could prove somewhat hazardous. This manuscript focuses on the latter methodology to build a precise 'image' of execution.

Cryptographic algorithm identification facilitates malware analysis in several ways, but in this case, when assessing ransomware strains, it yields a starting point for investigation which is essential when analytical time is restricted. With the uncertainty surrounding an application's custom, undocumented or established cryptosystem, analysts struggle to maintain complete awareness of the field—which makes this task ideal for automation. To effectively model cryptographic execution, previous research has relied on several assumptions and observations. These features do not necessarily always depict cryptographic code, but provide a baseline for analysis. For instance, cryptographic algorithms naturally involve the use of bitwise integer arithmetic and logical operations. These activities frequently reside in loops, for example, block ciphers typically loop over an input buffer to decrypt it block-by-block. Lutz [8] also postulated that any encrypted data is likely to have a higher information entropy than decrypted data.

Deep learning studies intricate Artificial Neural Networks (ANNs) with multiple hidden computational layers [16] that effectively model representations of data with multiple layers of abstraction, in which the high-level representations can amplify aspects of the input that are important for discrimination. A Convolutional Neural Network (CNN) [17], is a specialised architecture of ANN that employs a convolution operation in at least one of its layers. A variety of substantiated CNN architectures have been used to great effect in computer vision [18] and even Natural Language Processing (NLP), with empirically distinguished superiority in semantic matching [19], compared to other models.

CryptoKnight (https://github.com/AbertayMachineLearningGroup/CryptoKnight) is developed in coordination with these ideas. We introduce a learning system that can easily incorporate new samples through the scalable synthesis of customisable cryptographic algorithms. Its entirely automated core architecture is aimed to minimise human interaction, thus allowing the composition of an effective model at run-time. We tested the framework on several externally sourced applications using non-library linked functionality. Our experimental analysis indicates that CryptoKnight is a flexible solution that can quickly learn from new cryptographic execution patterns to classify unknown software. This manuscript presents the following contributions:

- Our unique convolutional neural network architecture fits variable-length diagnostic data to map an application's time-invariant cryptographic execution.
- Complimented by procedural synthesis, we address the issue of this task's disproportionate latent feature space.
- The realised framework, CryptoKnight, has demonstrably faster results compared to that of previous methodologies, and is extensively re-usable.

2. Related Work

The cryptovirological threat model has rapidly evolved over the last decade. A number of notable individuals and research groups have attempted to address the problem of cryptographic primitive identification. The following passages will discuss the consequences of their findings.

2.1. Heuristics

Heuristical methods [20] are often used to locate an optimal strategy for capturing the most appropriate solution and have previously shown great success in cryptographic primitive identification. A joint project from ETH Zürich (Zürich, Switzerland) and Google, Inc. (Menlo Park, CA, USA) [8] detailed the automated decryption of encrypted network communication in memory, to identify the location and time a subject binary interacted with decrypted input. From an execution trace which dynamically extracted memory access patterns and control flow data, Lutz [8] was able to identify the necessary factors required to retrieve the relevant data in a new process. His implementation was successfully able to identify the location of several decrypted webpages in memory fetched using *cURL*/*OpenSSL*, and even successfully extracted the decrypted output from a *Kraken* malware binary. The entropy metric was found to negatively affect the recognition of simple substitution ciphers as they do not typically effect the information entropy, which was also found to affect the analysis of *GnuPG*. Gröbert et al. [9], Gröbert [21] also used fine-grained dynamic binary analysis to generate a high-level control flow graph which was evaluated using three heuristics. The chains heuristic measured the ordered concatenation of all mnemonics in a basic block, comparing known signatures; the *mnemonic-const* heuristic extended the former method by assessing the combination of instructions and constants; and the *verifier* heuristic confirmed a relationship between the I/O of a permutation block. Trialled on *cURL*, the tool detected both Rivest-Shamir-Adleman (RSA) and Advanced Encryption Standard (AES) in a traced Secure Sockets Layer (SSL) session. When tested on a real-world malware sample, *GpCode*, the operation successfully detected the cryptosystem and extracted its keys, though the trace took fourteen hours to complete with an extra eight hours for analysis. The *Crypto Intelligence System* [22] integrates several these heuristical measures to counter the problem of situational dependencies, deriving a set of requirements that a detection framework should satisfy in order to be effective. In addition to correct classification, such tools should be able to handle packed programs, must not be too invasive, provide a range of input for alternate approaches and be somewhat extendable. Evaluating the aforementioned detection techniques [8,21], Matenaar et al. [22] found many different advantages but noted that many heuristics have 'immanent weaknesses' that prevent adequate generalisation.

2.2. Data Flow Analysis

Representational patterns of cryptographic data through dynamic analysis are often quite distinctive [23,24]. By closely monitoring an application's I/O it is possible to pinpoint a matching algorithm and highlight the specific code at run-time. For instance, Li et al. [15] assessed the 'avalanche effect' as a unique discriminatory feature, monitoring small changes in the input which would dramatically alter the output. Unfortunately, these methods often fail to adapt in the face of heavy obfuscation, as there are several simplistic techniques to bypass such filters. A tool by the name of *CryptoHunt* [14] was recently developed to identify cryptographic implementations in binary code despite advanced obfuscation. The implementation tracked the dynamic execution of a reference binary at instruction level, to further identify and transform loop bodies into boolean formulas. Each rule was designed to successfully abstract the particular primitive, but remain compact to describe the most emblematic features. Unlike sole I/O verification, this semantic depth more prominently revealed distinguishable features regardless of obfuscation. However, similar success was shown by Calvet et al. [25] who actually focused on I/O discrimination. Another unique and fairly effective approach for the identification of symmetric algorithms in binary code was based on subgraph

isomorphism and static analysis. Lestringant et al. [11] resolved each cryptographic algorithm to a Data Flow Graph (DFG), normalising the structure without breaking semantics, and then compared it to signatures of XTEA, Message Digest 5 (MD5) and AES with 100% accuracy. However, the formula relied on the manual selection of appropriate signatures which distinguished the applicable algorithms. For their three instances, generation was elementary, but would not realistically scale to the dimensions sought in this paper. All aforementioned tools ultimately suffer from similar issues, requiring either pre-existing reference implementations or arduous manual integration.

