Digital Identity Security Architecture in Ethos

W. Michael Petullo
mike@flyn.org

Jon A. Solworth
solworth@rites.uic.edu

Department of Computer Science
University of Illinois at Chicago
Chicago, IL 60607

ABSTRACT

Certificate systems often provide a foundation for distributed system security. A certificate is a signed statement; the user's private key must have been used to create the certificate's signature and the resulting certificate is tamper evident. Despite being based on sound theory, certificate system implementations are often exploited. Furthermore, certificate systems are often complex, to the extent that user-space programmers avoid certificates in favor of less secure, but easier to program, mechanisms.

We describe the certificate system for Ethos, an experimental Operating System (OS) that has been designed for security from the ground up. We reexamine and redesign the layering of certificate creation across kernel and user space, and discuss the beneficial security properties that result. The design enables certificates to be a pervasive authentication mechanism, private keys to be protected, and policy-based restrictions on the statements that a given application may sign. These protections are essential to protect digital identity systems from attack.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*authentication*

General Terms

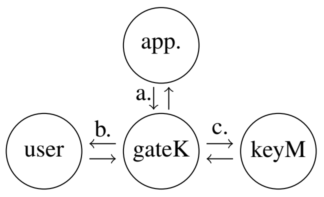
Design, Security

Keywords

Operating systems, certificates, digital identity, PKI, authentication, authorization

1. INTRODUCTION

The properties of certificates make them very useful for solving identity problems in distributed systems. Certificates are digitally signed statements [18], enabling very precise statements to be made by the signatory. It is reasonable to rely on such a signed statement only if it is highly probable that the apparent signer really signed it. And this, it turns out, is a subtle problem.



$$\begin{tikzcd} & \text{app.} \arrow[d, shift right, "a."'] & \\ \text{user} \arrow[r, shift right] & \text{gateK} \arrow[u, shift right] \arrow[l, shift right, "b."'] \arrow[r, shift left, "c."] & \text{keyM} \arrow[l, shift left] \end{tikzcd}$$

Figure 1: User-application-gatekeeper-keymaster model

Certificates are based on sound theory [27], but attackers often exploit flaws in their implementations. At issue are the many different software and hardware components involved in creating signatures. Each of these components introduces vulnerabilities. A security architecture should be structured to minimize the number of components which could be subverted and to reduce an attacker's ability to subvert each component. We shall describe a security architecture which does this, but before doing so we must introduce some terminology.

Figure 1 illustrates the *actors* involved in creating a certificate. A **user** is a human who explicitly or implicitly directs a system to use a private key to sign a certificate. In our model, there are three software components: an **application** is a program that prepares a certificate for signature; a **gatekeeper** asks for user approval and then requests a digital signature for a prepared certificate; and a **keymaster** has access to a private key and can use it to sign certificates. Unlike hand-written signatures, a digital signature cannot be realistically produced by a human; therefore, a keymaster must act on his behalf. To sign a certificate (a) an application generates an unsigned certificate and provides it to its gatekeeper (b) the gatekeeper requests approval from the user and (c) the gatekeeper requests that the keymaster sign the unsigned certificate and provides the result back to the application. The actors—users, applications, gatekeepers, and keymasters—all interact within the confines of a system security policy.

At first glance, it may seem that there are too many actors. Although some architectures may map multiple actors to a single software component, we chose to separate into three software actors because (1) a gatekeeper's size and privileges should be reduced to minimize its attack surface, and thus to prevent the keymaster from signing certificates against the security policy; (2) the keymaster needs to be a carefully crafted component, it is of paramount importance to prevent private key leakage; and (3) the application can then be unprivileged. We believe that this model best maps to the natural isolation and authorization layers within an operating system.

