A Linux kernel cryptographic framework: Decoupling cryptographic keys from applications* [extended version]

Nikos Mavrogiannopoulos
Dept. of Electrical Engineering/COSIC
Katholieke Universiteit Leuven

Miloslav Trmač
Red Hat

Bart Preneel
Dept. of Electrical Engineering/COSIC
Katholieke Universiteit Leuven

May 2013

Abstract

This paper describes a cryptographic service framework for the Linux kernel. The framework enables user-space applications to perform operations with cryptographic keys, while at the same time ensuring that applications cannot directly access or extract the keys from storage. The framework makes use of the higher privilege levels of the operating system in order to provide this isolation. The paper discusses the relevant security requirements and expectations, and presents the design of the framework. A comparison with alternative designs is also provided.

1 Introduction

There are many types of attacks against software applications, such as code injection, control flow modification [30, 1], etc., attributed to software vulnerabilities. Their elimination has been proven to be a Sisyphean task and today, software vulnerabilities are mitigated using a combination of defensive programming [27] and retroactive patching. As a result, there is a risk of an attacker 'taking control' of an application exposed to a public network (such as a web server), maybe for some limited time until the vulnerability is known and the system is updated.

The cost of such compromise may be significant for the stakeholders. One example are on-line shops today that typically use TLS [11] to authenticate to the client and setup a secure communications channel. The secure channel is setup using a server certificate and a corresponding private key. Because

*A shorter version of this paper was presented at the 27th ACM Symposium on Applied Computing. http://dx.doi.org/10.1145/2245276.2232006

in typical web server applications the private key resides in the same address space[1] as the server, its compromise is equivalent to a compromise of the private key. Thus, an adversary with the ability to compromise the web server could retrieve the private key and obtain the ability to masquerade as the original shop to its clients. This would be possible even after such a compromise is no longer possible (e.g., caused by an automatic software upgrade, or by installing a newer operating system). This might be catastrophic for the reputation of the shop.

A defense mechanism for the above attack is decoupling cryptographic keys from the software that uses them. This is typically achieved using hardware security modules or smart cards [29], that store the cryptographic keys in a medium that allows operations without exposing them. That approach effectively raises the protection of cryptographic keys from the cost of an application compromise, to the cost of compromising a hardware module. However, there is a possibility to increase the protection levels without involving new hardware. Operating systems (OS) are typically designed with security in mind [18], and provide protection mechanisms such as segregation of the OS with user applications and isolation of independent processes. These mechanisms can be used to decouple cryptographic keys from applications, providing an additional layer of security to a system where hardware security modules are not practical or are too costly (e.g. low-cost servers, or mobile phones).

Today, frameworks offering isolation of cryptographic keys from applications exist [24, 23], based on the separation of processes and the access control mechanisms offered by the OS. In this paper we present a cryptographic framework that decouples cryptographic keys from applications based on the separation between the OS and its processes. Apart from the description, we discuss the difficulties this approach imposes as well its advantages and disadvantages comparing to an alternative design. Moreover, in order to ensure the safety of the stored keys, we base the design of the framework on principles that delegate the protection of stored keys to the strength of the supported algorithms.

The next section provides a background on existing work. In Section 3 we formalize the threats on current operating systems and set the basic requirements for our framework. In Section 4 we describe our framework on a high level and present its architecture. A security analysis is provided in Section 5. In Section 6 a performance analysis is provided and the following section discusses the differences between our approach and alternatives. Section 8 concludes.

2 Background and related work

Hardware security modules (HSM) and smart cards are the main medium used today for decoupling cryptographic keys from software and are often required by governmental standards [26, 17]. Their operation is based on a logical and physical separation of cryptographic keys from the applications using them.

[1] By the term 'address space' we mean memory accessible in a process. Each process has its own address space.

For example a typical PKCS #11[2] smart card or a TPM module [29] allow for cryptographic operations (e.g., RSA decryption) using stored keys, without exposing them.

Because the deployment of hardware security modules is not always feasible, a few software-only approaches to separation already exist, based on existing isolation mechanisms[3]. The CNG API [24] is a redesign of the old Microsoft CryptoAPI, that allows the decoupling of cryptographic keys from applications. It uses a special 'key isolation process' executing with the permissions of a system user which provides services, such as cryptographic key storage and operations to other applications. A similar approach is used by another framework, the lite security module [23] (LSM). It provides a PKCS #11 API to applications in order to access a central daemon. The daemon executes under a system user and provides key storage and operations to the other applications. Both frameworks allow persistent keys that cannot be exported.

The protection level of security modules, however, is not easy to assess. That is because the available operations should be selected in way that no combination of them is sufficient to obtain or reconstruct the stored keys. This requirement, although simple to formulate it has been shown in practice that it is hard to achieve in real-world security modules [5, 7].

The goal of a formal proof of key safety in security modules is on-going work, and currently we only have indications of security in restricted designs [10, 6] that support few operations. Thus, in our design we make sure that the cryptographic operations implemented and made available are not sufficient to extract the stored keys. We will use a minimal set of operations carefully chosen for attack resistance properties. E.g., a signing operation using algorithms that are resistant to adaptive chosen-message attacks [13].

The term security module is used in this paper to describe any framework or hardware that provides operations on cryptographic keys without exposing them.

3 Model and Requirements

We chose our model based on the current single level[4] operating system design. In that model software is comprised of an operating system running in a different protection ring[5] than normal applications. This requirement ensures that normal applications cannot access data available to the operating system only. We also assume that attacks on the operating system are far less likely to

[2] A cryptographic API that provides logical separation between keys and operations. It is mainly used to access operations on hardware security modules and smart cards.

