A Study on White-Box Cryptography based Integrity Verification Techniques for On-Device AI Model and Their Performance

Han Bin Lee, Jun Young Cho, TaeGuen Kim

Department of Cyber Security, Korea University Sejong, Republic of Korea

Received: October 05, 2025; Revised: November 15, 2025; Accepted: December 10, 2025; Published: December 23, 2025

Abstract

This paper studies White-Box Cryptography (WBC)–based integrity verification for protecting on-device AI models deployed on resource-constrained and potentially hostile platforms. Conventional hash- or MAC-based integrity checks assume that keys and verification logic are isolated from adversaries, an assumption that fails when attackers control the execution environment and can inspect or modify AI binaries and models. WBC embeds secret keys inside heavily obfuscated implementations, making integrity checks more resilient against static and dynamic analysis and key extraction. In this study, methods of using WBC, which offers such security advantages, are investigated to protect on-device AI embedded within embedded systems. To assess the feasibility of WBC-based protection techniques for on-device AI, existing WBC-based integrity verification approaches are selected, implemented, and tested in constrained benchmark environments. Using the algorithmic analysis and experimental results, functional and non-functional requirements are then derived.

Keywords: Whitebox Cryptography, Data Integrity, Message Authentication Code.

1 Introduction

On-device AI has become a core component of modern embedded systems, ranging from smartphones and IoT sensors to automotive control units and industrial controllers [1]. Instead of sending data to powerful cloud servers, inference is increasingly performed locally to meet latency, privacy, and availability requirements [2]. As AI models and their supporting software gain direct access to sensitive data and critical control paths, ensuring their integrity becomes a central security requirement [3]. If an attacker can tamper with the model parameters or the surrounding code, the system may leak confidential information, behave unpredictably, or be deliberately steered into unsafe states [4]. Traditional software integrity mechanisms, such as cryptographic hashes, message authentication codes, or code-signing schemes, are typically designed under an assumption that secret keys and verification logic are kept separate from and hidden from adversaries. However, in many real-world deployments of on-device AI, this assumption no longer holds. Attackers may have full control over the device, can dump firmware images, debug and trace execution, and freely inspect or modify binaries and models. Under such white-box conditions, a conventional integrity check can often be bypassed by extracting keys, patching verification routines, or emulating expected outputs.

White-Box Cryptography (WBC) was proposed precisely to address such hostile environments [5]. In WBC, cryptographic algorithms are implemented in a way that embeds secret keys into heavily obfuscated tables and logic, making it significantly harder for an attacker with full code and memory access to recover the keys or to modify computations without detection [6]. These properties make WBC an attractive candidate foundation for protecting on-device AI, where integrity verification must operate in the presence of powerful local adversaries. Nevertheless, the practical feasibility of WBC-based integrity verification on embedded

platforms remains insufficiently understood. The execution time and memory consumption of WBC-based algorithms are often reported to incur substantial overhead compared with conventional implementations [7].

This paper focuses on WBC-based integrity verification for protecting on-device AI models deployed on resource-constrained and potentially hostile platforms. Methods of using WBC, which offers such security advantages, are investigated to safeguard both the code and parameters of AI workloads embedded within embedded systems. To assess the feasibility of this approach, existing WBC-based integrity verification schemes are selected, implemented, and evaluated in constrained benchmark environments [8]. Execution time and memory usage are measured across multiple scenarios to quantify the performance impact on realistic AI workloads. Based on algorithmic analysis and experimental results, concrete functional and non-functional requirements are derived. This requirement-oriented perspective aims to bridge the gap between cryptographic white-box designs and system-level engineering, providing practical guidance for the deployment of WBC-based integrity verification in future on-device AI systems [9].

2 Related Works

The following summarizes prior research on protecting on-device AI models and, for each study, discusses how White-Box Cryptography (WBC) could be leveraged or integrated in that context. SecDeep[10] is a lightweight TrustZone-based framework that protects data confidentiality and model integrity by executing only a small portion of sensitive tensor-handling code in the Secure World, keeping the secure TCB around 1K sLoC while preserving GPU acceleration. Its integrity checks rely on MD5/SHA-1 and format-preserving encryption replacing or complementing these with WBC-based integrity MACs would enable similar protection on devices without TrustZone and harden the integrity path against local key extraction. ASGARD[11] is a virtualization-based TEE framework on Armv8-A that uses a pKVM-based TEEvisor and a debloated Microdroid enclave to protect DNN models while securely passthroughing the NPU, achieving ≈2% latency overhead and reduced TCB. In this architecture, WBC-based integrity verification can serve as a portable software anchor for validating binaries and configurations inside and outside the enclave, especially on heterogeneous edge platforms. Soter[12] is an SGX-based partitioned framework that secures DNN confidentiality and integrity by transforming associative operators into morphed GPU-executable forms and restoring correct outputs inside the enclave, combined with oblivious fingerprinting for integrity. A WBC-based integrity engine could replace or augment this fingerprint layer, embedding verification keys in white-box MACs that validate GPU results even when SGX is unavailable. Partition-and-Merge[13] improves memory efficiency and confidentiality for large DNNs on constrained IoT devices by splitting models into encrypted/obfuscated sub-networks and merging them at runtime, enabling full on-device execution on 64–128 MB devices with moderate latency overhead. Attaching WBC-based integrity checks to each partition or the merge logic would detect tampered sub-networks and corrupted merge paths, complementing obfuscation with stronger runtime integrity.

DarkneTZ[14] protects model privacy on edge and mobile devices by running only privacy-critical layers in TrustZone's Secure World and the remainder in the Normal World, keeping overhead below about 10% while significantly reducing privacy attacks. WBC-based integrity verification can additionally safeguard the partitioning policy and sensitive-layer binaries, preventing silent downgrades or malicious layer replacement. Slalom[15] mitigates SGX performance and memory limits by offloading linear layers to an untrusted GPU and keeping nonlinear operations inside the enclave, combining Freivalds checks for integrity with blinding for input privacy and achieving large throughput and energy gains. WBC-based integrity primitives could replace or enhance Freivalds, embedding verification keys in white-box MACs that remain robust even if SGX boundaries are partially weakened. DeepAttest[16] enforces device-level IP protection and usage control by embedding a device-unique fingerprint into DNN weights and verifying it in a TEE before and during inference, with negligible accuracy loss and modest latency overhead. WBC-based integrity verification can protect the attestation code and fingerprint-checking logic themselves, enabling similar device-bound attestation on platforms without TEEs and reducing the risk of forged or bypassed checks. DeepSigns[17] is

an end-to-end watermarking framework for DNN ownership that embeds watermarks into internal activation distributions rather than weights, preserving accuracy and resisting fine-tuning, pruning, and overwriting attacks. Integrating DeepSigns with WBC-based integrity mechanisms would couple ownership proof with runtime integrity enforcement, ensuring that only authorized, untampered binaries and parameters run in a purely software-based protection stack. While these works address system-level protection, model partitioning, attestation, and watermarking, they generally treat cryptographic primitives as black boxes and do not directly analyze the behavior of concrete WBC constructions themselves. In contrast, classical WBC research on AES provides concrete designs and attack results that can be instantiated and benchmarked as integrity engines for protecting on-device AI models.