A primary goal of an identity system is that the signature scheme is **unforgeable**, meaning only user U can create U's signature on message m, and **universally verifiable**, meaning any user can verify that the signature on m is valid [2]. Together, these provide the property of **non-repudiation of origin** [40] that is the cornerstone of many distributed authentication systems[1]. If a keymaster properly isolates a sufficiently long cryptographic key and uses it with an appropriate cryptographic algorithm, then it is known that the keymaster signed a given message. In our model, a system must extend this property of non-repudiation back through a gatekeeper to the user that authorized the message.

Collectively, a keymaster, gatekeeper, and user that produces a certificate is the **signer**. The **relying party** verifies and possibly acts on a certificate.

We believe that many of the flaws in existing certificate systems are due to insufficient security analysis and an improper layering of the systems' application, gatekeeper, and keymaster components. It is difficult to design a certificate system and previous attempts have failed because they are unable to:

1. ensure that keymasters adequately protect private cryptographic keys (§2.1)
2. guarantee that the user is signing what he thinks he is signing (§2.2)
3. guarantee that the signer and relying party share a single meaning for a given certificate (§2.3)
4. restrict the certificates for which a given gatekeeper may request signatures (§2.4)

External smart cards are often used to address (1). Sometimes, smart cards provide a trusted input and output channel; this can help address (2), although this may significantly increase production cost.

This paper describes the design and implementation of certificates in Ethos, an experimental OS that has been designed for security from the ground up. Ethos is a clean-slate design, enabling its software layering to be engineered to meet security requirements at the most appropriate system level. We have leveraged this advantage to address all of the above listed shortfalls.

Ethos provides the following contributions:

- Stronger isolation properties, made possible by a sign system call that allows the keymaster component to be fully implemented in the OS kernel
- A type system that allows the system to bind certificates to a fixed semantic meaning
- An authorization framework that can restrict what a given gatekeeper may sign; we call this **certificate set authorization**

In the remainder of this paper, we discuss related work (§2), security requirements (§3), the design of certificates in Ethos (§4), and an evaluation of Ethos' implementation (§5).

[1] We use non-repudiation as a technical, rather than a legal, term.

2. RELATED WORK

2.1 Isolation

Insufficient isolation has enabled many attacks on certificate systems including:

Keylogging/PIN collection An attacker collects user input, possibly including a smart card PIN, after installing malicious hardware or software. This can result in **false authentication**, whereby a malicious program authenticates to a computing resource using a user's credentials; **fraudulent signatures**, whereby a program performs a digital signature without a user's approval; and **remote smart card control**, whereby a program makes the smart card's services available to third parties over a network [8].

Keyjacking An attacker undermines the notification mechanism of a certificate system, allowing certificate operations to occur without notifying the legitimate user [20]. This violates the requirement that a user's private key is only used with his approval.

Many systems have tried to isolate private keys in user-space. For example, Plan 9's factotum addresses the leaking of keys through debugging interfaces, swap space or willing disclosure [7]. The designers of Plan 9 took care to ensure that the `proc` filesystem restricted access to factotum and that the system would avoid swapping factotum to disk. Another example is SSH [39], which provides an authentication system that is based on public key cryptography. Like Plan 9, SSH attempts to isolate private keys, it protects them with an optional password and requires restrictive file permission settings. But such user-space isolation is often compromised by malicious software [16] because (1) malicious software runs with the user's privileges and (2) passwords are optional or the password choice is at the user's discretion. It is especially hard to close these potential security holes when individual applications implement disparate protection systems.

Side-channel attacks further threaten proper key isolation [17, 4]. An attacker may analyze the timings or power usage of various cryptographic operations to learn a private key. Careful implementation—including introducing random, input-independent delays; ensuring operations' inputs do not affect their timing; or using data masking [21]—can mitigate these attacks. Some algorithms are resistant to software-based countermeasures without a severe performance loss.