[3] Software that provides emulation to a security module but provides no isolation between cryptographic keys and the application was not considered in our study.

[4] Single level is used here to distinguish between Multi-Level Security (MLS) systems.

[5] We use the term protection ring to assume a strict separation of data and code between different access control levels. This terminology was described during the design of Multics in [32] and today it is a standard component of Operating Systems and CPUs [16, Privilege Levels p. 5–9].

succeed than attacks on specific software. This assumption is supported by [30] which concludes that "Application vulnerabilities exceed OS vulnerabilities".

A user called administrator, is able to directly control the OS executing in the highest protection ring and there is at least one user that operates solely on a lower protection ring. The applications running on a system are assumed to be running with the privileges of a normal user. We use the common terms user-space to reference the lower privilege rings and kernel-space for the higher privilege ring the OS operates.

3.1 Threats

In this model of an OS, applications operate in user-space and utilize cryptographic keys. We assume that application vulnerabilities might exist for some time until the system is patched. Those give adversaries a time window to execute malicious code at the same ring as the applications running in the system. Thus our main threats are:

- An adversary that has taken control for limited time of a user-space application and is trying to recover the protected keys; the adversary might be collaborating with another legitimate user;
- An adversary that is a legitimate system user and wants to extract raw keys; in contrast to the former adversary, this one is not restricted by time.

We want to protect keys in the presence of such adversaries. It should be noted that in this model, there is no distinction between operating system access and administrative access. This is due to the design of the modelled system.

3.2 Requirements

Table 1: Features and supported algorithms.

Feature	Supported Algorithms
Key generation	Symmetric keys and Private/public key pairs
Encryption and Decryption	RSA PKCS #1 v1.5, RSA PKCS #1 OAEP, AES-CBC, AES-CTR, AES-ECB, 3DES-CBC, 3DES-CTR, 3DES-ECB, CAMELLIA-CBC, CAMELLIA-CTR, CAMELLIA-ECB, RC4
Signing and verification	DSA, RSA PKCS #1 v1.5, RSA PKCS #1 PSS, HMAC-MD5, HMAC-SHA1, HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA-384, HMAC-SHA-512, MD5, SHA1, SHA-224, SHA-256, SHA-384, SHA-512
Key wrapping	AES-RFC3394, AES-RFC5649
Key derivation	Diffie-Hellman as in PKCS #3

A list of high level requirements for the framework concerning the cryptographic keys to be protected, follows.

1. Users should be allowed to perform cryptographic operations using the keys, but the key material should not be extractable;
2. Keys should be protected from all known side-channel attacks;
3. All cryptographic material such as keys must not be accessible from applications that use them, or from any other application, either directly or through a process memory dump;
4. Keys should be transferable to other systems in a secure way.

Our requirements for the features on the implementation were set to include the required functionality for TLS 1.2 [11] protocol implementation, and for its interface to be translatable to the PKCS #11 API. The operations and algorithms implemented are summarized in Table 1.

4 Architecture

In our design we want to emulate the hardware isolation of cryptographic operations from actual keys using the ring separation between the operating system and user-space as shown in Figure 1. Our goal is to completely isolate cryptography from the processes that actually use it. For that reason we implement a kernel service –in our prototype a Linux kernel module– that provides operations to cryptographic keys without exposing them. This would fulfill all of our requirements, as shown in Section 3.

Our framework, named NCR, is implemented as a self-contained Linux kernel module that provides access to cryptographic operations. It supports a synchronous API for communication with the user-space based on the `ioctl()`[6] system call. We emulate the hardware isolation of cryptographic operations from actual keys using the ring separation between the OS and user-space as shown in Figure 1. Our goal is to completely decouple cryptography from the processes that use it.

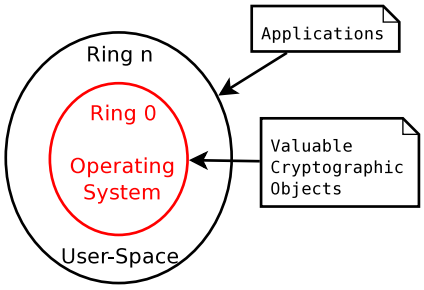

Figure 1: High level architecture

[6] `ioctl()` is a primitive communication interface between kernel and user-space. It typically works by allowing the caller to specify the operation and a structure with the operation input data.

4.1 Usage of keys

Applications should be allowed to perform operations using a set of keys in a way that only the outcome will be available to them. For this reason keys should be accessible and referenced as abstract objects, without exposing their contents. However, not all keys should be treated the same. Security protocols, typically operate using long term keys, used for a key exchange that outputs short term keys valid only for the current session (session keys). Typical cryptographic protocols such as the Internet security protocols [11, 20], satisfy the *known session keys* property of [4], which ensures that the knowledge of past session keys does not affect protocol security. Thus, in order to avoid protocol specific key derivation algorithms being present in the framework, we explicitly distinguish keys as 'long term' and 'session' keys.

This distinction strikes a balance between pragmatism and need for key protection. Operations on long term keys will be protected by the proposed framework, while session keys will be generated and used in user-space. The distinction is not arbitrary. Session keys are valid for limited time and protect data in the application address space. Hence, if one is able to extract those keys during their validity period, it is not unrealistic to claim that the data they protect could also be extracted.

