IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 3, pp. 1–31. DOI:10.46586/tches.v2025.i3.1-31

KeyVisor – A Lightweight ISA Extension for Protected Key Handles with CPU-enforced Usage Policies[†]

Fabian Schwarz[1,‡], Jan Philipp Thoma[2,‡], Christian Rossow[1] and Tim Güneysu[2,3]

[1] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany, {fabian.schwarz,rossow}@cispa.de
[2] Ruhr University Bochum, Bochum, Germany, first.last@rub.de
[3] DFKI GmbH, Bremen, Germany

Abstract. The confidentiality of cryptographic keys is essential for the security of protection schemes used for communication, file encryption, and outsourced computation. Beyond cryptanalytic attacks, adversaries can steal keys from memory via software exploits or side channels, enabling them to, e.g., manipulate confidential information or impersonate key owners. Therefore, existing defenses protect keys in dedicated devices or isolated memory, or store them only in encrypted form. However, these designs often provide unfavorable tradeoffs, sacrificing performance, fine-grained access control, or deployability.
In this paper, we present KeyVisor, a lightweight Instruction Set Architecture (ISA) extension that securely offloads the handling of symmetric crypto keys to the CPU. KeyVisor provides CPU instructions that enable applications to request protected key handles and perform AEAD cipher operations on them. The underlying keys are accessible only by KeyVisor, and thus never leak to memory. KeyVisor's direct CPU integration enables fast crypto operations and hardware-enforced key usage restrictions, e.g., keys usable only for de-/encryption, with a limited lifetime, or with a process binding. Furthermore, privileged software, e.g., the monitor firmware of TEEs, can revoke keys or bind them to a specific process/TEE. We implement KeyVisor for RISC-V based on RocketChip, evaluate its performance, and demonstrate real-world use cases, including key-value databases, automotive feature licensing, and a read-only network middlebox.

Keywords: Security Extensions, Microarchitecture, Key Management

1 Introduction

Cryptographic operations such as data encryption or authentication form the foundation of various security applications, ranging from TLS communication, VPNs, and file protection to device attestation and distributed IoT networks. In any of the underlying schemes, the secret keys are a critical asset that must be protected against attacks. If the keys

[†]Partly funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC2092 CASA - 390781972, by the DFG under the Priority Program SPP 2253 Nano Security (Project RAINCOAT 2 - Number: 440059533), by the European Commission, RIA (Project CONVOLVE - Number: #101070374) and by the German Federal Ministry of Education and Research (BMBF) under project SASPIT (16KIS1859). "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies." Date of this document: April 10th, 2025

[‡]These authors contributed equally.

Licensed under Creative Commons License CC-BY 4.0.
Received: 2025-01-15 Accepted: 2025-03-15 Published: 2025-06-05

are leaked, all higher-level security mechanisms based on them lose all (or most of) their security guarantees. However, as applications require direct access to these keys, attackers gaining access to the system memory, for instance, via local software exploitation (e.g., ROP, use-after-free), side-channel attacks, or remote exploits like Heartbleed [Syn20], can directly leak the keys and thus break the security of the applications.

To mitigate the risk of a key leakage, solutions for secure key management prevent direct key access by applications. Instead, they protect keys and their cryptographic operations in isolated environments, and provide a controlled interface to the keys via so-called *key handles*. Applications can request crypto operations via these handles without gaining access to the underlying plaintext keys, thus preventing leakage. Common approaches to implement such key handles can be grouped into three categories. First, hardware security modules (HSMs) are dedicated crypto devices that manage keys in isolated memory and support key generation and crypto operations, as well as additional access control and authentication features for them. A widely-used example is the TCG-standardized Trusted Platform Module (TPM) [Gro19], or comparable vendor-specific chips, e.g., Apple's Secure Enclave [App24, ope, Mic24]. Second, trusted execution environments (TEEs) support trusted code and data isolation and have therefore been used to implement secure key managers in software, e.g., based on Arm TrustZone [Goo24, LHX18] or Intel SGX [CBV17, Int, Int23]. Third, the Intel Key Locker CPU extension provides [Key20] CPU-protected AES key handles by introducing CPU instructions that generate and use CPU-encrypted handles for AES block operations (no AES modes).

However, existing technologies face limitations regarding their performance, integration, or enforceable key usage policies, restricting their use cases. TPM-like devices [Gro19, App24, ope, Mic24, CHB19] support many algorithms and key distribution scenarios. Their complex design can make application adoption highly non-trivial. In addition, they typically provide slower performance than CPU accelerators, rendering them unsuitable for high throughput symmetric crypto scenarios, e.g., network communication. Furthermore, they can only enforce coarse-grained access policies, but no specific and refined rules based on a caller's process identifier. While virtual software HSMs or TPMs provide more flexibility and sometimes higher performance, they are often less secure or more dependent on specific platform technologies, e.g., TEEs [Int, BCG+06, RSW+16]. Furthermore, HSMs and key manager services based on TEEs [BCG+06, RSW+16, LHX18, Goo24, CBV17, Int23] often suffer from high context switching overhead for invocation of a crypto operation, and are usually not supported on embedded systems. Moreover, various side-channel attacks against TEE-based de-/encryption have leaked symmetric crypto keys despite their isolation [CCX+20, GESM17, BMW+18]. The Intel Key Locker CPU extension [Key20] provides a further approach in this context, which is limited in its feature set and focused on (symmetric) AES keys. While providing fast, CPU-accelerated AES block operations, Intel Key Locker does not provide hardware support for encryption modes (ciphers), limiting control on how handles are used. In particular, Intel Key Locker's single securely supported usage policy is to make a handle usable only in kernel space, and Intel Key Locker can revoke handles only all at once. Furthermore, Intel Key Locker does not support secure key provisioning across systems. Moreover, when using Intel Key Locker within TEEs, the keys are only secure if Intel Key Locker has been configured with a hardware-generated protection key, and the handles are still vulnerable to leakage via side-channel attacks.

To overcome these limitations, we envision a CPU key handle extension for protecting symmetric crypto keys that combines a high-performance but lightweight system integration (HW/SW) with strong key usage controls. Similar to Intel Key Locker, we consider handles usable only via dedicated CPU instructions, enabling the CPU to mediate all operations. However, we design a new CPU-integrated policy engine that can enforce *how* and *by whom* each key handle can be used, and enables selective *revocation* of key handles. Leveraging the register-encoded execution context including the privilege level and process/TEE identifiers, the CPU can pinpoint the caller context. Furthermore, this context can be augmented with a handle-specific state to track a handle's usage controls and lifetime. With in-CPU support for (symmetric) authenticated encryption ciphers (AEAD), e.g., AES-GCM and ChaCha20-Poly1305, the CPU policy engine can enforce *how* key handles can be cryptographically used, including the ciphers and operations (en-/decrypt) permitted by a caller. Such fine-grained policies enable security schemes beyond those supported by TPMs or Intel Key Locker. By adding support for secure remote key imports, even more become feasible, ranging from read-only TLS inspection to CPU-enforced license keys.

In this paper, we present *KeyVisor*, a CPU extension design for protected key handles of symmetric keys that follows the principles outlined above. KeyVisor provides fast, easy-to-use crypto operations for AEAD ciphers (authenticated encryption with associated data), with user-defined handle restrictions enforced by the CPU. Like Intel Key Locker, KeyVisor builds on handles that are wrapped by a CPU-internal key and can only be unwrapped by the CPU. In addition, KeyVisor introduces several CPU-enforced usage restriction policies that can be associated with a key handle, and enables fine-grained handle revocation—all without sacrificing AEAD performance. KeyVisor focuses on AEAD ciphers as they are widely used due to their combination of encryption and authentication, e.g., AES-GCM in TLS or ChaCha20-Poly1305 in the Wireguard VPN [Don17]. Furthermore, AEAD ciphers enable us to design additional en-/decrypt-only key controls, not possible for many non-AEAD (stream) ciphers, e.g., AES-CTR. In particular, our current implementation of KeyVisor for RISC-V, based on AES-GCM, allows to control (1.) the permitted AES-GCM operations (encrypt/decrypt), (2.) permitted caller context, e.g., process, (3.) counter-based handle revocation (e.g., to enforce a one-time key), and (4.) selective on-demand handle revocation. In addition, KeyVisor introduces a local trusted key provisioner that enables remote services to securely export symmetric keys as restricted key handles to the local system—without leaking the plaintext key to the local software.

We prototype KeyVisor for AES-GCM on the 64-bit RISC-V (RV64) RocketChip CPU and release it as an open-source project for the research community[1]. KeyVisor introduces four new CPU instructions tailored to creating, using, and revoking key handles, as well as an efficient CPU-internal handle caching structure. We demonstrate the benefits of KeyVisor's usage policies by integrating it into three real-world use cases: web services using a vulnerable key-value database, a(n offline) licensing scheme for automotive pay-per-use features, and a read-only network middlebox for guaranteed tamper-free traffic analysis. We measure the area overhead of KeyVisor, which shows its lightweight design. Furthermore, we evaluate KeyVisor's performance by comparing it against related hardware- and software-based approaches and by analyzing a TLS-based real-world use case.

In summary, we make the following contributions:

- We design KeyVisor, a lightweight ISA extension providing protected key handles with CPU-enforced usage policies for (symmetric) AEAD ciphers.
- We provide a RISC-V (RV64) hardware implementation of KeyVisor for AES-GCM and evaluate its area requirements and performance, showing its practical feasibility.
- We compare KeyVisor against software and hardware implementations to give an idea of its applicability in real-world use cases.
- We demonstrate KeyVisor's benefits and flexible key usage policies by integrating it in three real-world use cases.

[1] KeyVisor's open-source code is available at: `https://github.com/Chair-for-Security-Engineering/KeyVisor`

2 Design Goals and Threat model

Existing solutions for protected key handles—most prominently including TPM-like co-processors [Gro19, App24, ope, Mic24, CHB19] or the Intel Key Locker ISA extension [Key20]—are not flexible enough to meet the needs of sophisticated use cases. TPM-like devices focus on storing and supporting many types of crypto keys and ciphers, measured boot [Mic22], and are limited in high-performance settings. In contrast, Intel Key Locker achieves high-performance via an in-CPU AES accelerator but has narrow key control, mainly limited to hiding the plaintext AES keys from memory. Therefore, we envision KeyVisor's design to enable a wide set of new use cases ranging from high-performance to embedded settings. We regard a lightweight ISA extension as a key enabler for achieving this, because (1.) it enables direct access to the CPU register values and thus execution contexts—essential for fine-grained handle controls—and (2.) can provide high-performance (symmetric) AEAD and policy operations while (3.) being affordable for high-end and low-end CPUs alike. We will now motivate KeyVisor's goals (G_x) based on concrete use cases (UC-1 to UC-3) where TPM-like solutions and Intel Key Locker are insufficient, before summarizing the threat model and presenting our full design.

UC-1: Preventing leakage via Process Binding. Existing key handle solutions lack fine-grained control over *who* can use a handle. While key handles mitigate remote attackers from stealing keys because the handles are unusable on the remote host, handles are still vulnerable to local leakage. For instance, consider a web service that uses a local key-value database (e.g., Redis) for storing ephemeral web session data. The web service wants to prevent the database from reading the stored data to protect against a potential compromise of the database. Therefore, the web service passes values only in encrypted form to the database. However, if the encryption keys of the web service are leaked, e.g., via a remote information leakage vulnerability or a local side-channel attack, attackers can decrypt the user data stored in the database (e.g., session credentials). Even if keys are wrapped inside handles, local attackers can exfiltrate and abuse the handles. The underlying issue is the incapability to enforce fine-grained local controls on what execution contexts are permitted to use a key handle. Therefore, we envision a new key handle extension to be able to *bind handles to specific local processes* (G_1) or *privilege levels* (G_2). That way, the web service could bind the handles to its own process, guaranteeing not only that the keys never leak to remote parties, but also that local attacker processes cannot abuse the key handles.

UC-2: Key Revocation and Remote Control. Two important concepts for key handles include revocation and remote provisioning. Revocation enables users to make handles unusable, e.g., for a key rollover or if key usage should be denied after a security incident. However, TPMs and Intel Key Locker only feature coarse-grained control for set-wise revocation—Intel Key Locker even supporting only an all-or-nothing revocation. For some use cases, it would be beneficial to even have the CPU automatically revoke a given handle after a specified number of crypto operations, e.g., to enforce one-time handles. Such a feature is particularly interesting when a remote service wants to temporarily provision a key to a client system, with CPU-enforced lifetime restrictions. That is, export a key as a restricted handle, without leaking the plaintext key.