Cold boot attacks provide another threat to key isolation [11]. These attacks exploit the data retention properties of DRAM to extract data from a memory chip after rebooting a machine. Zeroing memory is the most obvious countermeasure for cold boot attacks ([22] mentions limiting the existence of plain-text passwords), but some keys are necessarily persistent for a system to operate and often no single program is aware of where in memory all keys reside.

2.2 Seeing what you sign

Since it is not feasible for a person to create a digital signature by hand, some agent must act on a user's behalf. Due to this requisite indirection, it is difficult to guarantee that a user is signing what he intends [14]. This presents the possibility of a class of attacks called What You See Is Not What You Sign (WYSINWYS).

The most disadvantaged position for a user to be in is when an external party both generates and displays the data to be signed. A point-of-sale system that requests approval from a customer's smart card is an example of one such situation. The desirable properties of a smart card are sometimes at odds with this scenario. In order to increase their trustworthiness, smart cards are designed to have very limited functionality, focusing on providing cryptographic operations. A widely deployed smart card must be portable and inexpensive, so trusted input and output may be prohibitive. A vendor could defraud a customer by displaying one value on a point-of-sale system, but submitting another value to a customer's smart card for approval by digital signature. Only one of secure input or output is necessary for security in such a point-of-sale system [10]. But such cost-saving single-channel solutions are feasible only in situations where the statement to be signed is very simple.

Guaranteeing that users sign only what they intended to requires a trusted path between the display of a certificate and the input with which a user authorizes a signature. Of course, adding complexity to this secure path increases the difficulty of its assurance. This means that it might not be feasible to include the application that generates the input for signature in the secure path.

An external signing device that photographs a document, performs optical character recognition processing on the image, and signs the resulting text has been proposed [13]. But a user who just finished generating a long document may not be willing to carefully review it using a secure component before signing it.

2.3 The meaning of what you sign

Even with a secure means of reviewing material before signing it, it may be difficult to know whether to sign it. The **semantic-level difference** is the difference in cognitive understanding of the meaning of a digital document between the signer and the relying party. The **syntax-level difference** refers to the representation of data at the syntax level, between two points within a system [1].

Many attacks may exploit a semantic difference. For example, a **replay attack** exploits a certificate whose meaning is insufficiently clear; a signed copy of a vague statement is dangerous because it lacks context [35]. A **Dalí attack** exploits a polymorphic file; a single file displays different contents depending on what application is used to view it [6]. For example, a single file may contain data which is valid both as PDF and TIFF. Using a program capable of displaying either format will choose which of the possibly unrelated representations to display. Furthermore, it is possible to digitally sign a document that references external material. If this is done without care, one can alter the external material without invalidating the digital signature [15].

2.4 The authorization of signatures

PorKI [31] is a portable device that attempts to allow users to operate various workstations. The system stores keys on an external device that restricts access to the keys based on a security policy and the trustworthiness of a particular workstation. PorKI is somewhat coarse-grained in that it restricts keys based on hosts, but enforces no similar restriction based on application or certificate type; this type of authorization is left to the workstation. We consider Ethos to be complementary to PorKI because Ethos could serve as the basis of a more robust workstation. Furthermore, the PorKI PDA itself could be built upon Ethos (§4.4).

Many existing systems attempt to require that users explicitly authorize signature operations. But present software layering is very complex and makes this difficult. It is possible to fool common OSs and web browsers into either revealing a secret key or authenticating using client-side SSL without notifying the user [20]. Even if the user is warned that a signature is about to take place, users will often ignore security warnings [12].

2.5 Existing systems

The state-of-the-art certificate systems are susceptible to attack. The Department of Defense (DoD) Public Key Infrastructure (PKI) uses a smart card—the Common Access Card (CAC)—to perform cryptographic operations and is one of the largest PKI installations in the world [24]. The CAC isolates a user's private key from the rest of the system. However, the system also uses commodity software on a legacy OS. Although the CAC is a trusted component, it is used with an untrusted OS and applications. Furthermore it typically does not have trusted input or output. As a result, CAC-enabled systems are susceptible to PIN collection or WYSINWYS attacks, resulting in the possibility of false authentication, fraudulent signature and remote control [8]. As DoD networks have been successfully attacked, such malicious software could be introduced to them [19].