To prevent attacks due to interaction between different algorithms the public and private key pairs are tied to a single algorithm, such as RSA or DSA, and we enforce the same rule for secret keys as well. For example a key that is tagged to be used with the AES algorithm cannot be used with RC4 algorithm. In addition we allow for some keys to be used as input to a cryptographic hash or MAC function, in order to be consistent with the usage of secret keys for key exchange as in TLS with pre-shared keys (TLS-PSK) [12].

4.2 Framework components

Our design consists of two basic components, one is the 'cryptographic backend' and the NCR component that provides operations to user-space applications. Each component is described in detail in the following paragraphs, and a high level interplay of the components is shown in Figure 2.

4.2.1 Cryptographic backend component

This component of the design is based partially on the existing Linux kernel asynchronous symmetric cipher and hash API, and partially on libtomcrypt and libtommath. The latter libraries were used to implement the asymmetric cryptography backend that is not part of the Linux kernel cryptographic API. The libraries were modified for the Linux kernel and were used for the implementation of RSA, DSA and Diffie-Hellman algorithms. The Linux random number generator [14] was used to provide randomness.

The main purpose of the cryptographic backend component is to provide the main NCR component with a set of algorithms to support the required operations as in Table 1.

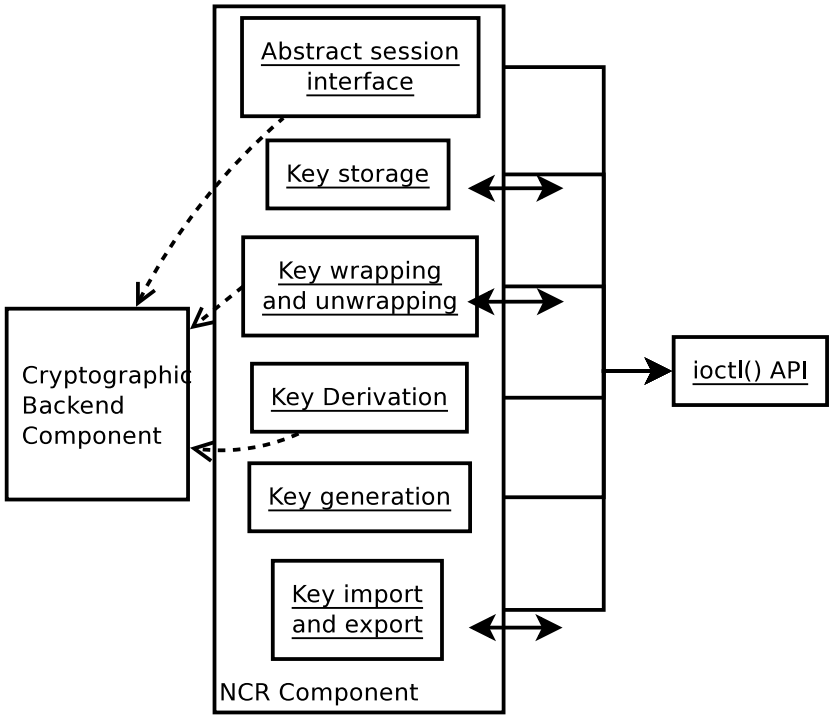

Figure 2: Component view of the framework. The entry and exit points for cryptographic keys are marked with double arrows.

4.2.2 NCR component

The NCR component contains the main implementation of the provided features. It operates on key objects that are only available within the component. The key objects are marked on generation or import with policy flags that indicate exportable status, wrappable status, or wrapping[7] status. This approach allows for key objects of a gradual security level. The highest being the keys that have no policy flags set and thus cannot be exported nor wrapped with another key, and the lowest being keys that are marked as exportable. We refer to keys with no policy flags as safe keys.

To enforce the security flags throughout the component, key objects can enter and leave the component through specific points only, the 'Key storage' and possibly the 'Key wrapping' subsystem as shown in Figure 2. Keys marked as exportable can also use the 'Key export' subsystem. The security flags are checked and enforced by the above subsystems, which also ensure that keys cannot be decoupled from their flags.

Key objects A mandatory access control mechanism is used to restrict access to cryptographic keys. Each key object on creation is tied to a particular algorithm and is given an owner and a group. The key access policy is being applied by the subsystem that is performing the operations, the abstract session interface. The usage of a mandatory access control instead of the familiar in Linux discretionary has the advantage of preventing the transfer of keys from users under some outsider's control to other unauthorized users of the system.

Abstract session interface This interface provides a generic way for the supported diverse algorithms to be used. All of the supported algorithms operate on some input data and their functionality is summarized in Table 2. signature verification as well as message digest operations. These operations are mapped to three actions, or `ioctl()` calls, being `NCRIO_SESSION_INIT`, `NCRIO_SESSION_UPDATE` and `NCRIO_SESSION_FINAL`. The `NCRIO_SESSION_INIT` action initializes the algorithm in the proper mode, i.e., encryption mode or decryption mode, and associate the operation with a key object. Subsequent `NCRIO_SESSION_UPDATE` actions operate on input data, i.e., by hashing or encrypting, and might generate output data depending on the operation. The `NCRIO_SESSION_FINAL` will terminate the operation and if required output data (e.g. in signature generation). That way, a class of different looking operations is served using three basic actions, being compatible with the PKCS #11 API.

Moreover, to avoid the need of multiple system calls for simple operations, i.e., encryption of a given data block we introduced `NCRIO_SESSION_ONCE`, which combines the functionality of the above operations in a single system call, something that is shown in Section 6 to have impact in performance.

