Repurposing Neural Networks for Efficient Cryptographic Computation

Xin Jin
The Ohio State University
jin.967@osu.edu

Shiqing Ma
University of Massachusetts Amherst
shiqingma@cs.umass.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

***Abstract*—While neural networks (NNs) are traditionally associated with tasks such as image recognition and natural language processing, this paper presents a novel application of NNs for efficient cryptographic computations. Leveraging the Turing completeness and inherent adaptability of NN models, we propose a transformative approach that efficiently accelerates cryptographic computations on various platforms. More specifically, with a program translation framework that converts traditional cryptographic algorithms into NN models, our proof-of-concept implementations in TensorFlow demonstrate substantial performance improvements: encryption speeds for AES, Chacha20, and Salsa20 show increases of up to 4.09×, 5.44×, and 5.06×, respectively, compared to existing GPU-based cryptographic solutions written by human experts. These enhancements are achieved without compromising the security of the original cryptographic algorithms, ensuring that our neural network-based approach maintains robust security standards. This repurposing of NNs opens new pathways for the development of scalable, efficient, and secure cryptographic systems that can adapt to the evolving demands of modern computing environments.**

I. Introduction

Modern technical innovations, e.g., cloud computing, autodriving, and virtual reality, generate massive private user data daily. For instance, over 90% of data in the world was generated in the past two years [1]. With such massive data, efficient cryptographic computations are more important than ever for various tasks, including cloud storage with cryptographic file system [2], 3D data protection [3], and data services for streaming, bank and other financial institutions [4]. Zoom, for instance, provides end-to-end encryption to safeguard meeting conversations by utilizing AES and TLS [5]. The encryption and decryption of Zoom meetings with 50 attendees entail of 4MB data, i.e., 262,144 AES blocks, per second [6]. However, traditional cryptographic implementations are known to be slow [7], which cannot match this emerging need for efficient cryptographic computations.

Deep Neural Network (DNN) is a key enabler for a lot of Artificial Intelligence (AI) tasks. Numerous research efforts [8] have been trying to accelerate the computation of DNNs, from architecture designs to engineering better hardware and software stacks. Prior work has demonstrated that DNNs, e.g., RNN — Recurrent Neural Network, can be Turing-complete [9, 10], which potentially enable other applications to leverage the recent advances in modern DNN acceleration. Although this has been known for years, limited exploration has been done to explore diverse applications. In this paper, we explore the possibility of accelerating cryptographic algorithm computation by repurposing DNNs, and demonstrate the superiority of this approach compared with existing state-of-the-art. To assist the transition, we also introduce a novel approach to automatically translate existing cryptographic algorithm implementations to NN implementations.

Besides the Turing-complete capability of DNNs, the potential of repurposing NN is further bolstered by a vibrant ecosystem of AI frameworks that can accelerate computing with better resource (e.g., memory) scheduling and management, nurtured collaboratively by both software and hardware communities. This approach streamlines programming across various hardware accelerators with popular and easy-to-use development tool sets. Recent years have witnessed a proliferation of hardware accelerators integrated into various platforms, such as Google Tensor Processing Units (TPUs) [11] on cloud servers and Apple A16 [12] on mobile devices. In tandem, providers of deep learning frameworks, including Google's TensorFlow [13] and Meta's PyTorch [14], have introduced user-friendly interfaces applicable to an array of hardware and platforms [13]. Additionally, established AI compiling frameworks like TVM [15] and XLA [16] support diverse hardware and software platforms, performing software optimizations. This collective progress empowers us to seamlessly program a wide range of hardware accelerators without being encumbered by engineering details.

Our system, TensorCrypt, leverages DL frameworks as out-of-the-box computing acceleration tools for more efficient cryptographic computation. It initiates the process by abstracting two distinct domain-specific languages (DSLs): one dedicated to cryptography and the other tailored for NN. These DSLs serve as representations for the transformation source and target, respectively. Leveraging these DSLs, we introduce a program transformation framework that is capable of converting programs written in the cryptographic DSL (CRYPT) into equivalent programs in the neural network (NN) DSL. This transformation is designed to uphold trace and bisimulation equivalence between the two programs. In simpler

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240955
www.ndss-symposium.org

terms, for a given input, both the overall program output and the internal states (such as internal memory values) of these programs remain identical. This critical property ensures that the transformed program does not reveal any new information to potential analyzers, even if an adversary were to observe the internal program execution. Consequently, the security level achieved by the transformed program matches that of the original. Following the transformation, we embark on further optimization of the NN. This process commences with the implementation of operator fusion, involving the definition of a set of fusion rules based on operator types. Simultaneously, we enhance the data transmission process through the adoption of load-on-use memory management. This combined approach empowers our NN-based cryptographic algorithms to operate with remarkable efficiency, particularly when complemented by hardware accelerators and optimization techniques. Significantly, these algorithms are designed for broad deployability across diverse hardware and software systems.

Our TENSORCRYPT prototype has undergone rigorous testing across AES, Chacha20, and Salsa20 implementations. Notably, AES and Chacha20 serve as the recommended ciphers for encrypting large-volume data in TLS 1.3 [17, 18]. The experimental findings demonstrate the superior efficiency of TENSORCRYPT models when compared to baseline approaches. Specifically, the TENSORCRYPT AES, Chacha20, and Salsa20 models exhibit remarkable speed advantages, achieving up to 4.09×, 5.04×, and 5.44× faster execution rates, respectively, compared to their baseline counterparts. Furthermore, the versatility of TENSORCRYPT models is a striking feature. They demonstrate compatibility with a wide array of hardware and software stacks. This encompasses diverse hardware configurations, ranging from mobile and IoT devices like the Google Pixel XL and Raspberry Pi 3, to hardware accelerators including Nvidia GPUs and Google TPUs. Likewise, the software landscape covered multiple operating systems such as Linux, Windows, and MacOS, and deployment languages spanning Java and Python. The integration of proposed optimizations significantly enhances the efficiency of TENSORCRYPT models. Operator fusion, for instance, leads to a reduction in latency of up to 79.79×, while load-on-use memory management accelerates encryption speeds by up to 92.5×. These optimizations further underline the substantial gains in performance achieved through our approach.

In summary, this paper introduces an innovative application of NNs to accelerate cryptographic algorithms, repurposing neural computation, employing a combination of program transformation and optimization techniques. While the challenge of cryptographic algorithm acceleration is well-established, we highlight the limitations of existing approaches that predominantly lean on hardware accelerators. These solutions often exhibit hardware specificity, achieving only suboptimal performance despite substantial engineering investments. The unique strength of our approach lies in harnessing both hardware and software capabilities, fully maximizing the potential of accelerators while significantly reducing engineering complexities. This novel approach leads to demonstrably superior empirical results in terms of efficiency, complemented by an intuitive and user-friendly programming interface. Additionally, we introduce domain-specific languages, novel program transformation methods, and domain-specific optimizations, enhancing the overall comprehensiveness of our solution. By tackling this longstanding problem, we establish a new paradigm for accelerating cryptographic algorithms.

Scope. This paper focuses on symmetric ciphers using parallel modes like ECB/CTR of AES, which benefit from GPU acceleration. We do not anticipate TENSORCRYPT's use in sequential modes like CBC whose performance bottleneck is caused by dependencies. Currently, as a prototype, TENSORCRYPT transformation does not cover the broader class of ciphers, e.g., ECC and post-quantum cryptography [19], or general-purpose computations.

II. BACKGROUND AND RELATED WORKS

Cryptographic Implementation Acceleration. Recently, secure transmission of large-scale data has become increasingly important and has gained significant attention [2, 20]. In the esteemed protocol TLS 1.3, ChaCha20 and Advanced Encryption Standard (AES) algorithms are recommended for large-scale data encryption and decryption [18]. AES is a widely used block cipher with parallelizable modes of operation [21]. Chacha20 and its variant, Salsa20, are stream ciphers, specifically designed for secure and efficient encryption of large-scale data [22, 23]. In response to the growing need for accelerated cryptographic implementations, numerous solutions have been proposed for different hardware [24, 25]. For instance, Intel introduces AES-NI to accelerate AES on Intel processors [24].

Moreover, there has been a substantial research focus on GPU-based cryptographic implementations [2, 18, 20, 26–31] due to the prevalence of graphic cards. Notably, these GPU-based implementations have shown superior performance. For example, GPU-based solutions are up to 13.60 times faster than AES-NI and FPGA-based implementations [20, 28]. Apart from AES, GPU-based acceleration solutions are also introduced to Salsa20 [31] and Chacha20 [18, 27]. In addition to pioneering new designs, researchers have also utilized these efficient cryptographic implementations in various applications, such as cryptographic file systems [2], random number generation [32], and software memory encryption [27].

Deep Neural Network. As of now, the most powerful AI models are in the deep neural network (DNN) architecture, which simulates the layered structure of human brains, and each layer performs certain operations that benefit the final task [33]. Recently, it has achieved wide-range success in modeling complicated data, e.g., images, natural languages, and code [33–36]. The success of DNNs is largely attributed to their ability to automatically engineer features. Such great power comes with a price which is the black box nature of DNNs [37]. Specifically, DNNs function as end-to-end systems but it's hard to understand their internal components (e.g., weights and bias). Despite recent efforts in Explainable AI (XAI) [38], interpreting general DNNs remains challenging and unresolved.

From the computational perspective, recurrent neural networks (RNNs) have been proven to be Turing-complete [9]. Recent work has also designed other types of DNNs that are Turing-complete [10]. This property indicates that DNNs can theoretically simulate any procedures. For example, an RNN iterates over computational cells to process data, and each cell consists of pre-defined operations. Some operations maintain the system state (i.e., context information) while others evaluate the input [39]. Specifically, unlike common learning-based DNNs that derive weights from data, non-statistical DNNs can be constructed without relying on data-driven pattern mining [38, 40]. For instance, DeepMind's Tracer translates human-readable programs into weights of transformer models, enhancing their interpretability [38]. Similarly, in this paper, we build TENSORCRYPT models in a non-statistical manner.

III. Design

In this section, we present the design of our transformation framework. We first present our threat model in §III-A. In §III-B, we introduce two domain-specific languages (DSLs) for cryptography and neural networks (NNs), then define transformation rules, and finally provide an example of this transformation process by demonstrating the application of TENSORCRYPT to AES. In §III-C, we introduce optimizations of TENSORCRYPT transformed NN models, i.e., operator fusion and load-on-use memory management. Finally, we provide an analysis of the transformation, rigorously proving its property of trace and bisimulation equivalence, in §III-D.

A. Threat Model

In line with the conventional threat model employed in cryptographic analysis, the primary objective of the adversary is to ascertain the secret key [41, 42]. The distinctive feature of our system, as opposed to existing ones, lies in the implementation of the same algorithm. Consequently, we operate under the assumption that the adversary possesses the capability to observe the system, akin to existing implementations. Our security objective is to ensure that our implementation does not divulge any more information than its existing counterparts.

B. Program Transformation

1) Domain Specific Languages: Cryptographic algorithms are mathematical operations that take byte/integer sequences, e.g., plaintext or ciphertext, as input and convert them into different representations. These algorithms are composed of arithmetic operations (e.g., addition), bit operations (e.g., shift) and array operations (e.g., slice). We formalize a simplified language representation, denoted as CRYPT DSL in Fig. 1, which contains all these operations as well as control flow statements (e.g., conditional statements). In this language, a program processes the input by a sequence of statements and generates ciphertext or plaintext as encryption or decryption results.

Unlike cryptographic programs, neural networks (NNs) are usually represented as computational graphs [43]. In the bottom-up view, the computational graph nodes are usually defined as *operators* (or *operations*), e.g., **add** operator

```
⟨program⟩          P ::= input(arg); s*; output(v)
⟨statement⟩        s ::= x := e | x := y ⟨op⟩ z
                     | if (v) : s1* else : s2*
                     | while (x>0) : s*
⟨expr⟩             e ::= v | x | x[v] | x[vi:vj]
⟨operator⟩        op ::= + | − | ∗ | / | XOR | ...
⟨⟩                  v ∈ Value
⟨⟩              x,y,z ∈ Identifier;
```

Fig. 1: Language Abstraction of CRYPT DSL

```
⟨NN⟩               M ::= input(arg); l*; output(v)
⟨layer⟩            l ::= x := add(·) | sub(·) | mul(·)
                     | bitShift(·) | bitAnd(·)
                     | slice(·) | lookup(·)
                     | cond(v, l1*, l2*) | loop(x>0, l*)
⟨layer-add⟩  add(v) ::= (+ v)
⟨layer-loop⟩ loop(x>0, l*) ::= (l[0] ∘ l[1] ∘ ...)
⟨⟩             arg, x, v ∈ Tensor;
```

Fig. 2: Language Abstraction of NN DSL

can sum two tensors together. The same operators are then aggregated as layers. That is, a neural network *layer* is a collection of operators that perform a specific functionality together. In NN models, the first and last layers are **input** and **output** layers. And the hidden layers take inputs from previous layers and produce outputs to following layers.