As an example, consider an automotive system for which the vendor offers extra features locked behind a licensing system. Customers can purchase temporary licenses to gain access to specific features for a limited number of activations, e.g., a sport mode providing more motor power. As part of a feature activation, the car connects to a remote vendor service to check for a valid license. However, in such a design, customers cannot user their purchased features if the car is in offline mode (e.g., no coverage). Therefore, vendors require a secure mechanism to locally verify licenses.

One solution could include the remote service provisioning a feature-specific signing key to the car, that enables the user to locally authenticate feature enabling from the respective control unit, e.g., motor unit. By wrapping the key in a handle, it can be isolated from local system-level attackers trying to bypass the licensing enforcement. If the number of signing operations could be limited to the number of acquired feature uses, the remote vendor could ensure that the handle is automatically revoked, even if the car is in offline mode. However, neither TPMs nor Intel Key Locker support such counter-based lifetime policies, and Intel Key Locker does not even support remote key provisioning. Therefore, for KeyVisor, we envision two new features: (1.) *selective key handle revocation* (*G_3*)—on demand, or via lifetime counters—and (2.) *confidential remote key provisioning (G_4)*. That way, the key could be securely provisioned to KeyVisor and transformed into a counter-restricted handle, which is automatically revoked by the CPU when the number of licensed feature activations has been reached.

UC-3: Share Read-only Access to Encrypted Data. Beyond controlling *by whom* and *how long* key handles can be used, we deem control on *how* they can be used important. For instance, consider local enterprise clients that communicate with external services via end-to-end-encrypted (E2EE) connections. The company might want to deploy a remote traffic monitoring service (e.g., on-path intrusion detection/prevention system, short: IDS/IPS) to scan the plaintext traffic for suspicious activities, e.g., data leakage or malware. However, the company wants to guarantee that the monitor preserves the integrity of the connections to prevent a compromised service from tampering with the traffic. Common approaches for live traffic decryption either cannot rule out data tampering, e.g., those directly sharing connection keys or performing certificate-based MITM attacks, or require dedicated ports of monitoring services into trusted hardware [PLPR18, DWY+19], or protocol changes that enforce read-only access to the plaintext traffic [LSL+19]. Instead, the company wants to wrap the connection keys in key handles that deny encryption operations required for data injections. However, TPMs do not control if software uses handles for encrypt or decrypt operations, and is too slow for live network monitoring. Intel Key Locker is CPU-accelerated and can selectively deny de-/encrypt AES block operations. However, due to Intel Key Locker's lacking in-CPU support for AEAD ciphers, attackers can bypass the de-/encrypt-only handle restrictions, e.g., by *decrypting* TLS traffic using AES-GCM *encrypt* operations (cf. § 5.3 for details). Therefore, we desire key handles that have full control of the AEAD ciphers in hardware and are adjusted to *securely enforce encrypt- or decrypt-only handles (G_5)*. That way, the company clients could remotely provision decrypt-only handles to the traffic monitoring service, enabling it fast but read-only traffic access, without requiring TLS changes.

2.1 Threat Model

We now summarize the threat models (TM) that we derived for KeyVisor from the above use cases. We assume the local CPU as trusted, and consider that local user or system software (incl. OS) use symmetric crypto keys (e.g., AES keys) as a basis for their higher-level security protocols. The local software is benign but vulnerable, i.e., might become the target of an attack. The attackers' goal is to gain uncontrolled access to the keys in order to abuse them for attacks against the security protocols (cf. use cases). To mitigate this threat, local software wrap their cryptographic keys using CPU-provided key handles to securely enforce usage restrictions on them and to prevent the (unrestricted) keys from leaking to attackers.

The cryptographic keys can be generated locally, or be provisioned by a remote system. In either way, the generation and wrapping phase of the keys must be protected. Otherwise, attackers can steal the keys from memory or tamper with their restriction policies before both are protected by a key handle. Therefore, we assume that each local service (cf. UC-1) is uncompromised while generating and wrapping keys (e.g., on startup), however, might get attacked or compromised *after* creating the key handles. We denote this attack model as $\mathbf{TM_{local}}$. In remote provisioning settings (cf. UC-2 or UC-3), remote services generate keys and want to confidentiality share them as restricted key handles with the local system. In this case, we assume that the remote systems are fully trusted, while all local software (incl. OS) might already be compromised by an attacker. We refer to this attack model as $\mathbf{TM_{remote}}$. That is, remote keys are securely generated, but we assume local attackers trying to leak or tamper with the provisioned remote keys and their restrictions policies before they are wrapped into protected local key handles. Our assumptions for both settings are in line with the threat models of TPMs and Intel Key Locker (also cf. § 8)—however, note that Intel Key Locker lacks remote key provisioning.

In both settings, we consider several attacks against local key handles: Attackers might compromise the user or OS process owning a key handle to gain direct access to the key handle (e.g., via code execution exploits), or use local timing or cache/memory side-channel attacks to steal them across process boundaries [YF14, GBK11]. Similarly, attackers might steal key handles via the network (e.g., Heartbleed) or from physical DRAM (e.g., DMA or cold boot attack). Attackers can try to tamper with key handles or interact with our key handle CPU extension via the local instruction interface. Furthermore, we consider timing or cache/memory side-channel attacks against our CPU extension. However, we exclude other physical side channels (e.g., power) and other physical attacks targeting the CPU and its internal memory, e.g., through fault injection or voltage glitch attacks [NSUH21, TBE+21, BJKS21]. We refer to orthogonal hardware and software defenses for tackling such strong attackers [GAMG+23, SFSG23]. In addition, we exclude Denial of Service (DoS) attacks by system-level attackers trying to delete or revoke key handles, since such attackers have the ability to terminate selective processes or the whole system anyway.

Optional Process or TEE Bindings. As motivated in UC-1 to UC-3, one of the intended usage restrictions is the binding of key handles to execution contexts, rendering them usable only within a specific process or TEE enclave (cf. § 3.4). Note that throughout the paper, we assume local attackers with control of user or OS processes that try to attack local key handles, including their usage restrictions and remote provisioning procedure (cf. $\mathrm{TM_{local}}$ and $\mathrm{TM_{remote}}$). Regarding TEE bindings, we support TEEs based on physical memory protection (PMP), e.g., Keystone enclaves [LKS+20], and are in line with their threat models by assuming the TEEs and their monitor firmware to be trusted ($\mathbf{TM_{bTEE}}$). For the *optional* process binding feature (only), we must relax our threat models: In the current RISC-V ISA and OS designs, processes and their IDs (identifiers) are assigned and managed by the OS. Therefore, process bindings can only be reliable while the OS is uncompromised ($\mathbf{TM_{bProc}}$), as OS-level attackers can manipulate the binding IDs. Designing OS-isolated, fully CPU-managed process IDs is out of scope. KeyVisor's other key handle restrictions (e.g., revocation) stay secure within $\mathrm{TM_{local}}$ and $\mathrm{TM_{remote}}$.

3 Design Concepts

In the following, we introduce the design concepts of *KeyVisor*, our hardware extension for protecting symmetric keys using key handles with CPU-enforced usage policies. Without loss of generality, we present the concepts tailored to 128-bit user keys for AEAD cipher operations, and target the 64-bit RISC-V ISA. Our choice is motivated by the open-source nature of RISC-V and the wide usage of AEAD ciphers, such as AES-GCM in TLS, Ascon in IoT settings, or ChaCha20-Poly1305 in the WireGuard VPN [Don17]. While KeyVisor's focus is on symmetric keys for AEAD ciphers, KeyVisor could support most symmetric

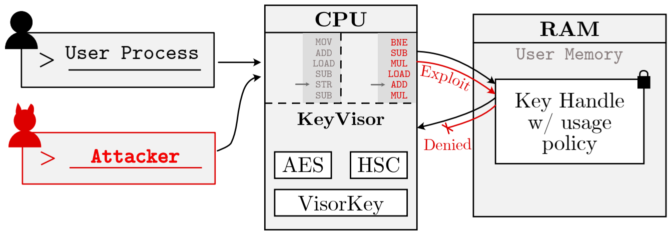

Figure 1: KeyVisor allows CPUs to replace in-memory keys with protected key handles usable only via new CPU instructions in a policy-defined way, e.g., by the owner process.

non-AEAD ciphers without limitations (cf. § 5.3). We will briefly discuss the support of non-AEAD ciphers and asymmetric (private) keys in § 7.

3.1 KeyVisor in a Nutshell

KeyVisor offloads the confidentiality protection and usage control of symmetric user keys from vulnerable software services to the CPU. That way, KeyVisor can provide fast cryptographic operations in hardware (de-/encryption) while enforcing strict key isolation based on the CPU-encoded user context. With KeyVisor, software can use a new CPU instruction to securely wrap their keys (*user keys*) with a *visor key* that is only accessible by the *local* CPU (see Figure 1). The resulting key handles never leak the plaintext keys and can thus be managed in unprotected memory and storage. Software can use key handles for cryptographic operations only via new CPU instructions that enforce usage restrictions before securely unwrapping the keys and performing the requested AEAD operations efficiently in hardware (e.g., user data encryption). The plaintext keys can be wiped from memory, thus preventing any leakage to local or remote attackers.

Among KeyVisor's key concepts that distinguish it from existing key protection solutions are its fine-grained key usage control and revocation management. When a user key is transformed into a key handle, users can specify a usage restriction policy that governs how and by whom the handle can be used to encrypt or decrypt data. KeyVisor securely associates the policy with the handle and enforces its rules on each handle-based operation, without sacrificing performance. The handle policies support high-level rules that specify the permitted AEAD cipher (e.g., AES-GCM) and types of crypto operations, e.g., "only permit decryption", as well as lifetime rules limiting the number of permitted handle uses, e.g., one-time keys. In addition, KeyVisor's tight CPU integration—in contrast to external solutions like TPMs—enables context-sensitive policy rules based on the CPU-exposed caller information, e.g., the current process ID or CPU privilege level. That way, KeyVisor can bind key handles to specific caller contexts, e.g., the kernel or a specific user process, which renders stolen handles unusable even by local attackers. In addition to restricting handle policies, KeyVisor enables (authorized) software to request revocation of a handle's key via a CPU-internal key handle allowlist, which is based on an efficient hardware caching structure, called *Handle State Cache* (*HSC*). Furthermore, KeyVisor introduces an (optional) authenticated *Remote Key Provisioner* which can securely receive user keys and restriction policies from a remote system and forward them to KeyVisor's handle unit. That way, remote services can share keys that never leak in plaintext to the local software and whose usage is tightly controlled. In § 6.3, we will describe how KeyVisor efficiently solves the web, network, and automotive key protection challenges described in § 2, and explore further scenarios in § 7.

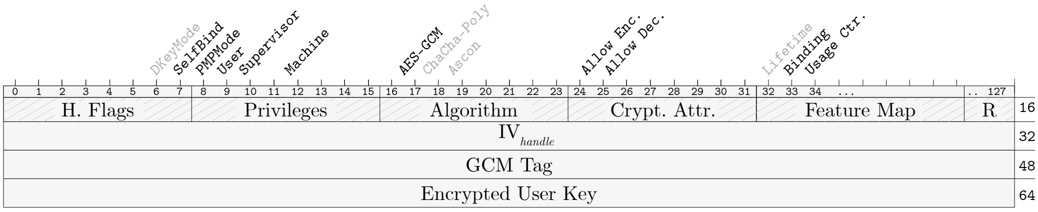

Figure 2: The Key handle format contains the encrypted user key, the AES-GCM IV and tag, and the handle usage policy (bits 0 to 128). The policy is signed as GCM AAD.

3.2 Transforming Keys into Protected Handles

In order to benefit from KeyVisor's protection guarantees, software services must transform their raw *user keys* (e.g., AES keys) into secure key handles and then wipe the unwrapped keys from untrusted memory. KeyVisor adds a new CPU instruction for generating protected key handles that services can directly execute. In KeyVisor, one key handle represents one user key and its associated usage restriction policies. In contrast to OS file or socket descriptors which only encode an index into an internal OS table containing all associated data, KeyVisor encodes the key and most of the usage policies directly in the key handle object itself. That way, KeyVisor avoids large expensive CPU-internal memory and lets software store the key handles in untrusted memory or disk storage.