Key wrapping and unwrapping The key wrapping subsystem allows transferring keys securely to other systems by encrypting a key with another. The

[7] Key wrapping refers to the technique of encrypting a key with another.

Algorithm type	Operations
Symmetric ciphers	Encryption, Decryption, Encryption & Authentication, Decryption & Verification
Asymmetric ciphers	Encryption, Decryption, Signature generation, Signature verification
Hash algorithms	Message digest generation
HMAC algorithms	MAC generation, MAC verification

Table 2: Operations available for each algorithm type.

subsystem accepts key encryption keys if they are marked as wrapping keys. The marking of a key as wrapping or unwrapping is a privileged operation (requires administrative access). An integrity check is performed by all the algorithms supported by the subsystem to ensure that random data cannot be unwrapped to a valid key. The type and associated algorithm with the key are included with the wrapped data to distinguish between different keys. The calls for wrapping and unwrapping keys are `NCRIO_KEY_WRAP` and `NCRIO_KEY_UNWRAP`. Because this operation separates the keys from their associated flags, it is a privileged operation.

Key storage A difference with user-space security module frameworks is that kernel-space provides no ability of data storage[8]. For this reason we adopted a design where the storage is delegated to the user-space applications. The protection of safe keys is being done by encrypting and authenticating everything with a system-wide master key. The keys, meta-data such as flags and an access control list are encrypted and authenticated, and handed out as binary data to user-space. Hence, the key storage facility –provided by the framework's user– is agnostic on the actual keys it stores. It is not able to recover any key, and can only pass it to kernel-space for loading. No naming scheme is imposed by the framework and the user can adopt any convenient scheme for the stored key objects. The corresponding calls for storing and loading keys for storage are `NCRIO_KEY_STORAGE_WRAP` and `NCRIO_KEY_STORAGE_UNWRAP`.

System-wide master key As previously discussed a master key is required to allow the key storage subsystem to operate. The only requirement for this master key is that it has to be loaded by a privileged user –the administrator. In a typical deployment we expect that the key will be provided either as part of the boot process or protected using the filesystem access control mechanism[9]. It should be noted that the overall system security is only as strong as this master key.

[8]Note that the Linux kernel can work without any writable filesystem and its subsystems should be operational without any facility for storage.

[9]Storing the system key in a hardware security module, when available, might seem to increase the defenses of the framework even against an adversary with administrative access. However, although such protection would prevent him from recovering the storage key, it would allow him to recover all the locally stored long-term keys. Because we expect long-term keys to be locally stored, the framework would not protect against this adversary.

Key derivation Unlike the ciphers and other algorithms supported by the 'Abstract session interface', the Diffie-Hellman key exchange, cannot fit the `NCRIO_SESSION_INIT`, `NCRIO_SESSION_UPDATE` and `NCRIO_SESSION_FINAL` formality of operations. To solve that issue, PKCS #11 defines a way to operate it in software by using a key derivation API, which we adopt. This key derivation API is a single action `NCRIO_KEY_DERIVE` that accepts an input key, an output key and parameters.

Key generation This subsystem supports two actions `NCRIO_KEY_GENERATE` and `NCRIO_KEY_GENERATE_PAIR`. The former action generates a random key of a given bit size to be used in MAC and symmetric ciphers. The latter generates a public and private key pair suitable for an algorithm such as RSA, DSA or Diffie-Hellman. All generated keys are tied to a single algorithm, and special parameters are allowed to fine-tune the key generation, i.e., by specifying bit lengths for symmetric keys, or prime and generator for Diffie-Hellman keys etc. Special flags can be given to keys during generation time. They are split in operation and policy flags as in Table 3. However, not all combinations are allowed, to prevent interactions between different algorithms (e.g., prevent a key to be used for RSA signing and encryption) [19]).

Table 3: Flags used to mark keys. More than one flag may be applicable to a key.

Operation flags	
Encryption	The specified key can be used only for encryption
Decryption	The specified key can be used only for decryption
Signature	The specified key can be used for signing only
Verification	The specified key can be used for verifying signatures only
Policy flags	
Wrapping	The specified key can be used for wrapping other keys
Unwrapping	The specified key can be used for unwrapping other keys
Wrappable	The key can be wrapped by other keys
External	The key was not generated within the framework
Exportable	The key can be exported from the component

Key Import and Export Keys that are explicitly flagged as exportable can be exported using the `NCRIO_KEY_EXPORT` action. Symmetric keys are exported in raw format, and public and private keys encoded using ASN.1 DER encoding rules. For public keys the SubjectPublicKeyInfo field of an X.509 certificate [8] is used. and for private keys an encoding was used, that is compatible with other libraries such as OpenSSL and GnuTLS[10].

[10] An alternative would be to use PKCS #8[28] encoding for private keys, but it was not used because it involved higher complexity than the current approach. This might be reconsidered in a future revision.

4.3 Extensibility

Since the `ioctl()` system call is limited to static structures and thus does not provide much room for extensibility, we created a hybrid-API based on `ioctl()` and the netlink[11] attributes. That way we allow for a number of attributes to be specified per call to provide an extensible API.

5 Security analysis

In this section we discuss how this implementation defends against attacks on cryptographic key isolation frameworks and side-channel attacks. In Appendix A we demonstrate through a human-directed proof how a particular set of security objectives, based on our requirements, are met, in the context of a TLS server application.