In Fig. 2, we define a neural network as a sequence of connected layers, starting from the input layer to the output layer. Each layer essentially is a function, performing arithmetic operations (e.g., **add**), tensor manipulations (e.g., **slice**), bitwise operations[1] (e.g., **bitShift**), etc. In Fig. 2, we list two special layers. The **add** layer performs value addition operations. The **loop** layer creates a sub-computational graph, representing a loop. Other special layers can be layers for random or pseudo-random number generation, which is useful in cryptographic applications. In most deep learning frameworks, a layer can be customized by defining a lambda function, which makes TENSORCRYPT extendable and generalizable to new layers that are not defined in Fig. 2. For example, a **lambda** layer [13] can be customized for various functionalities using Python lambda functions or any user-defined functions. AI frameworks will also automatically handle data transmission between layers, and we omit such details in our discussion.

Additionally, we define the semantics of CRYPT and NN DSLs in Fig. 3. These DSLs are subsets of existing programming languages (e.g., C and Python), extendable to other languages.

2) Transformation Rules: Fig. 4 presents the rules for transforming programs in CRYPT DSL into NN DSL, including: **Statement transformation rules** — transforming program statements into NN layers as follows.

- **[T-LOOKUP]**. In cryptography algorithms, table lookup operations are very important to perform nonlinear mapping, e.g., element lookup. These statements can be defined as

[1]Bitwise NNs are popular with better robustness, interpretability, and efficiency [44]. Therefore, bitwise operations are supported in major DNN frameworks, e.g., Tensorflow [13], ONNX [45] and Pytorch [46].

Definitions: $\sigma, \phi \in Store: Variable \rightarrow Value$; σ for cryptography (CRYPT) DSL, ϕ for neural network (NN) DSL.

Evaluation context rules:

$$E_s ::= E_s;s \mid [\cdot]_s \mid x{:=}[\cdot]_e \mid x{:=}[[\cdot]_e] \mid x{:=}[[\cdot]_e : e] \mid x{:=}[v : [\cdot]_e] \mid x{:=}[\cdot]_e \ \langle op\rangle \ e$$
$$\mid x{:=}v \ \langle op\rangle \ [\cdot]_e \mid \texttt{if } [\cdot]_e : s_1 \texttt{ else } : s_2 \mid \texttt{while } [\cdot]_e : s^*,$$
$$E_l ::= E_l;l \mid [\cdot]_l \mid x{:=}[\cdot]_e \mid x{:=}[[\cdot]_e] \mid x{:=}[[\cdot]_e{:}e] \ x{:=}[e{:}[\cdot]_e] \mid x{:=}l([\cdot]_e, e) \mid x{:=}l(v, [\cdot]_e)$$

Expression Rule: $\sigma : e \xrightarrow{e} v$ [E-CRYPT] $\phi : e \xrightarrow{e} v$ [E-NN]

$\sigma : v \xrightarrow{e} v$ [E-CONST-CRYPT] $\phi : v \xrightarrow{e} v$ [E-CONST-NN]

$\sigma : x \xrightarrow{e} \sigma(x)$ [E-VAR-CRYPT] $\phi : x \xrightarrow{e} \phi(x)$ [E-VAR-NN]

Statement/Layer Rule: $\sigma, s \xrightarrow{s} \sigma', s'$ [STMT-CRYPT] $\phi, l \xrightarrow{l} \phi', l'$ [STMT-NN]

$\sigma : x := y\,[v] \xrightarrow{s} \sigma[x \rightarrow \sigma(y[v])]$, skip [LOOKUP-CRYPT] $\phi : x := \textbf{lookup}(y, v) \xrightarrow{l} \phi[x \rightarrow \phi(y[v])]$, skip [LOOKUP-NN]

$\sigma : x := y\,[v_i : v_j] \xrightarrow{s} \sigma[x \rightarrow [\sigma(y[v_i]), ..., \sigma(y[v_j])]]$, skip [SLICE-CRYPT] $\phi : x := \textbf{slice}(y, [v_i : v_j]) \xrightarrow{l} \phi[x \rightarrow [\phi(y[v_i]), ..., \phi(y[v_j])]]$, skip [SLICE-NN]

$\sigma : x := y + z \xrightarrow{s} \sigma[x \rightarrow \sigma(y) + \sigma(z)]$, skip [OP-ADD-CRYPT] $\phi : x := \textbf{add}(y, z) \xrightarrow{l} \phi[x \rightarrow \phi(y) + \phi(z)]$, skip [OP-ADD-NN]

$\sigma : \texttt{if}(v) : s_1 : \texttt{else} : s_2 \xrightarrow{s} \sigma, s_1$, if $v = true$ [IF-T-CRYPT] $\phi : \textbf{cond}(v, l_1, l_2) \xrightarrow{l} \phi, l_1$, if $v = true$ [IF-T-NN]

$\sigma : \texttt{while}\ (x > 0) : s_x^* \xrightarrow{s} s_x^*, s$, if $x > 0$ [LOOP-E-T-CRYPT] $\phi : \textbf{loop}(x > 0, l_x^*) \xrightarrow{l} l_x^*, s$, if $x > 0$ [LOOP-T-NN]

Global Rules:

$$\frac{\sigma : e \xrightarrow{e} v}{\sigma, E[e]_e \rightarrow \sigma, E[v]_e} \text{ [G-EXPR-CRYPT]} \qquad \frac{\phi : e \xrightarrow{l} v}{\phi, E[e]_e \rightarrow \phi, E[v]_e} \text{ [G-EXPR-NN]}$$

$$\frac{\sigma : s \xrightarrow{s} \sigma', s'}{\sigma, E[s]_s \rightarrow \sigma', E[s']_s} \text{ [G-STMT-CRYPT]} \qquad \frac{\phi : l \xrightarrow{l} \phi', l'}{\phi, E[l]_l \rightarrow \phi', E[l']_l} \text{ [G-STMT-NN]}$$

Fig. 3: Semantics of CRYPT DSL and NN DSL

Statement Transformation: $s \xrightarrow{ctx} l$

$x := y\,[v] \xrightarrow{ctx} \boxed{x := \textbf{lookup}(y, v)}$ [T-LOOKUP] $x := y\,[v_i : v_j] \xrightarrow{ctx} \boxed{x := \textbf{slice}(y, [v_i : v_j])}$ [T-SLICE]

$x := y + z \xrightarrow{ctx} \boxed{x := \textbf{add}(y, z)}$ [T-OP-ADD] $x := y\,\hat{}\,z \xrightarrow{ctx} \boxed{x := \textbf{bitXOR}(y, z)}$ [T-OP-XOR]

$$\frac{s_1 \xrightarrow{ctx} l_1 \quad s_2 \xrightarrow{ctx} l_2}{\texttt{if}(v) : s_1^* \texttt{ else } : s_2^* \xrightarrow{ctx} \boxed{\textbf{cond}(v, l_1^*, l_2^*)}} \text{ [T-IF]} \qquad \frac{s_x \xrightarrow{ctx} l_x}{\texttt{while}(x > 0) : s_x^* \xrightarrow{ctx} \boxed{\textbf{loop}(x > 0, l_x^*)}} \text{ [T-LOOP]}$$

Program Transformation: $P \xrightarrow{ctx} M$

$$\frac{s \xrightarrow{ctx} l}{\texttt{input}(arg);\ s^*;\ \texttt{output}(v) \xrightarrow{ctx} \boxed{\texttt{input}(arg); l^*; \texttt{output}(v)}} \text{ [T-INPUT-ASGN]}$$

Fig. 4: Transformation Rules. We skip listing some rules due to space limitations, e.g., [T-OP-SUB] for substitution operations.

$x := y\,[v]$, where v-th element of y is looked up. This rule transforms lookup statements into $x = \textbf{lookup}(y, v)$ layers.

- **[T-SLICE]**. The slice statement is defined in many languages to obtain elements from the start index to the end index from arrays. It can be formally defined as $x := y\,[v_i : v_j]$, where y is sliced from v_i to v_j With this rule, we transform this statement into the **slice**(y, $[v_i : v_j]$) layer.
- **[T-OP]**. Arithmetic statements and bitwise operations are fundamental building blocks for cryptographic algorithms. To incorporate these computations into TENSORCRYPT models, we propose this rule for transforming arithmetic and bitwise statements. Moreover, we define a set of sub-rules for basic operations, e.g., [T-OP-ADD] and [T-OP-XOR], for addition and bit XOR operations.
- **[T-IF]**. We formalize the basic control flow statement `if-else` as **if**(v) : s_1^* **else** : $s_2^{*\prime}$. Accordingly, this rule transforms `if-else` statements into **cond**(v, l_1^*, $l_2^{*\prime}$) layers. The execution branches (l_1^* and l_2^*) are determined by the condition tensor v.
- **[T-LOOP]**. The loop statements are commonly utilized to iterate through encryption or decryption rounds. In this rule, $(x > 0)$, x, and s_x^* are the loop condition, iterator, and body. The loop condition can be expressed in different ways, e.g., (x in [1, N]). This statement repeatedly executes the loop body s_x^* when its condition holds true. [T-LOOP] transforms this statement to a **loop**($x > 0$, l_x^*) layer.

Program transformation rules transform the whole programs into NN models. In [T-INPUT-ASGN], a program P can be formalized as **input**(arg); s^*; **output**(v), where P sequentially executes a list of statements s^* based on input arguments arg to produce output v. We define an NN model M as **input**(arg); l^*; **output**(v), where arg, l^*, and v are input tensors, hidden layers, and output tensors. With TENSORCRYPT, each statement of s^* is transformed into one layer of l^*.

```
# encrypt function definition
input(msg, key, Nr);
# AddRoundKey: expanded key slicing and XOR bytes
k = key[0:15];
# byte-wise XOR of k and msg
msg = [k[0] ^ msg[0], …, k[15] ^ msg[15]];
# iterate Nr-1 rounds
for r in [1, 2, …, Nr-1]:
   # SubBytes: S-box lookup
   msg = Sbox[msg[0], …, msg[15]];
   # ShiftRows: row index array lookup
   msg = msg[0, 5, 10, 15, …, 12, 1, 6, 11];
   # MixColumn: gf_mul table lookup and XOR
   # msg[0] multiplies with 02 in GF(2^8)
   mul0_2 = gf_mul2[msg[0]];
   # msg[1] multiplies with 03 in GF(2^8)
   mul1_3 = gf_mul3[msg[1]];
   xor2_3 = msg[2] ^ msg[3];
   msg[0] = mul0_2 ^ mul1_3 ^ xor2_3;
   … # similar steps for msg[1, 2, 3] and other columns
   … # skip similar steps for AddRoundKey

# last round: SubBytes, ShiftRows, and AddRoundKey
… # skip similar steps
output(msg);
```

(a) AES Implementation in CRYPT DSL

```
# encrypt model transformed                                        ①
input(msg, key, Nr);
# AddRoundKey: expanded key slicing and XOR bytes
k = slice(key, [0:15]);                                            ②
# byte-wise XOR of k and msg
msg = bitXOR(k, msg);
# iterate Nr-1 rounds                                              ③
loop(r = [1, 2, …, Nr-1], [
   # SubBytes: S-box lookup                                        ④
   msg = lookup(Sbox, [msg[0], …, msg[15]]);
   # ShiftRows: row index array lookup                             ⑤
   msg = lookup(msg, [0, 5, 10, 15, …, 12, 1, 6, 11]);
   # MixColumn: gf_mul table lookup and XOR
   # msg[0] multiplies with 02 in GF(2^8)
   mul0_2 = lookup(gf_mul2, msg[0]);                               ⑥
   # msg[1] multiplies with 03 in GF(2^8)
   mul1_3 = lookup(gf_mul3, msg[1]);
   xor2_3 = bitXOR(msg[2], msg[3]);
   msg[0] = bitXOR(mul0_2, bitXOR(mul1_3, xor2_3));                ⑦
   … # similar steps for msg[1, 2, 3] and other columns
   … # skip similar steps for AddRoundKey
]);
# last round: SubBytes, ShiftRows, and AddRoundKey                 ⑧
… # skip similar steps
output(msg);                                                       ⑨
```

(b) AES Implementation in NN DSL

Fig. 5: AES Implementations. The code blocks in NN DSL are transformed from those of CRYPT DSL with the same colors.

Note that the design and implementation of DSLs (e.g., Python vs. PyPy) are different and vary in complexity. TENSORCRYPT DSLs are based on existing C++ (for CRYPT DSL) and TensorFlow (for NN DSL), because of their widespread adoption in respective domains in the real world. Additionally, both platforms boast mature ecosystems that support a wide range of hardware platforms with accelerators and software stacks. We prioritized clear and simplified interfaces, aligning with established principles of software engineering [47], to enhance readability and usability. The memory models of our DSLs align with C++ memory model [48] for the CRYPT DSL and a TensorFlow model for the NN DSL as well.

3) A Transformation Example Based on AES: To facilitate readers' comprehension of TENSORCRYPT, we provide a concrete example of how the transformation works using AES. Specifically, Fig. 5a presents an AES program in CRYPT DSL, abstracted from an existing AES implementation. Additionally, Fig. 5b shows a transformed AES model in NN DSL, abstracted from an NN model.

Advanced Encryption Standard. Advanced Encryption Standard (AES) is a symmetric block cipher that maps 16-byte input blocks to 16-byte output blocks through the use of multiple rounds. Each AES round (except for the final round) comprises four fundamental operations.

- `AddRoundKey` adds round-specific keys to the intermediate ciphertext `state` by bitwise XOR operations.
- `SubBytes` performs non-linear substitution by replacing `state` bytes with substitution box (S-box) elements.
- `ShiftRows` shifts rows of `state` by a predefined order.
- `MixColumn` maps each byte of a column of `state` into a new byte by matrix multiplication of `state` column matrices and a MixColumn matrix (MC).