3. SECURITY REQUIREMENTS

For a system to be robust against attack, it must have a well thought out set of security requirements, a design which meets those requirements, and a careful implementation. Herein, we offer a brief description and explanation of certificate-related security requirements.

Trust relations specify the entities that are relied upon in order to maintain the security of a system. For the user-application-gatekeeper-keymaster model, this includes:

1. The authors of gatekeepers.
2. The authors of the OS, including the keymaster.
3. The manufacturers of hardware included in the system's Trusted Computing Base (TCB), including the gatekeeper's input device and display.
4. Administrators who must be able to reason about and configure the system.
5. Users who review certificates and approve signatures on them.
6. Individuals with physical access to the trusted system components.

The model requires almost no trust in applications themselves. However, applications can undermine security by overwhelming a user with security-related decisions (e.g., a user that is prompted to review many certificates is likely to become complacent). Thus, applications should not unnecessarily burden their user.

Given the trust relations and requirements above, we now consider preventing attacks by untrusted entities. Users review and approve signatures; therefore, the security policy must assist users in this task. After all, a smart card may help isolate a private key but cannot help a user decide *what* to sign. Administrators must be able to reason about a system, to restrict misuse, and enable proper use. This requires that software is layered in a way to maximize simplicity through the use of natural, pervasive, and compulsory isolation mechanisms. The following section describes our design, which was developed to enforce these security requirements.

4. DESIGN

The central goal of Ethos' design is to provide the most secure mechanisms possible. To achieve this goal, Ethos forgoes compatibility with existing systems. So that this incompatibility does not hinder Ethos adoption [25], legacy OSs are relied upon to provide legacy compatibility. Ethos is simpler than its predecessors;

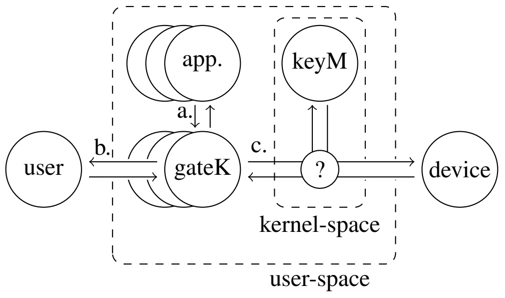


Figure 2: Ethos' user-application-gatekeeper-keymaster model

it allows programmers to have a better understanding of what their programs do and increases the ability of system administrators to reason about their security policies. The advantage is a reduced number of bugs in an installation and therefore a reduction in the number of security holes.

4.1 Architecture

Herein, we describe a high-level view of the Ethos certificate architecture, forgoing details until succeeding sections. An OS-wide, compulsory authorization affects how applications, gatekeepers, and keymasters interact within Ethos. This mechanism reduces the burden on administrators by enabling them to ensure certain policy enforcement without auditing application code; it also allows them to focus on a single set of mechanisms without the need to understand idiosyncratic application settings.

Within Ethos, the keymaster is able to provide strong isolation of the private key because it is implemented within the OS kernel.

Gatekeeper implementations are designed using the principle of least privilege [29]. To meet this goal, gatekeepers are separated from the applications they serve. The Ethos authorization system allows an administrator to limit the types of certificates that a gatekeeper may sign; restrict an application to communicating with a single, properly controlled gatekeeper; and disallow non-gatekeeper programs from requesting signatures. Gatekeepers must provide trusted input and output so that a user may review and approve the signature of certificates.

Thus, the trusted software components of the Ethos certificate system are the bootloader; the Ethos kernel, which enforces an authorization policy and isolates private keys; gatekeepers, which provide a trusted path to a user for certificate approval and request signatures; and various administrative tools.