2.3. Machine Learning

Attempting to address a difficulty with past methodologies, one thesis [26] studied the suitability of machine learning. While automated and highly efficient, thresholds [9,11,14] often require manual adjustment to manage the identification of new algorithmic samples. Hosfelt [26] sought to emphasise the ease of model retraining by analysing the performance of: Support Vector Machines (SVMs), Kernels, Naive Bayes, Decision Tree and K-means Clustering. The study was met with varying success, but ultimately suffered from a limited sampling of the latent feature space, preventing adequate scaling for more complex data—i.e., multi-purpose applications that may use cryptography in addition to other functions that can unintentionally obfuscate the control flow. With only 317 hand-crafted binary files, providing an equivalent number of vectors after metric-based feature extraction, the model ultimately under-fit. Baldwin and Dehghantanha [27] recently specialised a similar approach in the density-based detection of circumstantial op-codes consistent with the execution of crypto-ransomware. The authors used static analysis and SVMs to build a unique model comprised of benign and malicious binaries, showing exceptionally high accuracy in the discrimination of 443 unique opcodes; 100% in binary classification and 96.5% in family separation. Of particular importance to this research is their ranking methodology based on the reflection of all opcode occurrences. They show that certain instructions with similar densities prevent efficient discrimination, but unfortunately this distinction required significant manual interpretation. EldeRan [28] assessed a regularized logistic regression classifier, used in conjuncture with a dynamic feature extractor to produce a dataset of 582 ransomware and 942 'good' applications. While the results in this case were positive (Area Under Curve (AUC): 0.995), the methodology required a large number of malicious samples to have been run to generate a sufficient distribution.

2.4. Overview

There are several concerns with previous techniques, most of which relate to the difficulty of manually extracting signatures. Fundamentally, many systems rely on the researcher's ability to connect low-level ideas that correspond to high-level representations in an efficient manner. This is simply not doable in many situations, especially when presented with many alternate representations in a constricted time period. Malware signatures can radically change and undocumented primitives will likely go undetected. It is therefore the intent of this manuscript to detail a framework for the effectual verification of cryptographic signatures in unknown binary code with an emphasis on generalisation when presented with new conditions to minimize human-error. Intricate neural networks are well-suited to this task because of their ability to learn by example. Although prior literature has examined machine learning to an extent, this is the first work that has specifically addressed the use of deep learning. Earlier data variability issues are addressed by our method of procedural synthesis, to build an effectively large dataset with significant variability that prevents risk of adverse exposure. The full system, as in Figure 1, is comprised of three high-level stages:

1. Procedural generation guides the synthesis of unique cryptographic binaries with variable obfuscation and alternate compilation.
2. Assumptions of cryptographic code aid the discrimination of diagnostics from the dynamic analysis of synthetic or reference binaries, to build an 'image' of execution.

3. A DCNN fits variable-length matrices for ease of training and the immediate classification of new samples.

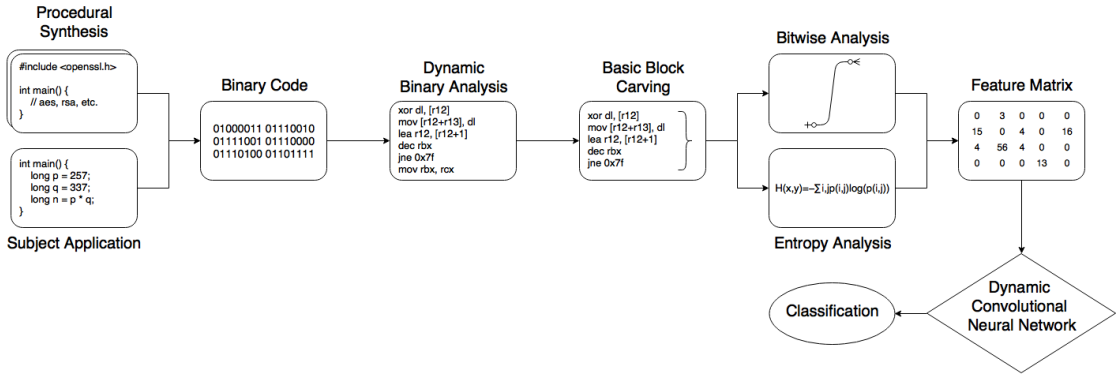


Figure 1. Framework Architecture.

3. Methodology

To construct a reasonably sized dataset with enough variation to satisfy the abstraction of cryptographic primitives, it is not enough to simply hand-write a small number of applications with little diversity in terms of operational outliers. For example, extracting features from the execution of a manually implemented single-purpose binary may give an appropriate feature vector, but re-running the extraction process will not provide any variation for repeating labels, outside of environmental setup. This methodology leverages procedural generation to include elements that provide some obfuscation without directly altering the intended control flow. For the three main algorithmic categories—symmetric, asymmetric and hashing—interpretation should correlate the related components to dynamically construct a unique executable.

3.1. Artefacts

OpenSSL is an open source cryptographic library that provides an Application Programming Interface (API) for accessing its algorithmic definitions. Review of its documentation revealed several similarities in the intended implementation of the each function. These specificities are either: *variable*, differ for each primitive; or *constant*, true for each category. This approach exclusively examined *C* in experimentation, but *C++* is also suitable.

Via appropriate headers, an application first imports the libraries which provide the expectant functionality when later compiled. Each primitive naturally requires contrastive functionality, so this is variable. Within the scope of an application's main body however, each symmetric algorithm requires the specification of a key and Initialization Vector (IV), asymmetric algorithms require a certificate declaration whereas hashing definitions do not expect either. These rules are categorically constant, therefore, their definitions can be specified by type. Next, a plaintext sequence is loaded into memory—directly or from a file—and ciphertext memory is allocated, also constant. Each sample will then employ its unique algorithm through differing declarations, reading in the plaintext, key or IV.