AES Implementation in CRYPT DSL. Fig. 5a shows an AES implementation in CRYPT DSL, where we group the program lines based on their functionalities into blocks. Block ① defines the input of `encrypt` function with three arguments, i.e., a 16-byte plaintext block `msg`, the round `key`, and round number `Nr`. Block ② defines the process of `AddRoundKey`, where the first round-specific key `k` is obtained by slicing `key` and then `k` is XOR'ed with `msg` at the byte level. Block ③ iterates over $(Nr-1)$ encryption rounds with a loop. Block ④ performs the `SubBytes` operation by looking up S-box with `msg` bytes. Next, block ⑤ defines the `ShiftRows` operation by looking up a predefined index array. Block ⑥ and ⑦ show the `MixColumn` operation, where every column of `msg` is operated with four fixed polynomials over $GF(2^8)$ [49]. While the calculations over the 16 bytes are similar, we only list these for the first byte for simplicity. In common practice, multiplication over $GF(2^8)$ can be performed by looking up the pre-calculated tables for efficiency purposes. Therefore, block ⑥ defines the table lookup operations on `gf_mul2` and `gf_mul3` for multiplications with 0x02 and 0x03. Block ⑦ performs XOR operations over the bytes. In each encryption round, the last operation is `AddRoundKey`, which is similar to block ② . The last encryption round of AES is different from the other rounds, e.g., without the `MixColumn` operation. In block ⑧ , we skip such details as the operations have been introduced. Ultimately, block ⑨ specifies the output statement that will produce the ciphertext `msg` as the end result.

Transformed AES Implementation in NN DSL. Fig. 5b presents the AES model transformed from the implementation in Fig. 5a. Since there are mainly four types of statements in Fig. 5a, i.e., slice, XOR, lookup, and loop, we focus on demonstrating transformation on these statements.

- **Slice Statement.** For the slice statement in block ② , we first map variables into identifiers in [T-SLICE]: x = `k` and

y = key. Afterwards, we evaluate slicing indices [0:15] by applying the execution context rule x:=[[·]$_e$: e] and x:=[v : [·]$_e$] to iteratively evaluate the expressions. The same process applies to other slice statements.

- **XOR Statement.** Block ② also declares a XOR statement, in which we have omitted the intermediate steps: msg[0] = k[0] ∧ msg[0], ..., msg[15] = k[15] ∧ msg[15]. To transform this XOR statement, we first need to transform these intermediate steps. Thus, we apply [T-OP-XOR] with mapping variables and identifiers, e.g., msg[0] = bitXOR(k[0], msg[0]). Next, we merge the layers, transformed from intermediate steps, into the bitXOR layer because many deep learning frameworks have element-wise layers, e.g., TensorFlow tf.bitwise.bitwise_xor. Other XOR statements can be transformed by applying [T-OP-XOR].
- **Lookup Statement.** In block ⑤, msg is looked up by a row index array. Similar to XOR statement transformation, we have omitted the intermediate steps msg[0] = Sbox(msg[0]), ..., msg[11] = Sbox(msg[11]) for simplicity. By applying [T-LOOKUP], lookup layers are generated, which are further merged as an element-wise lookup layer, similar to XOR statement transformation. The remaining lookup layers undergo the same transformation process.
- **Loop Statement.** At block ③, there is a for-loop statement. To apply [T-LOOP], we first identify the loop condition (r in [1, 2, .., Nr-1]) and the iterator r. The statements from lines 9 to 21 in the loop body correspond to s_x^*, which can be transformed into layers using the same approaches mentioned earlier. Similarly, the skipped statements are successfully transformed into layers, although their details have been omitted.
- **Program Transformation** To transform the encrypt function into a model, we apply the [T-INPUT-ASGN] rule. In particular, we utilize the mappings arg = (msg, key, Nr) and v = msg to transform the **input** and **output** statements. The function body statements s^* are transformed in a similar way to the transformation of loop body statements.

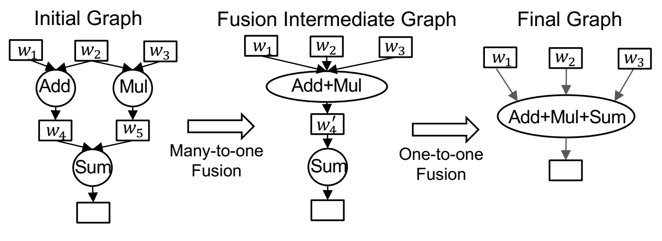
To this end, we successfully transform the AES implementation from CRYPT DSL to NN DSL in Fig. 5. In this figure, we omitted details of SubBytes, ShiftRows, and AddRoundKey operations in the last round for better readability. The omitted code follows the same implementation (but with different indices and variables) with the detailed code of the earlier steps we show, e.g., lines 3-–6 for AddRoundKey and lines 9—10 for SubBytes.

C. TensorCrypt *Model Optimizations*

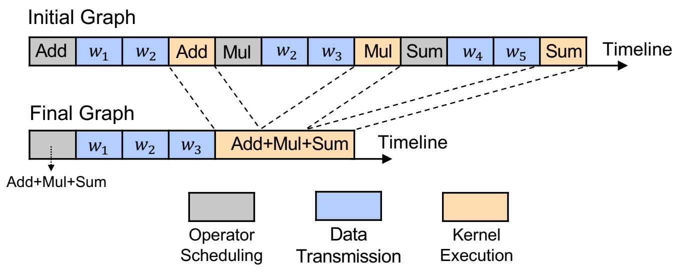
Hardware accelerators are introduced for high-performance computing, but there is a trade-off in using these accelerators. On the one hand, these devices are designed to enhance performance and achieve parallelism for some computations, e.g., floating-point computations. On the other hand, they are *slower* devices, which introduce extra management (e.g., scheduling) and data movement (e.g., through PCIe bus), compared to using main memory. Existing works [7, 50] leveraged the parallelization capabilities of these devices but failed to reduce the overhead introduced by unnecessary management and data movement. In light of this observation, we optimize models to address these limitations.

1) Operator Fusion: As discussed in §III-B1, neural networks can be represented as computational graphs. Fig. 6a presents an example, where the input tensors w_1, w_2, and w_3 are processed by three operators, i.e., Add, Mul, and Sum. To execute them, there are three main steps, i.e., operator scheduling, data transmission, and operator kernel execution [51]. For example, to perform the calculation with the Add operator, the host first schedules and picks up this operator to be executed based on the graph structure (step 1). Then, tensors w_1 and w_2 are loaded and moved to the GPU memory (step 2). Finally, the kernel of Add is executed (step 3).

Non-optimized computational graphs have redundant management and data movement operations. Fig. 6a gives an example of computing $w_1 + w_2 + w_2 \times w_3$. The initial computational graph contains three operators, i.e., Add, Mul, and Sum. Fig. 6b shows its execution process: the host first schedules the Add operator, then transfers the input tensors w_1 and w_2 to the GPU memory, and finally executes the Add operator. Then, the host schedules the Mul operator, transfers the input tensors w_2 and w_3 to the GPU memory, and executes the Mul operator. Finally, the host schedules the Sum operator, transfers the input tensors w_4 and w_5 to the GPU memory, and executes the Sum operator. In this execution process, all inputs and internal results have to be frequently loaded to (or offload from) the GPU memory, which leads to high overhead. Furthermore, despite its simplicity, this graph necessitates multiple rounds of scheduling (i.e.,



(a) Operator Fusion Steps



(b) Overhead Analysis

Fig. 6: Operator Fusion Workflow. In sub-figure (a), rectangles, cycles, and arrows are tensors, operators, and data transmission, respectively. In sub-figure (b), dashed lines denote equal kernel execution overhead.

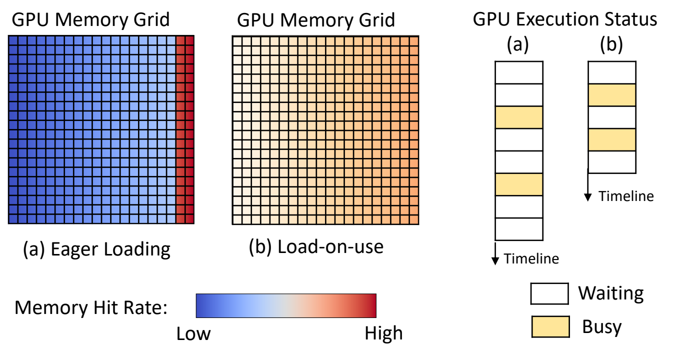


Fig. 7: GPU Memory Usage and Execution Status Difference between Common Practice (Eager Loading) and Our Optimization (Load-on-use).

positively correlated to the number of operators), introducing more overhead. The key idea of reducing such overhead is to fuse graph operators, which is known to be challenging [52]. While existing methods are search or heuristic-based, our key observation is that most cryptographic operators are many-to-one (e.g., addition) and one-to-one (e.g., slice), which allows us to perform easy rule-based operator fusion to reduce the scheduling and data movement overhead. Similar to Fig. 5, we omitted unoptimized operations for readability in Fig. 6.

Many-to-one Operator Fusion. For many-to-one operators, our fusion rule looks for operators that depend on the same inputs and simply merge these operations into one by appending one after the other. If these operators do not have dependencies, the order of operations does not matter. Otherwise, we follow the dependency order to maintain the correctness of the computation. For example, in Fig. 6a, the `Add` and `Mul` operators can be fused into one `Add-Mul` operator because they share the same input tensor w_2.

One-to-one Operator Fusion. For the one-to-one operator, we can simply merge it with its predecessor or successor by prefixing or appending the operator logic, respectively. For the Fig. 6a example, after fusing the `Add` and `Mul` operators, the `Sum` operator can be fused with the `Add-Mul` operator by appending the `Sum` logic to the `Add-Mul` operator. As such, we reduce the total number of operators to one, reducing the scheduling overhead by removing two operators and avoiding loading w_2, w_4, and w_5 (as shown in Fig. 6b).

Fusion Order. One-to-one operators do not have dependencies on other operators, thus the fusion order for them does not matter. Many-to-one operators can lead to the creation of one-to-one operators. As such, we first perform many-to-one operator fusion and then perform one-to-one operator fusion.

2) Load-on-use Memory Management: Another key factor that affects GPU computation performance is memory management. Similar to the CPU main memory, the GPU memory is also a shared resource and has limited capacity. If the GPU memory is full, the host has to evict some data from the GPU memory to make room for new data. This process is called *memory swapping* and is time-consuming. GPU memory management is more challenging. Firstly, unlike the main memory, which is controlled by the kernel, GPU memory is managed by users. Namely, a user can determine when to load data to the GPU memory and when to evict data from the GPU memory. A poorly programmed application may use the GPU memory inefficiently, leading to poor performance. We profiled the GPU memory usage of a popular cryptographic implementation [26]. For profiling, we used the NVIDIA GeForce GTX 1080 Ti GPUs that have 11 GB GDDR5X memory and fed the ciphers with over 10^7 input plaintext blocks. It took 0.0368 and 0.141 seconds to perform data transmission and encryption, respectively.

Fig. 7 (a) shows the hit-rate heatmap of the GPU memory. As we can see from the figure, the GPU memory is not fully utilized. Most of the GPU memory has very low hit rates and only a small portion of the GPU memory is frequently used. This is because the implementations loaded the large bulk of plaintext data to the GPU memory at the beginning of the encryption process, which is a common practice [26, 53], known as eager loading. The plaintext data is usually large and occupies most of the GPU memory. As such, the more frequently used values have to be frequently loaded and evicted from the GPU memory, which leads to poor encryption performance. Moreover, operator fusion is a typical optimization strategy that trades space for time: it reduces the number of operators but increases the number of inputs in each operator. For example, in the Fig. 6a, the final operator `Add-Mul-Sum` has three inputs, i.e., w_1, w_2, and w_3, more than any of the original operators. In the real world, if an operator has too many inputs, it can lead to inefficient memory usage and execution, similar to the presented case above.

While observing such a performance bottleneck, we propose to optimize the data transmission process with load-on-use, a lazy loading strategy. Namely, we would postpone the data loading as much as possible until the data is actually needed. To do this, we cluster computations requiring the same inputs and reorder the statements to ensure that statements using the same inputs are close to each other while maintaining the dependencies unchanged. Meanwhile, many ciphers employ a technique where plaintext is not directly encrypted but instead combined with intermediate ciphertext [22]. This approach is commonly used in many block and stream ciphers, e.g., AES (CTR mode), Salsa20, Chacha20, RC4, HC-128, etc [22]. We optimize such ciphers and only transfer plaintext from CPU to GPU when it is needed, rather than loading it into GPU memory from the very beginning. As shown in Fig. 7 (b), with our optimization, all the GPU memory is available and dedicated to cipher execution.

D. *Transformation Analysis*

In this subsection, we show that our program transformation guarantees the correctness of the generated program by showing its bisimulation equivalence to the original program. Moreover, we also prove that the transformed program is trace equivalent to the original program, achieving the same level of security as the original program under our threat model

(§III-A). To begin with, we first introduce the definitions of bisimulation equivalence and trace equivalence:

Definition 1 (Trace Equivalence)**.** Given programs P and M, $S^P_{i,t}$ represents the program execution state of program P at execution step t for a given input i. We use T^P and T^M to denote all possible execution steps for program P and M. Programs P and M are **trace equivalent** ($P \simeq_T M$) when all their execution states are equivalent, i.e., $\forall i, S^P_{i,T^P} = S^M_{i,T^M}$.