KeyVisor must protect the key handle-encoded data against tampering and key extraction. Otherwise, attackers might overwrite the usage restrictions to gain uncontrolled handle access, or even leak the encoded user key. Therefore, KeyVisor introduces a CPU-internal AES-GCM engine and one internal AES key, referred to as the *visor key*. KeyVisor uses the AES-GCM engine and visor key to wrap each key handle, i.e., encrypt and sign it. The visor key is securely generated by the CPU using a true random number generator (TRNG) and stored in a protected CPU register only accessible by KeyVisor. Consequently, only KeyVisor can decrypt the key handle and therefore extract and use the user key.

To be precise, on handle generation, KeyVisor performs an AES-GCM operation in hardware that uses the visor key k_{visor} to authentically encrypt the user-provided key k_{user}. A handle usage policy is derived from the user-specified restrictions and is used as additional authenticated data (AAD), i.e., it is signed but not encrypted. That way, the policy is bound to k_{user}, protected against tampering, and still readable by user software. For each new key handle, KeyVisor adds an entry to its CPU-internal allowlist which keeps track of all valid handles, as we will explain in § 3.4 and § 3.5. The resulting key handle is shown in Figure 2, consisting of the encrypted k_{user}, the authentication tag and initialization vector IV_{handle} of the authentic encryption, and the usage policy. KeyVisor calculates a fresh IV_{handle} for each handle using a linear feedback shift register (LFSR) (cf. § 3.3 and § 4) to prevent IV collisions. Otherwise, IV collisions would break the security of AES-GCM and thus allow for leaking information on the associated user keys. We will explain the usage policies of KeyVisor's handles in § 3.4, and briefly discuss alternative key wrapping schemes in § 7.

3.3 Handle-based Data De-/Encryption

Software services can use KeyVisor's protected key handles to perform AEAD operations (encrypt-sign, decrypt-verify) with the policy-permitted cipher, e.g., AES-GCM. KeyVisor ensures that only valid handles can be used and that their restriction policies are securely enforced by the CPU on each operation—without leaking information on the plaintext key. KeyVisor defines one CPU instruction for encryption and decryption respectively that can

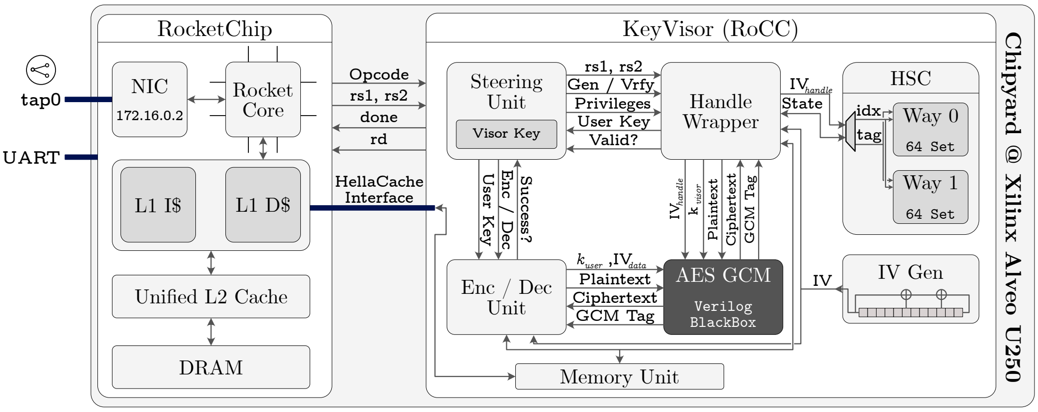

Figure 3: Simplified block diagram of KeyVisor's hardware implementation.

be directly executed by software in line with the policy, e.g., permitted privilege levels. The instructions take two input registers storing memory pointers to the key handle and an I/O structure containing pointers to the required plaintext/ciphertext and cryptographic metadata, e.g., the authentication tag for decryption.

On instruction execution, KeyVisor checks the key handle and usage policies before performing the requested crypto operation. KeyVisor first loads the key handle and unwraps it, i.e., verifies its integrity and decrypts the embedded user key using the visor key and GCM tag. If an attacker has tampered with the key handle data, including IV_{handle} and usage policy (used as AAD), the unwrapping fails and denies the operation. Upon success, KeyVisor checks if the handle is valid using its internal allowlist (HSC, § 3.5) and enforces the handle-associated usage restrictions, e.g., permitted operations or process-binding (discussed in the next section). If any of the checks fails, the requested crypto operation (encrypt/decrypt) is denied. By completely loading and checking the handle and its restriction policy before usage, KeyVisor prevents Time-of-Check-to-Time-of-Use attacks [Jin05] that try to concurrently tamper with the handle in memory, e.g., to bypass restrictions. Finally, KeyVisor loads the crypto and user data (block-wise) from the given input addresses and performs the actual AEAD de-/encrypt operation.

In principle, KeyVisor lets the user control the IV_{data} used for encryption and decryption operations, like existing designs (e.g., OpenSSL, Intel Key Locker). However, there is one important exception: users *cannot* choose the IV_{data} used by encrypt-only key handles. Otherwise, attackers could exploit that stream-cipher based AEADs allow to *decrypt* data using the *encrypt* operation, bypassing encrypt-only handle restrictions—as we will explain in § 5.3. Therefore, KeyVisor uses a hardware full-cycle LFSR [WM12] to generate a fresh, collision-free IV_{data} on each operation of encrypt-only handles, preventing such attacks. In that case, the resulting integrity tag and used IV_{data} are output to the I/O structure.

3.4 Handle Usage Policies and Revocation

By default, KeyVisor's key handles would be usable for crypto operations by whoever has memory access to them. Therefore, to protect handle access, KeyVisor introduces CPU-enforced per-handle usage restriction policies that enable users to easily specify if, how, and by whom key handles can be used. KeyVisor associates a policy on handle creation (§ 3.2) and enforces it on each handle-based operation with a minimal overhead.

Restrict how handles are used. One set of KeyVisor's restrictions controls *how* key handles can be used. This currently includes the selection of the permitted AEAD cipher and cipher operations. Users can deny encryption or decryption operations for a handle, which enables en-/decrypt-only handles that add asymmetric usage restrictions to otherwise symmetric crypto keys. In § 6.3.3, we show how they can enforce read-only access to TLS connections for a traffic monitor. While KeyVisor's focus is on AEAD ciphers, in principle, KeyVisor can also support non-AEAD ciphers, e.g., block ciphers like AES-CBC or stream ciphers like AES-CTR. However, as we will explain in § 5.3, it is not possible to enforce secure en-/decrypt-only handles for existing non-AEAD stream ciphers.

Restrict if handles are valid. The second set of restrictions controls *if* a handle is valid and *when* it becomes invalid. KeyVisor manages an allowlist of valid handles to avoid the necessity to keep track of a potentially endless number of invalid or revoked key handles. Since the validity of a handle inherently changes over time, this information cannot be stored inside the handles. Otherwise, an attacker could copy a valid handle and, once the original handle has been revoked, keep using the still valid copy. Therefore, KeyVisor stores the handle allowlist in CPU-internal memory combined with its per-handle state cache entries (HSC). On handle creation (§ 3.2), KeyVisor flags the new HSC entry associated with the handle as valid. To invalidate a key handle, KeyVisor provides a new CPU instruction that takes the handle address as input. The instruction unwraps and checks the key handle (cf. § 3.3) before flagging the respective HSC entry as invalid. We discuss KeyVisor's current revocation strategies and who can revoke handles in Appendix A. Additionally, KeyVisor supports a counter-based lifetime restriction for key handles, that limits the number of allowed en-/decrypt operations of a handle and revokes it when reaching zero. That way, one-time or usage-limited handles can be implemented, e.g., as used for a licensing scheme in § 6.3.2. As the counters must be updated on each operation, similar to the handle validity, they cannot be stored inside the handles. Instead, each counter is stored in a handle's CPU-internal state cache entry (cf. § 3.5).

Restrict who uses handles. Finally, KeyVisor implements restrictions that control *by whom* a handle can be used. First, users can define as subset of permitted CPU privilege levels, including user space, kernel space, or monitor mode. KeyVisor's instructions check the caller's privilege level directly via the respective CPU register. That way, handles can, for instance, be bound to the OS kernel, preventing user space attackers from using them. Second, handles can optionally be bound to a process or TEE context, making them usable only by that specific context, e.g., application. KeyVisor associates a unique process or TEE identifier (ID) with a key handle to establish the binding. On each handle-based operation, the binding is checked using the respective ID.

What CPU-level identifiers KeyVisor can securely use to distinguish process or TEE contexts depends on the ISA, OS, and TEE (cf. § 2.1: $\mathrm{TM_{bProc}}$, $\mathrm{TM_{bTEE}}$). For instance, for our 64-bit RISC-V ISA with a Linux-based OS, the active user process can be identified using the SATP CPU register, which stores the address of the process-specific page table and its address space identifier (ASID). Alternatively, KeyVisor allows the binding of handles to PMP-based TEE instances (physical memory protection), e.g., Keystone enclaves [LKS+20]. The PMP memory partitions isolate TEEs from the OS and are identified via CPU-accessible PMP IDs. As some binding IDs might leak sensitive information, e.g., kernel addresses of a process, KeyVisor currently stores the (optional) binding ID, i.e., SATP value or PMP ID, in a handle's CPU-internal memory entry (HSC). Alternatively, the ID could be stored within the handle (policy) in a masked way, e.g., encrypted or hashed.

3.5 Handle State Cache

KeyVisor introduces a CPU-internal state cache for efficiently storing per-handle data, called *Handle State Cache (HSC)*. Each entry of the HSC is associated with one valid key handle and includes the handle's information of its stateful restriction policies, e.g., the current lifetime counter value. Since handle lookups must be fast and CPU-internal memory is expensive and limited, the HSC is designed as a small set-associative structure, similar to CPU caches or the Translation Lookaside Buffer (TLB). That is, the HSC is organized in *sets* and *ways* forming a table-like structure (shown in Figure 5; Appendix Appendix A). To efficiently lookup entries in the cache, we split the bits of each key handle's unique IV_{handle} into a cache index (selecting the set) and a tag, with the split depending on the implemented cache and IV size, e.g., 6 bit and 90 bit. As described in § 3.2 and § 4, IV_{handle} is generated in a collision-free and (pseudo-)random way, such that statistically, the handles evenly spread over the cache sets. The tag is stored as part of the handle entries. On a handle-based operation, KeyVisor can use the index to select the correct cache set and then concurrently compare the tag in each way to pick the correct handle entry. In Appendix B, we discuss how to overcome the in-CPU size limitations of the HSC using memory swapping, enabling practically unlimited handles.

3.6 Remote Key (Handle) Provisioning

So far, we explained how clients can protect local keys using KeyVisor. However, many use cases involving symmetric keys require secure remote key sharing or provisioning. For instance, consider a network IDS (NIDS) for TLS traffic as described in § 2, which requires the TLS keys to be shared between the connection peers and the additional NIDS host. Unfortunately, a system sharing keys with a remote host has no control on how they are used by that host. A compromised NIDS host could abuse the shared keys to stealthily tamper with the TLS traffic. In such cases, it would be beneficial if the keys would not directly leak to the NIDS host and could be restricted in usage, permitting only decrypt operations to enforce read-only traffic access even if the NIDS host gets compromised.

Therefore, KeyVisor foresees the integration of a *Remote Key Provisioner (RKP)* to enable such use cases. The RKP securely receives remote keys and usage policies and directly forwards them to the local KeyVisor handle unit for wrapping. That way, untrusted local software, e.g., the NIDS, never gains direct access to the remote keys as they will become usable only as protected key handles with KeyVisor-enforced usage policies.