5.1 Cryptographic keys protection

Several cryptographic APIs that enforce isolation of keys, have issues that result in modification of key attributes [7, 6] by an adversary. We counter those issues by enforcing keys and attributes to be handled as a single unit in all operations. When for example, keys are exported from the framework, i.e., for storage, the exporting function will not only encrypt but also authenticate the data [15] to prevent modifications. This authentication depends only on the system key, preventing attacks as in [6].

We summarize the actions allowed on each key type in Table 4, and discuss individual aspects of security of the different keys in the framework, in the following paragraphs.

5.2 Safe keys

Safe keys have no policy flags and only allow cryptographic and storage operations. The supported ciphers are known to be secure against chosen-ciphertext and chosen-plaintext attacks, and the signature algorithms against adaptive chosen-message attacks [13]. This ensures that cryptographic operations alone will not reveal the key. The storage operations as discussed above encrypt and authenticate the key itself, as well as all metadata using the system's master key. This way unauthorized modification or decryption by all types of adversaries is prevented. In addition to the file system's access controls that will be used to store the key, the storage key format includes all metadata, e.g., the corresponding cipher and an access control list that specifies the users allowed to unwrap it. Thus, an adversary that has successfully attacked one user account can not use it to load another user's keys.

[11] A framework for communication of user-space with kernel.

The safety of those keys depend solely on the storage subsystem since it is the only subsystem that can be used to export them. The import and key generation subsystems are the only subsystems that can be used to create these keys.

5.3 Wrappable keys

Wrappable keys cannot be exported in raw but have to be encrypted by another designated key available to the framework. It is used as a mechanism to allow transfer of keys to other systems in a secure way, i.e., without revealing them in the process. In [7] Clulow describes several attacks to PKCS #11 wrappable keys. We give a summary below:

1. Wrapping with a weaker key;
2. Inserting a known wrapping key and using it to wrap the key;
3. Decryption of a wrapped key using the wrapping key;
4. Key Conjuring: unwrapping a random key and storing it as a key;
5. Secret Key Integrity: splitting the wrapped key and unwrapping each part separately in order to perform a brute-force attack;
6. Private key modification: Replacing wrapped blocks with different values.

We counter attacks 4-6 by not allowing arbitrary and simple wrapping mechanisms but only algorithms that include an integrity check. Currently the supported ones are the key wrapping methods described by RFC3394 and RFC5649 [31, 15, 25]. The 3rd item, of using the wrapping key to decrypt the wrapped key, is countered by prohibiting encryption and decryption with keys marked as wrapping keys. The 2nd item is countered by having the addition of wrapping keys a privileged operation, thus prohibiting non-privileged users from inserting their own wrapping keys.

5.3.1 Keys marked as hashable

Keys that are marked as hashable only, allow in addition to operations on safe keys, using the raw key as input to a cryptographic hash or HMAC. Because of that there are might be cases where weaknesses of the available hashes can be used to recover the used key. To prevent that we allow hashable keys to be used only with algorithms that are known to guarantee the 1^{st} preimage resistance property of a cryptographic hash function.

5.4 Exportable keys

Keys marked as exportable cannot be protected by this framework. They should be considered as having the same security level as any keys available to user-space.

Table 4: Operations that are allowed on different key types. The marked boxes indicate what kind of operations are allowed on each key.

Allowed operations	Safe key	Wrappable key	Hashable key	Exportable key
Cryptographic	⋆	⋆	⋆	⋆
Storage	⋆	⋆	⋆	⋆
Wrapping		⋆		
Hashing			⋆	
Export				⋆

5.5 Side channel attacks

Adversaries of such a framework have incentive to take advantage of side-channel attacks such as the ones described in [3, 22], to use timing and other information obtained by operations in order to extract the raw keys. For the purposes of this project we implemented the counter measures for the RSA algorithm, known as RSA blinding [22]. To counter timing attacks in the Linux kernel symmetric cipher subsystem, constant time ciphers should be added. This was not performed as part of our prototype but would be required for a real-world deployment.

6 Performance Analysis

In this section we show that the performance cost of decoupling of cryptographic keys from user-space processes. We compare the NCR framework (`/dev/ncr`) against a key isolation framework based purely in user-space. Moreover we include performance results of non-isolation frameworks such as the Linux port of the OpenBSD framework[12] [21] (`/dev/crypto`), and a user-space implementation of the AES. The latter two would reveal the performance "loss" by the usage of a key isolation framework.

For the benchmark we created a utility that submitted requests for encryption on each framework of chunks of data with different size and a fixed key. The AES cipher in CBC mode, and the NULL (dummy) cipher were used in our tests. The same implementation of AES was used in all frameworks under test, and all were tested on the same hardware and Linux kernel version 3.0.0.

For a submission of a chunk of data for encryption, the `/dev/crypto` interface requires 3 system calls, the same as the NCR framework. However NCR allows for submission of requests using a single system call, the `NCRIO_SESSION_ONCE`. We called the latter '`/dev/ncr` single' variant and included it to our benchmark to assess the optimizations obtained by reducing the number of system calls.

The only existing user-space key isolation framework that was available for the

[12]The Linux port of the OpenBSD framework was selected because of its performance advantage [9] over the native Linux cryptographic API (AF_ALG).

same system was LSM [23], but its design is based on the sockets API, requiring TCP/IP processing and data copying between the processes. A comparison with the NCR or the `/dev/crypto` frameworks that are zero-copy[13] would be negatively affected by such overhead. For that reason we implemented a user-space benchmark that would operate using the zero-copy principle. It involves two applications sharing memory and using semaphores for synchronization of operations.