The following studies focus on WBC algorithms themselves in this work, these constructions are implemented as integrity verification engines for on-device AI, and their benchmarked overheads are used to derive concrete functional and non-functional requirements for WBC-based integrity protection.

Chow et al.[18] introduce the white-box attack context and present the first white-box AES construction, in which the key is fused into large lookup tables and surrounded by linear and nonlinear encodings to hinder key extraction even when an adversary has full access to code and memory. Their work demonstrates how table-based implementations can achieve functional correctness while obscuring internal key material, and it has become the de facto baseline for subsequent WBC designs and attacks. In the present study, a Chow-style white-box AES instance is used as one of the WBC-based integrity verification engines, allowing its performance and resource footprint to be evaluated side by side with conventional integrity mechanisms. Xiao and Lai[19] propose a secure implementation of white-box AES that specifically addresses structural weaknesses of the Chow design, mainly by embedding the ShiftRows operation into matrix products and redesigning output encodings so that previously known key-extraction strategies are less effective. Their scheme refines the encoding structure to make algebraic decomposition attacks harder, while maintaining compatibility with standard AES functionality. As part of the benchmark campaign in this work, an implementation following the Xiao–Lai design is instantiated as a second WBC-based integrity primitive, enabling comparison of its execution time and memory usage against both Chow's scheme and non-WBC integrity checks under identical on-device AI scenarios. McMillion and Sullivan[20] systematically evaluate practical attacks against several deployed white-box AES constructions, including Chow-type and related designs, and show that generic techniques such as differential computation analysis can recover embedded keys in realistic settings with modest effort. Their results highlight the gap between theoretical white-box goals and the actual resistance of many WBC schemes, and they provide concrete attack costs and methodologies that any security evaluation must take into account. Building on these insights, the benchmark tests in this study treat the selected WBC implementations not as unbreakable primitives but as concrete integrity engines with known security margins, and focus on quantifying their performance and memory overheads under realistic deployment constraints for on-device AI, thereby informing functional and non-functional requirements for future, more robust WBC-based integrity designs.

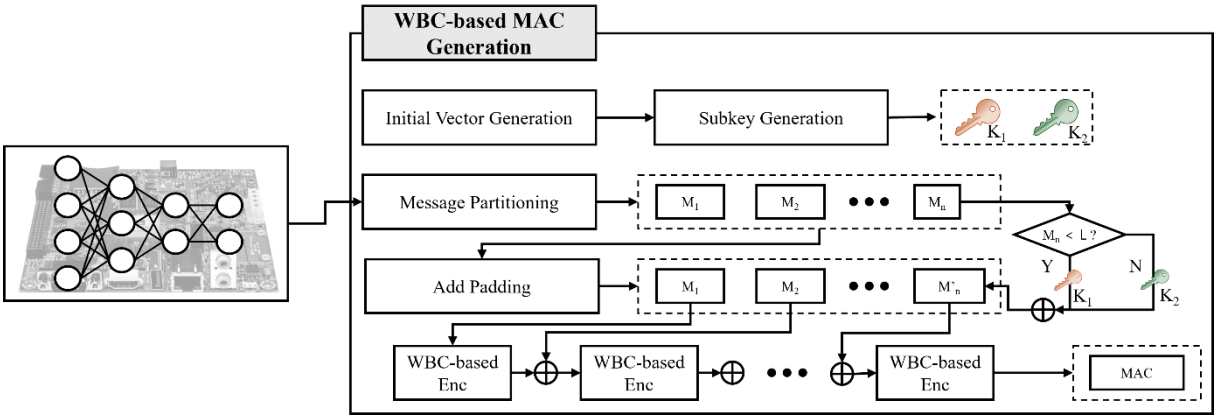

Figure 1: Processing Flow of WBC-CMAC Generation

3 WBC-Based Integrity Verification Framework Proposal

In this section, we propose the architecture of a MAC generation framework that leverages White-Box Cryptography (WBC) for on-device integrity verification. The proposed framework illustrated in Fig. 1, is a security software system that applies white-box cryptography to provide strong integrity verification and protection for AI models executed in on-device environments that are disconnected from external networks or exposed to high security risks. Unlike cloud-based services, on-device AI models are deployed directly to user devices, where an attacker may obtain full control or physical access and perform memory dumping, reverse engineering, or side-channel analysis. In such settings, the execution environment is effectively a white-box attack context in which all code and data can be observed and manipulated. Conventional symmetric-key MAC algorithms and simple model-encryption schemes have a fundamental weakness in this context at verification time, the secret key must inevitably appear in cleartext in memory or CPU registers when the MAC is computed or the model is decrypted. An attacker with white-box access can then extract the key and forge or tamper with the protected model at will. To address this problem, the proposed framework adopts the standard CMAC structure but replaces the core block-cipher engine $E_K(\cdot)$with a white-box cryptographic implementation $WBC(\cdot)$. In this design, the secret key is not stored in any explicit variable or memory region; instead, it is dispersed and fused into complex arithmetic logic and large lookup tables, heavily obfuscated across the implementation. As a result, the encryption key is never present in cleartext at any point during runtime, and integrity verification can be performed safely and autonomously on general-purpose processors, even when no TEE or secure hardware is available and no network connectivity can be assumed. The framework thus operates as an optimized, software-only security solution that preserves the trustworthiness of AI models even under extreme threats where an attacker can fully observe and analyze internal system behavior.

The framework is designed to support a wide range of deployment scenarios, including public networks, private networks, and offline or isolated environments in which on-device AI models operate. In all of these cases, white-box cryptography is used as the core security mechanism to perform integrity verification without exposing secret keys. In connected settings, the framework can complement or partially replace conventional CA-based digital signatures by allowing each device to perform local, standalone integrity checks, thereby reducing the load on centralized authentication servers and maintaining continuous security even when network connectivity is unstable. In offline and physically isolated environments such as smart factories or defense systems, where network separation is mandatory and hardware TEEs or real-time key distribution infrastructures are undesirable or unavailable, the framework functions as a practical integrity-verification alternative that still offers strong security. Because the white-box implementation never stores keys in cleartext but embeds them into obfuscated tables and logic, the system can reliably detect model tampering even when an attacker has physical access to the device and can perform memory dumps or reverse engineering in a full white-box attack setting. At the algorithmic level, the operation of the framework is based on CMAC, with the key-dependent encryption step replaced by a white-box block cipher. Conceptually, the CMAC encryption call $E_K(\cdot)$is substituted by a white-box function $WBC(\cdot)$that implements the same block-cipher transformation but with the key hardwired and obfuscated. The message authentication process comprises four main stages: subkey generation, message partitioning and preprocessing, final-block preprocessing, and iterative chaining.