Definition 2 (Trace Equivalent Transformation)**.** Given a program transformation Γ and programs P and M, $M = \Gamma(P)$, Γ is a trace equivalent transformation $\iff P \simeq_T M$.

Definition 3 (Bisimulation Equivalence)**.** Given two programs P and M, they are **bisimulation equivalent** ($P \simeq_B M$) if $\forall i, P(i) = M(i)$ where i is any input.

Intuitively, two programs are *trace equivalent* when their internal execution states are the same during the whole execution for the same input. Program transformation that preserves such a property is a trace equivalent transformation (Definition 1). Meanwhile, we refer to two programs as *bisimulation equivalent* as they implement the same functionality, i.e., two programs generate the same output for the same input, which guarantees the correctness of two implementations but does not guarantee the same security. For example, merge sort and quick sort algorithms can give the same output for the same input (i.e., bisimulation equivalent), but they are not trace equivalent. Bisimulation equivalent transformation can also be defined in this way, and we omit such details. For a cryptography algorithm, we require the transformation to be trace equivalent to ensure that the transformed program achieves the same level of security as the original program. This is to avoid the possibility that inner variables of the transformed program leak critical information (e.g., the secret key).

Based on the definitions, we have:

Corollary 1 (Functionality Guarantee)**.** Programs P and M are trace equivalent $\Rightarrow$ P and M are bisimulation equivalent, i.e., $P \simeq_T M \Rightarrow \ P \simeq_B M$.

Proof. We have $P \simeq_T M$. Based on its definition, we have $\forall i, S^P_{i,T^P} = S^M_{i,T^M}$. When t^P and t^M are the terminal states of program P and M, respectively, $\forall i, S^P_{i,t^P} = S^M_{i,t^M} \Rightarrow P(i) = M(i)$. Thus, $P \simeq_T M \Rightarrow \ P \simeq_B M$. □

To show the security of our transformation, we first introduce a few lemmas.

Lemma 2. Our proposed program transformation is a trace equivalent transformation.

This lemma can be proven by induction. Details can be found in §A.

Based on Corollary 1 and Lemma 2, we can make the following claim about the correctness guarantee of our program transformation:

Corollary 3 (Correctness Guarantee)**.** DNN M generated by the proposed program transformation provides the same functionality as its original implementation P.

Proof. We use Γ to denote our program transformation, thus $M = \Gamma(P)$. According to Lemma 2, our program transformation Γ is a trace equivalent transformation. According to Corollary 1, $\forall i, M(i) = P(i)$ where i is an input. Thus, M provides the same functionality as P. □

Lemma 4. Under the threat model where the adversary can observe all internal execution states, cryptographic programs M and P have the same security guarantees when $P \simeq_T M$.

Proof. Under the threat model, the goal of the adversary is to recover the key K from the program source code or its super set, the program execution states. We use $\mathbb{P}(K|S^P)$ and $\mathbb{P}(K|S^M)$ to denote the probability of the adversary guessing the correct key. The attack process is to make new observations on program states S and update this probability $\mathbb{P}(K|S)$, and when $\mathbb{P}(K|S)$ is high enough (can be solved under given time and computing resources constraints), the adversary wins. When P and M are trace equivalent, $\forall i, \forall k, S^P_{i,k} = S^M_{i.k}$. Namely, the adversary will make the same observations at any given time. Thus, $\forall i, \forall k, \mathbb{P}(K|S^P_{i,k}) = \mathbb{P}(K|S^M_{i,k})$.

In conclusion, trace equivalent programs M and P have the same security guarantees under this white-box setting. □

With this, we can confirm the security guarantee of our method by proving the following proposition:

Proposition 5 (Security Guarantee)**.** DNN M generated by our proposed program transformation provides the same security guarantees as its original implementation P under the threat model.

Proof. We use Γ to denote our program transformation, thus $M = \Gamma(P)$. According to Lemma 2, our program transformation Γ is a trace equivalent transformation. Thus, programs P and M are trace equivalent. According to Lemma 4, M and P provide the same security guarantee under the threat model. □

Now, we have proved that our program transformation guarantees the correctness of the transformed program and the same security as the original implementation under our threat model. Notice that such correctness and security guarantees abstract the compiling and optimization of the program conducted by the compiling framework. That is, if the compiling framework contains malicious behaviors [54], the proof is potentially not valid. Also, our transformation analysis is performed on DSLs instead of implementations. While the implementation adheres to the transformation framework and DSL, our optimizations enhance implementation efficiency without affecting the program transformation itself. We also want to point out that this security is consistent with existing cryptanalysis, which views the compiling framework (and lower-level computing infrastructures) as part of the trust base. In practice, we can also verify that the compiler intermediate representation (IR) or compiled binary code is trace equivalent. In Appendix §B, we provide such an example. For other potential threats (e.g., side channels), we have discussed security implications and side channel attacks in §III-A.

IV. Implementation

With TensorCrypt, we seek to transform existing cryptographic programs into NN models. To achieve this, we leveraged TensorCrypt to transform the C/C++ cryptographic programs (CRYPT DSL) into TensorFlow [13] models (NN DSL). Moreover, our transformation source programs are venerated C/C++ implementations, i.e., the OpenSSL AES/Chacha ciphers [55] and the official Salsa implementation [22]. Next, we transform these source programs into Tensorflow Models with TensorCrypt. Specifically, TensorCrypt utilizes the mappings outlined in Table II (Appendix §C) to build concrete TensorFlow models and layers. For TensorCrypt model optimizations (§III-C), we implemented load-on-use memory management with Tensorflow APIs (e.g., `tf.device` and `tf.constant` and directly leveraged XLA [16] for operator fusion.

Correctness. To verify the correctness of all implementations, we use NIST/RFC test vectors [56, 57] for AES and cipher specifications [22] for Salsa20 and Chacha20 on TensorCrypt models (as well as our baselines). Specifically, after building TensorCrypt models, we perform encryption on given input vectors and compare the cipertext with given test vectors to ensure exact match. The same process is applied on decryption with the TensorCrypt models.

V. Evaluation

We aim to answer the following research questions:

- **RQ1**: For effectiveness and efficiency, how do TensorCrypt models compare to state-of-the-art cryptographic implementations?
- **RQ2:** How general is our technique? Namely, is it limited to specific ciphers, hardware (e.g., mobile and IoT devices) or software (e.g., operating systems)?
- **RQ3**: How can our proposed optimization methods help reduce the overhead of TensorCrypt models?
- **RQ4**: To what extent can other factors, such as cipher modes, key sizes, and encryption/decryption options, impact the performance of TensorCrypt models?

A. Evaluation Setup

In this paper, we evaluate AES, Salsa20, and Chacha20. We use them for evaluations because they are recommended for large-volume data encryption in TLS 1.3 [58]. For AES, we follow previous studies [2, 59] and focus on the CTR mode due to its parallelism and superior security guarantee. Since different key sizes do not give notable performance differences (see §V-E for more details), we mainly report results on 128-bit keys for simplicity. In addition, our evaluation results are presented primarily on the encryption process because AES, Chacha20, and Salsa20 are symmetric ciphers [41], while it's worth noting that we also assess the decryption process, which is discussed in more detail in §V-E.

Baselines. To show how effective and efficient TensorCrypt models are, we compare them with current GPU-based cryptographic implementations that have achieved state-of-the-art performance [20, 26, 27]. But as far as we know, there are no such implementations available from reputable vendors, e.g., OpenSSL. Therefore, we concentrate on GPU-based cryptographic implementations found in literature and public repositories for baseline selection. We skip evaluating the implementations based on the other acceleration methods due to their lower performance, e.g., the AES-NI [24] are 6.52x-13.60x slower than GPU-based AES implementations [20, 28].

Despite many prior works proposing to accelerate cryptographic implementations using GPUs, most of them have not released their implementations. Moreover, some of the open-source implementations are incorrect [53]. For instance, the bugs are found in the existing research, e.g., hard-coded keys during compilation for implementations by Hajihassani et al. [7]. Therefore, we adopt a 3-step approach to select baselines considering availability, correctness, and state-of-the-art performance. First, we identify recent GPU-based cryptographic implementations and literature. Next, we test the open-source implementations to verify their correctness. Finally, we select implementations with state-of-the-art performance. Note that, we observe that baseline methods were evaluated using outdated GPUs in their original research, different from ours. Therefore, we use the same evaluation setup as TensorCrypt on all baselines for fair comparisons. We have chosen the GPU-based Chacha20/Salsa20 implementations [27] and AES benchmarks [20] as our baselines. (see Appendix §E).

Maintenance and Optimization Costs. During evaluations, we observe some one-time costs for both TensorCrypt models and baselines (e.g., model saving and loading for TensorCrypt models), which can be amortized in cipher execution processes (discussion in §F). For example, TensorFlow models undergo the time-consuming tracing process [60], which can be amortized by pre-loading the model in memory and reusing the traced model. Moreover, TensorCrypt models are first transformed from source programs, optimized using our proposed methods, and saved as `.pb` and `.tflite` files. For maintenance, TensorCrypt models and baselines involve minimal storage. Specifically, after transformation, TensorCrypt models are first stored in `.pb` files, whose sizes are comparable to baseline binaries, e.g., AES models and baselines have file sizes of 704 KB and 646 KB, respectively. TensorCrypt model sizes can be further reduced as `.tflite` models, e.g., the AES `.tflite` model is 149 KB, which is portable for mobile/IoT devices.

Test Environments and Methods. The specifications of the machines and devices used for evaluations are presented in Table I of §D. For performance testing, we process input messages into plaintext/ciphertext blocks (the block sizes for AES, Salsa20, and Chacha20 are 16-byte, 64-byte, and 64-byte) and

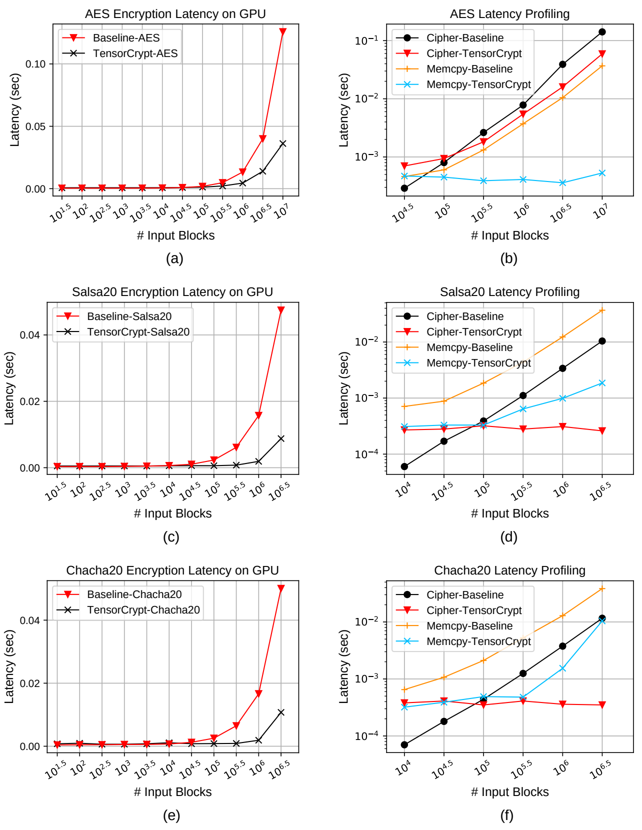


Fig. 8: Latency and Profiling of TENSORCRYPT Models and Baselines for AES, Salsa20, and Chacha20 on GPUs

run ciphers multiple (> 30) times. All steps are guaranteed to be the same for the baselines and TENSORCRYPT models.

B. **RQ1**: *Effectiveness and Efficiency*

To answer RQ1, we first evaluate both baselines and TENSORCRYPT models with GPUs and then perform an in-depth overhead analysis to comprehend the underlying reasons.

Overall Performance and Baseline Comparison. We evaluate the performance of TENSORCRYPT models and baselines on a GPU server, i.e., Lambda workstation in Table I, which has 4 NVIDIA GeForce GTX 1080 Ti graphics cards and NVIDIA CUDA Toolkit 11.4 as GPU driver. To fully utilize the GPU resources, we configure models and baselines as follows. For TENSORCRYPT models, we use the best-practice TensorFlow GPU computing strategy `tf.distribute.Strategy`. For baselines, we experiment with multiple thread numbers per block (i.e., `dimBlock`) and choose the one that yielded the best performance. We dynamically set the number of blocks per grid (i.e., `dimGrid`) based on the input size. In order to replicate the exponential data growth trend described in §I, we increase the size of plaintext and ciphertext exponentially based on number of input blocks[2].

