This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections.
Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use
https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms

Keymix: Protecting Large Resources from Key Exfiltration

Marco Abbadini[1], Michele Beretta[1], Sabrina De Capitani di Vimercati[2], Dario Facchinetti[1], Sara Foresti[2], Stefano Paraboschi[1], Matthew Rossi[1], and Pierangela Samarati[2]

[1] Department of Management, Information and Production Engineering, Università degli Studi di Bergamo, Dalmine 24044, Italy
{marco.abbadini,michele.beretta,dario.facchinetti, stefano.paraboschi,matthew.rossi}@unibg.it
[2] Department of Computer Science, Università degli Studi di Milano, Milan 20122, Italy
{sabrina.decapitani,sara.foresti,pierangela.samarati}@unimi.it

Abstract. When transmitted over public networks, high-value resources must be protected against unauthorized access. Although traditional encryption schemes provide confidentiality in transit, they offer no guarantees if the encryption key is exfiltrated through side-channel attacks or other means. Leakage-resilient schemes aim to remain secure under partial key leakage; however, existing constructions require strong assumptions on leakage timing or restrict the adversary to inaccessible memory regions, rendering them impractical for real deployments. In this paper, we introduce Keymix, a novel encryption scheme designed to protect large resources in the presence of key exfiltration. Keymix supports encryption with a configurable big-key $\mathcal{K}$ (up to GiBs in size), ensuring that a plaintext $\mathcal{R}$ (with $|\mathcal{R}| \geq |\mathcal{K}|$) cannot be fully recovered unless the $\mathcal{K}$ is fully exfiltrated, even when the adversary possesses the complete ciphertext. Unlike prior work, security depends solely on the exfiltrated portion of $\mathcal{K}$, without assumptions on leakage timing or inaccessible memory regions. We implement Keymix using AES and achieve a throughput of 1.6 GiB/s on commodity hardware. Moreover, we demonstrate its practicality through integration with modern cloud infrastructure inside a Kubernetes cluster, where it transparently applies exfiltration-resistant encryption to storage blocks during service restore and backup.

Keywords: Key Exfiltration Resistance, Leakage-resilient Cryptography, Big-key Encryption, Symmetric Encryption, Cloud Storage Security

1 Introduction

Modern cloud-native architectures rely on the continuous movement of sensitive workloads across network boundaries. Operations such as service deployment [46, 48], workload migration, and disaster recovery require transferring a

large amount of highly valuable data between clusters (e.g., proprietary machine learning models or confidential backups). To protect these assets in transit and distributed storage [7], organizations rely on standard symmetric encryption schemesi [25, 42] that provide strong confidentiality using compact keys. Although these cryptographic primitives are secure, the underlying infrastructure is far from perfect. Real-world systems are often compromised through vulnerabilities, side-channel attacks, or supply chain flaws [2,17,51]. In this context, the compactness of standard encryption keys becomes a critical issue. Even under strict network isolation and monitoring [10], where exfiltrating massive datasets would immediately trigger alarms, a stealthy attacker can easily leak a 256-bit key unnoticed. Once the key is stolen, the adversary can capture the encrypted workload from the network and easily decrypt it offline. To mitigate this threat, the cryptographic community has extensively studied security under partial key leakage. The *Bounded Retrieval Model* (BRM) [8, 19, 26] formalizes scenarios where an adversary can extract only a limited amount of information. Building on this foundation, *leakage-resilient* schemes [14, 15, 33] employ massive keys that are inherently harder to completely exfiltrate. By increasing the key size, these schemes raise the threshold for unnoticed exfiltration, giving defenders time to detect anomalies and interrupt the attack. However, existing constructions have practical limitations, often assuming leakage occurs only before encryption or restricting probing to specific memory regions, and none encrypts resources without first sampling the entire key.

To address these limitations, we introduce *Keymix*, a complementary symmetric encryption layer applied before network transfer to protect valuable multi-cluster resources against key exfiltration. Keymix uses configurable large keys (called big-keys) and ensures that an adversary with a full encrypted resource $\mathcal{R}_e$ (with $|\mathcal{R}| \geq |\mathcal{K}|$) cannot fully recover the plaintext unless the entire big-key is exfiltrated. To achieve this, our approach enhances butterfly networks [22] to guarantee complete interdependency of the bits within the big-key, resisting partial exfiltration. Unlike prior work, the security model of Keymix depends solely on the amount of exfiltrated information, avoiding restrictive assumptions about adversary probing. To summarize, we make the following contributions. First, we explain why existing encryption techniques fail to provide protection in the exfiltration scenario. Second, we propose Keymix, an exfiltration-resistant symmetric encryption primitive that supports arbitrarily large big-keys and whose security depends only on the amount of information exfiltrated. We provide an AES-based implementation of Keymix reaching throughputs up to 1.6 GiB/s on commodity hardware. Finally, we demonstrate a concrete integration of Keymix inside a Kubernetes cluster, where it transparently applies exfiltration-resistant encryption to storage blocks during service backup and restore.

2 Background on Data Exfiltration

Implementation flaws in widely deployed software remain a primary breach vector. For example, Heartbleed [24] enabled remote memory disclosure. Even with-

out explicit bugs, cryptographic implementations can leak secrets via timing side-channels. Kocher [36] and subsequent research [17] demonstrated that minor variations in execution latency suffice to recover private keys. Mitigating these channels requires deep expertise in both algorithmic design and compiler behavior [31]. Microarchitectural attacks on modern CPUs provide yet another vector. Transient-execution vulnerabilities (e.g., Spectre [35], Meltdown [38]) and cache-timing techniques [51] allow adversaries to bypass isolation by observing microarchitectural states. Despite hardware and OS mitigations, residual leakage channels persist [27, 31], enabling the gradual exfiltration of confidential data [35]. Physical side-channels (including electromagnetic [45], power [34], and resource contention signals [28]) can also exfiltrate data, allowing to reliably extract small secrets even when software sandboxing [3–6, 47] and hardware defenses are active [49]. All mentioned vulnerabilities expose a fundamental limitation of conventional encryption: recovering even a fraction of a compact secret key compromises the confidentiality of large resources. This motivates Keymix, a solution whose security solely depends on the volume of exfiltrated information.