In the subkey generation stage, the framework derives two independent subkeys $K_1$and K_2that determine how the last block is treated depending on whether padding is required. First, a b-bit block of zeros 0^b(where $b$is the block size of the underlying white-box cipher) is encrypted using the white-box function, yielding an initial block $L = WBC(0^b)$. Because this computation is performed inside the white-box engine, it never exposes the key in cleartext while still producing a value fully tied to the internal key-dependent tables, thus preserving resistance against key-recovery attacks. To derive K_1, the most significant bit (MSB) of $L$is examined. If $\mathrm{MSB}(L) = 0$, then $K_1$is obtained by left-shifting $L$by one bit. If $\mathrm{MSB}(L) = 1$, the left-shifted value is further XORed with a fixed constant $R_b$defined over the underlying Galois field. This follows the standard CMAC doubling operation in $\mathrm{GF}(2^b)$to avoid structural weaknesses for certain input patterns. The second subkey $K_2$is generated similarly from K_1: the MSB of $K_1$is checked, and the same shift-and-XOR rule with $R_b$is applied. This hierarchical subkey derivation ensures that $K_1$and $K_2$are algebraically related yet distinct, making it difficult for an attacker to infer whether padding was applied based solely on the final MAC value

or intermediate states. Once the subkeys are prepared, the input AI model file or data is normalized into a form suitable for processing by the white-box cipher during the message partitioning and preprocessing stage. The message Mcan have arbitrary length, but the cipher operates on fixed b-bit blocks. Therefore, $M$is split into a sequence of blocks $(M_1, M_2, \dots, M_r)$, where the first $r-1$blocks are full b-bit blocks, and the last block $M_r$may be either full-length or shorter than $b$bits. This partitioning step clarifies the block boundaries and determines whether the final block is complete or incomplete, which in turn drives the padding rules and subkey usage in the next stage.

The final-block preprocessing stage applies distinct subkeys and, if necessary, padding to the last block to prevent structural vulnerabilities in variable-length message authentication. Without this step, a simple chaining MAC could be vulnerable to extension attacks: an adversary could append data to a valid message and use the previous MAC value as the initial chaining state to forge a new valid MAC. To prevent such attacks, the preprocessing ensures that the last block is cryptographically marked as the end of the message. If the last block $M_r$is a complete block of exactly $b$bits, the framework computes $M'_r = M_r \oplus K_1$, where $\oplus$denotes bitwise XOR. If $M_r$is shorter than $b$bits, a padding function is applied: a single bit '1' is appended to the message data, followed by as many '0' bits as needed to reach $b$bits, yielding a padded block $\widetilde{M}_r$. The framework then computes $M'_r = \widetilde{M}_r \oplus K_2$. Using $K_1$for the complete-block case and $K_2$for the padded-block case allows the verification process to cryptographically distinguish between naturally terminated messages and those requiring padding, closing off subtle forgery vectors. Finally, the iterative chaining stage, structurally similar to CBC mode, combines all blocks into a single MAC value that captures the integrity of the entire message. An initial chaining variable $C_0$is set to the all-zero b-bit string. For each block $M_i$with $1 \le i < r$, the framework computes

$$C_i = WBC(C_{i-1} \oplus M_i),$$

so that each new chaining value depends on the previous one and the current plaintext block. This ensures that any one-bit change in any block propagates through all subsequent chaining values due to the avalanche effect of the block cipher, making it infeasible for an attacker to modify a portion of the message without altering the final MAC. In the last step, instead of using the original last block M_r, the preprocessed block $M'_r$is used:

$$C_r = WBC(C_{r-1} \oplus M'_r).$$

The final chaining value $C_r$is output as the message authentication code (MAC) that certifies the integrity of the AI model or data under the embedded white-box key. Because the encryption is implemented as a white-box cipher, the entire computation can be observed, but the key remains non-extractable in practice, and the MAC cannot be forged without replicating the same obfuscated implementation or breaking the white-box construction.

4 Benchmark Testing

In the benchmark evaluation, the WBC-MAC framework is compared against a range of conventional and white-box cryptographic algorithms on multiple hardware platforms. The baseline algorithms include software-based AES (SAES), standard AES, DES, and 3DES, while the white-box family consists of three AES implementations: CHOW[18], XIAO[19], and a hardened white-box AES variant, HARDENED-WBC[20], which incorporates countermeasure ideas inspired by the attack methodologies analyzed in McMillion and Sullivan. For each algorithm, we evaluate both single-file encryption and CMAC-based integrity-tag generation under identical conditions. The experiments are carried out on four representative hardware platforms that span from high-performance to embedded-class devices: an AMD Ryzen 5600G desktop processor, an Intel N100 low-power processor, a Raspberry Pi 4 single-board computer, and an NVIDIA Jetson Nano edge AI platform. This setup allows us to observe how WBC-induced overheads scale

across realistic on-device AI deployment environments. To quantify performance overhead, we measure the execution time required by each algorithm within the WBC-MAC framework. For input data sizes ranging from 512 KB to 10 MB, the wall-clock time to perform single-file encryption and CMAC tag generation is recorded on each hardware platform. All timings are reported in seconds, rounded to the fourth decimal place, enabling fine-grained comparison between conventional ciphers and the CHOW[18]-, XIAO[19]-, and HARDENED-WBC[20]-based implementations, and providing concrete guidance on acceptable latency budgets for on-device AI integrity verification.

4.1 Execution Time Measured in Benchmark Test Environment

The benchmark test results are presented in Table 1-4. Across all four benchmark platforms, the execution-time results show a clear, roughly linear increase with input size for every algorithm, but with very large gaps between conventional ciphers and WBC-based implementations. On the high-performance Ryzen 5600G, AES and SAES complete encryption plus CMAC tag generation for a 10 MB file in about 0.001–0.35 seconds, with DES and 3DES staying below 0.3 seconds as well. In contrast, the CHOW[18] and XIAO[19] white-box AES implementations require roughly 15.1 seconds and 37.4 seconds respectively for the same input size, already about two to three orders of magnitude slower than standard AES. The HARDENED-WBC[20] variant is the most expensive, with its runtime growing from about 59.6 seconds at 512 KB to roughly 1,168.8 seconds (over 19 minutes) at 10 MB, representing another order of magnitude slowdown even relative to CHOW and XIAO.