Ethos provides several methods for approving and producing signatures:

Method 1 a native, kernel-based method that is explicitly authorized by a user

Method 2 a native, kernel-based method with implicit authority

Method 3 a smart card-based method

Method 4 a method using an external device that integrates trusted input and output.

For each pairing of method m and application a, $\mathsf{gk}_{m,a}$ serves as its gatekeeper[2]. Ethos' authorization and type system ensure that each $\mathsf{gk}_{m,a}$ is restricted to (1) be invoked only by application a, (2) sign certificates only of the appropriate types, and (3) act only on behalf of allowed users.

[2]In fact, a single program, g_m, is used as the gatekeeper for method m; an administrator replicates g_m within the filesystem to allow different security labels with different authorization restrictions, i.e., $\mathsf{gk}_{m,a}$.

Native Ethos signatures.

Consider method one and two in Figure 2. Method one is the most straight forward and is sufficient for the most common requirements. A summary of this description appears in Table 1.

For method one, application a may execute its $\mathsf{gk}_{1,a}$ and provide it with the data to sign (labeled a). $\mathsf{gk}_{1,a}$ displays the data for the user and prompts the user to authorize the signature (labeled b). Upon approval, $\mathsf{gk}_{1,a}$ requests a signature from the keymaster by issuing a sign system call (labeled c); the keymaster grants the request if the security policy permits it. $\mathsf{gk}_{1,a}$ now has a signed certificate and can provide it back to a.

In some cases, explicit user approval is not needed for a program to generate a certificate. For example, a service might sign on behalf of the organization. To support implicit user approval, an administrator can authorize application a to use $\mathsf{gk}_{2,a}$ which implements method two by skipping the user authorization step in Figure 2 (labeled b). Doing so implies that the user upon whose behalf a operates pre-approves a's requested signatures.

External signing devices.

Ethos is engineered to provide more assurance than existing systems, so the need for a separate smart card is less compelling. However, there may be cases where an external signing device is beneficial. For example, a user may have to operate a workstation that is not trusted, an organization may want strong guarantees that the system does not leak a user's private keys, or it may be easier for users to maintain the physical security of a small device. We again reference Figure 2, this time to describe how Ethos supports external signing devices.

An external signer may or may not have trusted input and output. When it does not, Ethos performs the review and approval process labeled b in Figure 2 (in this case, the approval process might require that the user provide a PIN to authenticate to the device). When the external device provides trusted input and output, Ethos skips step b because it is known that the device will perform the review and approval process on its own. In either case, Ethos treats the device as the keymaster and the sign system call requests a signature from the device instead of calculating the signature within the kernel.

4.2 Isolation

Ethos is able to better isolate private cryptographic keys because they are kept in kernel space—they are never provided to user space. Ethos implements signing as a system call (application request to the kernel). This allows user space applications to request a signature of a certificate without having access to the private cryptographic key and allows the OS to restrict what may be signed.

Ethos' certificate system carefully implements cryptographic and other sensitive operations in the Ethos kernel to resist side-channel attack. Likewise, Ethos can zero private keys after use in order to reduce the effectiveness of a cold boot attack; unlike many certificate systems, Ethos' restriction of private keys to kernel space means that sensitive memory locations are known by the kernel.

The design of Ethos allows further isolation using vTPM [3]. The presence of a Trusted Platform Module (TPM) facility would allow the OS to encrypt users' private keys using a host's TPM private key in addition to a user's password for persistent storage. Decrypting the keys prior to using them to execute a signature would thus require two-factor authentication.

	Method 1	Method 2	Method 3	Method 4
Gatekeeper	**Ethos user space** asks for approval and then requests signature	**Ethos user space** immediately requests signature	**Ethos user space** asks for approval and then requests signature	**External device** asks for approval and then requests signature[a]
Keymaster	**Ethos kernel space** performs signature if authorized by system policy		**External device** performs signature if authorized by device	

Table 1: Signature methods

[a]In this case, the Ethos user-space gatekeeper does not need to prompt the user for approval; instead, $\mathsf{gk}_{4,a}$ immediately requests a signature from the keymaster which, in turn, forwards the request to the external device.