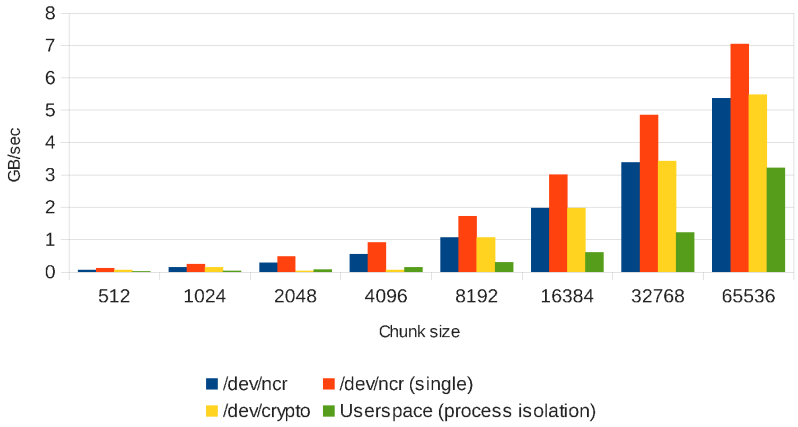

Figure 3: Performance comparison of `/dev/ncr`, `/dev/crypto`, and a user-space crypto server on the NULL cipher.

In Figure 3 we show a comparison of all frameworks using the NULL cipher and in Figure 4, a comparison using the AES cipher. Since the NULL cipher performs no operations, the former results reveal the overhead of the framework and its maximum performance limit. Comparing to the `/dev/crypto` interface we see that the NCR implementation has no noticeable performance difference. This illustrates the fact that decoupling of cryptographic keys from processes does not introduce any performance cost to a kernel framework. We can also see from the performance of NCR with a single system call, that the impact of the reduction of system calls from 3 to 1 is important. That is because system calls require context switches (switch execution from user-space to kernel-space) which come at a considerable cost.

The user-space based isolation framework shows very poor performance compared on the others. Profiling of the benchmark applications showed that this performance difference is due to the usage of semaphores for synchronization of the two processes. In the NULL cipher case, 50% of the execution time on each process was spent in synchronization. That is because semaphores in Linux although they are based in user-space futexes, do not completely avoid context switches, causing the observed poor performance. The difference between user-space and kernel-space implementations is reduced to $\approx 10\%$ when a software algorithm such as AES is involved, even though, on the NULL cipher

[13]By zero-copy we mean that no data are copied from the memory of the requesting application to memory of the serving application or kernel.

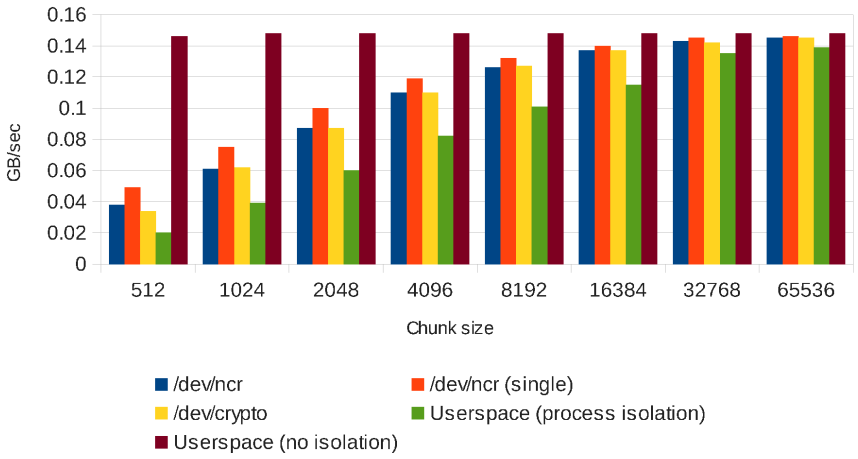

Figure 4: Performance comparison of `/dev/ncr`, `/dev/crypto`, a user-space isolation framework and a user-space non-isolation framework, on the AES-CBC cipher.

the difference exceeds 50%. Thus over the Linux kernel, a user-space approach to cryptographic key isolation would have disadvantage on high-rate or high-bandwidth transactions.

Both isolation frameworks, NCR and the userspace, are in a disadvantage comparing to userspace non-isolation framework as shown in Figure 4. Although the performance difference seems to become insignificant as larger data chunks are used, large chunks of data might not be the typical use-case of such a framework. For that we would suggest that a key isolation framework should be used after the benefits of isolation are weighted against the performance loss.

7 Comparison with alternative approaches

The differences between different framework designs, such as based on the separation of processes or the separation with the OS are not limited to performance. A separation framework might have additional goals from the requirements we set in Section 3. For example the CNG framework [24] abstracts the usage of hardware security modules, smart cards or software isolation using the same interface. This is not easily implemented in a framework residing in the OS, since functionality such as access to smart cards[14] is not typically available. Portability is another concern. A cross-platform solution cannot rely on a particular kernel interface.

On the other hand, there are advantages of an OS based framework. This is mainly performance as illustrated in the previous section, as well as efficient ac-

[14]For example parsing PKCS #15 structures or PKCS #11 modules requires an addition of large subsystems to the OS.

Table 5: Comparison of the different approaches for cryptographic key isolation.