The RKP requires the following features to enable such secure key sharing: an authentication mechanism, (confidential) isolation from the local system, and a direct interface to the KeyVisor CPU extension. Remote peers authenticate the RKP to establish a secure channel for sharing keys (and usage policies), e.g., based on a key exchange protocol and hardware-based remote attestation. The isolation and direct interface of the RKP guarantee that the received remote keys do not leak to the local software and can be directly passed with their policies to KeyVisor for a secure key handle transformation (cf. $\text{TM}_{\text{remote}}$ in § 2.1). We deem multiple implementation variants possible, e.g., a full hardware module or extension, a lightweight CPU extension cooperating with an external key manager such as a TPM [EGLA24], or a trusted software module isolated by a CPU-provided TEE. Our prototype implementation is based on the Keystone TEE [LKS+20], as discussed in § 4.

4 Implementation

To demonstrate our concepts, we implemented an open-source RISC-V (RV64) prototype of KeyVisor1[0] as a hardware extension to RocketChip [AAB+16] and the Chipyard framework [ABG+20]. We use Chipyard to instantiate a 64-bit RocketCore CPU equipped with a 5-stage in-order pipeline, and we integrate the Keystone project [LKS+20] to enable TEE support in the form of secure enclaves. Our extension is implemented in the Chisel hardware description language and focuses on key handles for AES-GCM operations. In § 7, we will explain how KeyVisor could be ported to other ISAs (e.g., x86 or Arm), and discuss in how far KeyVisor could be extended to support other types of ciphers, e.g., non-AES or non-AEAD ciphers.

As shown in Figure 3, our prototype consists of a main steering unit, the handle wrapper unit and HSC for handle management, the en-/decryption unit for handle-based operations, as well as an IV generation and AES-GCM unit. The steering unit serves as the main control unit of the KeyVisor extension and integrates the new key handle instructions using the RoCC interface provided by RocketCore: `wrapkey`, `encrypt`, `decrypt`, and `revoke`. The AES-GCM and IV units are used by the other units for the actual cryptographic operations. Note that as we use AES-GCM for the key wrapping, we can use the same hardware logic for the wrapping and the user-requested en-/decrypt operations, which further decreases our area overhead (cf. § 6.1). To enable memory access, KeyVisor is connected to the memory interface of the CPU L1 caches using RocketChip's *HellaCache* interface, allowing to fetch data from the cache or RAM. In production environments, the visor key is securely generated by the CPU's TRNG on startup, or loaded from a secure non-volatile storage. In the current prototype, we load the key from memory instead. The prototype does not yet implement the HSC swap memory (cf. Appendix B) and executes the instructions in a blocking way, halting the single-issue pipeline of the RocketChip CPU during computation—both could be improved in future versions.

AES-GCM Unit. KeyVisor's current implementation builds on an AES hardware unit with support for AEAD in order to protect the key handles and provide handle-based crypto operations. Our current prototype focuses on the frequently used GCM mode. We implement the AES unit as a black box based on an open-source AES128-GCM implementation [Ber23]. In principle, other AES-GCM implementations can be used to achieve different area-performance tradeoffs (cf. § 6.1). Moreover, as mentioned before, AES-GCM could be replaced or augmented with other (non-AES) AEAD ciphers, e.g., ChaCha20-Poly1305 [NL15] (cf. § 7).

The AES unit is only accessible by KeyVisor's new CPU instructions. KeyVisor queries the AES unit for AES-GCM operations by transferring an AES key and the required data, e.g., plaintext/cipher, AAD, or authentication tag. On key handle generation, the AES unit authentically encrypts the plaintext user key with the visor key, and signs the usage policies as AAD. On handle-based crypto operations, the AES unit performs the user-requested operation (encrypt/decrypt) with the handle's decrypted user key.

Collision-free IV Generation. KeyVisor must ensure that the IV's used for key handle generation (IV_{handle}) and encrypt-/decrypt operations (IV_{data}) are collision-free. As AES-GCM is insecure under IV collisions, otherwise, attackers might try to exploit collisions to recover information on the plaintext of encrypted user data or on the GCM authentication key in order to spoof tags. While the IVs must be unique, they need not be random. Therefore, we implement a four tap LFSR with $n = 96$ and maximum cycle length [WM12] to sample 96 bit IVs. The LFSR is clocked every time a new IV is required, thus ensuring that collisions can occur only after 2^{96} handle generations or crypto operations. Note that an attack purposely trying to overflow the LFSR would require more than $2.5 \cdot 10^{12}$ years when assuming 1 ns per operation (1 GHz clock), and is thus infeasible in practice.

CPU-Internal Registers and Memory. KeyVisor adds new CPU-internal registers and SRAM to store its internal state information. As our current implementation uses AES-128-GCM for the key handle protection, KeyVisor adds a secure 128 bit register for the visor key that is only accessible by KeyVisor. We implement the HSC in CPU-internal SRAM as a 2-way set-associative caching structure with 64 sets. Accordingly, we use the lower 6 bit of each IV_{handle} as the index to select the set, and the remaining 90 bit as tag for selecting the way (cf. Figure 5, Appendix). We store the tag in the respective

HSC handle entries together with the key handle's 64 bit binding ID and the current 8 bit lifetime usage counter if enabled. Our implementation does not store the validity flags of the key handle allowlist mechanism in the HSC for efficiency reasons. Instead, we implement a 128 bit-field register. Each bit in the register indicates the validity of the key handle associated with one of the 128 (2×64) HSC entries and is set/unset on handle creation and revocation accordingly.

Handle Wrapper. The handle wrapper unit is responsible for creating, unwrapping, and revoking key handles, as well as enforcing the associated restriction policies. Accordingly, the handle wrapper is involved in all new CPU instructions added by KeyVisor.

As shown in Figure 2, KeyVisor currently implements 512 bit key handles that embed the encrypted user key and the associated usage policy. As we build on AES with 128 bit keys, the cipher and tag length are 128 bit accordingly. Similarly, we reserve up to 128 bit for IV_{handle}, e.g., AES-GCM and ChaCha20-Poly1305 require 96 bit by default. The lower 128 bit of a handle are used for the key handle-embedded usage policy information. The current calling convention of KeyVisor's instructions passes handles via memory. Therefore, the key handle fields are 64 bit-aligned to enable faster CPU access. Alternatively, as the key handles fit into four extended 128 bit registers, a future implementation could support a faster register-based calling convention. The usage policy data is divided into five groups. The fields are encoded as space-efficient bit-fields which select the AEAD cipher (`Algorithm`), permit en-/decrypt operations (`Crypt.Attr.`) or caller privilege levels (`Privileges`), or enable usage restrictions, e.g., process-binding or usage counters (`Feature Map`). As process binding requires OS support to retrieve the target process ID (cf. § 3.4), we added an extra handle flag (`SelfBind`) that enables user processes to directly bind handles to their current process. Process and enclave binding are overloaded using the `PMPMode` switch, i.e., only one can be used for a key handle at a time. The gray fields and reserved block (`R`) in Figure 2 indicate future extensions (cf. § 7).

The key handle creation is implemented by KeyVisor's `wrapkey` CPU instruction. It takes two memory references: one to a 384 bit `handlegen` structure, containing the user's AES user key (plaintext) and usage policy data, and one defining the output address of the resulting key handle. The policy data is similar to that included in the key handle (Figure 2), but can additionally include a 64 bit binding target ID (process or PMP ID) and an 8 bit usage counter. The `keywrap` instruction transforms the user key into a valid KeyVisor key handle. First, the usage policy data (except of the binding ID and usage counter) is copied from the `handlegen` into the key handle. Afterwards, a new IV_{handle} is generated using the LFSR. The handle wrapper then derives a cache index based on the IV_{handle} and uses it to add a new entry to the HSC, storing the tag, binding ID, and counter. Afterwards, the handle wrapper starts the authentic encryption operation by loading the required data into the AES-GCM module: the visor key as key, the user key as data, the usage policy as AAD, and IV_{handle}. The resulting cipher and GCM tag are written into the key handle together with the used IV_{handle}, completing the handle generation.

On a handle-based operation (e.g., encrypt), the handle wrapper is responsible for unwrapping the key handle and enforcing its usage restrictions. First, the handle wrapper loads the cipher, GCM tag, IV_{handle}, and AAD (usage policy) from the key handle and inputs it into the AES-GCM unit to perform the decryption and signature verification using the visor key. Afterwards, the handle wrapper looks up the HSC entry based on the cache index derived from IV_{handle} and checks if the key handle is valid and the usage policy restrictions are satisfied. On success, the handle wrapper can continue the requested operation, e.g., by forwarding the plaintext user key to the en-/decryption unit, or performing a handle revocation.

KeyVisor's revocation instruction removes a key handle from the CPU-internal allowlist.

First, the handle wrapper unwraps and checks the handle. If the operation is permitted (cf. Appendix A), the handle wrapper identifies the handle's bit-field entry in the allowlist register based on the position (set, way) of its HSC entry. Finally, the valid bit is unset to revoke the handle, and the HSC entry can be reused. If all key handles bound to a given process or PMP ID should be revoked, KeyVisor revokes the handles of all HSC entries with matching IDs.

Handle-based De-/Encryption Unit. The de-/encryption unit performs handle-based crypto operations on user data, as described in § 3.3. KeyVisor implements two respective CPU instructions: `encrypt` and `decrypt`, which take a pointer to an I/O structure and the key handle as inputs. First, the handle wrapper unit verifies the key handle, and decrypts and forwards the contained user key to the de-/encryption unit. The de-/encryption unit loads the required data (plaintext/cipher) and AAD from the memory addresses given in the I/O structure, either generates a fresh 96 bit IV_{data} using the LFSR (iff encrypt-only handle) or loads the user-given IV_{data}, and if a decryption was requested, loads the authentication tag. The unit forwards the information to the AES-GCM-128 unit which block-wise performs the requested encrypt/decrypt operation in place, i.e., directly writing to the input data address given in the I/O structure, for zero-copy processing.

TEE-based Remote Key Provisioner. We implemented the RKP (see § 3.6) as a trusted Keystone enclave [LKS+20]. Keystone enclaves are hardware-isolated using RISC-V PMP and support remote attestation. Thus, they enable remote services to verify the authenticity and security of the RKP before sharing AES keys. Remote services can establish a remote channel by performing a key exchange as part of the attestation process [KSC+18, SR20] to send the key and associated usage policy. As KeyVisor's instructions can be called from within Keystone enclaves, the RKP enclave can directly transform the key into a protected key handle and wipe the plaintext key from memory. A local service can host an RKP enclave instance and proxy the secure remote connection to the enclave, receiving the resulting key handle(s) via shared memory. The plaintext key is never leaked. In § 6.3, we present two use cases (UC-2 and UC-3) enabled by KeyVisor and an RKP.

5 Security Analysis

In this section, we discuss the security of KeyVisor.

5.1 Impact of Local Software Attackers.

As described in our threat model (cf. § 2.1), we assume local user- and OS-level attackers trying to gain unrestricted access to user keys by attacking local key handles (TM_{local}) or the remote key provisioning ($\text{TM}_{\text{remote}}$). We assume that local software transforms their user keys into protected key handles while being uncompromised. Otherwise, their keys and usage restrictions would already be under control of the attackers. For remotely-provisioned keys, KeyVisor's Remote Key Provisioner ensures that trusted remote services can securely send the keys and restriction policies to KeyVisor, without risking leakage to or tampering by local attackers ($\text{TM}_{\text{remote}}$). The Remote Key Provisioner is hardware-isolated from user- and OS-level attackers, supports authenticated E2EE communication, and directly forwards the keys to KeyVisor to transform them into protected key handles (cf. § 3.6). Even strong OS-level attackers cannot leak the unrestricted (plaintext) user keys from key handles or bypass the enforcement of their restrictions. The visor key is CPU-generated and accessible only by KeyVisor. Therefore, only KeyVisor can unwrap key handles to access the user keys. Furthermore, this implies that handles can only be used via KeyVisor's CPU instructions, guaranteeing the enforcement of a handle's usage restrictions. That is, even if local attackers gain direct control over key handles, e.g., by compromising the owning process or stealing key handles across process boundaries via side-channel attacks, attackers cannot reveal the wrapped user key or disable a handle's usage restrictions. Any manipulated or bogus key handle or usage policy is detected and refused by KeyVisor as part of the integrity checks (here: AES-GCM-based).