3.2. Obfuscation

Two primary transformation mechanisms were highlighted by Xu et al. [14]. The first technique discusses the abstraction of relevant data groups to decrease their perceptible mapping. For example, a multidimensional array may be concatenated into a single column and either expanded or accessed as necessary. The second technique concerns itself with the splitting of variables [29] to disguise their representation. Colloquially known as *data aggregation* and *data splitting*, these methods partially obfuscate the data flow without subtracting from an application's distinct activity. Outside of such well-defined obfuscation, the inclusion of structured loops, arithmetical or bitwise operations create

discriminatory irregularity in an otherwise translucent process. Analogously, many training images for computer vision will contain noise that suitably detract from the trivial classification of its subject, and this process aims to replicate such uncertainty.

3.3. Interpretation

Formatting each respective variable artefact to allow ease of parsing in a similarly structured markup tree will allow the interpretation of unique cryptographic applications with alternate obfuscation. Stochastically generated keys, IVs and plaintext will add additional variation into each image. Algorithm 1 outlines the pseudo-code for this procedure.

Algorithm 1 Cryptographic Synthesis

Input: Cryptographic Constants & Variables
Output: Application Codes
1: select obfuscation—(aggregation, split, normal)
2: write to file:
import statements
abstracted keys
encryption routines
3: inject randomised arithmetic
4: **return** relative location

When compiling the resultant collection of cryptographic applications, data variability can be further increased. With alternate compilers or optimisation options, the resultant object code will dramatically fluctuate. Real-world instances are rarely compiled identically so multivariate output can provide further assurance for generalisation.

3.4. Feature Extraction

Cryptographic execution is time-invariant, therefore a reference binary may employ its associated functions at any point within a trace. Unintentional obfuscation of the control flow will negatively affect discriminatory performance, as discovered by Hosfelt [26], so granularity needs to be high. This following approach opts to draw appropriate features from a reference binary using dynamic instrumentation via Intel's Pin API. Through the disassembly of run-time instruction data, this section's outlined measurements principally assess the activity's importance in relation to the assumptions of cryptographic code.

3.4.1. Basic Blocks and Loops

A Basic Block (BBL) is a sequential series of instructions executed in a particular order, exclusively defined when there is one branch in (entry) and one branch out (exit). A BBL ends with one of the following conditions:

- unconditional or conditional branch—direct/indirect,
- return to caller.

Each instruction is evaluated in a linear trace, if any criteria are met, it is marked as a *tail*. The following instruction is delimited as the *head* of the subsequent sequence, but can similarly be identified as a *tail*. The 'stack' of execution stores relevant data from each instruction, and two boolean expressions indicate the predetermined blocks. As each BBL is dynamically revealed, non-executed instructions will unfortunately not be observed [9], but we can monitor indirect branches.

Many high level languages share a distinctly strict definition of a loop, contrarily, common interpretations of amorphous code are loose. Extending previous definitions [30,31] we hence delineate a loop upon the immediate re-iteration of any BBL, as output from Algorithm 2.

Algorithm 2 Instruction Sequencing, BBL Detection

Input: Run-Time Hook
Output: Path of Execution

```
head = false
if (last instruction is tail) then
    head = true
end if
if (instruction is branch, call or return) then
    tail = true
else
    tail = false
end if
if (write) then
    get entropy (memory write)
end if
return stack
```

3.4.2. Instructions

Conventional architectures use a common instruction format, interpretable as: opcode operand (destination/source). In x86, there are zero to three operands (separated by commas), two of which specify the destination and source. For example, when AES performs a single round of an encryption flow it calls the instruction *66 0f 38 dc d1* which can be disassembled as *aesenc xmm2, xmm1*. Directly operating on the first operand, in this case *xmm2*, it performs a round of AES encryption using the 128-bit round key from *xmm1*. Although this is an interesting example, the Advanced Encryption Standard New Instructions (AES-NI) architecture presents a problem for later generalisation as cryptographic acceleration prevents detailed analysis.

Alternate object code is typically quite distinctive, a primitive may employ several operations, in any order, and it is important to not dwell on specificities—i.e., exact semantics. For each instruction in the 'carved' linear trace we weight its ratio of bitwise operations upon the cross-correlation of prominent operators from a pool of cryptographic routines for discriminatory emphasis.

3.4.3. Entropy

As characterized by Rényi et al. [32], the associated uncertainty of a finite discrete probability distribution $p = (p_1, p_2, \ldots, p_n)$ can be measured using Shannon's Entropy. Suppose $p_k > 0$ $(k = 1, 2, \ldots, n)$ and $\sum_{k=1}^{n} p_k = 1$, distribution p is measured by quantity $H[p] = H(p_1, p_2, \ldots, p_n)$ hence defined as:

$$H(p_1, p_2, \ldots, p_n) = \sum_{k=1}^{n} p_k log_2 \frac{1}{p_k} \tag{1}$$

Upon detecting a memory write, the respective location's contents can be replicated. Casting its value to distribution p will allow the immediate calculation of $H[p]$. Related memory can then be deleted to prevent unnecessary exhaustion. Each BBLs absolute entropy increase/decrease can then be scored by its relation to prior activations over opposing registers and then summated as in Figure 2 where each BBL is $\in \mathbb{W}$.