3 Requirements and Existing Solutions

3.1 Threat Model

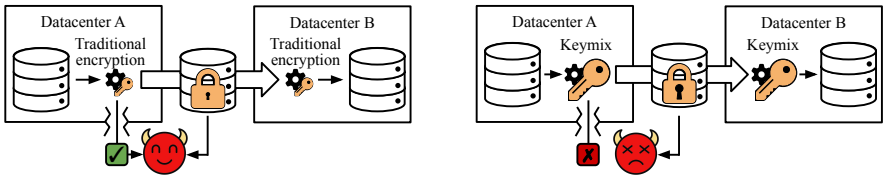
We assume a company deploying workloads across monitored datacenters. Communications among datacenters are protected by encryption. Despite best practices, covert data exfiltration remains a realistic threat (as explained in Section 2). Consistent with the Bounded Retrieval Model (BRM) [8, 19, 26], we assume an attacker can extract information until the exfiltrated volume reaches a threshold ℓ, triggering detection and channel termination. Unlike prior works, we impose no restrictions on the compromise method, attack timing, or the presence of inaccessible memory regions. The adversary's goal is to exfiltrate the key to decrypt resources intercepted over the public network. Specifically, we target one-wayness (unrecoverability of the full plaintext rather than indistinguishability) up to an effective threshold $\min(|\mathcal{K}|, |\mathcal{R}|)$. Thus, for large resources ($|\mathcal{R}| \geq |\mathcal{K}|$), security depends solely on $|\mathcal{K}|$ relative to ℓ: larger keys increase the likelihood of detection before sufficient material is extracted (Figure 1).

3.2 Requirements

Any solution that aims to secure resources in the presence of data exfiltration as presented in Section 3.1 must satisfy the following three requirements.

$\mathsf{R_1}$ Configurable big-key size. The scheme must support arbitrarily large encryption big-keys, allowing the user to increase the exfiltration bound ℓ according to the sensitivity of the resource and the type of protected workload.

$\mathsf{R_2}$ Complete big-key dependency. Any bit of the encrypted resource must depend on every bit of the big-key. This prevents selecting smaller, insecure encryption keys.



(a) Traditional encryption schemes (b) Keymix encryption scheme

Fig. 1: In the exfiltration scenario, the adversary is able to extract compact keys used to protect resources shared among datacenters. Larger keys increase the likelihood that the adversary is detected, and the covert exfiltration channel is interrupted before sufficient key material is extracted.

R_3 No-shrinking effect. The output of the encryption process, as well as any intermediate results or observable states during computation, must be at least as large as the exfiltration bound ℓ. This ensures that an adversary must exfiltrate at least ℓ bits of information to recover the plaintext, with no possibility of compressing the entropy of the big-key into a significantly smaller intermediate value that would be easier to extract.

3.3 Existing solutions

In the following we discuss the advantages, drawbacks, and degree of satisfaction of the requirements of existing solutions. Table 1 presents a summary.

One-Time Pad. The One-Time Pad provides perfect secrecy by XOR-ing the plaintext with a key of equal length. It naturally satisfies R_1✓ and R_3✓ because the key must be as large as the resource. However, it violates R_2✗ because XOR is applied bitwise. Moreover, each key is single-use, which imposes significant key-management overhead when resources are updated.

Table 1: Requirements satisfaction summary.

Approach	R_1	R_2	R_3
One-Time Pad	✓	✗	✓
AEAD [25, 42]	✗	-	-
Big-key [14, 15]	✓	✗	✗
TPM	✗	-	-
TEE [9, 11]	✓	-	-
Our proposal	✓	✓	✓

✓ = yes ✗ = no - = n/a

AEAD. Authenticated Encryption with Associated Data (AEAD) schemes simultaneously provide confidentiality, integrity, and authenticity for data in transit. The two most widely deployed constructions are AES-GCM [25] and ChaCha20-Poly1305 [42]. The former combines the AES block cipher in CTR mode with the GHASH hash function for authentication, and the latter pairs the ChaCha20 stream cipher with the Poly1305 one-time MAC. While extremely performant and secure when the key remains secret, they do not support configurable large keys (R_1✗): once the compact key is exfiltrated, all past and future ciphertexts can be decrypted.

Big-key encryption. Leakage-resilient schemes [33] address partial key leakage by using large keys. Early works stored the key in an expanded redundant

form [18, 23]. Later works formalized the setting under the Bounded Retrieval Model [8, 19, 21, 26], allowing an adversary to obtain up to ℓ bits of information. Within this line of research, Bellare et al. [14, 15] proposed two symmetric schemes that derive a short unpredictable key from a big-key via random sampling and encapsulation, then use a standard cipher for encryption. While achieving indistinguishability under pre-encryption leakage and satisfying $\mathbf{R_1}$✓, they fail $\mathbf{R_2}$✗ and $\mathbf{R_3}$✗ because the derived key is compact and independent of the full big-key (whereas Keymix trades subkey sampling for full-memory exfiltration resistance by mixing the entire big-key). Furthermore, these constructions assume leakage occurs only before encryption or is restricted to the big-key, limiting their practical applicability under our threat model (Section 3.1).