Fig. 8 (a) presents the latency of AES models and baselines on GPU. Our overall results indicate that both TENSORCRYPT models and baselines exhibit an increasing trend in encryption latency. For small input (# blocks $\leq 10^4$), our baselines and TENSORCRYPT models exhibit similar performance. For large input ($> 10^4$), TENSORCRYPT models are 2.33× faster than the baselines on average. This performance gap becomes more significant when block number $> 10^6$ with our models achieving a speedup of 4.09× compared to the baselines. The 10^6 input blocks correspond to the data required for encryption and decryption in a Zoom meeting with 200 attendees, the application scenarios of TENSORCRYPT as elucidated in §I. Meanwhile, in Fig. 8 (c) and (e), Salsa20 and Chacha20 TENSORCRYPT models outperform the baselines as well, i.e., they are 5.44× and 5.06× faster than baselines on average. Our main objective for developing TENSORCRYPT was to enable efficient cryptographic computations, and the significant outperformance of our models compared to the baselines demonstrates the effectiveness of our approach.

Performance Analysis. To gain a better understanding of the reasons behind the superior performance of TENSORCRYPT models, we profiled both our models and the baselines. This analysis reveals three primary steps that cause the overhead: (*i*) key generation and input padding, (*ii*) cipher execution on the GPU, and (*iii*) data transmission between the CPU and GPU devices. Step (*i*) is found to be very efficient, taking less than 0.00001 seconds for both the baselines and TENSORCRYPT models. Therefore, we focused our performance profiling on the remaining two steps, (*ii*) and (*iii*), particularly for large input sizes where we observed significant performance differences between the models and baselines.

Fig. 8 (b) presents the profiling results for AES models and baselines. AES TENSORCRYPT models surpass baselines in both cipher execution and memory copy operations. Specifically, when with $> 10^5$ input blocks, TENSORCRYPT models (Cipher-TENSORCRYPT) exhibit an average cipher execution speed 1.93× faster than baselines (Cipher-Baseline). Similarly, TENSORCRYPT models (Memcpy-TENSORCRYPT) outperform baselines (Memcpy-Baseline) in data transmission operations. Fig. 8 (d) and (f) present profiling results for Salsa20 and Chacha20, where TENSORCRYPT models also exhibit better performance. As an illustration, on average, Salsa20 and Chacha20 models (Cipher-TENSORCRYPT) demonstrate cipher execution speeds that are 9.50× and 8.09× faster than the baseline (Cipher-Baseline), respectively. In data transmission, the Salsa20 model (Memcpy-TENSORCRYPT) demonstrates 2.29×, 5.61×, and 12.44× faster performance than the baseline (Memcpy-Baseline), given 10^4, 10^5, and 10^6 input blocks.

RQ1 Answer: TENSORCRYPT models are more efficient than baselines (e.g., up to 4.09× faster for AES encryption). Our profiling results demonstrate their superior efficiency in cipher execution and data transmission, e.g., up to 12.44× improvement for Salsa20.

C. *RQ2: Generalizability*

To answer RQ2, we evaluate TENSORCRYPT models on multiple platforms with various software and hardware stacks.

[2]We round some block numbers to integers, such as 31 for $10^{1.5}$

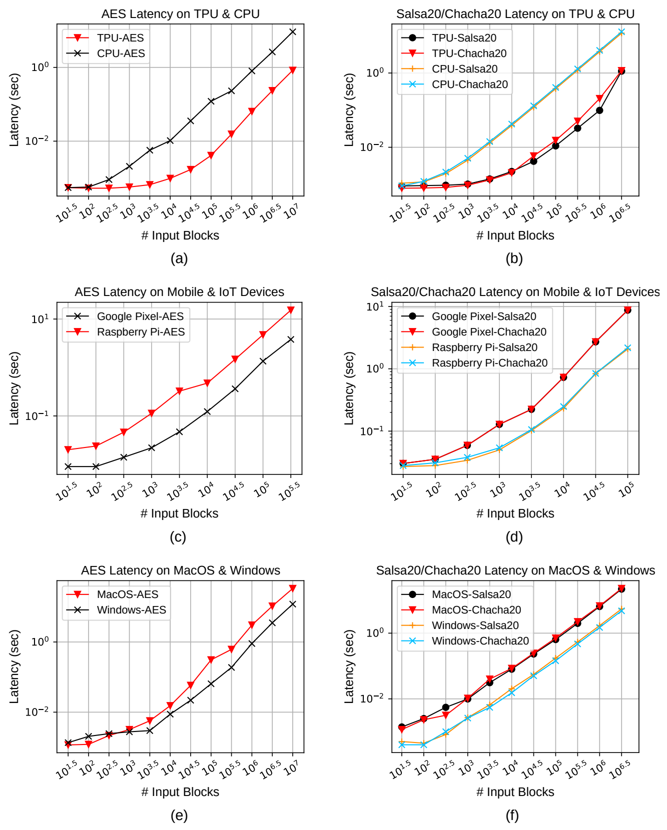


Fig. 9: Latency of TENSORCRYPT Models on Diverse Platforms with Various Software and Hardware Stacks.

Other Hardware Accelerators. To assess the deployability of TENSORCRYPT models on other hardware accelerators, we create a virtual machine (VM) on Google Cloud (Table I) with a Google Tensor Processing Unit (TPU) V2-8 node [11]. We skip evaluating baselines since TPUs are specifically designed for tensor operations and currently do not support CUDA GPU programs. Instead, we opt to evaluate TENSORCRYPT models on the CPU of the created VM, which enables us to showcase the advantages of TENSORCRYPT. For model deployment, we only need to change several lines of GPU deployment codes to use TPUs with the `tf.distribute.TPUStrategy` API.

Fig. 9 (a) presents the encryption latency of AES TENSORCRYPT models on TPU and CPU. Overall, TENSORCRYPT models perform 10.53× faster on TPU compared to CPU. With large input sizes (# blocks $\geq 10^4$), TPUs can significantly enhance the model's performance. For instance, with 10^5 input blocks, the model is 29.43× faster when executed on TPU compared to CPU. Fig. 9 (b) shows the latency of Salsa20 and Chacha20 models with TPU and CPU devices (since both Salsa20 and Chacha20 have the same block sizes, we present their performance on the same figure). The Salsa20 and Chacha20 models are 13.87× and 15.60× faster on TPU than on CPU, respectively.

Mobile and IoT Devices. DNN models are widely used in mobile and IoT applications [61], critical for IoT security [62]. We demonstrate the deployability of TENSORCRYPT models on mobile devices by running them on a Google Pixel XL smartphone with Android 8.1 installed. Although this device has a graphics card, TensorFlow mobile GPU delegate [63] only supports a very limited number of operators and tensor types without allowing users to add customized ones at the time of paper writing. Consequently, we build an Android application to execute the models on the CPU with 4 threads. For IoT devices, we deploy our models on a Raspberry Pi 3 board. To support cross-platform deployment, we convert TENSORCRYPT models into `.tflite` models with TensorFlow. We compare the deployment performance of both `.tflite` models and `.pb` models for the IoT device and find that the models in `.pb` format outperformed the others. Therefore, we report the performance of `.pb` models for IoT devices.

Fig. 9 (c) and (d) present the latency of the AES, Salsa20, and Chacha20 models on mobile and IoT devices. Overall, we notice comparable upward trends in latency for all three ciphers on both devices, e.g., it takes 150.1× more time to encrypt the input message with 10^5 blocks than with 10^2 blocks. The AES, Salsa20, and Chacha20 TENSORCRYPT models show respective speedups of 3.64×, 2.26×, and 2.13× on the Google Pixel smartphone compared to the Raspberry Pi 3. This difference in performance is reasonable considering the different CPU and memory capabilities of these two devices.

Operating Systems. In addition to evaluating the performance of our models on Linux (on the Lambda workstation) and Android (on the Google Pixel phone), we also assess their performance on Windows and MacOS. It is worth noting that Windows and MacOS constitute a significant share of the desktop market, with Windows holding 75.2% and MacOS holding 15.9% [64]. The test machines consist of a Windows Desktop running Windows 10 and a MacBook Pro machine running MacOS 12.5. We utilize the CPUs of these machines for evaluations as TensorFlow requires NVIDIA GPU drivers which are currently not compatible with Intel and AMD graphics cards. Fig. 9 (e) and (f) present the latency of the AES, Salsa20, and Chacha20 models running on Windows and MacOS machines. Overall, the model execution is faster on the Windows machine compared to the MacOS machine due to the better Windows machine specifications. Moreover, AES, Salsa20, and Chacha20 models exhibit respective speedups of 3.25×, 4.07×, and 4.27× for large input messages ($> 10^3$ blocks) on MacOS compared to Windows.

Programming Languages. To deploy TENSORCRYPT models, it is required to write codes to load and execute them, potentially using various programming languages. We have previously shown examples of deploying models using Python and Java in our evaluations (see Section V-B). Specifically, we utilized the Python `tensorflow` and `tflite_runtime` APIs to construct and import `.pb` and `tflite` files across multiple platforms, e.g., the Google Cloud VM and IoT devices. To deploy the models on mobile devices, we utilized the Java `org.tensorflow.lite` APIs. In this process, we have also demonstrated the ability of our models to utilize various API sets for a single language, e.g., `tensorflow` and `tflite_runtime` for Python. Moreover, developers have the option to convert our TensorFlow models into models of other deep learning frameworks using ONNX [65], providing them with even greater deployment flexibility. Although we have evaluated the two most popular DNN deployment languages [66], there are still other languages, such as C and C++. A more comprehensive study is left for future work.

RQ2 Answer: TENSORCRYPT is applicable to different ciphers (i.e., AES/Salsa20/Chacha20). TENSORCRYPT models can be deployed with diverse hardware and software stacks, i.e., hardware accelerators (GPU/TPU), operating systems (Linux/Windows/MacOS/Android), languages (Python/Java), and devices (mobile/IoT/desktop/-cloud server).

D. **RQ3***: Effectiveness of Optimization*

To answer RQ3, we assess the effectiveness of our proposed optimizations, i.e., operator fusion and load-on-use memory management. Based on our evaluations, these optimizations have similar performance benefits for AES, Salsa20, and Chacha20 models. For instance, operator fusion improves the performance of AES, Salsa20, and Chacha20 models by 79.79×, 85.72×, and 74.69×, respectively. Therefore, in this section, we focus on presenting the evaluation results for AES to provide a detailed analysis.

Operator Fusion. We investigate the impact of operator fusion on accelerating TENSORCRYPT models by enabling and disabling it during experiments. Fig. 10 (a) presents the encryption latency of models with and without this optimization. In general, we note a considerable performance improvement due to this optimization. To be specific, on average, models with operator fusion are 44.46× faster compared to those without it. Furthermore, for messages with a small size ($\leq 10^4$ input blocks), the performance gap is less than 27.30×. However, for larger input sizes, the latency difference becomes much more significant. For example, there is a 79.79× improvement in performance for messages containing 10^6 input blocks.

Performance Improvement Analysis. Having observed a significant performance improvement with operator fusion, we aim to uncover its underlying cause. First, we examine the changes in the computational graph of deep neural networks by dumping DNN programs with and without enabling operator fusion using the introspection flag [67]. Fig. 14 in Appendix §G presents the DNN programs with operator fusion applied. From line 3 to 13, multiple operators are fused into a single operator (`%fused_computation`). Furthermore, operator fusion reduces operators in the while loop (from line 17 to 28) from 1,404 to 24, indicating a decrease of 98.3% in the number of operators. Additionally, among these 24 operators, 12 are fused operators that can be identified by their specific identifier names, e.g., `%fused_computation.3`. We also profile the overhead of data transmission and operator scheduling, which we analyzed in §III-C1. Fig. 10 (b) presents the profiling results. Compared to the small overhead with optimization (all < 0.0025 seconds), the overhead without optimization exhibits

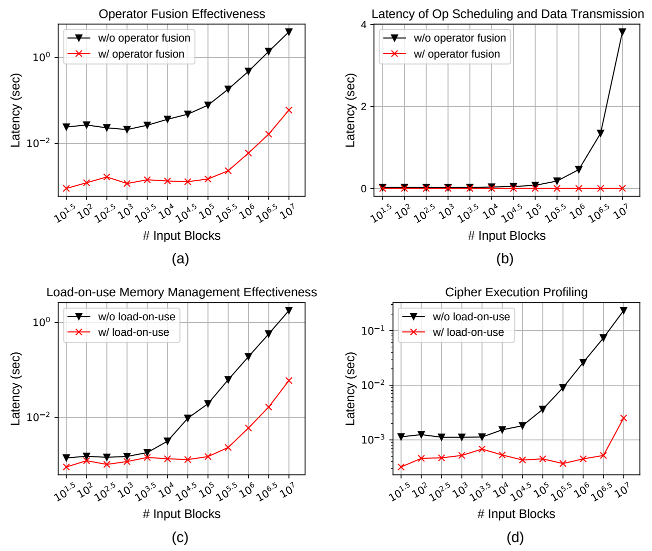


Fig. 10: Effectiveness and Latency Profiling of Operator Fusion and Load-on-use Memory Management Optimizations

an exponential incremental trend. In general, operator fusion reduces overhead by 519.29×. For large input plaintexts, the optimization proves to be even more effective, reducing overhead by 1514.92× for 10^7 input blocks.

Load-on-use Memory Management. Fig. 10 (c) presents the overall encryption latency of our models, comparing the performance w/wo load-on-use memory management. According to this figure, it significantly reduces latency with large input ($\geq 10^4$ input blocks), e.g., 2.45× improvement with 10^4 input blocks, and up to 4.92×. As discussed in §III-C2, load-one-use memory management optimizes the GPU memory for the encryption process. Therefore, we also profile this process. Fig. 10 (d) presents the overhead of counter encryption with and without this optimization. We find that the encryption process is significantly accelerated. When the plaintext is large, its overhead can be reduced by up to 92.5×.