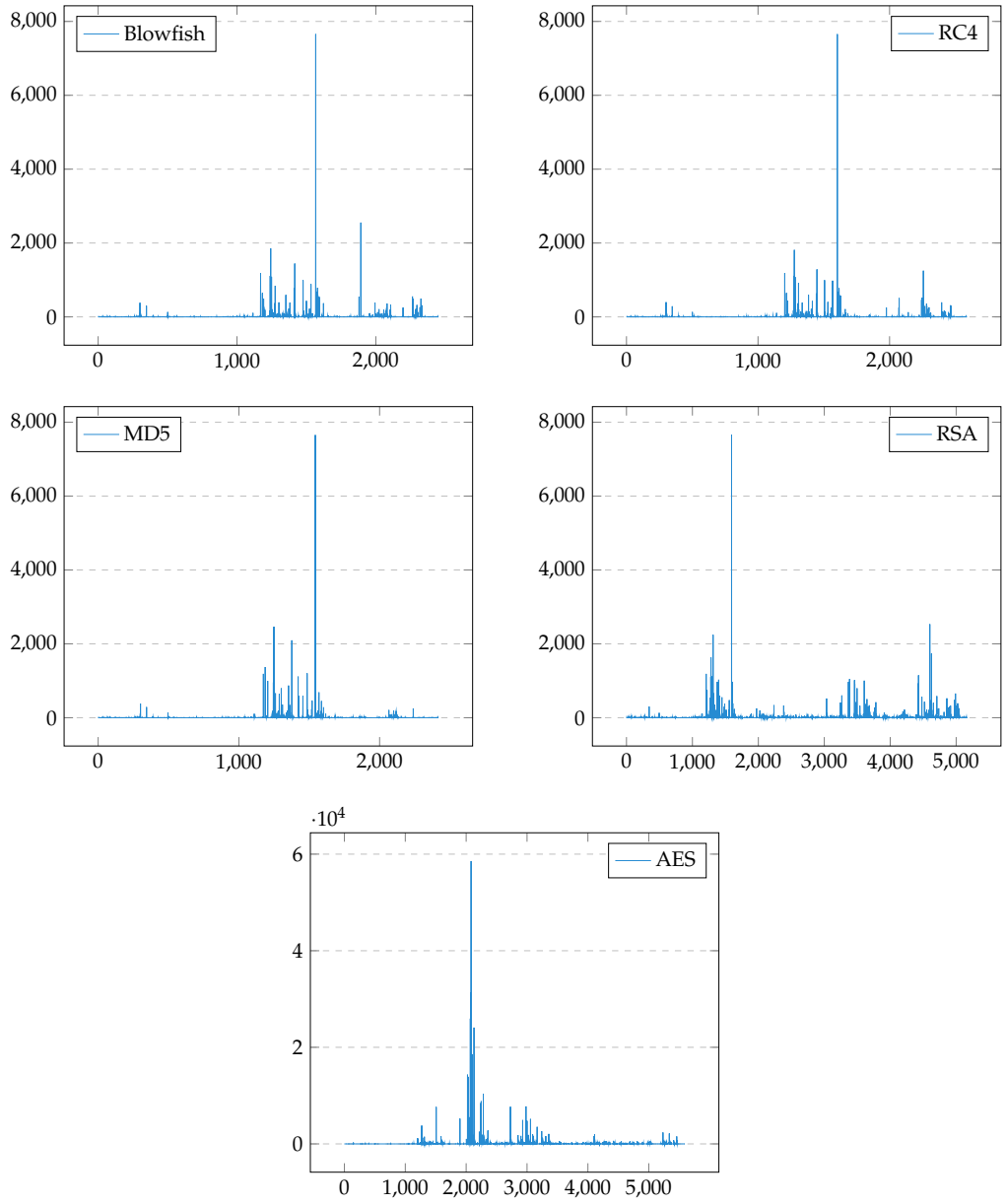


Figure 2. Entropy Scoring. The relative entropy scoring (y) of each BBL (x) within the traces from a sample set of algorithms. Each pattern is unique and should aid discrimination.

3.5. Model

Founded in earlier research [33], this proposed definition of DCNN treats the input in a manner similar to that of a sentence. For each word, embeddings are defined as $\mathbf{d}$, where $\mathbf{d}_i$ corresponds to the total weight of a particular operation, multiplied by its entropic score. Feature vector $\mathbf{w}_i \in \mathbb{W}^d$ is a therefore a column in sentence matrix $\mathbf{s}$ such that $\mathbf{s} \in \mathbb{W}^{d \times s}$. Let's say $\mathbf{w} = 128$, exclusively assessing pure arithmetic impact would make $\mathbf{d} = 12$, so $\mathbf{s}$ would equal 12×128. The model itself combines several one wide convolutions with (dynamic) k-max pooling and folding to map variable-length input. Topologically presented in Figure 3, the model combines several one wide convolutions with (dynamic) k-max pooling and folding to map variable-length input.

Mathematically, convolution is an operation on two functions of a real-valued argument, in this case a vector of weights $\mathbf{m} \in \mathbb{W}^m$ and a vector of inputs $\mathbf{s} \in \mathbb{W}^s$, to yield a new sequence $\mathbf{c}$. With *kernel* $\mathbf{m}$, the convolved sequence in a one-wide convolution takes the form $\mathbf{c} \in \mathbb{R}^{s+m-1}$, thus preserving

the number of defined embeddings. Equation (2) describes the selection of k, a distinct subset of $\mathbf{s}$ that most relevantly depicts an l-th order's progression. Based on the total number of convolutional layers L, the current layer l, the projected sentence length s and the predefined final pooling parameter k_{top}, any particular layer's selection is delimited as:

$$k_l = \max(k_{top}, [\frac{L-l}{L}s]) \tag{2}$$

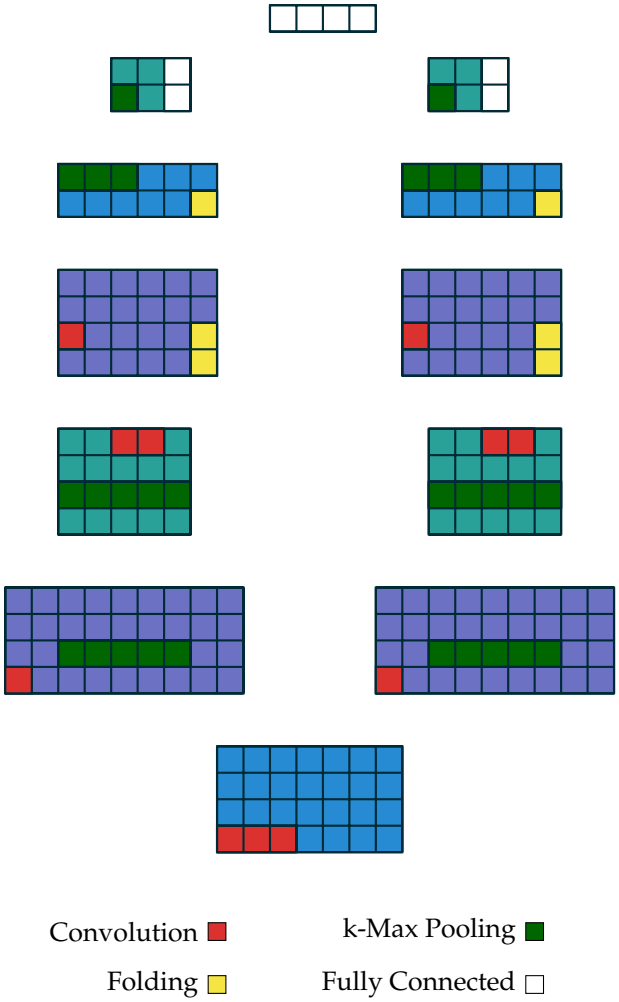


Figure 3. Dynamic Convolutional Neural Network. The architecture of a DCNN as illustrated by [33]. In this case, the model is intended for a seven word input sentence of embedding $d = 4$ with two convolutional layers ($m = 3$ and $m = 2$), two (dynamic) k-max pooling layers ($k = 5$ and $k = 3$) within two feature maps.

With value k and sequence $\mathbf{p} \in \mathbb{R}^p$ of length $p \geq k$ the base k-max pooling operation selects the subsequence $\mathbf{p}^k_{\max}$ of the k most active features. Completely unreactive to positional variation, it not only preserves the original perspective, but can also distinguish repetitious features. Since identifiable cryptographic routines may execute at any point in a sequential trace, this operation fits perfectly.

Another significant phase in this procedure simplifies the way the model perceives complex dependencies over rows. Veritable feature independence is removed through component-wise summation of every two rows, shrinking $d/2$. Nested between a convolutional layer and (dynamic) k-max pooling, a folding layer halves the respective matrix.

While the model itself can automatically scale with new variants, several hyper-parameters may need to be adjusted to better suit the sample pool. Manual tuning is an inefficient, costly process that frequently offers no advantage. Bergstra and Bengio [34] empirically show that randomly chosen trials are more efficient for hyper-parameter optimization than manual or grid search over the same domain—calculating the most viable constants in a fraction of the time. Taking the shuffled Cartesian product of all hyper-parameter subsets allows the ambiguated selection of distinct constants for trial on a small number of epochs.

3.6. Experimentation

Table 1 presents a range of popular algorithms which were selected for experimental analysis as they are widely employed for both legitimate and fraudulent purposes. CryptoKnight was configured to build two feature maps from its first convolution through a round of *k*-max pooling in combination with the Parametric Rectified Linear Unit (PReLU) activation function. Eighteen further hidden one-dimensional wide convolutions interspersed with Rectified Linear Unit (ReLU) non-linear activations and *k*-max pooling further reduced the feature space. Due to the number of embeddings the model used three folding operations, one at the start and two simultaneously prior to the penultimate pooling layer, after its convolution. The final convolution built three additional feature maps and then pooled on the stipulated topmost magnitude of 800. A linear transformation, of output size 200, was then applied with softmax to fully connect the model. Within this, a dropout probability of 0.5 was selected to improve regularization as recommended by Hinton et al. [35]. Filter widths were specified for the first and last convolutional layers of 99 and 358 respectfully, the remaining hidden layers shared a width of 9. In total, the model employed 20 convolutions to efficiently reduce the extensive feature space.

Table 1. Cryptographic Algorithms.

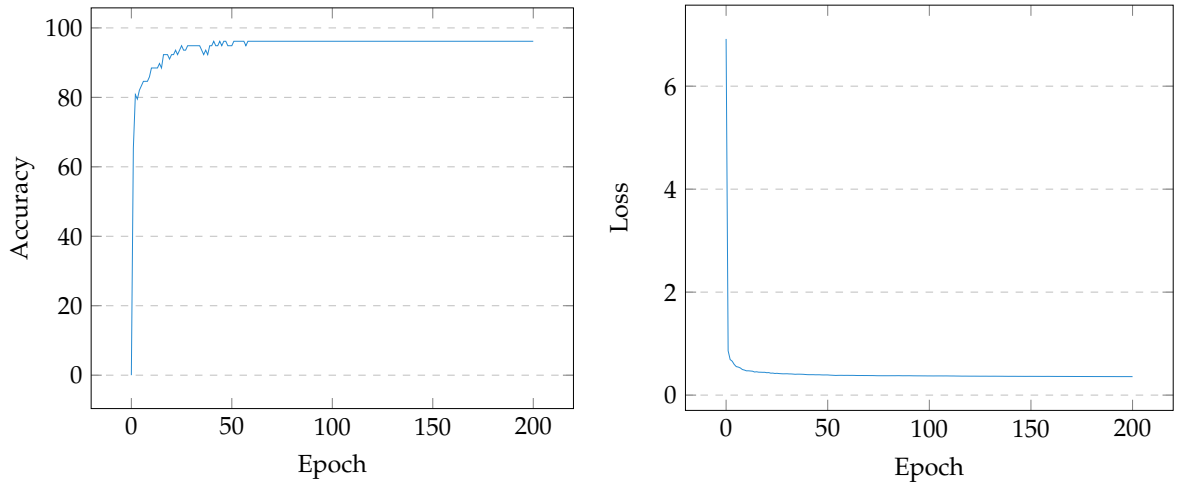
Category	Algorithm
Symmetric	AES RC4 Blowfish
Asymmetric	RSA
Hashing	MD5