TPM. Trusted Platform Modules (TPMs) provide hardware storage for cryptographic keys and certificates, performing sensitive operations in an environment isolated from the main processor. While this mitigates direct memory probing, TPMs have limited storage capacity, hence cannot store arbitrarily large big-keys ($\mathbf{R_1}$✗). Moreover, recent research demonstrated that even TPMs remain vulnerable to exfiltration attacks [30, 40].

TEE. Trusted Execution Environments (TEEs), such as Intel SGX, enable the execution of sensitive workloads and cryptographic operations within hardware-isolated enclaves. By storing large keys in the enclave memory and allowing the implementation of custom protocols (e.g., TaLoS [11], SCONE [9]), TEEs support configurable big-key size ($\mathbf{R_1}$✓). Nevertheless, they demand extensive re-implementation of services to use the enclave APIs, and are available only on selected CPU platforms (e.g., Intel Xeon). Moreover, recent works have shown that TEEs are not immune to exfiltration attacks [41], confirming the need for exfiltration-resistant schemes such as Keymix.

4 Design

Keymix is a symmetric stream cipher that derives a resource-specific keystream from a big-key $\mathcal{K}$. The key is stored inside the protected datacenter and its size is user-configurable ($\mathbf{R_1}$✓). A compact $\mathcal{K}$ (e.g., a few KiBs) yields the highest throughput but offers weaker resilience to exfiltration, whereas a large $\mathcal{K}$ (e.g., a few GiBs) provides strong protection but lower throughput.

To produce the keystream, Keymix applies several passes over the big-key $\mathcal{K}$, alternating the execution of a cryptographic primitive that operates on fixed-length blocks (MixCTR, described in Section 4.1) with repeated and organized memory swaps. These two operations fully mix $\mathcal{K}$, guaranteeing that every bit of the generated keystream depends on every bit of $\mathcal{K}$ ($\mathbf{R_2}$✓), and that no intermediate result or observable state smaller than ℓ is produced ($\mathbf{R_3}$✓).

4.1 MixCTR

MixCTR is a cryptographic primitive that leverages a Pseudorandom Permutation (PRP) to transform a fixed-size input block b into an equally-sized out-

Algorithm 1 The MixCTR primitive.

E: Symmetric cipher with key E_k and input E_{IV}, s.t. $|E_k| = m|E_{IV}|$ for $m \in \mathbb{N}^+$
b: Input block with size $|b| = |E_k| + |E_{IV}|$
b': Output block with size $|b'| = |b|$

function MixCTR(b)
  $E_k, E_{IV} \leftarrow \text{split}(b)$
  $b' \leftarrow \varepsilon$ ▷ Initialize empty output
  $n \leftarrow |b|/|E_{IV}|$ ▷ Number of encryption operations
  for $c \leftarrow 0$ **to** $n - 1$ **do**
    $b' \leftarrow b' \parallel E(E_k, E_{IV} + c)$
  end for
  return b' ▷ Return the mixed sequence
end function

put block b' in such a way that every bit of b' depends on every bit of b. To illustrate its operation, we implement MixCTR with AES-256, which uses a 256-bit key and operates on 128-bit blocks. With this configuration, MixCTR processes a 384-bit input sequence b and produces a 384-bit output b'. The primitive first splits b into a 256-bit key E_k and a 128-bit initialization vector E_{IV}, and uses them as input to the AES-256 cipher in counter mode (CTR), requiring exactly three AES invocations to generate the required output. The procedure is formalized in Algorithm 1 and shown in Figure 2. From a security perspective, the CTR mode of operation transforms a standard block cipher into a provably secure stream cipher [37]. Consequently, it is computationally infeasible to reconstruct the MixCTR input b from the output b' alone. This task is equivalent to recovering the key and initialization vector of AES-256 given only the corresponding ciphertext block, a problem that is intractable under the standard ciphertext-only attack model for a secure PRP. Moreover, MixCTR produces an output that depends on the entire input by construction: any change to either E_k or E_{IV} yields a completely different permutation with AES-256 ($\mathbf{R_2}$✓).

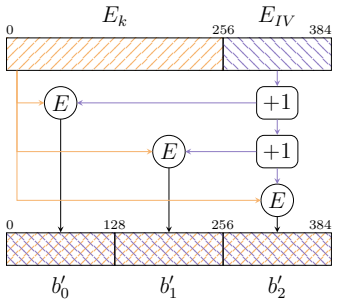


Fig. 2: MixCTR with AES-256.

4.2 Keymix

Keymix extends the complete-dependency property of MixCTR to arbitrarily large big-keys. In detail, given an input big-key $\mathcal{K}$, it outputs an equally-sized sequence $\mathcal{K}'$ such that each bit of $\mathcal{K}'$ depends on each bit of $\mathcal{K}$.

To this end, Keymix follows a multi-layer structure inspired by butterfly networks [12, 13, 22]. Let N be the total number of fixed-size chunks in $\mathcal{K}$, and let f be the fanout, i.e., the number of chunks concatenated to form a valid MixCTR input block b. The algorithm executes then exactly $\log_f N$ mixing layers. Importantly, the fanout f cannot assume arbitrary values. First, f must divide the MixCTR block size $|b|$ evenly so that the chunk size $|c| = |b|/f$

Algorithm 2 Keymix function.