RQ3 Answer: Our proposed optimizations make TENSORCRYPT models more efficient, e.g., operator fusion reduces the latency by up to 79.79× and load-one-use memory management accelerates encryption by up to 92.5×.

E. **RQ4***: Impacts of Other Factors*

Encryption and Decryption. In addition to demonstrating high encryption performance in various scenarios in the above sections, we also investigate the decryption performance of TENSORCRYPT models. We skip evaluating the decryption performance of Salsa20 and ChaCha20 since their encryption and decryption processes are identical [23]. Fig. 11 (a) presents the latency of AES models and baselines for encryption and decryption. Overall, there are no notable latency differences between encryption and decryption for both TENSORCRYPT models and baselines, i.e., the average latency gaps for models and baselines are 1.6% and 0.3%, respectively, which is to be expected given that AES is a symmetric cipher [68].

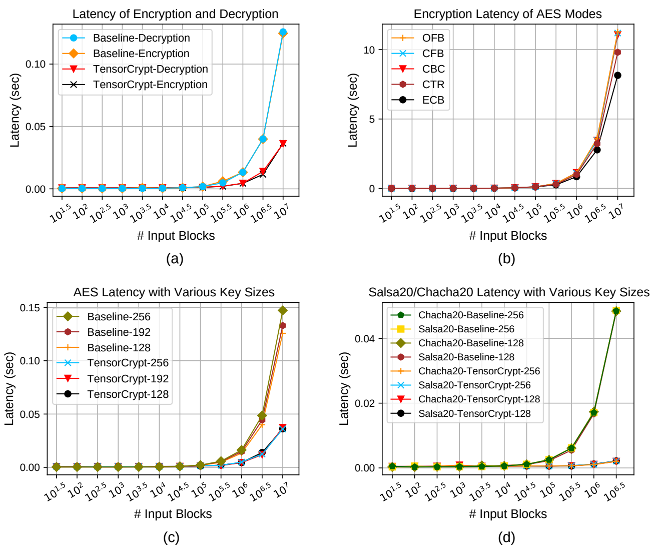


Fig. 11: Latency of Models and Baselines in Encryption/Decryption, Different AES Modes and Various Key Sizes

Different AES Modes. While we mainly report evaluation results of CTR mode because of its better security and parallelism (see §V-A), we also report the performance of other AES modes. Specifically, we execute AES TensorCrypt models in the ECB, CBC, CFB, OFB, and CTR modes using the Lambda workstation's CPUs (Table I) to give a fair comparison, since CBC, CFB, and OFB are sequential modes that cannot be parallelized. Fig. 11 (b) presents the latency of different AES modes. According to the results, the parallel modes such as ECB and CTR are faster compared to the sequential modes, e.g., the CTR mode is 1.05×, 1.06×, and 1.12× faster than CBC, CFB, and OFB modes on average.

Various Key Sizes. AES, Salsa20, and Chacha20 support multiple sizes of keys. Specifically, AES supports key sizes of 128, 192, and 256 bits, where different key sizes correspond to different encryption rounds. In Fig. 11 (c), we present the latency of the AES TensorCrypt models and baselines with 128-bit, 192-bit, and 256-bit keys on GPUs. Notably, we find that the key size has an impact on the encryption latency of the baselines, but it does not notably affect TensorCrypt models. For instance, the baseline with 128-bit keys is 1.18× faster than the baseline with 256-bit keys on average. However, the TensorCrypt-128 model is 1.003× faster than TensorCrypt-256, which is trivial enough to allow users to freely switch between models with different key sizes.

Salsa20 and Chacha20 support 128-bit and 256-bit keys, which result in different key streams as initial states. Fig. 11 (c) shows the latency of Salsa20 and Chacha20 models and baselines in 128-bit and 256-bit keys. While TensorCrypt models consistently outperform the baselines, we do not observe notable performance differences among different key sizes of Salsa20 and Chacha20 models. Specifically, the average latency gaps between the 128-bit and 256-bit Salsa20 and Chacha20 models are only 0.9% and 0.7%, respectively. From the efficiency perspective, the trivial performance differences across various key sizes allow us to showcase TensorCrypt models with one specific key size, such as the 128-bit key.

RQ4 Answer: TensorCrypt models have negligible performance differences in encryption and decryption and across various key sizes. AES latency varies on different modes, e.g., ECB mode is 1.32× faster than CFB mode.

VI. Discussion

Multi-GPU Parallelism. We do not include multi-GPU evaluations because our baseline methods do not support such configurations [2, 18, 20, 26–31]. To ensure a fair comparison, we evaluate both TensorCrypt and the baselines under the same single-device environment. Our goal is to demonstrate that our superior performance stems from the design of TensorCrypt itself, rather than the use of additional hardware accelerators. That said, modern AI frameworks like TensorFlow [69] have extensive support for multi-GPU setups, enabling efficient execution of massive models. TensorCrypt is implemented in TensorFlow and thus can leverage multi-GPU configurations.

Implementation Effort Analysis. In §I, we introduced the engineering challenges associated with harnessing hardware accelerators. These challenges encompass a range of issues, including but not limited to inadequate tool chain support, steep learning curves, a scarcity of experts, and significant time commitments [70]. The introduction of TensorCrypt offers a transformative solution—a program translation framework that necessitates only an existing cryptographic implementation. Since existing cryptographic experts are already well-versed in their implementations, transitioning to AI frameworks, characterized by their high-level abstractions, is relatively straightforward. Furthermore, the tool chain utilized for the development, maintenance, and deployment of our models is both mature and widely adopted. Conducting a comprehensive exploration of these factors, potentially through human-centered studies, could pave the way for interesting avenues in future research.

Security Implication Analysis. For the potential security implications, we analyze TensorCrypt models from three aspects. First, TensorCrypt leverages hardware accelerators, thus, it is potentially vulnerable to side-channel attacks, e.g., power side channels [71, 72] and time side channels [73] attack, like other other GPU-based cipher implementations [74]. Instead, we focus on improving efficiency over baselines. Second, deep neural networks are known to be vulnerable to adversarial machine learning attacks, e.g., data poisoning and backdoor attacks [75]. However, such attacks cannot be applied to TensorCrypt models since they are not trained. Instead, the weights are directly translated from a symbolic program, which makes it possible to explain and interpret the weights and execution. Finally, to support multi-platform deployment (§V-C), TensorCrypt does not sacrifice security but needs hardware resources for acceleration, e.g., GPUs, same as existing solutions (see §III-D).

Other Platforms and Implementations. Our evaluations are performed on diverse platforms listed in Table I. While there can be other platforms for deployment and evaluations, the performance observed on these platforms may differ from the results reported in this paper. We believe the evaluations on other platforms (e.g., other GPUs) will be interesting for future research. Additionally, we selected the baseline implementations to the best of our ability; however, more advanced baseline implementations may emerge in the future. For TENSORCRYPT models, we only implemented them on TensorFlow, and we believe there are frameworks for implementations. While our primary contributions focus on the novel application of neural networks (NN) for cryptography, we leave the exploration of alternative implementations as future work.

VII. Conclusion

In summary, we present an approach to accelerate cryptographic algorithms through the TENSORCRYPT system. This addresses the escalating demand for efficient cryptographic computations, facilitated by user-friendly programming interfaces and optimal performance. TENSORCRYPT employs DSL-guided transformations to seamlessly convert cryptographic programs into AI models, guaranteeing trace and bisimulation equivalence. Moreover, our introduced optimization techniques substantially amplify model efficiency. Empirical evaluations robustly demonstrate that TENSORCRYPT models outperform existing solutions, underscoring their adaptability across diverse platforms. TENSORCRYPT artifact is available at: https://github.com/OSUSecLab/TensorCrypt.

Acknowledgment

We appreciate anonymous reviewers for their insightful feedback. The authors have been supported in part by TrojAI W911NF-19-S-0012, and NSF awards 2112471, 2207202, 2319944, and 2342250. Any opinions, findings, conclusions, or recommendations presented are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Bernard Marr, "How much data do we create every day? the mind-blowing stats everyone should read," https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/?sh=458cc54e60ba.

[2] V. Eduardo, L. C. E. De Bona, and W. M. N. Zola, "Speculative encryption on {GPU} applied to cryptographic file systems," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019, pp. 93–105.

[3] H. Park, Y. Huo, and S.-E. Yoon, "Meshchain: Secure 3d model and intellectual property management powered by blockchain technology," in *Advances in Computer Graphics: 38th Computer Graphics International Conference, CGI 2021, Virtual Event, September 6–10, 2021, Proceedings 38*. Springer, 2021, pp. 519–534.

[4] S. A. Manavski, "Cuda compatible gpu as an efficient hardware accelerator for aes cryptography," in *2007 IEEE International Conference on Signal Processing and Communications*. IEEE, 2007, pp. 65–68.

[5] T. Isobe and R. Ito, "Security analysis of end-to-end encryption for zoom meetings," *Ieee Access*, vol. 9, pp. 90 677–90 689, 2021.

[6] "Zoom system requirements: Zoom web app and web client," https://support.zoom.us/hc/en-us/articles/11999201591949-Zoom-system-requirements-PWA-and-web-client, accessed on 10-16-2023.

[7] O. Hajihassani, S. K. Monfared, S. H. Khasteh, and S. Gorgin, "Fast aes implementation: A high-throughput bitsliced approach," *IEEE Transactions on parallel and distributed systems*, vol. 30, no. 10, pp. 2211–2222, 2019.

[8] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed dnn training acceleration," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 16–29.

[9] H. T. Siegelmann and E. D. Sontag, "On the computational power of neural nets," *Journal of computer and system sciences*, vol. 50, no. 1, pp. 132–150, 1995.

[10] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," *arXiv preprint arXiv:1410.5401*, 2014.

[11] G. Cloud, "Cloud tpu," https://cloud.google.com/tpu.

[12] Tom Brookes, "What is the apple a16?" https://nanoreview.net/en/soc/apple-a16-bionic.

[13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[14] A. Paszke, S. Gross, F. Massa *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[15] T. Chen, T. Moreau, Z. Jiang *et al.*, "{TVM}: An automated end-to-end optimizing compiler for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 578–594.

[16] Tensorflow, "Xla: Optimizing compiler for machine learning," https://www.tensorflow.org/xla.

[17] H. Böck, A. Zauner, S. Devlin, J. Somorovsky, and P. Jovanovic, "Nonce-disrespecting adversaries: Practical forgery attacks on gcm in tls." *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 475, 2016.

[18] Z. Wang, H. Chen, and W. Cai, "A hybrid cpu/gpu scheme for optimizing chacha20 stream cipher," in *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE, 2021, pp. 1171–1178.

[19] D. J. Bernstein and T. Lange, "Post-quantum cryptography," *Nature*, vol. 549, no. 7671, pp. 188–194, 2017.

[20] C. Tezcan, "Optimization of advanced encryption standard on graphics processing units," *IEEE Access*, vol. 9, pp. 67 315–67 326, 2021.

[21] J. Daemen and V. Rijmen, "Aes proposal: Rijndael," 1999.

[22] D. J. Bernstein, "The salsa20 family of stream ciphers," *New stream cipher designs: the eSTREAM finalists*, pp. 84–97, 2008.

[23] D. J. Bernstein *et al.*, "Chacha, a variant of salsa20," in *Workshop record of SASC*, vol. 8, no. 1. Citeseer, 2008, pp. 3–5.

[24] S. Gueron, "Intel(r) advanced encryption standard (aes) new instructions set white paper," https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf.

[25] S. Ghaznavi, C. Gebotys, and R. Elbaz, "Efficient technique for the fpga implementation of the aes mixcolumns transformation," in *2009 International Conference on Reconfigurable Computing and FPGAs*. IEEE, 2009, pp. 219–224.

[26] C. Wang and X. Chu, "Gpu accelerated aes algorithm," *arXiv preprint arXiv:1902.05234*, 2019.

[27] P. Santucci, E. Ingrassia, G. Picierro, and M. Cesati, "Memshield: Gpu-assisted software memory encryption," in *Applied Cryptography and Network Security: 18th International Conference, ACNS 2020, Rome, Italy, October 19–22, 2020, Proceedings, Part II*. Springer, 2020, pp. 323–343.

[28] R. K. Lim, L. R. Petzold, and Ç. K. Koç, "Bitsliced high-performance aes-ecb on gpus," in *The New Codebreakers*. Springer, 2016, pp. 125–133.

[29] J. W. Bos, D. A. Osvik, and D. Stefan, "Fast implementations of aes on various platforms," *Cryptology ePrint Archive*, 2009.

[30] C. Mei, H. Jiang, and J. Jenness, "Cuda-based aes parallelization with fine-tuned gpu memory utilization," in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 2010, pp. 1–7.