The same pattern holds on lower-end and embedded-class platforms. On Raspberry Pi 4, Intel N100, and Jetson Nano, conventional ciphers (AES/SAES/DES/3DES) finish within about 1 second for 10 MB inputs, whereas CHOW and XIAO stretch into tens or hundreds of seconds, and HARDENED-WBC rises into the multi-thousand-second range. Although absolute times differ per CPU class, with Ryzen 5600G being fastest and Raspberry Pi 4 generally slowest, the relative ordering of algorithms is consistent. AES ≈ SAES are fastest, followed by DES and 3DES, then CHOW, then XIAO, and finally HARDENED-WBC as the slowest. These results indicate that, while WBC-based integrity protection is functionally applicable across heterogeneous on-device environments, its raw execution-time overhead, especially for hardened designs, must be carefully budgeted, for example by limiting protected payload sizes, amortizing checks, or combining WBC-MAC with lighter, hardware-assisted mechanisms where available.

Table 1: Execution Time of WBC-CMAC Generation using Ryzen 5600G

	SAES	AES	DES	TDES	CHOW[18]	XIAO[19]	HARDENED[20]
512KB	0.018	< 0.001	0.005	0.014	0.74	1.854	59.556
1MB	0.035	< 0.001	0.011	0.028	1.533	3.72	118.534
1.5MB	0.052	< 0.001	0.016	0.041	2.246	5.578	177.482
2MB	0.07	< 0.001	0.021	0.055	2.945	7.439	235.842
2.5MB	0.087	< 0.001	0.027	0.071	3.816	9.283	294.394
3MB	0.104	< 0.001	0.032	0.083	4.421	11.127	352.881
3.5MB	0.122	< 0.001	0.037	0.096	5.117	13.003	411.387
4MB	0.139	< 0.001	0.043	0.11	6.013	14.867	468.581
4.5MB	0.157	< 0.001	0.048	0.124	6.692	16.764	526.942
5MB	0.174	0.001	0.053	0.138	7.421	18.599	584.749
5.5MB	0.192	0.001	0.059	0.151	8.155	20.493	643.088
6MB	0.209	0.001	0.064	0.165	8.922	22.37	702.08
6.5MB	0.227	0.001	0.07	0.179	9.635	24.258	759.573
7MB	0.243	0.001	0.075	0.192	10.628	26.118	817.421
7.5MB	0.262	0.001	0.08	0.206	11.339	28	875.921
8MB	0.279	0.001	0.086	0.22	12.226	29.877	933.69
8.5MB	0.295	0.001	0.091	0.233	12.896	31.677	992.402
9MB	0.313	0.001	0.096	0.247	13.367	33.569	1051.431
9.5MB	0.334	0.001	0.102	0.261	14.515	35.458	1110.242
10MB	0.348	0.001	0.107	0.275	15.138	37.362	1168.786

Table 2: Execution Time of WBC-CMAC Generation using Resberry Pi 4

	SAES	AES	DES	TDES	CHOW[18]	XIAO[19]	HARDENED[20]
512KB	0.068	-	0.018	0.049	3.431	8.953	318.113
1MB	0.079	-	0.036	0.098	6.78	17.736	634.737
1.5MB	0.112	-	0.059	0.146	10.418	27.072	946.382
2MB	0.149	-	0.116	0.195	13.69	35.997	1261.696
2.5MB	0.187	-	0.108	0.244	16.853	45.001	1577.737
3MB	0.224	-	0.109	0.292	20.24	53.984	1881.732
3.5MB	0.262	-	0.127	0.341	23.913	63.071	2192.743
4MB	0.3	-	0.145	0.388	27.518	71.228	2510.339
4.5MB	0.336	-	0.164	0.441	30.33	80.726	2817.812
5MB	0.374	-	0.182	0.488	33.679	90.61	3127.562
5.5MB	0.411	-	0.2	0.58	37.553	98.311	3433.774
6MB	0.446	-	0.218	0.584	38.659	108.358	3747.126
6.5MB	0.485	-	0.236	0.632	45.319	118.023	4056.219
7MB	0.526	-	0.255	0.683	46.571	126.176	4372.424
7.5MB	0.563	-	0.273	0.73	52.093	136.573	4670.239
8MB	0.642	-	0.291	0.78	51.88	143.779	4982.805
8.5MB	0.638	-	0.309	0.828	57.49	153.042	5295.5
9MB	0.671	-	0.327	0.922	56.82	163.334	5609.205
9.5MB	0.714	-	0.345	0.925	62.009	172.031	5916.884
10MB	0.746	-	0.363	0.974	67.393	179.777	6230.552

Table 3: Execution Time of WBC-CMAC Generation using Intel N100

	SAES	AES	DES	TDES	CHOW[18]	XIAO[19]	HARDENED[20]
512KB	0.037	< 0.001	0.006	0.018	1.283	2.849	102.306
1MB	0.071	0.001	0.012	0.037	2.562	5.675	199.376
1.5MB	0.109	0.003	0.019	0.048	3.857	8.512	301
2MB	0.147	0.002	0.025	0.063	5.134	11.431	393.742
2.5MB	0.183	0.002	0.033	0.084	6.425	14.363	466.624
3MB	0.215	0.002	0.038	0.095	7.796	17.082	558.081
3.5MB	0.252	0.002	0.044	0.115	9.019	20.069	650.874
4MB	0.287	0.003	0.049	0.131	10.279	22.805	745.507
4.5MB	0.324	0.003	0.056	0.142	11.589	26.018	852.673
5MB	0.356	0.004	0.061	0.16	12.774	28.572	926.229
5.5MB	0.396	0.004	0.072	0.177	14.215	31.636	1017.801
6MB	0.43	0.004	0.074	0.192	15.407	34.394	1112.793
6.5MB	0.468	0.004	0.081	0.205	16.648	37.202	1207.702
7MB	0.51	0.005	0.089	0.224	18.143	40.336	1303.07
7.5MB	0.537	0.005	0.1	0.239	19.352	43.171	1389.2
8MB	0.578	0.006	0.098	0.256	20.597	46.08	1476.676
8.5MB	0.608	0.006	0.107	0.276	21.881	48.824	1569.024
9MB	0.653	0.007	0.114	0.282	23.125	51.71	1658.027
9.5MB	0.681	0.009	0.118	0.301	24.435	55.103	1751.148
10MB	0.711	0.009	0.124	0.317	26.144	57.751	1840.78