$\mathcal{K}$: Big-key
f: Fanout, or number of chunks in the MixCTR input
$\mathcal{K}'$: Output with size $|\mathcal{K}'| = |\mathcal{K}|$

function KEYMIX($\mathcal{K}, f$)
  $\mathcal{K}' \leftarrow \mathcal{K}$ ▷ Copy $\mathcal{K}$ into working buffer $\mathcal{K}'$
  $N \leftarrow f|\mathcal{K}|/|b|$ ▷ Number of chunks in $\mathcal{K}$
  for $layer \leftarrow 1$ **to** $\log_f N$ **do**
    $\mathcal{K}' \leftarrow$ MIXPASS($\mathcal{K}', f, layer$)
  end for
  return $\mathcal{K}'$ ▷ Return the fully-mixed big-key
end function
function MIXPASS($\mathcal{K}_{layer}, f, layer$)
  $\mathcal{K}'_{layer} \leftarrow \varepsilon$ ▷ Initialize empty output
  for all g **in** GROUPS($\mathcal{K}_{layer}, f, layer$) **do**
    $c_0, \ldots, c_{f-1} \leftarrow g$ ▷ Extract the chunks in group g
    $b \leftarrow c_0 \,\|\, \cdots \,\|\, c_{f-1}$ ▷ Concatenate the chunks to form b
    $\mathcal{K}'_{layer} \leftarrow \mathcal{K}'_{layer} \,\|\, \text{MIXCTR}(b)$ ▷ Mix b and append to the result
  end for
  return K'_{layer}
end function

is an integer number of bits (ensuring compatibility with the underlying PRP). Consequently, supported big-key lengths take the form $|\mathcal{K}| = |c|f^l$, where $l \in \mathbb{N}^+$ and the base case $l = 1$ consists of f chunks, requiring one mixing pass.

Keymix begins by initializing a working buffer $\mathcal{K}'$ as a copy of $\mathcal{K}$. Then, in each layer, $\mathcal{K}'$ is partitioned into N/f disjoint groups of f chunks, each selected according to a layer-specific grouping pattern. For every group, the f chunks are concatenated into a single block b and passed to MixCTR. The outputs produced by all MixCTR calls are concatenated in the same order as the groups were processed, forming a new working buffer to be used in the next layer. The grouping pattern changes at each layer to ensure the value of each bit of the big-key $\mathcal{K}$ spreads exponentially. Indeed, after l layers, a chunk in the initial working buffer affects f^l output chunks. Consequently, after $\log_f N$ layers, every chunk in the final output depends on the entire big-key $\mathcal{K}$.

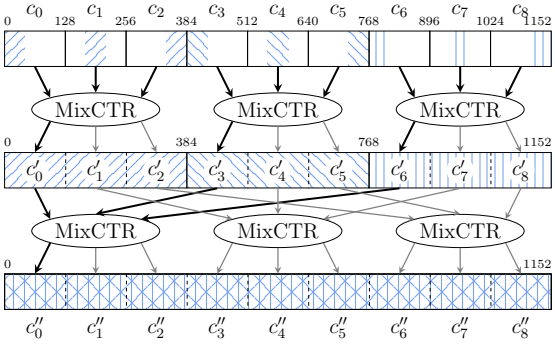


Fig. 3: Application of Keymix on a big-key $\mathcal{K}$ with 9 chunks, and fanout 3.

Figure 3 exemplifies the diffusion process for fanout $f = 3$ applied to a 144-Byte big-key (9 chunks). In the first layer, chunks are grouped consecutively as $\langle c_0, c_1, c_2\rangle$, $\langle c_3, c_4, c_5\rangle$, and $\langle c_6, c_7, c_8\rangle$; after the first mixing pass, c'_0, c'_1, and c'_2 depend on all the chunks in $\langle c_0, c_1, c_2\rangle$. In the second layer, the chunks are

regrouped as $\langle c'_0, c'_3, c'_6 \rangle$, $\langle c'_1, c'_4, c'_7 \rangle$, and $\langle c'_2, c'_5, c'_8 \rangle$; after the second mixing pass, every final chunk c''_i depends on all $\langle c_0, \ldots, c_8 \rangle$ original chunks of $\mathcal{K}$ ($\mathbf{R_2}$✓). The complete procedure is formalized in Algorithm 2, while implementation details on efficient grouping strategies are provided in Section 5. We remark that, by construction, for every level $|\mathcal{K}'| = |\mathcal{K}|$. Hence, no smaller (and easier to exfiltrate) intermediate result is ever produced, satisfying requirement $\mathbf{R_3}$✓.

4.3 Resource encryption

The Keymix function described in Section 4.2 produces a deterministic output from a big-key $\mathcal{K}$. To encrypt resources, this output is used to generate a non-repeating keystream that is XORed with the plaintext, in the spirit of a stream cipher. Three aspects must be addressed for practical use: (i) resources may be significantly larger than $\mathcal{K}$, requiring the keystream to be extended accordingly, (ii) only some portions of a large resource may be updated, and (iii) each resource must be encrypted under a unique keystream to prevent reuse attacks.

Arbitrarily large resources. Given a resource $\mathcal{R}$, ideally $|\mathcal{K}| \geq |\mathcal{R}|$ (see Section 4.4 for the security discussion). However, practical deployments can favor smaller big-keys to maximize throughput. In such cases, Keymix can be invoked multiple times on slightly modified versions of $\mathcal{K}$ to generate successive portions of the keystream. By construction, Keymix guarantees complete big-key dependency ($\mathbf{R_2}$✓): even a single-bit modification to $\mathcal{K}$ produces an entirely different output sequence $\mathcal{K}'$. Hence, to extend the keystream when $|\mathcal{R}| > |\mathcal{K}|$, we modify $\mathcal{K}$ by applying a 128-bit counter to the first MixCTR block before every invocation of Keymix, and concatenate the results into a single keystream to be XORed with $\mathcal{R}$. This process is illustrated in Figure 4. We emphasize that any approach that applies Keymix only once and subsequently feeds its output into a conventional stream cipher would violate requirement $\mathbf{R_3}$ (no-shrinking effect). Such a construction would introduce a compact intermediate state that is significantly easier for the adversary to exfiltrate, falling into the same pitfalls as the symmetric AEAD schemes discussed in Section 3.3.