[31] A. Khalid, G. Paul, and A. Chattopadhyay, "New speed records for salsa20 stream cipher using an autotuning framework on gpus," in *Progress in Cryptology–AFRICACRYPT 2013: 6th International Conference on Cryptology in Africa, Cairo, Egypt, June 22-24, 2013. Proceedings 6*. Springer, 2013, pp. 189–207.
[32] S. K. Monfared, O. Hajihassani, M. S. Kiarostami, S. M. Zanjani, D. Rahmati, and S. Gorgin, "Bsrng: a high throughput parallel bitsliced approach for random number generators," in *49th International Conference on Parallel Processing-ICPP: Workshops*, 2020, pp. 1–10.
[33] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. S. Iyengar, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–36, 2018.
[34] X. Jin, K. Pei, J. Y. Won, and Z. Lin, "Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1631–1645.
[35] X. Jin, J. Larson, W. Yang, and Z. Lin, "Binary code summarization: Benchmarking chatgpt/gpt-4 and other large language models," *arXiv preprint arXiv:2312.09601*, 2023.
[36] X. Jin and Z. Lin, "Simllm: Calculating semantic similarity in code summaries using a large language model-based approach," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1376–1399, 2024.
[37] D. Castelvecchi, "Can we open the black box of ai?" *Nature News*, vol. 538, no. 7623, p. 20, 2016.
[38] D. Lindner, J. Kramár, M. Rahtz, T. McGrath, and V. Mikulik, "Tracr: Compiled transformers as a laboratory for interpretability," *arXiv preprint arXiv:2301.05062*, 2023.
[39] L. Medsker and L. C. Jain, *Recurrent neural networks: design and applications*. CRC press, 1999.
[40] Y. Weng, M. Zhu, F. Xia, B. Li, S. He, K. Liu, and J. Zhao, "Neural comprehension: Language models with compiled neural networks," *arXiv preprint arXiv:2304.01665*, 2023.
[41] C. De Cannière, "Analysis and design of symmetric encryption algorithms," *Doctoral Dissertaion, KULeuven*, 2007.
[42] C. De Canniere, A. Biryukov, and B. Preneel, "An introduction to block cipher cryptanalysis," *Proceedings of the IEEE*, vol. 94, no. 2, pp. 346–356, 2006.
[43] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," *Journal of machine learning research*, vol. 18, 2018.
[44] M. Kim and P. Smaragdis, "Bitwise neural networks," *arXiv preprint arXiv:1601.06071*, 2016.
[45] "Onnx operator schemas," https://github.com/onnx/onnx/blob/main/docs/Operators.md, accessed on 10-10-2023.
[46] "Torch operators," https://pytorch.org/docs/stable/torch.html, accessed on 10-10-2023.
[47] H. Van Vliet, H. Van Vliet, and J. Van Vliet, *Software engineering: principles and practice*. John Wiley & Sons Hoboken, NJ, 2008, vol. 13.
[48] R. Morisset, P. Pawan, and F. Zappa Nardelli, "Compiler testing via a theory of sound optimisations in the c11/c++ 11 memory model," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 187–196, 2013.
[49] N. I. of Standards and Technology, "Advanced encryption standard," *NIST FIPS PUB 197*, 2001.
[50] T. Sanida, A. Sideris, and M. Dasygenis, "Accelerating the aes algorithm using opencl," in *2020 9th International Conference on Modern Circuits and Systems Technologies (MOCAST)*. IEEE, 2020, pp. 1–4.
[51] TensorFlow, "Optimize tensorflow gpu performance with the tensorflow profiler," https://www.tensorflow.org/guide/gpu_performance_analysis.
[52] M. Li, Y. Liu, X. Liu *et al.*, "The deep learning compiler: A comprehensive survey," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 708–727, 2020.
[53] W.-K. Lee, H. J. Seo, S. C. Seo, and S. O. Hwang, "Efficient implementation of aes-ctr and aes-ecb on gpus with applications for high-speed frodokem and exhaustive key search," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 6, pp. 2962–2966, 2022.
[54] K. Thompson, "Reflections on trusting trust," *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, 1984.
[55] OpenSSL, "Openssl project," https://github.com/openssl/openssl/tree/master/crypto.
[56] M. Dworkin, "Recommendation for block cipher modes of operation. methods and techniques," National Inst of Standards and Technology Gaithersburg MD Computer security Div, Tech. Rep., 2001.
[57] Y. Nir and A. Langley, "Chacha20 and poly1305 for ietf protocols," https://datatracker.ietf.org/doc/rfc7539/.
[58] E. Rescorla, "The transport layer security (tls) protocol version 1.3," Tech. Rep., 2018.
[59] H. Zhang, A. Anilkumar, M. Fredrikson, and Y. Agarwal, "Capture: Centralized library management for heterogeneous {IoT} devices," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 4187–4204.
[60] "Tracing," https://www.tensorflow.org/guide/function#tracing, accessed on 11-29-2023.
[61] Z. Sun, R. Sun, L. Lu, and A. Mislove, "Mind your weight (s): A large-scale study on insufficient machine learning model protection in mobile apps," in *30th USENIX security symposium (USENIX security 21)*, 2021, pp. 1955–1972.
[62] X. Jin, S. Manandhar, K. Kafle, Z. Lin, and A. Nadkarni, "Understanding iot security from a market-scale perspective," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1615–1629.
[63] TensorFlow, "Gpu delegates for tensorflow lite," https://www.tensorflow.org/lite/performance/gpu.
[64] StatCounter, "Desktop operating system market share worldwide," https://gs.statcounter.com/os-market-share/desktop/worldwide.
[65] J. Bai, F. Lu, K. Zhang *et al.*, "Onnx: Open neural network exchange," https://github.com/onnx/onnx, 2019.
[66] PwC, "Pwc's global artificial intelligence study: Exploiting the ai revolution," https://www.pwc.com/gx/en/issues/data-and-analytics/publications/artificial-intelligence-study.html.
[67] T. XLA, "Inspect compiled programs," https://www.tensorflow.org/xla#inspect_compiled_programs.
[68] M. J. Dworkin, E. B. Barker, J. R. Nechvatal, J. Foti, L. E. Bassham, E. Roback, J. F. Dray Jr *et al.*, "Advanced encryption standard (aes)," 2001.
[69] M. Abadi, A. Agarwal, P. BarhamAmini *et al.*, "Api documentation," https://www.tensorflow.org/api_docs.
[70] S. Mittal and J. S. Vetter, "A survey of cpu-gpu heterogeneous computing techniques," *ACM Comput. Surv.*, vol. 47, no. 4, 2015.
[71] Z. Lin, U. Mathur, and H. Zhou, "Scatter-and-gather revisited: High-performance side-channel-resistant aes on gpus," in *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*, 2019, pp. 2–11.
[72] D. Das, A. Golder, J. Danial, S. Ghosh, A. Raychowdhury, and S. Sen, "X-deepsca: Cross-device deep learning side channel attack," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.
[73] C. Luo, Y. Fei, and D. Kaeli, "Side-channel timing attack of rsa on a gpu," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 3, pp. 1–18, 2019.
[74] Z. H. Jiang, Y. Fei, and D. Kaeli, "A novel side-channel timing attack on gpus," in *Proceedings of the Great Lakes Symposium on VLSI 2017*, ser. GLSVLSI '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 167–172. [Online]. Available: https://doi.org/10.1145/3060403.3060462
[75] M. Rigaki and S. Garcia, "A survey of privacy attacks in machine learning," *arXiv preprint arXiv:2007.07646*, 2020.
[76] C. Lattner, "Llvm and clang: Next generation compiler technology," in *The BSD conference*, vol. 5, 2008.
[77] T. Jin, A. Eichenberger *et al.*, "Onnx mlir," https://github.com/onnx/onnx-mlir.
[78] C. Lattner, M. Amini, U. Bondhugula *et al.*, "Llvm ir target," https://mlir.llvm.org/docs/TargetLLVMIR/.
[79] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.
[80] G. Narayanan and N. Haneef, "Parallel aes cryptography," https://github.com/jmarcao/Parallel-AES.
[81] D. Stefan, "Analysis and implementation of estream and sha-3 cryptographic algorithms," Ph.D. dissertation, 2011.

APPENDIX

A. *Proof of Lemma 2*

Lemma 2. Our proposed program transformation is a trace equivalent transformation.

Proof. Proof by induction.

• $t = 0$: These two implementations do not initialize any global variables, and thus, they start from the same initial state, i.e., $\forall i, S^P_{i,0} = S^M_{i,0}$.

• $t = 1$: The first action of two cryptographic implementations is to read inputs based on given arguments, which is a system provided function and only changes the variable representing the input. Thus, $\forall i, S^P_{i,1} = S^M_{i,1}$. Similarly, for the terminal state, the two implementations both return the output value. We emit the details in our discussion.

• $t > 1$: Suppose that when $t = k$, the execution states of programs P and M are the same for any input i, i.e., $S^P_{i,k} = S^M_{i,k}$. Based on semantics of two DSLs in Fig. 3, their execution states can be represented by all memory traces, i.e., σ for the CRYPT DSL implementation and ϕ for the NN DSL implementation. In other words, $S^P = \sigma$ and $S^M = \phi$. When $t = k$, $\forall i, S^P_{i,k} = S^M_{i,k}$, thus, $\sigma_{i,k} = \phi_{i,k}$ for any i. Here, we aim to prove that $\forall i, \sigma_{i,k+1} = \phi_{i,k+1}$ which is equivalent to $\forall i, S^P_{i,k+1} = S^M_{i,k+1}$.

The goal is to show that $\forall i, \sigma_{i,k+1} = \phi_{i,k+1}$ holds for the next statements s_{k+1} (of the CRYPT DSL implementation) and l_{k+1} (of the NN DSL implementation). For our program transformation Γ, we have $l_{k+1} = \Gamma(s_{k+1})$. Notice that s_{k+1} can be any type of statements defined in the CRYPT DSL (Fig. 1). According to transformation rules of Γ (Fig. 4), for each type of statement in the CRYPT DSL, there exists one and only one rule to transform the statement into a single statement in the NN DSL (Fig. 2). Proof by exhaustion solves this problem.

– $s_{k+1} \xrightarrow{ctx} l_{k+1}$: $x = y + z \xrightarrow{ctx} \boxed{x = \textbf{add}(y, z)}$.

Based on rule `[OP-ADD-CRYPT]` (Fig. 3), statement s_{k+1} updates the execution state by $\sigma[x \to y+z]$. Namely, it updates the value of x to be the sum of y and z. Similarly, following the rule `[OP-ADD-NN]` in Fig. 3, statement l_{k+1} updates the execution state by $\phi[x \to y + z]$.

Because $\sigma_{i,k} = \phi_{i,k}$, we have $\sigma_{i,k}[y] = \phi_{i,k}[y]$ and $\sigma_{i,k}[z] = \phi_{i,k}[z]$. Thus, we have:

$$\begin{aligned}\sigma_{i,k+1}[x] &= \sigma_{i,k}[y] + \sigma_{i,k}[z] = \phi_{i,k}[y] + \phi_{i,k+1}[z] \\ &= \phi_{i,k+1}[x]\end{aligned}$$

Considering that both σ and ϕ only update the value of x, we can get $\sigma_{i,k+1} = \phi_{i,k+1}$ is true. Thus, we prove that when $s_{i,k+1}$ is $x = y + z$, $\forall i, \sigma_{i,k+1} = \phi_{i,k+1}$ if $\sigma_{i,k} = \phi_{i,k}$.

– Similarly, when s_{k+1} are other types of statements, we always have $\forall i, \sigma_{i,k+1} = \phi_{i,k+1}$ if $\sigma_{i,k} = \phi_{i,k}$. Intuitively, our transformation Γ ensures that there is a one-to-one mapping between CRYPT DSL statements and NN DSL layers. Global and semantic rules in Fig. 3 guarantee that corresponding statements have the same evaluation order, and statement and layer rules show that they preserve the same execution states. Rules defined in Fig. 4 and Fig. 3 with the same sub-tokens are corresponding rules, e.g., `[T-OP-ADD]`, `[OP-ADD-CRYPT]` and `[OP-ADD-NN]`. Due to space limit, we omit proof of others.

Thus, for $t > 1$, $\forall i, S^P_{i,k+1} = S^M_{i,k+1}$ when $S^P_{i,k} = S^M_{i,k}$.