Table 4: Execution Time of WBC-CMAC Generation using Jeson Nano

	SAES	AES	DES	TDES	CHOW[18]	XIAO[19]	HARDENED[20]
512KB	0.046	< 0.001	0.022	0.051	3.562	6.236	299.905
1MB	0.093	0.001	0.043	0.102	7.6	12.472	587.76
1.5MB	0.139	0.001	0.065	0.153	11.071	18.721	890.78
2MB	0.187	0.002	0.086	0.206	15.278	24.999	1159.26
2.5MB	0.231	0.002	0.108	0.256	18.654	31.192	1450.561
3MB	0.279	0.003	0.129	0.309	22.293	37.411	1737.559
3.5MB	0.324	0.003	0.151	0.358	26.501	43.716	2014.671
4MB	0.37	0.004	0.172	0.411	29.728	49.982	2303.321

4.5MB	0.416	0.004	0.193	0.461	34.538	56.301	2592.469
5MB	0.463	0.005	0.216	0.511	37.395	62.417	2874.981
5.5MB	0.509	0.005	0.237	0.562	41.319	68.733	3178.318
6MB	0.555	0.006	0.258	0.614	45.115	74.981	3449.922
6.5MB	0.607	0.006	0.28	0.675	48.272	81.232	3718.618
7MB	0.648	0.007	0.301	0.716	52.604	87.449	4009.309
7.5MB	0.695	0.007	0.323	0.767	56.302	93.717	4294.604
8MB	0.74	0.008	0.347	0.819	59.859	99.994	4591.454
8.5MB	0.786	0.008	0.366	0.871	63.958	106.165	4864.406
9MB	0.832	0.009	0.444	0.919	67.314	112.451	5137.22
9.5MB	0.879	0.009	0.408	0.971	71.39	118.747	5419.393
10MB	0.925	0.01	0.43	1.023	74.891	124.969	5730.347

4.2 Memory Overheads Measured in Benchmark Test Environment

The benchmark test results are presented in Table 5-8. Across all benchmark platforms, the memory-consumption results show that conventional ciphers occupy only a small and slowly growing working set, whereas WBC-based implementations require substantially more memory, dominated by their lookup tables and obfuscation logic. On the Ryzen 5600G, for example, AES and SAES use roughly 4–14 KB of memory as the input size increases from 512 KB to 10 MB, with DES and 3DES remaining in a similar range. In contrast, the CHOW[18] implementation grows from about 4.5 KB at 512 KB input to roughly 29.5 KB at 10 MB, XIAO[19] from about 48.9 KB to 105.7 KB, and the HARDENED-WBC[20] variant from about 11.7 KB to 73.7 KB over the same range. Similar patterns appear on Raspberry Pi 4, Intel N100, and Jetson Nano, classical ciphers stay within roughly 2–14 KB, while CHOW stabilizes around 25–31 KB, XIAO around 80–105 KB, and HARDENED-WBC between about 34–74 KB depending on the platform and input size.

The results also indicate that memory usage increases only moderately with input size for all algorithms, confirming that the dominant factor is the algorithm's internal state and table allocation rather than the data size itself. Among the WBC implementations, XIAO consistently shows the highest memory footprint, reflecting its larger and more complex table structure, while HARDENED-WBC typically occupies less memory than XIAO but still several times more than CHOW and an order of magnitude more than standard AES. Across platforms, this translates into a clear trade-off, adopting WBC-based integrity protection, especially stronger designs like XIAO and HARDENED-WBC, requires provisioning tens of kilobytes of additional memory per protected process, which is acceptable on desktop-class CPUs but may become a constraining factor on embedded or edge devices with tight RAM budgets. These measurements therefore provide concrete upper bounds and design guidelines for integrating WBC-MAC into on-device AI deployments where both execution time and memory footprint must be carefully controlled.

Table 5: Memory Consumption of WBC-CMAC Generation using Ryzen 5600G

	SAES	**AES**	**DES**	**TDES**	**CHOW[18]**	**XIAO[19]**	**HARDENED[20]**
512KB	4072	4032	2568	2572	4540	48860	11712
1MB	4380	4364	3148	3088	5840	52196	16028
1.5MB	5008	4856	3600	3600	7528	55468	19916
2MB	5424	5656	4176	4112	8740	58692	23640
2.5MB	5952	5992	4620	4624	10028	61484	29436
3MB	6520	6716	5140	5140	11320	64824	32464
3.5MB	7096	7124	5964	5996	12608	68112	33888
4MB	7632	7644	6508	6488	13904	71028	41308
4.5MB	8228	8156	6996	6992	15216	73896	45392
5MB	8476	8300	7492	7508	16504	77148	44988
5.5MB	8952	8976	8036	8108	17816	80580	51668
6MB	9508	9504	8544	8576	19088	82728	56716

6.5MB	9932	10032	9060	9096	20364	85980	54348
7MB	10516	10536	9548	9584	21720	89100	53904
7.5MB	11084	11020	10156	10116	23080	92180	60600
8MB	11656	11588	10580	10632	24276	95240	61772
8.5MB	12140	11992	11096	11140	25596	98452	63144
9MB	12592	12600	11596	11676	26828	100952	64492
9.5MB	13224	13132	12168	12108	28208	103124	68368
10MB	13764	13728	12628	12672	29472	105668	73700

Table 6: Memory Consumption of WBC-CMAC Generation using Rasberry Pi4

	SAES	**AES**	**DES**	**TDES**	**CHOW[18]**	**XIAO[19]**	**HARDENED[20]**
512KB	3360	-	2296	2300	4368	49188	12048
1MB	3808	-	2808	2812	5648	52644	15128
1.5MB	4320	-	3320	3324	7056	56484	18444
2MB	4896	-	3832	3836	8336	58404	22052
2.5MB	5408	-	4344	4348	9616	61092	24588
3MB	5912	-	4856	4860	11024	63908	27464
3.5MB	6432	-	5496	5372	12304	67620	30644
4MB	6928	-	6008	6012	13584	88868	33696
4.5MB	7456	-	6648	6524	14864	73124	36408
5MB	7960	-	7032	7036	16144	76068	39424
5.5MB	8480	-	7544	7548	17424	78628	42184
6MB	8992	-	8056	8188	18704	81828	44928
6.5MB	9504	-	8696	8572	20112	84388	47916
7MB	10016	-	9208	9212	21520	87076	50560
7.5MB	10528	-	9720	9724	22800	90660	53800
8MB	11040	-	10232	10236	23952	92708	56668
8.5MB	11552	-	10744	10748	25232	95908	59228
9MB	12064	-	11256	11260	26256	98596	62668
9.5MB	12560	-	11768	11772	27792	101284	66012
10MB	13088	-	12152	12156	28944	103844	68252