The relative simplicity of Ethos' kernel-based isolation is important even in the presence of an external cryptographic device such as a smart card, as described in §2.1. By restricting what a process may sign, the Ethos kernel can protect against many attacks that result in an unintended signature. For example, only a banking application may request signatures for withdrawal requests. Other applications are not authorized to sign such requests, so they may not be exploited to do so. An organization could audit the banking application to make it more robust against attack. Ethos is able to enforce these restrictions because of the presence of type checking and a carefully designed authorization policy.

4.3 Type checking

In Ethos, system objects—including files and directories—have a type associated with them. More specifically, an administrator provides a type's **layout**—that defines the type's variable fields—and **semantic description**—that describes the meaning of these fields—as the input to a hash function. The output of this function is a **type ID** and uniquely identifies the type. The kernel maintains a list of type definitions and corresponding IDs. The kernel data structure describing a directory contains a type ID field from this list and Ethos' type checking guarantees that all files placed in the directory are well-formed with respect to the type. As a result, all files in a given directory are of the same type. The combination of type checking, the sign system call, and a system authorization policy allows Ethos to restrict what certificates may be signed.

The header used by Ethos certificates is shown in Figure 3. If a type definition begins with fields matching the certificate header, then that type defines a certificate sub-type. The fields highlighted in gray are filled in by the OS and are therefore subject to the administrator's security policy. Each certificate contains the following fields:

Size The size of the certificate header and its payload

Version A version field that facilitates future modifications of the certificate format

Type ID The type ID

Public Key The public key that will validate the certificate's signature

Revocation Server The server that maintains a certificate revocation scheme

Revocation Server Public Key A key that may be used to authenticate the revocation server

Valid From, Valid To Fields that identify the time period during which the certificate is valid

Certificate Signature The certificate's digital signature

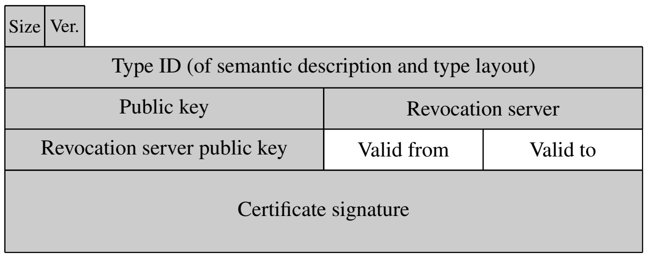

Size
Ver.
Type ID (of semantic description and type layout)
Public key
Revocation server
Revocation server public key
Valid from
Valid to
Certificate signature

Figure 3: Certificate header format

Following the certificate header is the *certificate body* which may contain arbitrarily defined fields. While certificate headers have the same structure across all certificates, certificate bodies are specific to the type of certificate. Certificate bodies contain typed fields, and can be viewed in a manner analogous to a paper form's fields. Non-variable data is never stored in a type field; instead it is contained in the semantic description that contributes to the corresponding type ID. The semantic description describes the certificate and its fields; it is bound to a certificate sub-type as described in the next paragraph.

In order to disambiguate its semantic meaning, the kernel stamps each certificate with a type ID as part of the sign system call. Thus, even with the existence of two types with the same layout the (i.e., either would pass the other's well-formedness check) the semantic meaning of a given certificate is unambiguous; a user cannot substitute a signed certificate of one type for another that has the same layout but which he is not allowed to sign.