	Kernel-space framework	User-space framework
Performance	+	
Combination with HSMs		+
Portability		+
Secure bootstrapping	+	
Efficient access to hardware cryptographic accelerators	+	

cess to cryptographic accelerators [21] and the secure bootstrapping property [2]. Cryptographic accelerators, when not implemented in the CPU instruction set are only available through a kernel-space driver. A kernel-space cryptographic framework is in advantage of using the driver directly without the cost of context switches. In a secure bootstrapping architecture each system module verifies the modules it loads (or uses). For example, the system loader would verify the OS and the OS would verify the loaded applications and so on. Having the cryptographic framework in the OS allows for a clear and simple design since the OS would already contain the functionality needed to perform signature verification, i.e., the algorithms, and this functionality has already been verified by the system loader as part of the OS. We provide a summary of the advantages and disadvantages in Table 5.

8 Concluding remarks

In the previous sections we have presented the implementation of a cryptographic framework for the Linux kernel. The main goal of the framework is to decouple cryptographic keys from the applications using them, by confining the keys to kernel-space. This confinement is achieved through the enforcement of rules on the way keys are generated and utilized in order to protect against known attacks on cryptographic APIs. The API is practical, in the sense that it can be used to fully implement cryptographic protocols such as TLS 1.2 [11], and can be used to provide services under the PKCS #11 API. It is available to all system users and the administrator role, although required, is limited to maintenance aspects of the framework, such as loading the master keys, and the generation of wrapping keys.

In addition to protecting cryptographic keys this framework provides user-space applications with access to cryptographic hardware accelerators in a similar way as in [21]. This is especially important in embedded systems with limited processors, where the software implementation of a cryptographic algorithm can be an order of magnitude slower than the hardware one.

Our performance evaluation showed that the throughput of the framework is equivalent to another kernel-based cryptographic framework that does not decouple keys from applications, indicating that there are no issues in the design that hinder performance. Moreover we show that alternative designs consisting

of user-space only components will have performance degradation, caused by the inter-process communication overhead. Thus we believe that our software implementation provides a practical additional layer of protection for cryptographic keys.

9 Acknowledgments

The authors would like to thank Steve Grubb for the inception of this project, Jan Filip Chadima for implementing a user-space interface to the NCR framework, Stephan Müller, Tomáš Mráz, Óscar Repáraz, Andreas Pashalidis, Nessim Kisserli, Jan Cappaert and the anonymous referees for valuable comments and feedback. This work was supported in part by the Research Council K.U.Leuven: GOA TENSE, by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy) and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen) SBO project.

References

[1] Aleph-One. Smashing the stack for fun and profit. *Phrack*, (49), November 1996.

[2] W. A. Arbaugh, D. J. Farbert, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE symposium on security and privacy*, pages 65–71. IEEE Computer Society, 1997.

[3] D. J. Bernstein. Cache-timing attacks on AES, 2005.

[4] S. Blake-Wilson, D. Johnson, and A. Menezes. Key agreement protocols and their security analysis. In *Crytography and Coding*, volume 1355 of *Lecture Notes in Computer Science*, pages 30–45. Springer Berlin / Heidelberg, 1997.

[5] M. Bond and R. Anderson. API-Level attacks on embedded systems. *IEEE Computer*, 34:67–75, 2001.

[6] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 260–269, New York, NY, USA, 2010. ACM.

[7] J. Clulow. On the security of PKCS #11. In *Cryptographic Hardware and Embedded Systems - CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 411–425. Springer Berlin / Heidelberg, 2003.

[8] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008.

[9] Cryptodev-linux. Performance comparison of Linux kernel crypto APIs, 2011.

[10] S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS #11. In *Computer Security Foundations Symposium, 2008. CSF '08. IEEE 21st*, pages 331 –344, June 2008.

[11] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008.

[12] P. Eronen and H. Tschofenig. Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). RFC 4279 (Proposed Standard), Dec. 2005.

[13] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17:281–308, April 1988.

[14] Z. Gutterman, B. Pinkas, and T. Reinman. Analysis of the linux random number generator, 2006.

[15] R. Housley and M. Dworkin. Advanced Encryption Standard (AES) Key Wrap with Padding Algorithm. RFC 5649 (Informational), Sept. 2009.

[16] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel, 2010.

[17] ISO. Common criteria for information technology security evaluation, July 2009.

[18] T. Jaeger. *Operating system security*. Morgan and Claypool Publishers, 2008.

[19] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), Feb. 2003.

[20] C. Kaufman, P. Hoffman, Y. Nir, and P. Eronen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 5996 (Standards Track), Sept. 2010.

[21] A. Keromytis, J. Wright, and T. de Raadt. The design of the OpenBSD cryptographic framework, 2003.

[22] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. pages 104–113. Springer-Verlag, 1996.

[23] C. Merli. LSM-PKCS #11: Lite security module.

[24] Microsoft. Cryptography API: Next generation, 2010.

[25] National Institute of Standards and Technology (USA). Aes key wrap specification, November 2001.

[26] National Institute of Standards and Technology (USA). Security requirements for cryptographic modules, November 2009.

[27] N. Provos. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, pages 231–242, 2003.

[28] RSA Laboratories. PKCS #8: Private-key information syntax standard, 1993.

[29] RSA Laboratories. PKCS #11: Cryptographic token interface standard v2.30, 2009.

[30] SANS. The top cyber security risks. Technical report, SANS, September 2009.

[31] J. Schaad and R. Housley. Advanced Encryption Standard (AES) Key Wrap Algorithm. RFC 3394 (Informational), Sept. 2002.

[32] M. D. Schroeder and J. Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15:157–170, 1972.

A Application to TLS key protection

This section describes in detail how the proposed framework could be used to protect the private keys used to identify a TLS [11] server. The approach is inspired by the security target format defined by the common criteria standard [17].