Table 7: Memory Consumption of WBC-CMAC Generation using Intel N100

	SAES	**AES**	**DES**	**TDES**	**CHOW[18]**	**XIAO[19]**	**HARDENED[20]**
512KB	4096	4096	2176	1664	3840	50432	16688
1MB	4608	4608	2688	1664	4352	53348	18968
1.5MB	5120	5120	3200	1536	4992	53960	19188
2MB	5632	5632	3712	1664	5632	54656	21848
2.5MB	6144	6144	4096	1664	6016	55292	27984
3MB	6656	6656	4736	1664	6656	56704	27616
3.5MB	7168	7168	5248	1920	7168	57600	29172
4MB	7680	7680	6016	1792	7680	62008	29860
4.5MB	8192	8192	6400	1920	8064	62660	30764
5MB	8704	8704	7040	1920	8448	63844	31876
5.5MB	9216	9216	7424	1920	9344	63620	31948
6MB	9728	9728	7936	1920	9728	64828	33004
6.5MB	10240	10112	8448	1792	10112	66292	34124
7MB	10752	10752	9088	1920	10624	65840	34304
7.5MB	11264	11264	9472	1792	11264	66628	35008
8MB	11776	11776	9984	1920	11648	67072	37636
8.5MB	12288	12288	10624	1664	12416	68644	36460
9MB	12800	12800	11008	1792	12800	71128	37772
9.5MB	13312	13312	11520	2048	13312	73884	38452
10MB	13824	13824	12160	1792	13696	74720	38544

Table 8: Memory Consumption of WBC-CMAC Generation using Jeson Nano

	SAES	AES	DES	TDES	CHOW[18]	XIAO[19]	HARDENED[20]
512KB	2952	2936	3944	3944	6088	33124	17164
1MB	3460	3436	5980	5988	8216	39480	21252
1.5MB	3944	3952	5980	5992	8508	46024	25348
2MB	4460	4484	5980	5992	10560	45976	25608
2.5MB	5260	4964	5984	5988	12632	48140	31620
3MB	5680	5432	8036	8084	12640	54444	31804
3.5MB	6144	6080	8340	8340	14700	64932	36220
4MB	6664	6640	8324	8340	14704	67104	40676
4.5MB	7120	7108	8352	10404	16772	69156	41000
5MB	7580	7800	10376	10388	18828	71260	45268
5.5MB	8096	8192	10380	10400	18844	75424	47328
6MB	8612	8536	10396	10400	20904	76400	49600
6.5MB	9112	9068	10392	10408	22972	79632	52828
7MB	9536	9572	12432	12452	22984	81844	55896
7.5MB	10332	10108	12432	12444	25044	83948	57436
8MB	10640	10700	12424	12444	25064	88080	60184
8.5MB	11260	11096	12436	14500	27100	88088	64112
9MB	11732	11624	14476	14492	29184	92324	68320
9.5MB	12192	12348	14480	14496	29204	94416	69868
10MB	12772	12620	14476	14488	31264	96476	73596

5 Requirements for WBC-based Integrity Checks for On-Device AI Protection

In this section, functional and non-functional requirements derived from the analysis of WBC and on-device AI are discussed. Functional requirements concern the capabilities needed to perform on-device AI integrity verification, whereas non-functional requirements relate to the required performance and other quality attributes.

5.1 Functional requirements for WBC-based Integrity Check

- **On-device integrity verification for AI models and data:** The system shall provide an integrity verification mechanism to protect AI models and related data executed in on-device environments. On-device AI models are vulnerable to tampering. A built-in integrity mechanism is therefore the core function that ensures the AI pipeline is operating on unmodified, trustworthy artifacts.

- **White-box–based protection of keys and internal structure:** The system shall use a white-box cryptography–based encryption or MAC module so that keys and structural information embedded in the AI model are not exposed through reverse engineering or static/dynamic analysis. In white-box attack settings, an adversary can fully inspect binaries and memory. Traditional crypto leaves keys briefly in plaintext; WBC embeds them into obfuscated tables and logic, raising the bar for key extraction and structural analysis.

- **Device-bound control of the integrity verification module:** The system shall provide a control mechanism that allows the integrity verification module to run only on authorized devices (e.g., based on a device identifier or secure binding). Without device binding, protected models and verification code could be copied wholesale to unauthorized devices. Device-level control supports licensing, IP protection, and prevents offline abuse on cloned hardware.

- **Built-in performance measurement and comparison:** The system shall provide functionality to measure and compare performance (e.g., memory usage and execution time) before and after applying the protection techniques. WBC-based integrity adds non-trivial overhead. Designers need visibility into how

much cost is introduced in specific deployments so they can tune policies (e.g., verification frequency, choice of WBC variant) and ensure SLAs are still met.

- **Identification and management of integrity-critical assets:** The system shall provide functionality to identify and manage critical assets subject to integrity verification, such as model parameters, weights, configuration files, and internal computation blocks. Not all artifacts require the same level of protection. Being able to explicitly mark and manage "integrity-critical" components allows more focused and efficient use of WBC-MAC, reducing overhead while still protecting what matters most.

- **Assessment of obfuscation and reuse resistance:** The system shall provide evaluation functionality to measure reuse resistance when model obfuscation or similar protection techniques are applied to AI models. One key goal of on-device protection is to prevent unauthorized reuse or repackaging of protected models. An explicit assessment function (e.g., measuring how easy it is to extract or transplant model components) allows quantitative evaluation of obfuscation strength.

- **Logging and reporting of integrity verification results:** The system shall be able to store and transmit on-device integrity verification results in the form of logs or reports. Auditability is essential for security operations, compliance, and incident response. Persisting verification outcomes and, optionally, sending them to a backend enables post-mortem analysis and continuous monitoring.

- **Response actions on verification failure:** The system shall be able to trigger warning and blocking procedures (e.g., halting execution, degrading service, or notifying administrators) when integrity verification fails. Detection without reaction is insufficient. When tampering is detected, the system must enforce a clear response policy to prevent further damage, stop unauthorized use, and alert responsible stakeholders.

- **CMAC-based structure with a white-box cipher core:** The system shall be implemented on a CMAC structure, where the underlying block cipher is replaced by a white-box cryptographic implementation. CMAC is a well-studied, standardized MAC construction with clear security properties. Replacing only the block-cipher core with a WBC implementation allows reuse of CMAC's proven design while upgrading the key-handling component to a white-box setting.

- **Local, offline integrity verification without external CAs:** The system shall complete integrity verification locally, without requiring communication with external authentication servers or CAs (Certificate Authorities). On-device AI often runs in disconnected or intermittently connected environments. Requiring online CA access is unrealistic; verification must be self-contained, with all necessary trust material embedded or provisioned locally.

- **Binary-level handling of diverse AI model formats:** The system shall be capable of processing various on-device AI model formats at the binary level. In practice, models are deployed in many formats (e.g., TFLite, ONNX, proprietary binaries). Tying integrity verification to a specific framework API would limit applicability; operating at the binary level makes the mechanism framework-agnostic and suitable for heterogeneous model ecosystems.