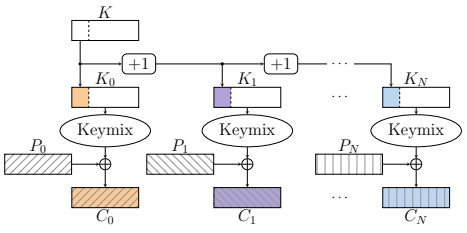


Fig. 4: Generating a keystream with Keymix by applying a counter to the first block in the big-key.

Incremental resource updates. Large resources are frequently modified via partial updates, as in file systems and distributed storage solutions such as Longhorn [44]. Since Keymix acts as a stream cipher, incremental updates only require recomputing the keystream segments corresponding to affected blocks. The required keystream is generated by applying a fresh 128-bit nonce to the first 384-bit block b of $\mathcal{K}$ before invoking Keymix, akin to the counter used for over-

sized resources. Nonce and counter spaces remain non-overlapping. Decryption relies on a compact log tracking each updated segment's offset and nonce.

Encrypting multiple resources. Ideally, each resource uses a dedicated big-key so partial key exfiltration cannot compromise unrelated assets. To save storage, one big-key can protect multiple resources by applying a unique 128-bit resource nonce to the first block b of $\mathcal{K}$. Since MixCTR's input has three 128-bit fields, the resource nonce, update nonce, and length counter fit without overlap, ensuring unique keystreams.

4.4 Security considerations

Exfiltration-resilient cryptographic schemes, as formalized in the Bounded Retrieval Model [8, 14, 15, 19, 21, 26], are designed to remain secure when an adversary extracts at most ℓ bits of information from a secret before detection and channel termination. In the threat model of Section 3.1, the security of Keymix therefore hinges on the relationship between the exfiltration bound ℓ and the configurable size of the big-key $\mathcal{K}$. In the following, we analyze the guarantees that Keymix provides in different scenarios.

AES security preliminaries. The MixCTR primitive relies on AES-256, which resists differential and linear cryptanalysis [32]. Since no practical attacks exist even under chosen-plaintext scenarios (e.g., Biclique cryptanalysis [16, 50] requires $2^{254.2}$ operations), we adopt brute-force key search as our baseline.

Leakage of $\mathcal{K}$ under known ciphertext. We analyze the security of Keymix when the adversary knows a portion of the big-key $\mathcal{K}$ together with the complete ciphertext $\mathcal{R}_e$, but possesses no information about the plaintext resource $\mathcal{R}$. To aid clarity, we set the fanout parameter to $f = 3$ and refer to the diffusion process illustrated in Figure 3. Consider a 9-chunk big-key requiring two mixing layers. Suppose the adversary has exfiltrated the chunks $c_0, \ldots, c_7$, while the last chunk c_8 remains unknown. Because the first layer groups chunks consecutively, the adversary can locally compute $c'_0, \ldots, c'_5$. However, c'_6, c'_7, c'_8 depend on the unknown chunk c_8 and cannot be obtained without a brute-force search. Focusing on the second layer, each final keystream chunk (e.g., c''_0) is produced by applying MixCTR to a regrouped triple such as $\langle c'_0, c'_3, c'_6 \rangle$. The adversary knows c'_0 and c'_3 but not c'_6. Moreover, the corresponding keystream chunk c''_0 cannot be recovered from the observed $\mathcal{R}_e$ because $\mathcal{R}_e$ is the XOR of the keystream with the unknown plaintext $\mathcal{R}$. Thus, recovering the unknown input chunk c'_6 (or equivalently c'_7 and c'_8) requires brute-forcing the 128-bit unknown portion of the MixCTR input block. Due to the PRP security of AES-256 in CTR mode, this search space of size 2^{128} is computationally infeasible. In general, given a fanout f, the butterfly structure of Keymix combined with MixCTR limits the adversary's progress to a brute-force search over $2^{|b|/f}$ possibilities per affected group (2^{128} for $f = 3$ with 384-bit blocks). Hence, probing $\mathcal{K}$ or intermediate layers reveals no plaintext chunks when $\ell \leq |\mathcal{K}| - |b|/f$. Conversely, if the adversary probes the final keystream $\mathcal{K}'$ (or $\mathcal{R}$) in working memory during encryption, each

exfiltrated chunk reveals only the corresponding chunk of $\mathcal{R}$ ($\mathcal{R}_e \oplus \mathcal{K}'$) without helping recover the rest of $\mathcal{K}'$ or $\mathcal{K}$, requiring $\min(|\mathcal{K}|, |\mathcal{R}|)$ bits for full recovery.

Leakage of $\mathcal{K}$ under known plaintext. Suppose the adversary knows the complete ciphertext $\mathcal{R}_e$, the plaintext $\mathcal{R}$, and a portion of the big-key $\mathcal{K}$ (or of any intermediate layer). This scenario is relevant under our threat model, since the adversary may probe any memory region at any time. In this setting, knowledge of both $\mathcal{R}_e$ and $\mathcal{R}$ allows the adversary to immediately recover the keystream of the final layer by computing $\mathcal{R}_e \oplus \mathcal{R}$. The adversary can then mount a bottom-up attack by inverting the Keymix layers one by one. For fanout $f = 3$, consider big-keys with length equal to f^l chunks ($l \geq 1$). To invert each layer without brute-forcing 2^{128} keys, the adversary must exfiltrate the fraction $1 - 1/f$ of the chunks at that layer forming the AES keys, plus those required from previous layers. The total number of chunks to be exfiltrated is $\sum_{i=1}^{l} (1 - 1/f) f^i = f^l - 1$. For example, with the 9-chunk big-key ($f = 3$) in Figure 3, in the worst case an adversary possessing c''_0 exfiltrates the two chunks forming the AES-256 key ($E_k = c'_0 \parallel c'_3$), applies AES decryption $E^{-1}_{E_k}(c''_0)$ to recover the IV chunk $E_{IV} = c'_6$, and ultimately recovers $\mathcal{K}$ by exfiltrating 8 of the 9 chunks. In general, Keymix resists full recovery of $\mathcal{K}$ (and $\mathcal{R}$) until at least $f^l - 2$ chunks are exfiltrated, bounding leakage to $\ell \leq (f^l - 2)|b|/f$.