The only type of handle restrictions affected by local OS-level attackers is the optional process binding of a key handle (cf. § 3.4). In the current RISC-V ISA and OS implementations, processes are managed by the OS, including the page table and address space information used by KeyVisor as their unique binding IDs. Therefore, OS-attackers can bypass key handle bindings, e.g., by assigning a malicious process the binding ID of a stolen key handle. Consequently, KeyVisor's optional process bindings are currently only reliable while the OS is uncompromised (cf. $\mathrm{TM}_{\mathrm{bProc}}$ in § 2.1). In contrast, the PMP memory partitions and IDs used for TEE bindings are managed by the monitor-mode firmware. Therefore, TEE bindings are reliable even if the local OS has been compromised. Note that the threat model of TEEs assumes a trusted monitor ($\mathrm{TM}_{\mathrm{bTEE}}$), but key handles and their other (non-binding) usage restrictions are also secure under a compromised monitor.

5.2 Attacks against KeyVisor's Hardware Components.

With KeyVisor, we present a hardware security extension that is integrated into the CPU microarchitecture. Thus, it is important to make sure that our extension does not introduce new vulnerabilities to the microarchitecture.

The KeyVisor extension can, in principle, be shared between multiple CPU cores. Since most of KeyVisor's operations depend on the AES hardware unit, new KeyVisor instructions must be stalled while the extension is busy. That is, if process A / core A is currently executing a KeyVisor instruction, and process B / core B also issues a concurrent KeyVisor instruction while the hardware extension is still busy, the pipeline of process B / core B needs to stall. Note that only KeyVisor instructions are affected. B can execute non-KeyVisor instructions concurrently without a pipeline stall. Due to the sequential processing, cross-domain data leakage during parallel usage of KeyVisor can be prevented easily. Naturally, the implementation must ensure not to output old data from internal registers, e.g., in case of a failed handle verification. This can be safely achieved by clearing internal registers at the end of each operation. Though, due to the shared (limited) hardware resources and resulting stalling, it is feasible for attackers to observe the utilization of KeyVisor by measuring the latency of KeyVisor instructions and thus launch a timing-based side-channel attack. However, such an attack only leaks information on whether another process is using KeyVisor but not about the user data itself[2] or the AES keys, i.e., the visor key or user keys. Similarly, depending on the implementation, the Handle State Cache may leak information on which key handles have recently been used based on the timing of the handle verification. Again this does not reveal information about the keys or the processed data. Note that if the workload heavily relies on KeyVisor, KeyVisor could be implemented as a pipelined architecture supporting new instruction invocations in every clock cycle, optionally including further performance optimizations (e.g., prefetcher leveraging KeyVisor's deterministic memory access patterns).

Our current focus is on software-based attackers while most physical hardware attacks are out of scope, as described in § 2.1. However, note that while KeyVisor's key handles prevent DMA or cold boot attacks from stealing unrestricted user keys from DRAM, it is important to ensure that KeyVisor's AES unit and critical hardware registers containing key material are protected from physical attackers, e.g., by using masked implementations. This challenge is shared with other hardware-based key handle designs, and particularly

[2] Note that the length of the processed data may be leaked by timing. This is an inherent problem of variable-sized inputs.

relevant in settings where attackers can gain easy hardware access (e.g., IoT). Note that KeyVisor is directly integrated into the CPU package and therefore can benefit from existing CPU hardening techniques, e.g., against physical reverse engineering.

5.3 Challenges of Encrypt-/Decrypt-only

The introduction of encryption- and decryption-only key handles (cf. § 3.4) poses some challenges from a security point of view. Intuitively, cipher modes that encrypt data using a traditional block cipher like AES rely on the inverse block cipher function to decrypt the data. This is, for example, the case for AES-ECB and AES-CBC. However, when the block cipher is used in a stream cipher mode of operation, e.g., in AES-CTR or AES-OFB, the inverse function is not used. Instead, the block cipher is used to generate a keystream that is XORed with the plaintext or ciphertext, resulting in identical encryption and decryption operations. Therefore, the operations of these ciphers cannot be restricted, because attackers can use encrypt operations to decrypt data and vice versa, thus bypassing de-/encrypt-only restrictions. Consequently, while KeyVisor could provide secure key handles with usage restrictions (e.g., revocation) for non-AEAD keys, KeyVisor cannot enforce de-/encrypt-only handles for stream ciphers like AES-CTR due to their design.

AEAD ciphers like AES-GCM or ChaCha20-Poly1305 face similar challenges for these usage restrictions. Therefore, Intel Key Locker's de-/encrypt handles are insecure for stream and AEAD ciphers. However, in the following, we explain how KeyVisor's hardware support for AEAD enables the design of secure de-/encrypt-only handles for AEAD ciphers.

Encryption-only. AEAD ciphers additionally perform data authentication, i.e., after data (`dt`) en-/decryption, they generate/verify an authentication tag over the cipher and optional additional data (`ad`). Generally, in such cases, it *does not* hold that

$$Enc_{k,iv}(Enc_{k,iv}(dt, ad), ad) = Dec_{k,iv}(Enc_{k,iv}(dt, ad), ad) \tag{1}$$

since the double-encryption on the left side of the equation would yield an authentication tag different from the encryption-decryption on the right side. Therefore, one could assume that the encrypt-only restriction would be trivially compatible with AEAD ciphers. However, for constructions like AES-GCM or ChaCha20-Poly1305, it holds that

$$Enc_{k,iv}(Enc_{k,iv}(dt, ad), ad) = (dt, t'), \tag{2}$$

with t' being the authentication tag of the second encryption. That is, the double encryption returns the correct plaintext and *only* deviates in the authentication tag t' from the encryption-decryption on the right side of Equation 1. Thus, if the attacker chooses to ignore the authentication tag, they can still decrypt the message using an encryption-only handle. To avoid this, for each encryption operation that has the encryption-only restriction, KeyVisor generates IV_{data} collision-free in hardware (cf. § 3.3 or § 4), thus rendering it impossible for an attacker to choose colliding IV_{data} and perform the double-encryption to bypass the encrypt-only restriction.

Decryption-only. The decrypt instruction of KeyVisor does not return an authentication tag. Instead, it compares the user-provided tag with the tag that is computed during the decryption, and indicates the result, i.e., the validity of the ciphertext, as a return value of the instruction. While an attacker can use a decrypt-only handle to produce ciphertexts by *decrypting* plaintexts under a chosen IV_{data}, the attacker cannot produce a valid authentication tag since the decryption never returns an authentication tag. Thus, the attacker would need to forge the authentication tag without knowing the key, which, by design, is not feasible for AEAD ciphers.

Table 1: Area utilization of our KeyVisor prototype.

	LUTs	FFs	% of LUT Overhead
CPU w/o KeyVisor	154 594	134 958	-
CPU w/ KeyVisor	179 118	138 888	total: 15.9 %
AES GCM	19 428	1 851	79.2 %
Enc / Dec Unit	1 296	278	5.2 %
Gen IV	2	96	0 %
handle wrapper	1 418	164	5.7 %
HSC	848	624	3.5 %
Mem. Access	834	271	3.5 %

6 Evaluation

In this section, we analyze KeyVisor's hardware costs and measure its performance. Moreover, we show how KeyVisor's concepts solve the challenges of the three real-world use cases presented in § 2.

6.1 Area

Table 1 shows the hardware utilization of KeyVisor on a Xilinx Alveo U250 data center accelerator card. Overall, KeyVisor adds ≈16 % lookup table (LUT) overhead (≈24.5 k), and ≈3 % flip-flop (FF) overhead (≈3.9 k) to the RocketChip CPU core. Note that the relative area overhead appears bigger due to the small size of the core itself (RocketChip is a single-issue in-order CPU). On larger processors, the relative overhead will be significantly smaller. Furthermore, about 80 % of the total LUT overhead introduced by KeyVisor results from the AES-GCM unit. If a smaller area overhead is preferred over performance, a more lightweight AES-GCM implementation can be used instead. The remainder of KeyVisor's hardware costs results from its handle wrapper, data en-/decryption logic, and the HSC. Note that the HSC requires additional SRAM memory which is not shown in the table. For the prototype, we implemented a 2-way set-associative HSC with 64 sets (128 entries). Each entry holds 162 bit (cf. Figure 5), i.e., 2.6 kB of SRAM memory are added.

6.2 Performance

In this section, we first present microbenchmarks of KeyVisor's new CPU instructions to determine the encryption and decryption performance. Then, we compare the performance of KeyVisor with two widely adopted software libraries (OpenSSL, mbedTLS), and analyze KeyVisor's overhead on top of CPU-based AES-GCM accelerators. In addition, we provide a *conceptual* performance comparison between KeyVisor and its closest existing related work: Intel Key Locker. Note that, as discussed in the respective section, a direct comparison of their performance is infeasible, due to the significant platform differences between our FPGA-based RISC-V prototype and commercial x86 CPUs.

6.2.1 Microbenchmarks

Figure 4 shows the encryption and decryption performance of KeyVisor with varying data and AAD sizes in four-byte steps. Each pixel of the heatmap is averaged over 100 iterations with random inputs. The de-/encryption of payloads with 4 B of data and AAD takes 188 clock cycles on average, and that of payloads with 200 B of data and AAD takes 421 cycles. Most of the latency results from the memory interactions, as the AES data blocks need to be loaded from and stored to memory in 64 bit chunks.

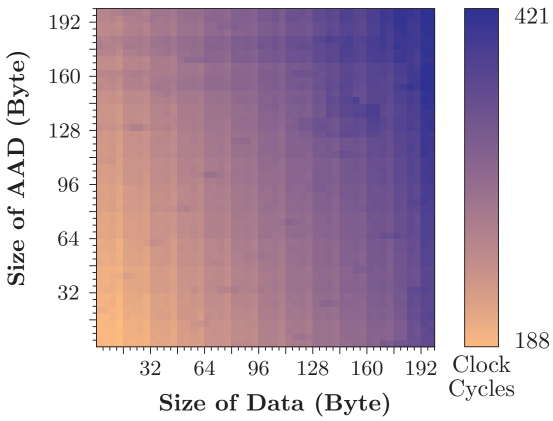

Figure 4: Performance of KeyVisor encryption and decryption for varying sizes of data (x-axis) and AAD (y-axis).

As shown in the figure, the latency of KeyVisor's encryption and decryption operations increases linearly with the input sizes of data and AAD. However, appending further data blocks increases the latency slightly faster than adding authenticated data. Note that handle restrictions like process binding or decryption restrictions do not affect the latency of KeyVisor's instructions. These restrictions are checked within a single clock cycle during the handle verification. The measurements in Figure 4 show a distinct grid pattern. Notably, executions where the data (or AAD) length is a multiple of the AES block size (16 B) have a slightly lower latency. This is because the AES-GCM unit and KeyVisor's de-/encryption unit are optimized to operate on full AES blocks. In addition, the memory access granularity and alignment can contribute to timing differences. RocketChip's memory interface is optimized to load 64 bit data blocks from memory, but requires multiple accesses to load shorter blocks, e.g., for 48 bit, a 32 bit followed by 16 bit fetch. A similar effect occurs for unaligned accesses as the memory unit restricts 2^w byte accesses to 2^w byte-aligned addresses.

6.2.2 Comparison to Software Libraries

To put the performance of our KeyVisor prototype further into perspective, we compare it to software-based encryption and decryption using OpenSSL in version 1.1.1q and MBedTLS in version 2.28.2 on the RocketChip CPU. We used the `-O3` compiler flag to ensure that the software is optimized for high performance. Note, however, that this is not an ideal comparison, as a software implementation is expected to be slower than a hardware implementation. Therefore, afterwards, we will additionally compare the performance to hardware implementations.

Like in the previous experiments, we measure the average performances over 100 runs with random input data and AAD. For payloads with 4 B of data and AAD, the encryption has an average latency of 16 232 clock cycles with OpenSSL and 22 716 clock cycles with MBedTLS. For 200 B of data and AAD, OpenSSL took 44 372 clock cycles and MBedTLS took 40 097 clock cycles on average. Notably, for the software libraries, decryption is faster than encryption. That is mostly due to the (non-TRNG) randomness initialization and the fact, that the authentication tag cannot be computed in parallel on encryptions. For the AES-GCM decryption, we measured 12 391 cycles and 9 774 cycles for 4 B of data and AAD, and 41 623 cycles and 37 727 cycles for 200 B for OpenSSL and MBedTLS respectively. To recall, the de-/encryption of 4 B of data and AAD using KeyVisor took 188 cycles, whereas the de-/encryption of 200 B of data and AAD took 421 cycles. Hence, KeyVisor is orders of magnitude faster than comparable software libraries, which lies in the nature of hardware-accelerated implementations.