- **White-box security against key extraction:** The system shall guarantee white-box security in the sense that, even if an attacker gains full control of the device and performs memory dumps, secret keys are never present in cleartext and cannot be extracted. The core motivation for using WBC is to protect keys in environments where the adversary can fully observe execution. This requirement forces all integrity-critical keys to be fused into obfuscated tables and logic, ruling out naïve implementations that merely "wrap" a conventional cipher and still leak keys in RAM or registers.

- **No dependence on TEEs or dedicated security chipsets:** The system shall be capable of performing integrity verification even on general-purpose processors that do not provide TEEs (e.g., TrustZone) or dedicated security chipsets. Many low-end or legacy devices lack TrustZone, SGX, or external secure

elements. This requirement maintains the design goal that WBC-MAC must stand alone as a software-only integrity mechanism, particularly in environments where hardware isolation is unavailable or untrusted.

5.2 None-Functional requirements for WBC-based Integrity Check

- **Low computational overhead on embedded / IoT devices:** The system shall incur sufficiently low computational overhead to operate on resource-constrained embedded and IoT devices, taking into account the higher cost of WBC-MAC compared to conventional ciphers. WBC-MAC is orders of magnitude slower than AES/CMAC, so integrity checks cannot be assumed "free." This requirement ensures that, even with WBC-based protection, overall verification latency remains compatible with practical deployment constraints for edge and IoT devices, instead of being limited to server-class hardware only.

- **Pure software operation on general-purpose processors:** The system shall operate purely in software on general-purpose processors, without depending on specific hardware security modules. Many on-device AI deployments (legacy devices, low-cost boards, or virtualized environments) do not have access to TEEs, secure elements, or proprietary accelerators. Requiring only a CPU and system memory makes the proposed WBC-based integrity verification broadly deployable across heterogeneous platforms.

- **Minimal impact on AI model quality:** After protection is applied, the system shall not cause significant degradation in AI model performance (e.g., accuracy and inference latency); model accuracy loss shall be kept within ±1 percentage point. Integrity protection must not invalidate the original purpose of the AI model. This constraint ensures that any embedding of cryptographic checks, code wrapping, or obfuscation does not materially change prediction quality or lead to unacceptable slowdowns in normal inference paths.

- **Execution-time bound on Raspberry Pi 4–class devices:** On Raspberry Pi 4–class embedded environments, integrity verification for a 1 MB AI model using a lightweight white-box algorithm (e.g., CHOW[18] or XIAO[19]) shall complete within 20 seconds. Benchmark results show roughly 6.8 seconds for CHOW-CMAC and 17.7 seconds for XIAO-CMAC at 1 MB, so a 20-second upper bound is realistic while still providing a safety margin. This requirement defines a concrete performance target for "small to medium" models on typical IoT boards.

- **Linear scalability with model size:** As the data size increases, verification time shall scale approximately linearly with the model size, and no abrupt exponential performance degradation shall occur. Experimental results show that CHOW-CMAC[18] time grows roughly in proportion to input size (e.g., from ~3.4 s at 512 KB to ~67.4 s at 10 MB). Requiring near-linear scaling avoids pathological behaviors (e.g., cache or paging blow-ups) that would make WBC-MAC unusable for larger on-device models.

- **Peak memory usage bound (≤ 100 MB) on edge devices:** When verifying models of up to 10 MB on Jetson Nano and Raspberry Pi 4, the peak memory usage of the integrity verification module shall remain below 100 MB. Benchmarks show that even the heaviest WBC variants stay in the tens of megabytes, which fits comfortably within typical IoT RAM budgets (1–4 GB). This requirement codifies a conservative upper bound that leaves room for the AI runtime and other processes while accounting for WBC's larger table and code footprint.

- **Adaptive algorithm selection based on RAM constraints:** Given that stronger WBC implementations such as XIAO[19] and HARDENED-WBC[20] exhibit roughly 2–4× higher memory usage than lightweight implementations like CHOW, the system shall be able to select algorithms dynamically based on available device RAM or enforce configurable memory thresholds. Measurements show that CHOW, XIAO, and HARDENED-WBC occupy noticeably different memory footprints. This requirement acknowledges that not all devices can afford the heaviest variant and mandates a policy layer that picks an appropriate WBC engine per device profile.

- **Cross-architecture software portability (x86_64 and ARM):** The system shall function correctly in pure software on both x86_64 architectures (e.g., Ryzen, Intel N100) and ARM architectures (e.g., Raspberry Pi 4, Jetson Nano), without relying on dedicated hardware accelerators. The benchmark suite explicitly covers both desktop-class x86_64 and embedded ARM boards, and the same WBC implementations are expected to run on all of them. This requirement ensures that the integrity framework can be reused across data-center, edge, and deeply embedded deployments.

- **Use in non–real-time integrity scenarios:** The WBC-MAC approach, being significantly slower than conventional public-key digital signatures (e.g., ECDSA/RSA verification), shall be designed for scenarios where strict real-time constraints are not required, such as one-time verification at boot, verification during update, or periodic integrity checks. The emphasis shall be on key protection and resistance to white-box attacks rather than low-latency verification. ECDSA verification on Raspberry Pi 4 completes in milliseconds, whereas WBC-MAC for 1 MB may take several seconds. This requirement clarifies the intended use cases: WBC-MAC is for high-assurance integrity in white-box settings, not for per-request, latency-sensitive checks.

- **Maintained white-box property despite higher memory usage:** Even if WBC-MAC consumes more memory at runtime than typical digital-signature verification, the white-box property—ensuring that the encryption key is never exposed in plaintext—shall be preserved. The additional memory cost due to obfuscated tables and logic shall be treated as an acceptable security–performance trade-off, provided that overall peak memory usage remains within the specified bounds. Benchmarks confirm that WBC-MAC uses significantly more memory than ECDSA verification due to large, obfuscated tables. This requirement explicitly accepts that overhead as long as it stays within the defined limits and continues to deliver the main benefit of WBC: robust key protection in hostile on-device environments.

6 Conclusion

In our research, we investigated White-Box Cryptography (WBC) as a software-only basis for integrity verification of on-device AI models deployed on resource-constrained and potentially hostile platforms. Traditional hash- and MAC-based integrity mechanisms assume that keys and verification logic remain hidden from adversaries, an assumption that fails when attackers can fully inspect firmware and binaries. To address this, we proposed a WBC-based CMAC framework in which the block-cipher core is implemented as a white-box primitive, embedding keys into heavily obfuscated tables and logic so that integrity verification is still possible under full white-box attacker capabilities. We instantiated concrete WBC-based integrity approaches and benchmarked the WBC-based integrity check approaches against conventional block ciphers on four representative hardware platforms, from Ryzen 5600G to Raspberry Pi 4 and Jetson Nano. The results show that WBC-based integrity checks incur one to three orders of magnitude higher execution time and require tens of kilobytes of additional memory per protected process, while still scaling roughly linearly with input size. In addition, Based on algorithmic analysis and these benchmarks, we derived functional and non-functional requirements for deploying WBC-based integrity verification in on-device AI systems. The functional requirements specify capabilities such as on-device integrity checks for models and data, white-box protection of keys, device-bound control, logging and response policies, and binary-level handling of diverse model formats. The non-functional requirements define acceptable bounds for execution time and memory usage, cross-architecture portability, minimal impact on model quality, and a focus on non–real-time integrity scenarios. We expect that this requirement-oriented perspective and the provided benchmark data will serve as a practical foundation for designing and deploying more robust, white-box–aware integrity protections in future on-device AI systems.