Based on a frequency analysis of simplified opcodes, the following operations were selected due to their prevalence in the cryptographic sample pool: *cmp*, *mov*, *test*, *lea*, *and*, *or*, *xor*, *pxor*, *add*, *sub*, *inc*, *dec*, *shr*, *shl*, *sar*, *not*. The final design matrix, of embedding 16, contained a variable number of vectors corresponding to the numeration and associated weightings of each basic block in a subject binary. A distribution of size $n = 750$ was drawn in which $\sim$75% was used for training and $\sim$25% remained for testing. After 200 epochs, the model successfully converged at 96% accuracy with a minimal loss of 0.35. Figure 4a shows the test accuracy over 200 epochs, and Figure 4b displays its simultaneous loss. Two further tests were conducted to trial the model on data without the alternate scaling techniques; in both cases, the performance was significantly poorer (Figure 5) but the effectiveness of entropy scoring is noted.

Table 2 diagnoses the pre-trained model's associated confusion on an additional collection of validation samples where $n = 150$ with an F1 score of 0.96.

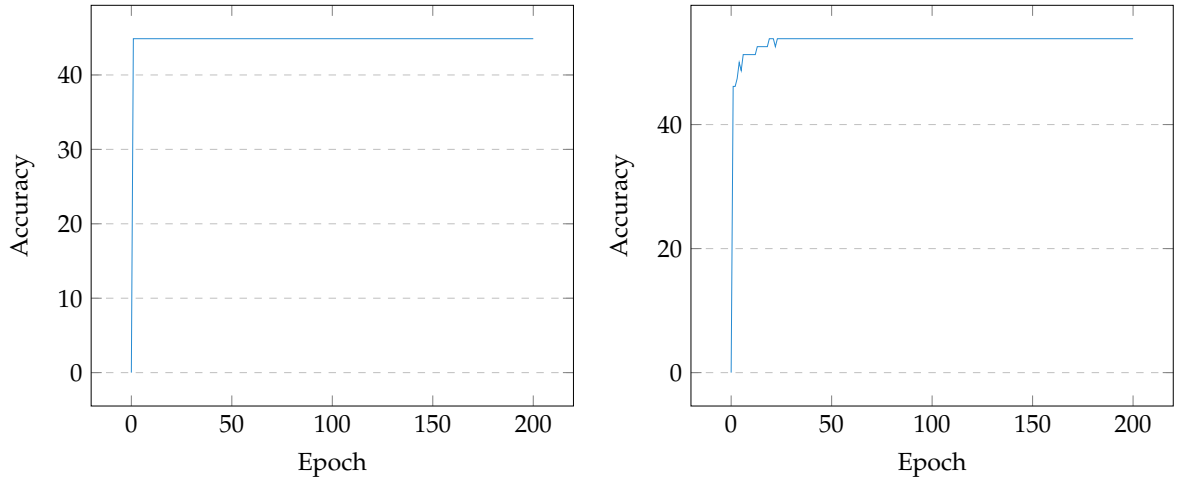
We collected and compiled several open source implementations for the sample pool, leveraging pure (non-library linked) cryptographic functionality to assess our methodology. The classification rate varied with hyper-parameter optimisations and distribution sizes, but was most notably able to classify 4/5 Rivest Cipher 4 (RC4), 4/5 Blowfish and 3/5 MD5 samples. Unfortunately no representational

RSA instances were identified at time of testing and AES was overlooked due to the model having been trained on cryptographically accelerated binaries.



(**a**) Accuracy (**b**) Loss

Figure 4. Training Performance—200 Epochs.



(**a**) No Entropy (**b**) No Loops

Figure 5. Training Performance (Comparison)—200 Epochs. Without scoring the associated entropy, the model immediately converges to ∼44% and clearly under-fits the data. Removing loop occurrence based feature scaling, the model fits at ∼53% which indicates that the entropy metric effectively boosts recognition.

Table 2. Validation Results. $x = predicted, y = label$.

	AES	**RC4**	**BLF**	**MD5**	**RSA**	**R/A**
AES	13	0	0	0	0	0
RC4	0	12	1	0	0	0
BLF	0	0	12	0	0	1
MD5	0	1	0	12	0	0
RSA	0	0	0	0	13	0
R/A	0	0	0	0	0	13

The model's performance in regards to the effectual discrimination of Blowfish was heavily reliant on hyper-parameter optimisations, whereas the other classes generalised easier.

Analysing *GnuPG 1.4.20*, the software was directed to encrypt an empty text document using 256-bit AES. With a trace time of ∼1 min 40 s, CryptoKnight predicted the use of AES and RSA. Finally, a real-world crypto-ransomware sample named 'GonnaCry' [36] was exclusively examined in a Virtual Machine (VM). This is an open-source demonstrative tool for examining the effects of targeted attacks through the recursive encryption of all files using AES and RSA. Although its execution was preemptively halted to limit its effect, CryptoKnight quickly identified the usage of RSA.

4. Evaluation

Traditional cryptographic identification techniques are inherently expensive [8,9,11,14] and heavily rely on human intuition. CryptoKnight was built to reduce this associated error-prone interaction, with refined sampling of the latent feature space, a procedurally synthesised distribution allowed our DCNN to map proportional linear sequences with a finer granularity than that of conventional architectures without overfitting, CryptoKnight converged at 96% accuracy through the optimisation of hyper-parameters based on a grid search. The model ultimately fit the synthetic distribution well, with performance on par to that of Kalchbrenner et al. [33].