Certificate types—and the semantic description bound to them—serve to establish a pre-defined logic between the signer and the relying party, decreasing the semantic-level difference between the two parties. Properly designed types also reduce syntax-level differences by enforcing a canonical representation. The authorization system regulates which certificates can be signed; unauthorized certificate types—such as those with machine-interpreted external references or re-playable meaning—can never be signed. System administrators restrict the certificates that may be signed to those with vetted semantics and they do so in a centralized place, the OS authorization policy.

4.4 Authentication and authorization

System policy.
Ethos provides for strong authentication of local and network users. The authentication of remote users is built into the system-level networking protocol implemented by Ethos. In addition, Ethos also authenticates at the host level in a manner transparent to user-

space programs. This allows for the confident enforcement of authorization policies based on both remote user and host.

Ethos has mandatory access controls [26]. A full discussion of Ethos' authorization system is beyond the scope of this paper. In summary, Ethos authorization is composable, analyzable, and provides a broad set of authorization properties. Herein we discuss only a host authorization policy; it is also possible to build authorization in a distributed manner, for example, see [23]. Ethos' authorization properties support the production of trusted paths and can restrict an application so that it may only communicate with its own gatekeeper, ensuring an application only uses signature methods approved for the application. Ethos provides further control through a certificate-specific permission that regulates the signing of data based on the target file descriptor's type—the maysign permission:

```
( user, program ) maysign ( label )
```

This can be read as the given program (most likely a $\text{gk}_{m,a}$ as discussed in 4.1), running on behalf of the given user, may request signatures over certificates with the given label. In Ethos, directories bear a label (used for authorization) and a type (defining the type of files the directory contains). As a result, the maysign permission restricts the generation and signing of certificates to authorized types. This control serves to guarantee that certificates signed by a given program are restricted to a set of authorized semantic meanings and disallow signatures on the classes of statements that are susceptible to the semantic attacks discussed in §2.3.

In addition, Ethos provides a verify program that receives a certificate from the network and performs the verification process. Other programs may be restricted to receive data only from verify. In this manner, an administrator can ensure that a given server program acts only on requests with valid signatures.

Ethos is particularly well suited to implement the external signing device discussed in §4.1. This is because Ethos permissions are dynamic; this means that Ethos may restrict a gatekeeper's ability to request signatures based on the host from which the gatekeeper accepts a network connection. A user may have several accounts, where each account has an associated signature keypair of a different assurance rating. In this manner, an external Ethos signer could regulate the keys available when interacting with a given untrusted system, much like PorKI. Ethos has an advantage in that these authorization decisions are regulated by the system's unified mandatory access controls. Furthermore, an Ethos-based signer supports type-enforced network connections, allowing the benefits previously discussed to extend to the device.

User authorization.

In addition to being granted permission by a system authorization policy, all signature operations should be approved by a user[3]. After all, certificates reflect the will of humans. Ethos' mandatory access controls allow an administrator to create a trusted path between a user and his gatekeeper so that the user may approve signatures. Using such a channel, the gatekeeper displays the certificate fields and the fields' semantic descriptions to the user. To guard against homograph attacks [9], Ethos gatekeepers maintain a bias towards a single language's character set when displaying certificates. The gatekeeper highlights non-native characters in its output. After displaying the certificate, the gatekeeper prompts the user to approve or cancel the operation. This compulsory user authorization serves to prohibit the attacks described in §2.2, primarily WYSINWYS attacks.

[3]As described in §4.1, signature method two allows for implicit approval that is restricted to administrator-selected applications.

System	Signatures per second
Ethos (Method 2)	546
OpenSSL (2048 bit key)	26
OpenSSL (4096 bit key)	4
System	**Verifications per second**
Ethos	243
OpenSSL (2048 bit key)	945
OpenSSL (4096 bit key)	262

Table 2: Signature measurements

4.5 Interface

The sign system call.

The sign system call takes as parameters a directory file descriptor and file name. After checking that the operation is authorized and that the file has not been previously signed, Ethos adds the directory's type to the certificate, populates the other fields in the header, signs the file, and writes it back to the filesystem. The reason Ethos writes the file back to the filesystem instead of providing the signed data in memory is to provide an audit trail on signed certificates.