6.2.3 Comparison to Hardware-Accelerated AES

To determine the difference between the performance of KeyVisor and an AES-GCM hardware implementation without handles (*pure* AES-GCM), we measure the time consumed for the key handle unwrapping and policy verification. After the unwrapping, the en/-decryption of KeyVisor uses the same mechanism as a memory-mapped hardware accelerator[3]. In our implementation, the overhead of the handle verification and unwrapping adds a constant overhead of 93 cycles when compared to a pure AES-GCM hardware implementation. That is, of the 188 cycles needed to encrypt 4-Bytes of data with 4-Bytes of AAD, 93 cycles are spent to verify and unpack the encryption key. The remaining 88 cycles include loading the plaintext from memory and the actual encryption. Since the handle verification overhead is static, the percentage overhead decreases for larger data inputs. For a typical TLS 1.2 packet, i.e., about 1 500 B of data and 13 B of AAD, KeyVisor requires 1439 cycles on average, i.e., ≈6 % overhead on top of an AES-GCM hardware unit. In our KeyVisor implementation, we utilize an open source hardware implementation. Note that, in principle, this implementation can be exchanged with any other hardware implementation, yielding different performance and area tradeoffs.

6.2.4 Comparison to Intel Key Locker

A direct comparison with Intel Key Locker (Intel KL) is not possible in a meaningful way, since the implementations are based on vastly different architectures. That is, Intel KL is available only on recent commercial grade CPUs that are trimmed for performance and efficiency, while our KeyVisor implementation is based on an open source RISC-V CPU running on an Field Programmable Gate Array (FPGA), targeting modifiability and academic experiments. Recent benchmarks of Intel KL show an encryption and decryption performance of about 3 400 MiB/s for AES-XTS on a 13th generation Raptor Lake CPU [Bae23]. These results may vary, since Intel CPUs implement dynamic frequency scaling which adjusts the processing performance to current demand. *Attempting* to compare, our KeyVisor prototype runs on a FPGA with a single core 100 MHz rocket chip CPU and achieves a performance of about 50 MiB/s for AES-GCM. Next to the performance gap between the CPU architectures, differences exist regarding the memory units, e.g., RAM, that affect the overhead when fetching the user's plaintext or cipher data for a crypto operation. Finally, Intel KL's instructions pass key handles via registers, while KeyVisor's current prototype uses memory references. Therefore, a fair comparison of the two is unfeasible.

However, we can compare KeyVisor's and Intel Key Locker's designs to estimate the expected performance difference. Both solutions unpack and verify key handles using AES. Intel Key Locker uses AES-GCM-SIV, which is slightly slower than plain AES-GCM but omits the need for collision-free IV generation, as implemented by KeyVisor using the LFSR. For Intel KL, handles with 128 bit keys are 384 bit in size, which is 128 bit smaller than the KeyVisor handles that, due to the use of AES-GCM instead of AES-GCM-SIV, additionally include an IV inside the handle. Hence, the unpacking for Intel KL can be faster by about 25 %. Moreover, depending on the configuration, KeyVisor needs to check more flags in the handle, which in some cases requires accessing the Handle State Cache (e.g., if the handle is bound to a process). This can require two additional clock cycles.

[3] A memory-mapped accelerator uses memory references as input to the accelerator. For example, the hardware-accelerated AES-NI utilizes extended registers for the I/O, and is therefore different to the approach taken with KeyVisor. Register-based approaches are typically faster, but require special registers in the CPU. An AES-NI-style register-based approach could be implemented with KeyVisor.

Based on the smaller handle size, the slightly slower AES-GCM-SIV, and the slightly faster flag-checking, we estimate that an implementation resembling Intel KL on our RocketChip CPU would reduce the static overhead for the handle unpacking and verification to about 69 cycles compared to the 93 cycles of KeyVisor. Once the unpacking and verification (incl. policy enforcement) is finished, the usage of the actual user key differs only in the cipher (i.e., AES mode of operation) between Intel KL and KeyVisor. That is, while Intel KL supports only plain AES block operations in hardware, KeyVisor integrates whole ciphers, e.g., AES-GCM, allowing for additional usage restrictions and a faster cipher performance. As discussed above, the unpacking and verification overhead is static and thus contributes only a small part to the overall de-/encryption overhead for larger payload sizes, e.g., $\approx 6\,\%$ for TLS 1.2 packets. Therefore, we expect KeyVisor to perform on par with Intel KL in real-world use cases of AEAD ciphers, as with Intel KL, major parts of the cipher must still be implemented in software, e.g., the GCM computation.

6.3 Use Case Analysis

We now revise the three real-world challenges introduced in § 2 and show how KeyVisor efficiently enables secure solutions for them. For each use case, we implement a simplified proof of concept (PoC).

6.3.1 UC-1: Ephemeral Key-value Storage for Web Services

Without KeyVisor, a web server can already store encrypted user session data (e.g., credentials) in an untrusted key-value database. While this prevents data leakage on a database compromise, without using key handles, it bares the risk that attackers can leak the key. Existing key handle solutions like TPMs or Intel Key Locker can protect the key to render *remote* attacks ineffective, as the handle is only locally usable. However, *local* attackers can still exploit leaked handles, because the handles are valid system-wide.

With KeyVisor, the web server can securely bind the key handle of its data encryption key to its process context. That way, even if a key handle is stolen by a local attacker, the CPU stops the attacker from using it, thus preventing any unauthorized data decryption. We implemented a PoC consisting of stub web and database services, interconnected via UNIX domain sockets. The web service generates a process-bound key handle to authentically encrypt a value, using the associated storage key as the AAD. The web service later queries the stored cipher using the storage key, and decrypts and verifies it using the key handle. As the key handle is only usable by the web service process and the key is part of the AAD, the web service knows that the data is secure and correct—an optional counter can additionally prevent rollbacks.

6.3.2 UC-2: Automotive Feature Licensing Control

In our second use case, we considered a licensing system for automotive pay-per-use features, that must operate securely in offline mode. The car vendors want to remotely provision lifetime-restricted signing keys to the car for authenticating feature-enable requests. However, TPM-like solutions and Intel Key Locker lack CPU-enforced revocation, and Intel Key Locker does not support remote key imports. In contrast, KeyVisor can revoke key handles based on usage counters. Moreover, KeyVisor can securely import remote keys with associated usage restrictions. That way, vendors can provision feature keys via the RKP, and KeyVisor enforces lifetime counters matching the licensed number of feature uses.

For our PoC, we use a simplified model consisting of a remote vendor licensing service, an automotive gateway processor running the majority of the car's software, and bus-connected computing units that control and enable features for car peripherals—here: a motor unit. The licensing service grants the license for the sport mode while the motor unit activates the feature when requested by the gateway using a valid license key. The gateway supports KeyVisor and a TEE-based RKP (cf. § 4) interacting with the licensing service via an authenticated (attested), E2EE connection [KSC+18, Enc21, SDH+22, SR20] to receive the license. The license includes the number of permitted feature activations (*cntr*) and a feature-specific AES key known by the motor unit, e.g., statically derived from a shared master key. The RKP transforms the key into a key handle and limits the handle usage to *cntr* authenticated encryption (signing) operations, before wiping the key from TEE memory. Afterwards, the handle is shared with the gateway service which can use it to request feature activation. To activate a feature, the gateway uses the license key handle to sign a nonce of the motor unit, and sends the request via the bus to the motor unit. The key handle is counter-restricted and will be revoked when the licensed number of uses has been completed. More formally, the key derivation, handle creation, and feature-enable request for a feature *ftr* can be described as:

$$\begin{aligned} \textit{aes-key}_{\textit{ftr}} &:= \textit{KDF}(\textit{aes-key}_{\textit{master}} || \textit{name}_{\textit{ftr}}) \\ \textit{cntr-khndl}_{\textit{ftr}} &:= \textit{handle-wrap}(\{\textit{cntr}\}, \textit{aes-key}_{\textit{ftr}}) \\ \textit{req}_{\textit{ftr}} &:= \textit{auth-enc}(\{\textit{cntr-khndl}_{\textit{ftr}}\}, \textit{name}_{\textit{ftr}} || \textit{nonce}) \end{aligned}$$

where "||" is the byte string concatenation and $\textit{req}_{\textit{ftr}}$ the request.

In our PoC, our KeyVisor-enabled FPGA represents the car with a Keystone-based RKP, and the connected workstation hosts the vendor service. The RKP enclave communicates with the vendor service via an encrypted and attested channel. A non-TEE gateway process receives the key handles from the enclave and performs the feature activation protocol with the motor unit process. We model the car bus via local UNIX domain sockets.

6.3.3 UC-3: Read-only TLS Traffic Monitor

Finally, we envision a third-party service that offers traffic monitoring (e.g., NIDS, on-path NIPS), e.g., to find attacks in the TLS traffic of monitored workstations. The monitoring service needs to have read-only access to the (decrypted) traffic, but should not be able to manipulate plaintexts. Existing approaches sharing the plaintext connection keys cannot prevent such manipulations, and also TPMs or Intel Key Locker cannot securely enforce decrypt-only key handles. With KeyVisor, it becomes possible to securely provision the TLS connection keys as decrypt-only handles for tamper-free traffic monitoring.

We envision the following design: The middlebox hosting the monitoring service supports KeyVisor with an RKP. The enterprise clients (workstations) use modified TLS libraries that securely send the AES connection keys with decrypt-only policies to the RKP. The RKP shares the decrypt-only key handles with the traffic monitor, which stores them together with the associated metadata (connection info, IVs) from the clients. The service can then look up the handles using the connection metadata, decrypt the client TLS traffic—captured on-path or forwarded by a router—and monitor the plaintext data. If the service becomes compromised, attackers cannot tamper with the connections as KeyVisor prevents usage of the decrypt-only handles for (re-)encrypting tampered packets.

We provide a PoC for clients using the mbedTLS library and TLS 1.2 connections with AES-128-GCM ciphers. We use our FPGA as both, the monitoring host and TLS target server. The TLS client runs on a separate system of the same local network. The client connects to the server to send an HTTP request. During the TLS handshake, the TLS library establishes a secure connection to the RKP enclave running on the FPGA (cf. previous use case) and shares the AES connection keys. The RKP passes the key handles to the traffic monitoring process via shared memory. For the traffic capturing, the monitoring process uses libpcap. In our PoC, the *synchronous* key sharing adds an overhead of about 15.2 % to 20.4 % on top of the TLS handshake. This is acceptable especially for mid-to-long-term connections and can be further reduced by *asynchronous* sharing.

7 Discussion and Future Directions

In this section, we discuss KeyVisor's portability to other platforms, ideas for additional usage restrictions and use cases, as well as support for alternative wrapping or user ciphers.

Porting to other Architectures. While KeyVisor's current implementation focuses on RISC-V with a TEE-based Remote Key Provisioner (RKP), KeyVisor's components could be ported to other architectures. KeyVisor's instructions must be adapted to the target ISA and the RISC-V hardware identifiers used for handle restrictions (cf. § 3.4) must be mapped to secure alternatives. Our current prototype passes instruction arguments (e.g., key handles) by reference, which is widely supported. Alternatively, arguments could be passed by value, e.g., using the 128 bit XMM or 256 bit YMM registers of x86 (if available). The RISC-V CPU privilege levels can be mapped to the respective x86 and Arm variants—e.g., user mode to x86's ring3 and Arm's EL0—and the process information from RISC-V's SATP register can be derived from the CR3 and TTBRx registers of x86 and Arm respectively. Reliable TEE binding IDs depend on the specific TEE (e.g., NS-bit for Arm TrustZone). KeyVisor's RKP could be based on a different TEE or even a full hardware extension, assuming it still satisfies the required security requirements (cf. § 3.6).