Discussion

The impediment of dynamic binary instrumentation was made clear by [9] who highlighted an extensive twenty two hour trace and analysis time. CryptoKnight's analysis time also varied, but not quite to this extent; for the sample binaries, analysis took up to a maximum of around one minute. However, synthesis saw exponential draw times of indefinable length, but since manual analysis often takes invariably longer than an adequate draw time, this footprint is arguably marginal. The entropy metric assumes that a cryptographic function's associated uncertainty is higher than that of conventional interaction. Lutz [8] discovered that this negatively affected the recognition of simple substitution ciphers; however, it did not affect CryptoKnight in the same way due to its scoring mechanics and demonstrably high accuracy in subjective tests without the metric. Figure 5a proves that the entropy metric distinctly benefited the task, improving accuracy far beyond plain loop scaling. Unfortunately, problems with cryptographic acceleration played an important role in the detection of native AES implementations. Intel's AES-NI extension was proposed in 2008 for boosting the relative speed of encryption and decryption on microprocessors from Intel and AMD. Akdemir et al. [37] describe the instruction set with regard to its breakthrough performance increase. As the six instructions, prefixed by AES, directly perform each of the cipher's operations on the Streaming SIMD Extensions (SSE) XMM registers, the natural progression of the cipher could not be fully observed, thus warranting further investigation.

Once trained, the proposed framework would be most beneficial as part of an analyst's toolkit—to quickly verify known cryptographic instances. The DCNN was intended to map time-invariant cryptographic execution despite control or data flow obfuscation; an intrinsic problem of prior literature—Section 2. While successful, this formulation still has a few preliminary issues to address. For instance, the predefined operator embeddings explicitly define the entire feature set, therefore new samples which perhaps deviate from traditional operation may be unidentifiable. Should the framework not immediately generalise to an additional algorithm, correlating its most predominant operators by interchanging embeddings or enlarging the scope should enhance cognition. Additional signatures from other cryptographic libraries (such as Libgcrypt or PolarSSL) would further benefit generalisation. Another fundamental part of CryptoKnight's supervised design is that it can only classify known samples; new cryptographic algorithms must be added to the generation pool, a process this framework has strived to simplify, but an unavoidable limitation of the proposed architecture nonetheless. This also makes custom cryptography more difficult to classify, as no high-level reference implementations would feasibly exist to import. Integrating an unsupervised component into the core model could facilitate the detection of non-cryptographic signatures, or alternatively advanced synthesis could negate the need for procedural generation entirely, to

further reduce the presently expensive time requirement. There are several additional optimizations to improve the existing framework's efficiency however; the new layers [33] would benefit from Graphics Processing Unit (GPU) acceleration through integrated *C* binaries, distribution synthesis could be multi-threaded to decrease draw time, and a multi-class element to learn application invariant primitives would remove the need to pre-combine functions. A similar model could also be re-purposed to decompile binaries with more accuracy than traditional methods—which typically only manage simplistic control flow dissections. Other work could include the detection of cryptographic functions in parallel with the execution of a subject binary, perhaps using a Recurrent Neural Network (RNN). In any case, the full code base has been published on GitHub for future researchers to use and improve.

5. Conclusions

Despite advanced countermeasures, the cryptovirological threat has significantly increased over the last decade. To aid malware analysis, our research has demonstrated that cryptographic primitive classification in compiled binary executables can be successfully achieved using a DCNN with ∼96% accuracy. Moreover, our implementation is fundamentally more flexible than that of previous work, marginalising the error prone human element with automated distribution synthesis, training and optimisation. The framework successfully detected many externally sourced non-library applications and maintained a distinctively high accuracy on our synthetic samples. When trialled on *GnuPG* and a real-world ransomware binary, CryptoKnight successfully distinguished the use of AES and RSA almost instantaneously.

Author Contributions: Investigation, G.H.; Supervision, X.B.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Young, A.; Yung, M. Cryptovirology: Extortion-based security threats and countermeasures. In Proceedings of the 1996 IEEE Symposium on Security and Privacy, Oakland, CA, USA, 6–8 May 1996; pp. 129–140.
2. Snow, J. CryptXXX Ransomware, 2016. Available online: https://blog.kaspersky.com/cryptxxx-ransomware/11939/ (accessed on 10 September 2018).
3. Chiu, A. *Player 3 Has Entered the Game: Say Hello to 'WannaCry'*. Available online: https://www.cybrary.it/channelcontent/player-3-has-entered-the-game-say-hello-to-wannacry/ (accessed on 7 September 2018).
4. Kharraz, A.; Robertson, W.; Balzarotti, D.; Bilge, L.; Kirda, E. Cutting the gordian knot: A look under the hood of ransomware attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*; Springer: Berlin, Germany, 2015; pp. 3–24.
5. Scaife, N.; Carter, H.; Traynor, P.; Butler, K.R. Cryptolock (and drop it): Stopping ransomware attacks on user data. In Proceedings of the 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), Nara, Japan, 27–30 June 2016; pp. 303–312.
6. Deane-McKenna, C. *NHS Ransomware Cyber-Attack was Preventable*; The Conversation, 13 May 2017. Available online: https://theconversation.com/nhs-ransomware-cyber-attack-was-preventable-77674 (accessed on 10 September 2018).
7. Beek, C. *McAfee Labs Threats Report*; Intel Security: Santa Clara, CA, USA, 2016.
8. Lutz, N. Towards Revealing Attackers' Intent by Automatically Decrypting Network Traffic. Master's Thesis, ETH Zürich, Zürich, Switzerland, August 2008. (A joint project between the ETH Zurich and Google, Inc.)
9. Gröbert, F.; Willems, C.; Holz, T. Automated Identification of Cryptographic Primitives in Binary Programs. In Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID 2011), Menlo Park, CA, USA, 20–21 September 2011; Springer: Berlin/Heidelberg, Germany, 2011; pp. 41–60.
10. IBM. Bucbi Ransomware. Available online: https://exchange.xforce.ibmcloud.com/collection/Bucbi-Ransomware-16eef23d3b7ea484ed69ecd78b6c1232 (accessed on 7 September 2018).