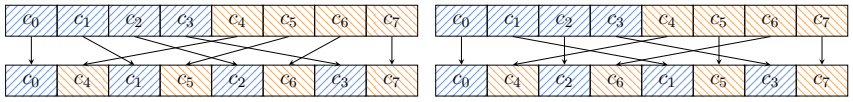
Sharing $\mathcal{K}$ for multiple resources. Encrypting multiple resources under the same big-key $\mathcal{K}$, or using a big-key smaller than the resource, does not alter the fundamental security relations derived in the previous paragraphs. However, it enhances the adversary's ability to perform inference. For example, suppose two resources $\mathcal{R}_A$ and $\mathcal{R}_B$ of identical size $|\mathcal{R}_A| = |\mathcal{R}_B| = |\mathcal{K}|$, both protected by the 9-chunk big-key from Figure 3 but using distinct nonces $nonce_A$ and $nonce_B$. These nonces are XORed into the first MixCTR input block before applying Keymix. To recover the first three chunks of $\mathcal{R}_A$, the adversary must exfiltrate the chunks c'_0, c'_3, c'_6 in the second layer. Once these are obtained, recovering the first three chunks of $\mathcal{R}_B$ requires only the additional exfiltration of the updated c'_0 (affected to $nonce_B$), because the nonce effect propagates to the full output only after the second mixing layer. Thus, by exfiltrating only 4 chunks, the adversary can recover 6 chunks of plaintext (3 from $\mathcal{R}_A$ and 3 from $\mathcal{R}_B$).

5 Implementation

We implement Keymix as an open-source C library.[3] In this section, we detail and clarify implementation choices.

Choice of symmetric cipher. The MixCTR primitive (Section 4.1) can use any secure symmetric cipher E. We adopt AES-256 because (i) no practical attacks compromising it are known as of 2026, and (ii) AES enjoys hardware acceleration on virtually all modern x86 CPUs. Using 256-bit keys lets MixCTR process 384-bit blocks (256-bit key plus 128-bit IV) per invocation, supporting

[3] `https://github.com/unibg-seclab/keymix`



a) Normal Butterfly structure b) Optimized Butterfly structure

Fig. 5: Two approaches to rearrange the output of Keymix before mixing (8-chunk big-key, fanout $f = 2$, layer $l = 3$). Lines and colors represent how chunks are interleaved.

larger fanouts and fewer mixing layers in Keymix. Moreover, the three CTR-mode AES-256 encryptions reuse the same key schedule; since key expansion is more expensive than encryption rounds, this reuse improves efficiency.

Identification of groups. As shown in Figure 3, Keymix achieves complete interdependency by mixing f chunks spaced $d = f^{l-1}$ positions apart at each layer l. While essential for diffusion, this spacing harms CPU cache locality for large big-keys and violates the contiguous-buffer requirements of libraries like OpenSSL. We resolve this by reorganizing the working buffer before each layer via an optimized butterfly network variant [22] that keeps $1/f$ of the chunks stationary, restoring locality while preserving diffusion (Figure 5).

Parallelization. Keymix supports efficient parallelization through the assignment of disjoint groups of chunks to parallel threads for the first $\log_f N - 1$ layers. Threads synchronize only in the final layer via a barrier before performing the last in-place chunk rearrangment. Moreover, when encrypting resources significantly larger than the big-key, successive Keymix invocations are fully independent, and they can therefore run concurrently with no coordination.

6 Experiments

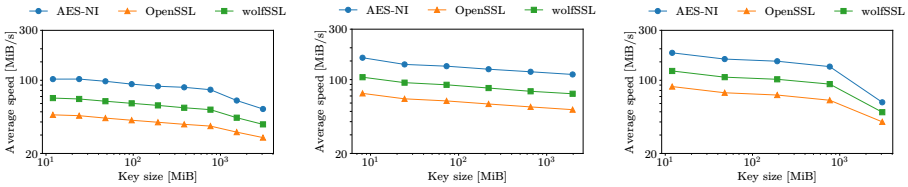
We evaluated Keymix on a Linux system with a 12-core AMD Ryzen 9 7900X CPU, 64 GiB of RAM, and a 2 TB SSD, comparing three AES backends: OpenSSL 3.3.1 [43], wolfSSL 5.7.2 [1], and Intel's AES-NI assembly [29].

6.1 Keymix performance

We first evaluated a single Keymix invocation varying the AES backend, fanout, and admissible big-key sizes (8 MiB to 3 GiB, constrained by the fanout as discussed in Sections 4.2 and 5). Figures 6 and 7 report the results.

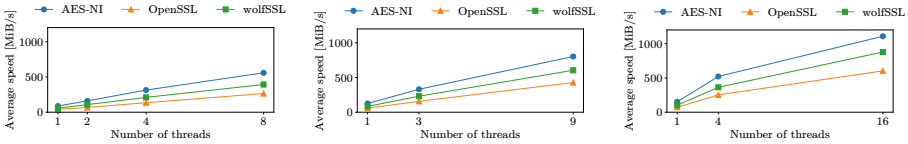
Intel's AES-NI assembly achieves the highest throughput. The gap between wolfSSL and OpenSSL stems from cipher context and memory management;[4] caching OpenSSL contexts narrowed this gap, though wolfSSL remained faster. Throughput decreases gradually with larger big-keys and lower fanouts, which

[4] `https://github.com/openssl/openssl/issues/17064`



(a) Fanout 2 (b) Fanout 3 (c) Fanout 4

Fig. 6: Single-threaded Keymix speed.