Additional Usage Restrictions and Use Cases. KeyVisor's key handle format is designed to support additional restrictions. As indicated in Figure 2, similar to a TPM, we could optionally (`DKeyMode`) generate new user keys directly inside the CPU, to never reveal keys to memory at all. Furthermore, KeyVisor could support alternative (AEAD) ciphers for user keys chosen at handle generation (`Algorithm`), e.g., ChaCha20-Poly1305 (used by Wireguard VPN) or Ascon (IoT use cases) [Don17, Chr21]. Further cipher options are discussed in the next paragraph. We also envision time-based lifetime restrictions (`Lifetime`) that incorporate a timestamp from a trusted local clock and a lifetime duration inside the key handle (e.g., in reserved bits `R`), defining when a handle gets revoked. Such a feature would be particularly useful to enforce periodic key renewal, e.g., when used for time-restricted authorization keys, similar to Kerberos tickets or X.509 certificates. KeyVisor's en-/decrypt-only key handles could additionally be used to mimic asymmetric cryptographic properties, mapping encrypt to signing, and decrypt to verifying. By replacing asymmetric keys with restricted key handles of symmetric keys, the computational workload of client devices could be decreased, especially in embedded settings. For instance, symmetric keys could be deployed into IoT boards during manufacturing as decrypt-only key handles, allowing the devices to decrypt (verify) messages from the manufacturer (e.g., trusted firmware updates) while blocking message spoofing.

Alternative Crypto Ciphers. Without loss of generality, KeyVisor's current implementation focuses on the symmetric AES-GCM AEAD cipher—for wrapping user keys into key handles as well as for user-requested crypto operations (i.e. en-/decrypt). The advantage of using AES-GCM for both is the ability to use the same hardware logic for wrapping and en-/decryption, thus, reducing the hardware footprint. However, KeyVisor could support alternative cipher schemes: KeyVisor could explore the adoption of wrapping schemes like NIST SP 800-38F [Dwo12], or AES-GCM-SIV (used by Intel Key Locker). As mentioned above, KeyVisor can also support alternative (non-AES) AEAD ciphers for user operations (or wrapping), e.g., the lightweight Ascon cipher for IoT settings, or ChaCha20-Poly1305. The permitted cipher algorithm of a key handle can be specified as part of its usage policy, as shown in Figure 2. KeyVisor could also support non-AEAD symmetric ciphers for user operations, e.g., AES-CBC or AES-CTR. However, as described in § 5.3, stream cipher modes (e.g., AES-CTR, AES-OFB) lose support for secure en-/decrypt-only handles.

Note that KeyVisor focuses on *symmetric* ciphers as they are frequently used in high-performance settings where CPU acceleration is beneficial. However, in principle, KeyVisor could also be generalized to asymmetric encryption or even signature schemes. In that case, KeyVisor could protect the private keys against local attackers and enforce usage restrictions, e.g., to remotely provision a single-use signing key for signature delegation. More special-purpose cipher candidates include the stateful post-quantum signature schemes XMSS [HBG+18, BDH11] and LMS [MCF19], since NIST recommends managing their stateful private keys securely in hardware [CAD+20]—a task that could be handled by KeyVisor's key handles. However, a detailed exploration and design of these generalizations and features is out of scope for this paper.

8 Related Work

KeyVisor is most related to cryptographic co-processors and CPU extensions aiming at key isolation. In addition, many projects provide TEEs or in-process isolation that can optionally protect crypto keys and their operations in separated domains.

Hardware Security Modules (HSMs). HSMs protect keys in dedicated memory and expose keys only as handles to the users. TPMs [Gro19] are specific crypto co-processors (HSMs) standardized by the TCG and offered by popular vendors, such as OpenTitan [ope] (based on Google's Titan chip), Apple's Secure Enclave [App24], and Microsoft's in-CPU Pluton [Mic24]. HSMs and KeyVisor follow a similar threat model: both root their trust in their processor (external vs. internal CPU), isolate keys from local attackers, and support secure remote provisioning. HSMs provide more advanced features, e.g., measurement-based attestation or X.509 certificates, but at the cost of a more complex hardware design, sometimes including extra firmware components (e.g., OpenTitan). KeyVisor's lightweight CPU extension is tailored for key protection and comes at lower area costs (without extra firmware), beneficial for embedded use cases, and with an easier, more flexible software integration via its directly callable CPU instructions. Furthermore, TPMs cannot access process or TEE identifiers as used by KeyVisor to enforce fine-grained key bindings. In addition, except for OpenTitan, all implementations are proprietary, and HSMs like TPMs have a slower de-/encryption throughput than in-CPU accelerators like AES-NI (x86) or KeyVisor. Virtual TPMs like vTPM [BCG+06] or fTPM [RSW+16] share performance issues, and, moreover, depend on additional trusted software, e.g., a hypervisor.

TEE-based Key Management Services (KMS). TEEs provide data and code isolation rooted in CPU extensions [CLD16, LKS+20, BBD+21] or dedicated co-processors [NSWM21]. They enable applications that distrust the local OS and rely only on trust in the processor, thus allowing for software KMS designs with a threat model comparable to that of KeyVisor. Intel developed an HSM inside their SGX TEE [Int] and showed with their KMRA [Int23] how to remotely provision and use private keys of web servers inside SGX. Chakrabarti et al. [CBV17] designed a KMS using SGX for OpenStack's Barbican. Similarly, Android provides secure key storage based on Arm TrustZone [Goo24] while TZ-KMS [LHX18] uses TrustZone to implement key distribution across cloud platforms. In contrast to KeyVisor, TEEs can run arbitrary user code and thus allow for tailored security schemes, e.g., for KMS in distributed systems. However, TEE-based KMS face performance and integration overhead, because applications must call into the TEEs for each crypto operation. In addition, they face a more complex attack surface than a lightweight CPU extension like KeyVisor. In contrast, KeyVisor focuses on protecting local and remotely-provisioned keys against local attackers. KeyVisor supports fast, directly usable key operations via its CPU instructions. Furthermore, while TEEs have no secure notion of a caller context, KeyVisor can enforce secure process or TEE binding policies. Thus, KeyVisor can enable TEEs to securely wrap keys in key handles for leakage protection (e.g., against side channels) and bind them to make them usable only within the correct TEE context. Moreover, as shown in § 4 and § 6.3, TEEs can be combined with KeyVisor to enable new security schemes.

Intel Key Locker. KeyVisor is inspired by the proprietary Intel Key Locker (Intel KL) CPU extension [Key20]. KeyVisor's threat model and high-level design shares properties with Intel KL, but substantially extends it with concepts that overcome several fallbacks and thus enable more advanced use cases (cf. § 6.3 and § 7). Intel KL supports only an AES-engine in hardware and thus has no control on how software uses its AES block operations, rendering CPU-enforced usage policies on specific AES modes or operations unfeasible. In contrast, KeyVisor integrates full AEAD ciphers, e.g., AES-GCM, enabling for instance secure decrypt- and encrypt-only handles as used for read-only access in § 6.3.3. Furthermore, KeyVisor provides process and TEE bindings, lifetime restrictions, as well as selective revocation, and considers the integration of a Remote Key Provisioner to securely import remote keys as usage-restricted key handles (cf. § 3.6).

Key Protection via Other Hardware Extensions. Prior work also aimed to protect crypto keys by leveraging more generic hardware extensions, such as memory tagging, taint tracking, or capabilities.

Memory tagging: ERIM [VOED+19] and libmpk [PLX+19] focus on using Intel MPK to create isolated in-process domains. One use case is to enable a web service to securely perform AES operations, comparable to KeyVisor's key handles. However, libmpk can be bypassed via malicious domain switches, e.g., by malicious libraries or code-reuse attacks. ERIM depends on a trusted OS module that vets services to prevent such malicious domain switches, but can still be bypassed by code injection attacks. Donky [SWS+20] implements a similar approach for RISC-V, but replaces the OS-dependency with an efficient user space monitor for policy management. However, in contrast to KeyVisor, Donky still depends on a trusted OS for its security guarantees, whereas KeyVisor is rooted in the CPU. In addition, the software adoption of Donky's domain definitions and cross-domain calls is much more complex than KeyVisor's instruction interface, and Donky cannot provide hardware-protected key handles that are robust against cross-process leakage.

Taint tracking: BliMe [EGLA24] enforce a taint-tracking policy at the ISA level (with HSM integration support) to prevent leakage of unencrypted client data. When client data gets unencrypted by a BliMe-enabled server, BliMe taints the data and enforces its confidentiality. However, software adoption of BliMe has strict requirements to avoid leakage, and the HSM is not part of BliMe's implementation. KeyVisor and BliMe are both RISC-V ISA extensions and do *not* depend on a trusted OS for their security guarantees. However, they differ in their focus as KeyVisor provides protected key handles with CPU-enforced usage policies and high performance crypto operations.

Capabilities: Hardware-based memory capability systems like CHERI [WWN+15] or Capstone [YWB+23] provide fine-grained memory isolation support, which could be used to isolate crypto keys. However, they lack a key handle abstraction with tailored usage policies and easy integration into existing real-world applications. Capacity [DDCNL23] implements object capabilities based on Arm PA, Arm MTE, and a kernel extension for fine-grained protection of file-based resources and memory. While Capacity can isolate a private key file and its memory buffers, Capacity relies on the security of the OS for *all* its policies and does not enforce crypto-specific policies, e.g., decrypt-only handles.

9 Conclusion

The protection of cryptographic keys is essential to the security of higher-level security schemes. Therefore, several existing designs remove the plaintext keys from unprotected memory to prevent leakage. Instead, they replace the keys with software-usable key handles that hide the plaintext from users and attackers. However, existing approaches are limited in settings requiring high-performance and fine-grained control on the usage, revocation, and deployment of key handles. For instance, external devices like TPMs feature a rather slow performance and limited insights into the CPU's execution context, while CPU extensions like Intel Key Locker lack control on how and by whom handles are used and might not support remote keys. In this paper, we therefore introduced KeyVisor, a lightweight but high-performance CPU extension for protected key handles with CPU-enforced usage and revocation control. KeyVisor combines CPU-exposed context information with new per-handle custom state to control *how* and *by whom* handles can be used, and when each handle is revoked. Furthermore, KeyVisor enables advanced remote use cases by supporting a trusted key provisioner that securely transforms remote keys to local key handles. Our open-source RISC-V prototype demonstrates how KeyVisor enables new security schemes ranging from high-performance networking use cases (e.g., TLS traffic monitoring) to embedded feature licensing schemes.

References

[AAB+16] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 4, 2016.

[ABG+20] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro*, 2020.

[App24] Apple Inc. Protecting keys with the Secure Enclave. https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/protecting_keys_with_the_secure_enclave, 2024.

[Bae23] Chang S. Bae. Patch v6 00/12 x86: Support key locker. Online, linux-kernel.vger.kernel.org, April 2023. https://lore.kernel.org/lkml/20230410225936.8940-1-chang.seok.bae@intel.com/.

[BBD+21] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *USENIX Security*, 2021.

[BCG+06] Stefan Berger, Ramon Caceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: Virtualizing the Trusted Platform Module. In *USENIX Security*, 2006.

[BDH11] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS-a practical forward secure signature scheme based on minimal security assumptions. In *Post-Quantum Cryptography: 4th International Workshop*. Springer, 2011.

[Ber23] Luca Berghella. AES-GCM 128-192-256 bits. https://github.com/BLu85/AES-GCM-128-192-256-bits, March 2023. Commit: 0b9bee5.

[BJKS21] Robert Buhren, Hans Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. In *ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[BMW+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*, 2018.

[CAD+20] David A Cooper, Daniel C Apon, Quynh H Dang, Michael S Davidson, Morris J Dworkin, Carl A Miller, et al. Recommendation for stateful hash-based signature schemes. *NIST Special Publication*, 800(208), 2020.

[CBV17] Somnath Chakrabarti, Brandon Baker, and Mona Vij. Intel SGX Enabled Key Manager Service with OpenStack Barbican, 2017.

[CCX+20] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. SgxPectre: Stealing Intel Secrets From SGX Enclaves via Speculative Execution. *IEEE Secur. Priv.*, 18(3), 2020.

[CHB19] Dhiman Chakraborty, Lucjan Hanzlik, and Sven Bugiel. simTPM: User-centric TPM for Mobile Devices. In *USENIX Security*, 2019.

[Chr21] Christoph Dobraunig and Maria Eichlseder and Florian Mendel and Martin Schläffer. Ascon v1.2: Lightweight Authenticated Encryption and Hashing. *J. Cryptol.*, 34(3), 2021.

[CLD16] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security*, 2016.

[Cor12] Jonathan Corbet. Supervisor mode access prevention. https://lwn.net/Articles/517475/, 2012.