11. Lestringant, P.; Guihéry, F.; Fouque, P.A. Automated Identification of Cryptographic Primitives in Binary Code with Data Flow Graph Isomorphism. In Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, Singapore, 14–17 April 2015; ACM: New York, NY, USA, 2015; pp. 203–214.
12. Moser, A.; Kruegel, C.; Kirda, E. Limits of static analysis for malware detection. In Proceedings of the Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), Miami Beach, FL, USA, 10–14 December 2007; pp. 421–430.
13. Luk, C.K.; Cohn, R.; Muth, R.; Patil, H.; Klauser, A.; Lowney, G.; Wallace, S.; Reddi, V.J.; Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, Chicago, IL, USA, 12–15 June 2005; ACM: New York, NY, USA, 2005; Volume 40, pp. 190–200.
14. Xu, D.; Ming, J.; Wu, D. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2017; pp. 921–937.
15. Li, X.; Wang, X.; Chang, W. CipherXRay: Exposing cryptographic operations and transient secrets from monitored binary execution. *IEEE Trans. Dependable Secur. Comput.* **2014**, *11*, 101–114. [CrossRef]
16. LeCun, Y.; Bengio, Y.; Hinton, G. Deep learning. *Nature* **2015**, *521*, 436–444. [CrossRef] [PubMed]
17. LeCun, Y. Generalization and network design strategies. In *Connectionism in Perspective*; Elsevier: New York, NY, USA, 1989; pp. 143–155.
18. LeCun, Y.; Kavukcuoglu, K.; Farabet, C. Convolutional networks and applications in vision. In Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS), Paris, France, 30 May–2 June 2010; pp. 253–256.
19. Hu, B.; Lu, Z.; Li, H.; Chen, Q. Convolutional neural network architectures for matching natural language sentences. In *Advances in Neural Information Processing Systems*; Curran Associates, Inc.: Red Hook, NY, USA, 2014; pp. 2042–2050.
20. Pearl, J. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*; U.S. Department of Energy: Washington, DC, USA, 1984.
21. Gröbert, F. Automatic Identification of Cryptographic Primitives in Software. In Proceedings of the 27th Chaos Communication Congress, Berlin, Germany, 27–30 December 2010.
22. Matenaar, F.; Wichmann, A.; Leder, F.; Gerhards-Padilla, E. CIS: The Crypto Intelligence System for Automatic Detection and Localization of Cryptographic Functions in Current Malware. In Proceedings of the 2012 7th International Conference on Malicious and Unwanted Software, Fajardo, PR, USA, 16–18 October 2012.
23. Zhang, P.; Wu, J.; Wang, X.; Wu, Z. Decrypted data detection algorithm based on dynamic dataflow analysis. In Proceedings of the 2014 International Conference on Computer, Information and Telecommunication Systems (CITS), Jeju, Korea, 7–9 July 2014; pp. 1–4.
24. Zhao, R.; Gu, D.; Li, J.; Yu, R. Detection and Analysis of Cryptographic Data Inside Software. In Proceedings of the 14th International Conference on Information Security, Xi'an, China, 26–29 October 2011; Lai, X., Zhou, J., Li, H., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 182–196.
25. Calvet, J.; Fernandez, J.M.; Marion, J.Y. Aligot: Cryptographic function identification in obfuscated binary programs. In Proceedings of the 2012 ACM Conference on Computer and Communications Security, Raleigh, NC, USA, 16–18 October 2012; ACM: New York, NY, USA, 2012; pp. 169–182.
26. Hosfelt, D.D. Automated detection and classification of cryptographic algorithms in binary programs through machine learning. *arXiv* **2015**, arXiv:1503.01186.
27. Baldwin, J.; Dehghantanha, A. Leveraging support vector machine for opcode density based detection of crypto-ransomware. In *Cyber Threat Intelligence*; Springer: Berlin, Germany, 2018; pp. 107–136.
28. Sgandurra, D.; Muñoz-González, L.; Mohsen, R.; Lupu, E.C. Automated Dynamic Analysis of Ransomware: Benefits, Limitations and use for Detection. *arXiv* **2016**, arXiv:1609.03020.
29. Drape, S. Intellectual Property Protection using Obfuscation. In Proceedings of the 2009 IEEE Sensors Applications Symposium, New Orleans, LA, USA, 17–19 February 2009.
30. Tubella, J.; Gonzalez, A. Control speculation in multithreaded processors through dynamic loop detection. In Proceedings of the 1998 Fourth International Symposium on High-Performance Computer Architecture, Las Vegas, NV, USA, 1–4 February 1998; pp. 14–23.

31. Moseley, T.; Grunwald, D.; Connors, D.A.; Ramanujam, R.; Tovinkere, V.; Peri, R. Loopprof: Dynamic techniques for loop detection and profiling. In Proceedings of the 2006 Workshop on Binary Instrumentation and Applications (WBIA), San Jose, CA, USA, 21–25 October 2006.
32. Rényi, A. On measures of entropy and information. In Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability, Berkeley, CA, USA, 20 June–30 July 1961; pp. 547–561.
33. Kalchbrenner, N.; Grefenstette, E.; Blunsom, P. A Convolutional Neural Network for Modelling Sentences. *arXiv* **2014**, arXiv:1404.2188.
34. Bergstra, J.; Bengio, Y. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.* **2012**, *13*, 281–305.
35. Hinton, G.E.; Srivastava, N.; Krizhevsky, A.; Sutskever, I.; Salakhutdinov, R.R. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv* **2012**, arXiv:1207.0580.
36. Marinho, T. GonnaCry. 2018. Available online: https://github.com/tarcisio-marinho/GonnaCry (accessed on 7 September 2018).
37. Akdemir, K.; Dixon, M.; Feghali, W.; Fay, P.; Gopal, V.; Guilford, J.; Ozturk, E.; Wolrich, G.; Zohar, R. *Breakthrough AES Performance with Intel AES New Instructions*; White Paper; Intel Corporatlon: Santa Clara, CA, USA, June 2010.