(a) Fanout 2, 192 MiB key (b) Fanout 3, 218 MiB key (c) Fanout 4, 192 MiB key

Fig. 7: Multi-threaded Keymix speed with fixed big-key size.

require more mixing layers: fanout 2 is noticeably slower, while fanouts 3 and 4 perform similarly, converging for larger keys.

To evaluate multi-threading (Figure 7), we fixed $|\mathcal{K}|$ to the smallest admissible size above 100 MiB for each fanout. Scaling threads yields speedups from +70% (fanout 2, 2 threads, OpenSSL) to +759% (fanout 4, 16 threads, wolfSSL) over single-threading, with a median gain of +80.92% per thread. Gains taper off beyond the 12 physical cores when relying on Simultaneous Multithreading.

6.2 Encryption performance

We next measured the speed of encrypting a 100 GiB resource while varying the number of concurrently processed keystream portions, using Intel's AES-NI assembly for Mix-CTR, fanout 3, 3 threads per invocation, and big-keys with size ranging from 8 MiB to 3 GiB. As shown in Figure 8, processing two keystream portions in parallel yields speedups from +84.21% to +134.16% over a single portion. Beyond four concurrent portions, throughput saturates for most big-key sizes: four portions with three threads each utilize all 12 physical cores, saturating the platform's compute capacity. Only the smallest big-key (8.11 MiB) continues scaling, as both the key and its derivation fit within the 64 MiB CPU L3 cache.

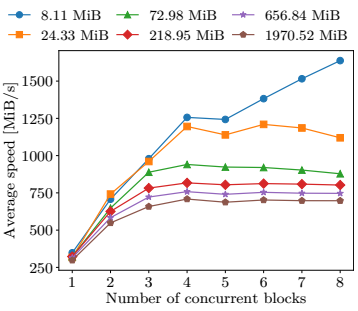
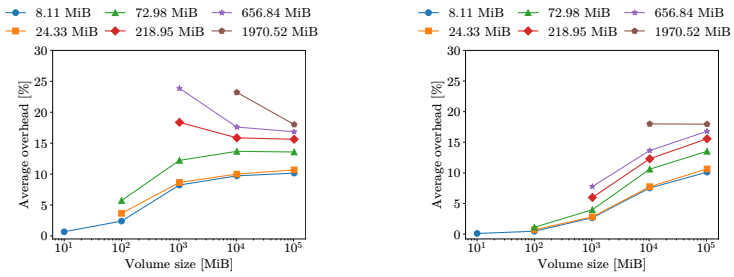


Fig. 8: Parallelized encryption of a 100 GiB resource with $f = 3$, AES-NI, and 3 threads.



(a) Backup (b) Restore

Fig. 9: Backup and restore operations overhead with different big-key sizes.

6.3 Transparent encryption of storage blocks

To evaluate our proposal in a practical scenario, we configured a single-node Kubernetes cluster on the same hardware, running Longhorn [44] for persistent volumes, MinIO [39] for backup object storage, and Chaos Mesh [20] for traffic-control bandwidth emulation. First, a file of a given size is created in the persistent volume and its hash is recorded ①. Longhorn then backs up the volume to MinIO ②. Next, we emulate a failure by deleting the volume ③ to trigger Longhorn's recovery ④, and finally compare the restored and original volume hashes ⑤. Against an unencrypted baseline, we evaluated Keymix using AES-NI, fanout $f = 3$, 3 threads per invocation, 4 concurrent keystream portions, and $|\mathcal{K}| \leq |\mathcal{R}|$. As shown in Figure 9, big-keys up to 72.98 MiB incur at most 13.58% overhead, while larger keys reach up to 23.87%. This highlights a favorable security-performance trade-off: scaling $|\mathcal{K}|$ from 72.98 MiB to 1.97 GiB increases the exfiltration threshold ℓ by 27× while adding only about 10 percentage points to the overall operational overhead.

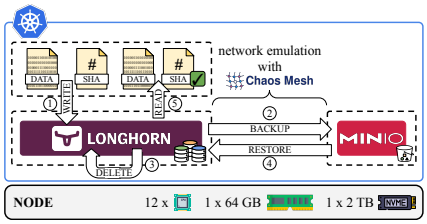


Fig. 10: Kubernetes workflow used to test backup and restore operations.

7 Conclusions

This paper introduced Keymix, a novel symmetric scheme designed to protect multi-cluster resources against key exfiltration. Keymix satisfies three critical properties: (i) it supports arbitrarily large big-keys $\mathcal{K}$, (ii) it enforces complete dependency of every keystream bit on the entire $\mathcal{K}$, and (iii) it produces no intermediate states smaller than $\mathcal{K}$. Its security guarantees thus scale with the volume of key material exfiltrated relative to $\mathcal{K}$, without assumptions on leakage timing or inaccessible memory. Built around AES-256, Keymix scales efficiently across multi-core systems and can integrate transparently with existing workflows to protect volume backup and restore operations with modest overhead.

Acknowledgments

This work was supported by MASE Mission Innovation 2.0 - D.M. 386/2023 under project EVFTEG (grant no. F53C25001680001).