[DDCNL23] Kha Dinh Duy, Kyuwon Cho, Taehyun Noh, and Hojoon Lee. Capacity: Cryptographically-Enforced In-Process Capabilities for Modern ARM Architectures. In *ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[Don17] Jason A. Donenfeld. WireGuard: Next Generation Kernel Network Tunnel. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[Dwo12] Morris Dworkin. Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping. *NIST Special Publication*, 800(38F), 2012.

[DWY+19] Huayi Duan, Cong Wang, Xingliang Yuan, Yajin Zhou, Qian Wang, and Kui Ren. LightBox: Full-Stack Protected Stateful Middlebox at Lightning Speed. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[EGLA24] Hossam ElAtali, Lachlan J. Gunn, Hans Liljestrand, and N. Asokan. BliMe: Verifiably Secure Outsourced Computation with Hardware-Enforced Taint Tracking. In *Network and Distributed System Security Symposium (NDSS)*, 2024.

[Enc21] Keystone Enclave. Attestation – Keystone Enclave. https://docs.keystone-enclave.org/en/latest/Keystone-Applications/Attestation.html, 2021.

[GAMG+23] Johannes Geier, Lukas Auer, Daniel Mueller-Gritschneder, Uzair Sharif, and Ulf Schlichtmann. CompaSeC: A Compiler-Assisted Security Countermeasure to Address Instruction Skip Fault Attacks on RISC-V. In *Asia and South Pacific Design Automation Conference*. ACM, 2023.

[GBK11] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *IEEE Symposium on Security and Privacy*, 2011.

[GESM17] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In Cristiano Giuffrida and Angelos Stavrou, editors, *European Workshop on Systems Security, EUROSEC*. ACM, 2017.

[Goo24] Google LLC. Hardware-backed Keystore | Android Open Source Project. https://source.android.com/security/keystore, 2024.

[Gro19] Trusted Computing Group. Trusted Platform Module (TPM). https://trustedcomputinggroup.org/work-groups/trusted-platform-module/, 2019.

[HBG+18] Andreas Hülsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: eXtended Merkle Signature Scheme. *RFC*, 8391, 2018.

[Int] Intel Corporation. eHSM (SGX Enclave Based Hardware Security Module). https://github.com/intel/ehsm.

[Int23] Intel Corporation. Intel Software Guard Extensions (Intel SGX) – Key Management Reference Application (KMRA). 2023. https://networkbuilders.intel.com/solutionslibrary/intel-sgx-kmra-on-intel-xeon-processors-technology-guide.

[Jin05] Jinpeng Wei and Calton Pu. TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study. In Garth Gibson, editor, *Proceedings of the FAST '05 Conference on File and Storage Technologies*. USENIX, 2005.

[Key20] *Intel Key Locker Specification*, 343965-001us, rev. 1.0 edition, 2020. https://www.intel.com/content/www/us/en/develop/download/intel-key-locker-specification.html.

[KSC+18] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. Integrating Remote Attestation with Transport Layer Security. *CoRR*, abs/1801.05863, 2018.

[LHX18] Shiyu Luo, Zhichao Hua, and Yubin Xia. TZ-KMS: A Secure Key Management Service for Joint Cloud Computing with ARM TrustZone. In *IEEE Symposium on Service-Oriented System Engineering*, 2018.

[LKS+20] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *European Conference on Computer Systems*, EuroSys, 2020.

[LSL+19] Hyunwoo Lee, Zach Smith, Junghwan Lim, Gyeongjae Choi, Selin Chun, Taejoong Chung, and Ted Taekyoung Kwon. maTLS: How to Make TLS middlebox-aware? In *Annual Network and Distributed System Security Symposium*, 2019.

[MAK+17] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback Protection for Trusted Execution. In *USENIX Security*, 2017.

[MCF19] David McGrew, Michael Curcio, and Scott Fluhrer. RFC 8554: Leighton-Micali hash-based signatures, 2019.

[Mic22] Microsoft Corporation. Measured boot and host attestation. https://learn.microsoft.com/en-us/azure/security/fundamentals/measured-boot-host-attestation, 2022.

[Mic24] Microsoft Corporation. Microsoft Pluton as Trusted Platform Module. https://learn.microsoft.com/en-us/windows/security/hardware-security/pluton/pluton-as-tpm, 2024.

[NL15] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. https://www.rfc-editor.org/info/rfc7539, 2015. RFC 7539.

[NSUH21] Shoei Nashimoto, Daisuke Suzuki, Rei Ueno, and Naofumi Homma. Bypassing Isolated Execution on RISC-V using Side-Channel-Assisted Fault-Injection and Its Countermeasure. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1), 2021.

[NSWM21] Pascal Nasahl, Robert Schilling, Mario Werner, and Stefan Mangard. HECTOR-V: A Heterogeneous CPU Architecture for a Secure RISC-V Execution Environment. In *ACM Asia Conference on Computer and Communications Security*, 2021.

[ope] Open source silicon root of trust (RoT) | OpenTitan. https://opentitan.org/.

[PLPR18] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. SafeBricks: Shielding Network Functions in the Cloud. In *USENIX Symposium on Networked Systems Design and Implementation*, 2018.

[PLX+19] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *USENIX Annual Technical Conference*, 2019.

[RSW+16] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. fTPM: A Software-Only Implementation of a TPM Chip. In *USENIX Security*, 2016.

[SDH+22] Fabian Schwarz, Khue Do, Gunnar Heide, Lucjan Hanzlik, and Christian Rossow. FeIDo: Recoverable FIDO2 Tokens Using Electronic IDs. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[SFSG23] Florian Stolz, Marc Fyrbiak, Pascal Sasdrich, and Tim Güneysu. Recommendation for a Holistic Secure Embedded ISA Extension. In Mehdi Tibouchi and Xiaofeng Wang, editors, *Applied Cryptography and Network Security - 21st International Conference, ACNS 2023, Kyoto, Japan, June 19-22, 2023, Proceedings, Part II*, volume 13906 of *Lecture Notes in Computer Science*, pages 62–84. Springer, 2023.

[SR20] Fabian Schwarz and Christian Rossow. SENG, the SGX-Enforcing Network Gateway: Authorizing Communication from Shielded Clients. In *USENIX Security*, 2020.

[SWS+20] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86. In *USENIX Security*, 2020.

[Syn20] Synopsys, Inc. Heartbleed Bug. https://heartbleed.com/, 2020.

[TBE+21] Thomas Trouchkine, Sébanjila Kevin Bukasa, Mathieu Escouteloup, Ronan Lashermes, and Guillaume Bouffard. Electromagnetic fault injection against a complex CPU, toward new micro-architectural fault models. *J. Cryptogr. Eng.*, 11(4):353–367, 2021.

[VOED+19] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *USENIX Security*, 2019.

[WM12] R. W. Ward and T.C.A. Molteno. Table of Linear Feedback Shift Registers. Technical Report 2012-1, Department of Physics, University of Otago, 2012.

[WWN+15] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE Symposium on Security and Privacy*, 2015.

[YF14] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.

[YWB+23] Jason Zhijingcheng Yu, Conrad Watt, Aditya Badole, Trevor E. Carlson, and Prateek Saxena. Capstone: A Capability-based Foundation for Trustless Secure Memory Access. In *USENIX Security*, 2023.

A Key Handle Revocation Strategies

KeyVisor enables flexible revocation strategies, depending on the usage policies of the respective key handles. Except for TEE-bound handles, software with higher CPU privilege levels is permitted to revoke a handle, even if not included in the usage policy. For instance, an OS kernel should be able to clean up the handles of a terminating process, though it might not be permitted to use them directly, similar to how SMAP works for user memory pages [Cor12]. KeyVisor currently enforces the following revocation policy:

For unbound key handles, i.e., not bound to an execution context (cf. § 3.4), KeyVisor permits handle revocation by any software with a CPU privilege level $\geq$ the smallest level permitted by the handle's usage policy. As unbound handles are meant to be easily shared

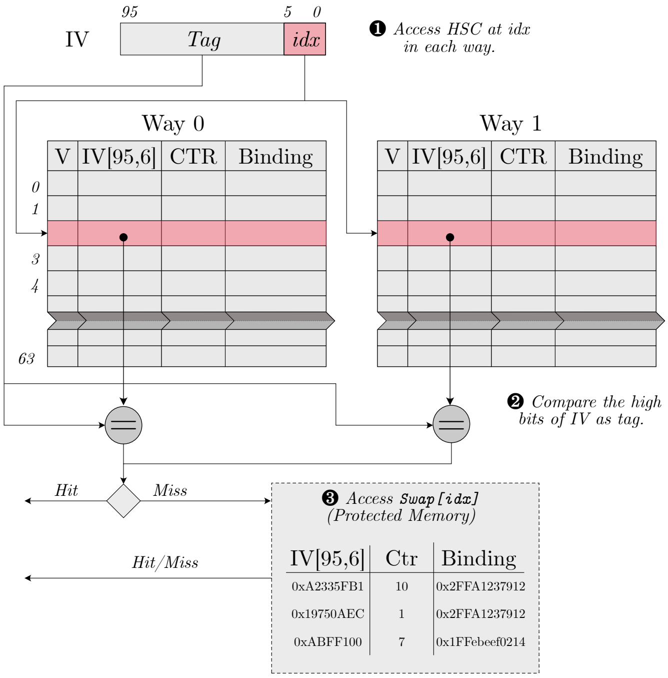

Figure 5: Overview of KeyVisor's Handle State Cache and its IV-based indexing, assuming 96 bit handle IVs ($\mathrm{IV_{handle}}$).

across processes via memory, a flexible revocation strategy is reasonable. Note that a key handle cannot be easily guessed by a local attacker process, e.g., to use or revoke an unbound handle, because of the unpredictable IV_{handle}, cipher, and tag data included in the handle. For process-bound handles, KeyVisor permits revocation by (1.) the process to which it is bound, and (2.) every context with privileges > the smallest level permitted by the handle policy. KeyVisor provides a CPU instruction for revoking all handles of a given process ID, enabling, e.g., an OS on a process termination to clean up all handles bound to that process, without the need to track the handles in memory. For handles bound to a PMP-based TEE (e.g., Keystone enclave), KeyVisor stays in line with the TEE threat model by only permitting revocation by (1.) the respective PMP context (TEE), and (2.) the PMP-managing monitor mode software (for TEE cleanup). In particular, the untrusted OS is not permitted to revoke handles bound to an isolated TEE.

System software integration of KeyVisor's revocation mechanisms can include extensions to the OS and monitor firmware. The process and TEE termination handlers can be extended to revoke KeyVisor's PID/PMP-based revocation instruction to invalidate all key handle's bound to the terminating process or TEE. Unbound key handles can be cleaned up by the users, or by the OS and monitor. The OS and monitor could create unbound handles on behalf of a user application/TEE, such that they can keep track of the respective key handles in descriptor tables, similar to those used for file or socket descriptors. Alternatively, KeyVisor could internally store the process/TEE IDs of the key handles' creator contexts, and revoke them together with the bound handles. We skip the engineering details as invoking KeyVisor's revocation instructions is straightforward and the OS/monitor integrations are not specific to KeyVisor and do not require new concepts.

B HSC Swapping to RAM (optional)

In principle, with key handles protected using AES-GCM, KeyVisor's Handle State Cache (HSC) could allow for up to 2^{96} valid key handles, as it derives the cache indices and tags from the 96bit GCM handle IVs (cf. Figure 5). However, since on-chip area for CPU-internal memory is expensive, a much smaller HSC is preferred, e.g., a 2-way cache with 64 sets as used in our prototype (cf. § 4). If the cache is full, no more handles can be created until the next revocation. To still allow for a practically unlimited number of handles, the HSC can optionally be extended to act as an LRU (least recently used) cache that swaps handles to memory when the HSC set is occupied. For this, a memory region must be reserved as a handle swap region. When entries are swapped out to non-secure memory, they are authentically encrypted using a CPU-internal storage key statically derived from the visor key, with the IV_{handle} tag and index being signed as AAD. In addition, a monotonic counter value must be included in the AAD to prevent rollback attacks against the swap region [MAK+17]. With swapping enabled, KeyVisor must inspect the swap memory on a cache miss using IV_{handle} and swap in decrypted entries on demand. As LRU-based swapping is well-known (e.g., page tables) and not specific to KeyVisor, we skip further details of a potential implementation.