Acknowledgements

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government (MSIT) (No.RS-2025-02215590, Development of AI

implementation obfuscation technology to prevent information leakage in On-Device AI)

References

[1] Wang, X., Tang, Z., Guo, J., Meng, T., Wang, C. H., Wang, T., & Jia, W. (2025). Empowering edge intelligence: A comprehensive survey on on-device AI models. ACM Computing Surveys, 57(9), 1–39.

[2] Heydari, S., & Mahmoud, Q. H. (2025). Tiny machine learning and on-device inference: A survey of applications, challenges, and future directions. Sensors, 25(10), 3191.

[3] Palo Alto Networks. (2025). AI model security: What it is and how to implement it. https://www.paloaltonetworks.com/cyberpedia/what-is-ai-model-security

[4] Yang, J., He, Q., Zhou, Z., Dai, X., Chen, F., Tian, C., & Yang, Y. (2025). EdgeThemis: Ensuring model integrity for edge intelligence. In Proceedings of the ACM Web Conference 2025 (WWW '25) (pp. 3136–3146).

[5] Chow, S., Eisen, P., Johnson, H., & van Oorschot, P. C. (2003). White-box cryptography and an AES implementation. In Selected Areas in Cryptography (SAC 2002) (Lecture Notes in Computer Science, Vol. 2595, pp. 250–270). Springer.

[6] Chow, S., Eisen, P., Johnson, H., & van Oorschot, P. C. (2003). A white-box DES implementation for DRM applications. In J. Feigenbaum (Ed.), Digital Rights Management (DRM 2002) (Lecture Notes in Computer Science, Vol. 2696, pp. 1–15). Springer.

[7] Wyseur, B. (2012). White-box cryptography: Hiding keys in software. NAGRA Kudelski Group.

[8] Albricci, D. G. V., Ceria, M., Cioschi, F., Fornari, N., Shakiba, A., & Visconti, A. (2019). Measuring performances of a white-box approach in the IoT context. Symmetry, 11(8), 1000.

[9] Shi, Y., Li, Y., Ouyang, Q., Gao, J., & Zhao, S. (2025). LSTable: A new white-box cipher for embedded devices in IoT against side-channel attacks. IEEE Transactions on Emerging Topics in Computing, 13(3), 1242–1258.

[10] Liu, R., Garcia, L., Liu, Z., Ou, B., & Srivastava, M. (2021, May). SecDeep: Secure and performant on-device deep learning inference framework for mobile and IoT devices. In Proceedings of the International Conference on Internet-of-Things Design and Implementation (pp. 67-79).

[11] Moon, M., Kim, M., Jung, J., & Song, D. (2025). ASGARD: Protecting On-Device Deep Neural Networks with Virtualization-Based Trusted Execution Environments. In Proceedings 2025 Network and Distributed System Security Symposium.

[12] Shen, T., Qi, J., Jiang, J., Wang, X., Wen, S., Chen, X., ... & Cui, H. (2022). {SOTER}: Guarding black-box inference for general neural networks at the edge. In 2022 USENIX Annual Technical Conference (USENIX ATC 22) (pp. 723-738).

[13] Xie, X., Wang, H., Jian, Z., Li, T., Wang, W., Xu, Z., & Wang, G. (2024, May). Memory-efficient and secure dnn inference on trustzone-enabled consumer iot devices. In IEEE INFOCOM 2024-IEEE Conference on Computer Communications (pp. 2009-2018). IEEE.

[14] Mo, F., Shamsabadi, A. S., Katevas, K., Demetriou, S., Leontiadis, I., Cavallaro, A., & Haddadi, H. (2020, June). Darknetz: towards model privacy at the edge using trusted execution environments. In Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services (pp. 161-174).

[15] Tramer, F., & Boneh, D. (2018). Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. arXiv preprint arXiv:1806.03287.

[16] Chen, H., Fu, C., Rouhani, B. D., Zhao, J., & Koushanfar, F. (2019, June). DeepAttest: An end-to-end attestation framework for deep neural networks. In Proceedings of the 46th International Symposium on Computer Architecture (pp. 487-498).

[17] Rouhani, B. D., Chen, H., & Koushanfar, F. (2019, April). Deepsigns: an end-to-end watermarking framework for protecting the ownership of deep neural networks. In ACM International Conference on Architectural Support for Programming Languages and Operating Systems (Vol. 3, p. 1).

[18] Chow, S., Eisen, P., Johnson, H., & Van Oorschot, P. C. (2002, August). White-box cryptography and an AES implementation. In International Workshop on Selected Areas in Cryptography (pp. 250-270). Berlin, Heidelberg: Springer Berlin Heidelberg.

[19] Xiao, Y., & Lai, X. (2009, December). A secure implementation of white-box AES. In 2009 2nd International Conference on Computer Science and its Applications (pp. 1-6). IEEE.

Author's Biography

Hanbin Lee received his B.S. degree in Artificial Intelligence Cybersecurity from Korea University (Sejong Campus) in 2024. He is currently pursuing his M.S. degree in the Department of Cybersecurity at the same university. His primary research interests include RF signal analysis, anomaly detection, and ROS-based data analysis.

Junyoung Cho received his B.S. degree in Artificial Intelligence Cybersecurity from Korea University (Sejong Campus) in 2024. He is currently pursuing his M.S. degree in the Department of Cybersecurity at the same university. His primary research interests include RF signal analysis, anomaly detection, and ROS-based data analysis.

TaeGuen Kim received his B.S. degree in Electronics and Computer Engineering in 2011 and his M.S. degree in Computer and Software Engineering in 2013, both from Hanyang University, South Korea. He earned his Ph.D. degree in Computer and Software Engineering from the same university in 2018. He subsequently worked as a Senior Research Engineer at Hyundai Motor Company (HMC), followed by an Assistant Professor position at Soonchunhyang University. Since September 2024, he has been serving as a faculty member at Korea University (Sejong Campus). His research interests include malware analysis, AI-driven security solutions, and automotive cybersecurity.