References

1. wolfSSL – Embedded SSL/TLS Library. `https://www.wolfssl.com` (2024)
2. CVE-2026-43284. https://nvd.nist.gov/vuln/detail/CVE-2026-43284 (2026)
3. Abbadini, M., et al.: Cage4Deno: A fine-grained sandbox for Deno subprocesses. In: ACM ASIACCS (2023)
4. Abbadini, M., et al.: Lightweight cloud application sandboxing. In: IEEE CLOUDCOM (2023)
5. Abbadini, M., et al.: NatiSand: Native code sandboxing for javascript runtimes. In: RAID (2023)
6. Abbadini, M., et al.: POSTER: Leveraging eBPF to enhance sandboxing of WebAssembly runtimes. In: ACM ASIACCS (2023)
7. Abbadini, M., et al.: Supporting Data Owner Control in IPFS Networks. In: IEEE ICC (2024)
8. Alwen, J., et al.: Public-key encryption in the Bounded-Retrieval Model. In: EUROCRYPT (2010)
9. Arnautov, S., et al.: SCONE: Secure linux containers with intel SGX. In: USENIX OSDI (2016)
10. Arnoldi, E., et al.: Blackbird: A Scalable and Resource-Efficient Framework for Distributed Network Monitoring. In: IEEE ICDCS (2026)
11. Aublin, P., et al.: TaLoS: Secure and transparent TLS termination inside SGX enclaves. Tech. Rep. DTRS17-5, Imperial College London (2017)
12. Bacis, E., et al.: Mix&slice: Efficient access revocation in the cloud. In: CCS (2016)
13. Bacis, E., et al.: Mix&slice for efficient access revocation on outsourced data. TDSC (2024)
14. Bellare, M., Dai, W.: Defending against key exfiltration: Efficiency improvements for big-key cryptography via large-alphabet subkey prediction. In: CCS (2017)
15. Bellare, M., et al.: Big-key symmetric encryption: Resisting key exfiltration. In: CRYPTO (2016)
16. Bogdanov, A., et al.: Biclique cryptanalysis of the full AES. In: ASIACRYPT (2011)
17. Brumley, D., Boneh, D.: Remote timing attacks are practical. Computer Networks (2005)
18. Canetti, R., et al.: Exposure-resilient functions and All-or-Nothing transforms. In: EUROCRYPT (2000)
19. Cash, D., et al.: Intrusion-resilient key exchange in the Bounded Retrieval Model. In: Theory of Cryptography (2007)
20. Chaos Mesh Authors: Chaos Mesh: A chaos engineering platform for kubernetes. `https://chaos-mesh.org/` (2026)
21. Di Crescenzo, G., et al.: Perfectly Secure Password Protocols in the Bounded Retrieval Model. In: Theory of Cryptography (2006)
22. van Dijk, M., et al.: Hourglass schemes: how to prove that cloud files are encrypted. In: CCS (2012)
23. Dodis, Y., et al.: On perfect and adaptive security in exposure-resilient cryptography. In: EUROCRYPT (2001)

24. Durumeric, Z., et al.: The matter of Heartbleed. In: IMC (2014)
25. Dworkin, M.J.: Recommendation for block cipher modes of operation: Galois Counter Mode and GMAC (2007)
26. Dziembowski, S.: Intrusion-resilience via the Bounded-Storage Model. In: Theory of Cryptography (2006)
27. Easdon, C., et al.: Rapid prototyping for microarchitectural attacks. In: USENIX Security (2022)
28. Genkin, D., et al.: Drive-by key-extraction cache attacks from portable code. In: ACNS (2018)
29. Gueron, S.: Intel Advanced Encryption Standard (AES) New Instructions Set. Tech. rep., Intel (2010)
30. Jacob, H., et al.: faulTPM: Exposing AMD fTPMs' deepest secrets. In: IEEE EuroS&P (2023)
31. Jancar, J., et al.: They're not that hard to mitigate: What Cryptographic Library Developers Think About Timing Attacks. In: IEEE S&P (2022)
32. Joan, D., Vincent, R.: The design of Rijndael: AES-the advanced encryption standard. Information Security and Cryptography (2002)
33. Kalai, Y.T., Reyzin, L.: A survey of leakage-resilient cryptography (2019)
34. Kocher, P., et al.: Differential power analysis. In: CRYPTO (1999)
35. Kocher, P., et al.: Spectre attacks: Exploiting speculative execution. In: IEEE S&P (2019)
36. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: CRYPTO (1996)
37. Lipmaa, H., et al.: CTR-mode encryption. In: First NIST Workshop on Modes of Operation (2000)
38. Lipp, M., et al.: Meltdown: Reading kernel memory from user space. In: USENIX Security (2018)
39. MinIO, Inc.: MinIO: High performance object storage. `https://min.io/` (2026)
40. Moghimi, D., et al.: TPM-FAIL: TPM meets Timing and Lattice Attacks. In: USENIX Security (2020)
41. Nilsson, A., et al.: A Survey of Published Attacks on Intel SGX (2020)
42. Nir, Y., Langley, A.: ChaCha20 and Poly1305 for IETF protocols. Tech. rep., RFC Editor (2018)
43. OpenSSL project: OpenSSL: Cryptography and SSL/TLS Toolkit. `https://www.openssl.org/` (2024)
44. Project, C.I.: Longhorn: Cloud-native distributed block storage for kubernetes. `https://longhorn.io/` (2026)
45. Quisquater, J., Samyde, D.: ElectroMagnetic Analysis (EMA): Measures and counter-measures for smart cards. In: ACM E-SMART (2001)
46. Rossi, M., et al.: POSTER: Policy-driven security-aware scheduling in Kubernetes. In: ACM ASIACCS (2025)
47. Rossi, M., et al.: POSTER: Transparent Temporally-Specialized System Call Filters. In: ACM ASIACCS (2025)
48. Rossi, M., et al.: Secure Kubernetes Workload Deployment with Automated Enforcement of Cluster-Defined Policies. In: IEEE CLOUDCOM (2025)
49. Spreitzer, R., et al.: Systematic classification of side-channel attacks: A case study for mobile devices. IEEE Communications Surveys & Tutorials (2018)
50. Tao, B., Wu, H.: Improving the biclique cryptanalysis of AES. In: ACISP (2015)
51. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. In: USENIX Security (2014)