• In conclusion, we prove Lemma 2 using proof by induction. □

```
define dso_local i32 @add(i32 %x, i32 %y) #0 {  entry:
   %x.addr = alloca i32, align 4 // memory allocation
   %y.addr = alloca i32, align 4
   store i32 %x, i32* %x.addr, align 4 // store inputs
   store i32 %y, i32* %y.addr, align 4
   %0 = load i32, i32* %x.addr, align 4 // load values
   %1 = load i32, i32* %y.addr, align 4
   %add = add nsw i32 %0, %1 // add operation
   ret i32 %add
}
```

(a) IR of Integer Value Addition Program Function

```
 1  func @add(%a0: ptr<i32>, %a1: ptr<i32>, %a2: i64, %a3:
              ptr<i32>, %a4: ptr<i32>, %a5: i64) ->
              struct<(ptr<i32>, ptr<i32>, i64)> {
 2     // get a struct object (%3) for the 1st input tensor
 3     %0 = undef : struct<(ptr<i32>, ptr<i32>, i64)>
 4     %1 = insertvalue %a0, %0[0] : struct<(ptr<i32>,
                                         ptr<i32>, i64)>
 5     %2 = insertvalue %a1, %1[1] : struct<(ptr<i32>,
                                         ptr<i32>, i64)>
 6     %3 = insertvalue %a2, %2[2] : struct<(ptr<i32>,
                                         ptr<i32>, i64)>
 7     // make %20 point to the 1st input tensor value
 8     %19 = extractvalue %3[1] : struct<(ptr<i32>,
                                         ptr<i32>, i64)>
 9     %20 = load %19 : ptr<i32>
10     ...... // omit similar steps as line 3 to 8
11     // make %22 point to the 2nd input tensor value
12     %22 = load %21 : ptr<i32>
13     // add values of the 1st and 2nd input tensors
14     %23 = add %20, %22  : i32
15     ...... // omit steps of storing %23 to %18
16     // return the result struct object (%18)
17     return %18 : struct<(ptr<i32>, ptr<i32>, i64)>
18  }
```

(b) IR of Integer Tensor Addition Neural Network Model

Fig. 12: An Example of Trace Equivalence.

```
from z3 import *
# create symbolic variables
x, y, x_addr, y_addr, v0, v1, v_add, a1, a4, v20, v22,
        v23, v18 = BitVecs(“x y x_addr y_addr v0 v1
        v_add a1 a4 v20 v22 v23 v18", 32)
# define function semantics
fCRYPT = And(x_addr==x, y_addr==y, v0==x_addr,
                        v1==y_addr, v_add==v0+v1)
fNN = And(a1==x, a4==y, v20==a1, v22==a4, v23==v20+v22,
        v18==v23)
# create logic implication
G = Implies(And(fNN, fCRYPT), v_add == v18)
# get a satisfiable problem & solve constraints
solver = Solver()
solver.add(Not(G))
print(solver.check())
```

Fig. 13: Equivalence Proof with SMT Solver

B. Example of Trace Equivalence

Fig. 12 presents an example to demonstrate trace equivalence of programs before and after transformation. To present our analysis steps, we choose the `[T-OP-ADD]` rule (Fig. 4) which is simple but non-trivial. As the transformation framework requires a source program, we first create a function (f_{crypt}) which takes two integer inputs (x, y) and returns the output calculated by the $z = x + y$ statement. With the `[T-OP-ADD]` rule, we construct a model (f_{nn}) performing the operation of adding two integer tensors with the add layer. For trace equivalence demonstration, we need to prove the memory state updates are the same for executing the statement

TABLE I: Machine and Device Specifications

Device name	Processor/CPU	OS	RAM	Storage	Accelerator
Lambda Workstation	Intel Xeon E5-1650	Ubuntu 18.04 LTS	64 GB	4 TB	NVIDIA GeForce GTX 1080 Ti
Google Cloud VM	Intel Skylake	Debian 10	104 GB	300 GB	Google TPU V3-8 and V2-8
Windows Desktop	AMD Ryzen 5600X	Windows 10	16 GB	1 TB	AMD Radeon RX 6600
MacBook Pro	Intel I5-7360U	MacOS 12.5	8 GB	256 GB	Intel Iris Plus Graphics
Google Pixel XL	Qualcomm Snapdragon 821	Android 8.1	4 GB	32 GB	Qualcomm Adreno 530
Raspberry Pi 3	ARMv7 rev 4	Raspbian 10 (buster)	1 GB	16 GB	N/A

and layer. Different from proof in §III-D, we prove this in LLVM IR level. LLVM offers intermediate representations (IR) that are language agnostic [76]. These representations capture low-level memory operations while abstracting away structural differences between programs and neural networks. Therefore, we compile the program, generate its LLVM IR (Fig. 12 (a)) with the `clang` compiler [76], and get model's LLVM (MLIR) IR (Fig. 12 (b)) with `onnx-mlir` [77].

In Fig. 12 (a), the inputs of `@add` function are stored into two allocated memory addresses from line 2 to 5. To carry out an addition operation, the inputs are retrieved from memory, and the statement at line 8 performs the addition of the retrieved values. Considering the memory state update, variable `%add` is updated from its initial state to $x + y$. In Fig. 12 (b), the `@add` layer takes 6 arguments, representing two input tensors. According to the IR function type rule [78], the first 3 arguments belong to the first tensor, in which `%a1` points to the data buffer of the first tensor's value (x), the `%a0` pointer is for freeing memory, and `%a2` is the address offset of the aforementioned two pointers. For simplicity, we only show the detailed memory address operations of the first input tensor from line 3 to 9. Specifically, the `undef` operation defines a `struct` object at line 3. The `insertvalue` operation inserts arguments `%a0, %a1, %a2` to this object and returns the modified `struct` object. Therefore, variable `%3` is an object holding all pointers and offsets of the first input tensor after executing lines 3 to 6, and the tensor value is stored at its second pointer. Line 8 and 9 load the tensor value pointer from `%3` and make `%20` point to the tensor value. With similar steps, `%22` will point to the second tensor's value (y). The `add` operation at line 14 adds the two input tensors' values to `%23`. We omit steps of storing `%23` into another `struct` object `%18`, which is finally returned. To this end, the memory state of `%23` is updated from its initial state to $x + y$.

By comprehending the fundamental operations and memory state updates depicted in Fig. 12, we prove the addition program function f_{crypt} and the addition model f_{nn} are equivalent. We prove this by using SMT solvers: we first encode memory variables and then prove that they will hold the same value after these instructions when initial values are the same. In Fig. 12, we ignore pointer manipulation operations as they will not change the memory states, and we focus on the memory read and write operations. To encode the semantics of f_{crypt} and f_{nn}, we first introduce symbols to represent intermediate variables. For f_{crypt}, the symbols are x_{in}, y_{in}, x_{addr}, y_{addr}, v_0, v_1, and v_{add}. With these symbolic

TABLE II: TensorFlow Modules/Operators Used

NN DSL	TensorFlow	NN DSL	TensorFlow
input	tf.function	sub	tf.math.subtract
lookup	tf.gather	bitAnd	tf.bitwise.bitwise_and
slice	tf.slice	bitShift	tf.bitwise.shift
add	tf.math.add	bitXOR	tf.bitwise.bitwise_xor
mul	tf.math.mul	loop	tf.while_loop
M	tf.Module	cond	tf.cond
output	tf.function		

variables, we formalize semantics of f_{crypt} as:

$$x_{addr} = x_{in} \land y_{addr} = y_{in} \land v_0 = x_{addr} \land v_1 = y_{addr} \land v_{add} = v_0 + v_1 \quad (1)$$

Similarly, we define symbolic variables a_1, a_4, v_{20}, v_{22}, v_{23}, v_{18} for f_{nn}, and formalize its semantics as:

$$v_{20} = a_1 \land v_{22} = a_4 \land v_{23} = v_{20} + v_{22} \land v_{18} = v_{23} \quad (2)$$

With the definitions of f_{crypt} and f_{nn} in equations Equation 1 and Equation 2, the proof of their value equivalence can be converted to whether the following proposition (F) to be VALID given $x_{in} = a_1, y_{in} = a_4$:

$$f_{crypt} \land f_{nn} \Rightarrow v_{add} = v_{18} \quad (3)$$

To prove F to be `VALID`, we prove $\neg F$ is unsatisfiable (`UNSAT`) by getting negation of F: `VALID`(F) $\equiv$ `UNSAT`($\neg F$).

Then, the problem is converted to the proof of satisfiability of $\neg F$, which can be solved by a `SMT` solver. Fig. 13 presents a satisfiability proof implementation for the equivalence of f_{crypt} and f_{nn} based on the `Z3 SMT` solver [79], which successfully proved f_{crypt} and f_{nn} to be trace equivalent.

C. *TensorFlow Modules and Operations*

Table II presents the mapping from NN DSL to Tensorflow modules and operators for constructing our AES TensorCrypt models.

D. *Machine and Device Specifications*

Table I presents the specifications of all machine and devices that we used for evaluations.

E. *Baseline Selection*

To find the baselines, we follow the 3-step baseline selection approach (§V-A) to evaluate various cryptographic implementations for AES, Salsa20, and Chacha20. The evaluation results are presented in Table III, in which the selected baselines are highlighted. Note that, some baselines are outdated/buggy or only accept fixed input in their initial

```
HloModule a_inference_call_1933__XlaMustCompile_true_config_proto_n_007_n_0...02_001_000__executor_type_.1508

%fused_computation (param_0: s32[10,16], param_1.1: s32[], param_2.2: s32[1,16], param_3.3: pred[]) -> s32[10,16] {
  %param_0 = parameter(0)
  %param_3.3 = pred[] parameter(3)
  %broadcast.8120 = pred[1,16]{1,0} broadcast(pred[] %param_3.3), dimensions={}
  %param_2.2 = parameter(2)
  %param_1.1 = parameter(1)
  %con.15264 = constant(0)
  %dynamic-slice.3907 = dynamic-slice(%param_0, %param_1.1, %con.15264), dynamic_slice_sizes={1,16}
  %select.9 = select(%broadcast.8120, %param_2.2, %dynamic-slice.3907)
  ROOT %dynamic-update-slice.1306 = dynamic-update-slice(%param_0, %select.9, %param_1.1, %con.15264)
}

...// skip declarations of other fused operators
%while_body_30_const_0__.61.clone (inputs.7 -> (s32[], s32[10,16], s32[1,16], s32[10,16], s32[176]) {
  %inputs.7 = (s32[], s32[10,16]{1,0}, s32[1,16]{1,0}, s32[10,16]{1,0}, s32[176]{0}) parameter(0)
  ... // skip 11 operations
  %fusion.10 = fusion(%get.8478, %get.8476, %get.8477, %copy.4), kind=kLoop, calls=%fused_computation.10
  %fusion.9 = fusion(%get.8478, %con.14946, %con.14944, %fusion.10), kind=kLoop, calls=%fused_computation.9
  %fusion.8 = fusion(%get.8478, %con.14946,  %con.14944, %fusion.9), kind=kLoop, calls=%fused_computation.8
  ... // skip declarations for fusion.7 to fusion.2, which are defined similarly
  %fusion.1 = fusion(%get.8478,  %con.14946,  %con.14944, %fusion.2), kind=kLoop, calls=%fused_computation.1
  %fusion.11 = pred[] fusion(%copy.4), kind=kLoop, calls=%fused_computation.11
  %fusion = fusion(%get.8475, %copy.4, %fusion.1, pred[] %fusion.11), kind=kLoop, calls=%fused_computation
  ROOT %tuple.3503 = tuple(%add.1321, %fusion, %fusion.1, %get.8477, %get.8478)
}
```

Fig. 14: Dumped DNN Program with Operator Fusion. Some information, e.g., operations and shapes, are omitted for simplicity.

TABLE III: Evaluations for Baseline Selection

Implementation	Availability	Latency
AES [2, 7, 28, 50, 53, 71]	✗	-
AES [80]	✓	0.015 (s)
AES [20]	✓	0.013 (s)
Salsa20 [31]	✗	-
Chacha20 [18]	✗	-
Salsa20/Chacha20 [27]	✓	0.016 (s)

prototypes, and we have updated and reproduced them to adapt to outer testing configurations, e.g., dynamic input. Our evaluations focus on implementations proposed after 2013, as older implementations may not deliver state-of-the-art performance. For instance, the Salsa20 implementation [81], proposed in 2011 and based on CUDA 4.0, is outdated and does not deliver advanced performance. To assess popularity, we consider the total number of stars, forks, and watchers of the repositories. For performance comparison, we conduct tests across different input sizes with the same evaluation environment. We report the averaged latency for 10^6 input blocks in Table III. Notably, for Salsa20 and Chacha20, the only open-sourced implementations available are proposed by Santucci et. al [27], with Chacha20 being the only one implemented initially. To solve the problem, as discussed in §V-A, we extend it to include Salsa20 strictly following its specification [22].

While the core ciphers have been implemented by baselines, we extend them to different modes and key sizes following the NIST standard [68]. For instance, the AES baseline only implements Electronic Code Book (ECB) mode with 256-bit keys. Since various AES modes and key sizes share the same core cipher, we extend ECB cipher to Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR) modes. Our baselines are also subjected to the same validation process (see §IV) as TensorCrypt models.

F. One-time Costs for Cipher Execution

We observe some one-time latency costs while evaluating baselines and TensorCrypt models. For TensorCrypt models, there are two one-time operations, including loading models and model recompilation. For model loading, the average latency is 0.148 seconds. For model recompilation, the latency comes from the dynamic shape of plaintext (or ciphertext) inputs. Specifically, given the different input size, the DNN compiler has to perform recompilation as it currently does not support dynamic shapes[3]. To avoid this, we compile several models with different fixed input shapes, e.g., (10, 16), (100, 16), etc. To use them for encryption, we pad zeros to plaintext inputs to match the fixed shapes. For instance, we pad 16 zeros to the input with a shape of (9, 16) and encrypt it with the (10, 16) model. After encryption, we retrieve ciphertext results by removing the padded bytes. The zero padding operation reduces latency cost and converts model recompilation time as one-time cost. Note that, we have considered and counted the latency of input padding operations. For both TensorCrypt models and baselines, all the one-time operation cost can be amortized in cipher execution processes.

G. Example of Operator Fusion Optimization

Fig. 14 presents the DNN program that we dumped for the TensorCrypt model with the operator fusion optimization. The identifier names (e.g., `%fused_computation` and `%fusion.10`) reveal the lines with operator fusion.

[3] https://groups.google.com/g/xla-dev/c/WgQ-xyRj9ZQ/m/8WDsXF0pDAAJ