A.1 Problem definition

The TLS 1.2 protocol uses a public-private key pair to identify the server and avoid man-in-the-middle (MITM) attacks: the public key is included in a certificate signed by a trusted third party and provided to the client as part of the TLS connection establishment process. The protocol can use either a RSA key pair, or a Diffie-Hellman key pair. Although the algorithm and key exchange method choice affects internal operation of the TLS handshake, for our purposes we can treat both kinds of keys in the same way: the private key must be available to the TLS server, and disclosure of the private key to an adversary allows him to successfully mount a MITM attack impersonating the legitimate server.

As Section 4.1 describes, protection of session key material does not provide a meaningful improvement in security, and an adversary is necessarily able to mount a MITM attack, or an equivalent information disclosure attack, during the time the TLS server is under the adversary's active control.

We therefore consider the following components of the problem of protecting a TLS server.

T1 Threat to be mitigated: An adversary could impersonate the TLS server for an unbounded time. This is based on the capabilities of the first adversary described in Section 3.1.

T2 Necessary functionality: The TLS server must have enough access to the private key identifying the server (referred to simply as "the key" from now on) to be able to serve legitimate requests.

We also assume that the TLS private key is managed by the administrator.

A.1.1 Security objectives

To handle the above-described problem, we have chosen a set of specific *objectives* that we want the TLS server deployment to meet fully.

O1 The TLS server process can remain compromised only for limited time.

O2 Only the TLS server process and processes started by the administrator can perform operations using the key.

O3 An adversary who controls the TLS server can not export the key to a different computer.

O4 Processes of the administrator are trusted not to reveal the key.

O5 The TLS server has enough access to the key to be able to perform RSA decryption, RSA or DSA signing and Diffie-Hellman shared secret establishment, depending on TLS key exchange method.

These objectives constitute one possible approach to handling the problem set out in Section A.1: O1–O4 are together designed to mitigate threat T1, and O5 to provide functionality required by T2.

Whether, and how effectively, the chosen objectives handle the problem set, is a question that must be assessed by human judgement.[15]

A.2 Requirements and policies

Finally, we design a set of *requirements* on the cryptographic module described in this article and *policies* that must be enforced by the server's administrators by system configuration or setting up human processes. The requirements and policies were chosen in a way that allows us to demonstrate that all objectives were met in a semi-formal way.

We use the following requirements which are met by the module and were verified by manual inspection of the source code:

R1 The only ways to make a pre-existing key available as a `/dev/ncr` key object is to load it using the key storage component, unwrap it, and to import it from a plaintext representation.

R2 The wrapping and exportable flag restrictions are correctly enforced.

R3 No other user than administrator is able to set the wrapping or exportable flag of an existing key.

R4 No user other than administrator is able to make a copy of an existing non-exportable key with one or both of the flags set if they were not set on the original without having access to a key with the wrapping flag set.

R5 A key wrapped for storage identifies users allowed to unwrap or to perform operations using the key, and this restriction is correctly enforced.

[15] For example, the above set of objectives ignores ways to mount threat T1 without attacking the computer on which the TLS server runs, such as discovering the private key from the public key by brute force attacks, or obtaining a copy of the key from a key escrow system.

The following policies are the responsibility of system administrators:

P1 Any TLS server compromise is discovered and fixed in a bounded amount of time.

P2 The TLS server runs under a dedicated user account.

P3 The key, wrapped for storage, is stored in a file that is accessible to the user account of the TLS server.

P4 The key is not wrapped, using the storage master key or any other key, in files that are accessible to any other user account, except perhaps for the administrator.

P5 The key wrapped for storage in P3 above identifies the user account of the TLS server, and no other account, as allowed to unwrap the key.

P6 The storage wrapping master key is available at most to the kernel and processes running as administrator on the TLS server computer.

P7 Processes of the administrator are trusted not to reveal the key.

P8 The key wrapped for storage in P3 above does not have the exportable flag set.

P9 The TLS server does not have access to any key with the wrapping flag set.

P10 The key is not stored in any file unencrypted, nor encrypted with a key available to processes not running as administrator.

A.3 Meeting security objectives

We can now demonstrate that the requirements and policies from Section A.2 are sufficient to implement the objectives we have chosen in Section A.1.1:

We start with two auxiliary claims:

L1 If a key does not have the wrapping or exportable flag set, then the TLS server can not wrap using the key, or export the key, respectively: ensured by combination of R2, R3, R4, P9.

L2 An adversary who controls the TLS server can not extract the key material to user space:

- The key is not available exported in plaintext: P8, L1, P10.
- The key can not be extracted through the key wrapping mechanism: P9, L1.
- The key can not be extracted using the storage wrapping mechanism: P6.

Moving on to the objectives:

- O1 is implemented by policy P1.
- To demonstrate O2 is implemented, consider:

  - Only the TLS server process and processes started by administrator can load the key into a `/dev/ncr` key object: R1, P4, P2, and P10.
  - An adversary who controls the TLS server can not make the key object available to other user accounts on the same computer: R5, P5, P9, L1.
  - An adversary who controls the TLS server can not extract the key material to user space: L2.
- To demonstrate O3 is implemented, consider:
  - The adversary can not extract the key material to user space: L2.
  - The adversary can not use the key wrapping mechanism to transfer the key material: P9, L1.
  - The adversary can not use the storage wrapping mechanism to transfer the key material: P6.
- O4 simply becomes policy P7.
- O5: The key is made available to the server by P3.