The verification process.

The verification process consists of a library call that makes various system calls. Like the sign system call, the verify library call takes as parameters a directory file descriptor and file name. Invoking the verification process implies that the calling program is acting on the certificate (unless it is found invalid).

The library call verifies that the certificate has not expired, the signer's credentials have not been revoked, the certificate's syntax hash matches the expected type, and the digital signature on the certificate is valid. The library call also logs that the system received the certificate; at a minimum, Ethos maintains this log for the lifetime of the certificate.

5. EVALUATION

5.1 Experiments

In this section, we compare our implementation of certificates in Ethos to OpenSSL on Linux. OpenSSL is commonly used to provide a subset of the capabilities provided by Ethos, implements digital signatures, and has been optimized for production use. Furthermore, OpenSSL provides benchmarking tools.

Experimental setup.

Ethos is a paravirtualized OS running on top of Xen 4.1. We created two unprivileged, 32-bit guest domain images containing Ethos and Linux 2.6.32.27 with OpenSSL 0.9.8p. Our test machines use AMD Athlon 64 X2 3800+ processors and 2GB of memory.

OpenSSL's speed utility provided the basis of our experiments. We used speed to measure OpenSSL's RSA implementation. Note that 1,024-bit RSA is considered unsafe today [38], and that 3,072-bit RSA (128-bit security) is equivalent to current Ethos security.

Implementation	Lines of code
OpenSSL (crypto only)	177,673
Ethos sign system call	138
Ethos verify library call	133
Reference application/gatekeeper	69
NaCl[a]	51,709

Table 3: Lines of code measurements

[a]NaCl includes the implementation of ciphers not used within Ethos. It also provides separate, optimized assembly source code for IA32 and x86_64 and several sub-architectures. Much of this is machine generated from qhasm code that is not distributed with NaCl. We have included all of this code in our count.

Performance.
Table 2 describes our results from comparing the signature rate of Ethos to OpenSSL. In all measurements, Ethos outperforms OpenSSL when performing signatures. We attribute this primarily to the speed of the NaCl library. Our results demonstrate that Ethos maintains competitive performance despite moving the sign operation into a system call.

Our implementation of certificates was only recently completed, and we have just begun the performance tuning of Ethos. We believe that our numbers have substantial room for improvement. Nevertheless, they are currently competitive with OpenSSL.

5.2 Code metrics

We used the tool cloc[4] to count the lines of code in Ethos and NaCl and to compare our results to OpenSSL. Table 3 displays our results. OpenSSL is a much larger code-base than Ethos because it provides many cryptographic options and runs on many operating systems. The design decisions behind Ethos have worked to keep its code simple and to shield application developers from making technical cryptographic decisions. As a result, Ethos' full implementation of certificates is small enough for a careful assurance review.

6. CONCLUSION AND FUTURE WORK

Ethos simplifies the generation and verification of certificates while providing certificate set authorization and isolating private keys. These protections are provided in the OS, so that they cannot be bypassed and attacks against applications do not subvert OS protections.

The design of the protections arises from a careful analysis of the potential attackers, the resources they might employ, and the classes of vulnerabilities which have been exploited in the past. Most of all, it considers *trust relations* and the entities that they rely upon for secure operation; trust relations are always issues of policy and thus must be configurable.

Ethos is able to reduce application complexity while providing strong protections which may not be bypassed. This was achieved through:

- co-design: has enabled the interaction of different parts of the certificate system to be simplified and analyzed.

- types and a sign system call: combined, these allow for certificate set authorization.

- isolation: Ethos ensures that private keys are never available in user-space.

- transparent mechanisms: specific details about signatures, cryptographic integrity, and authentication have been added in a way which is invisible to the application.

- higher abstraction level: allows security properties to be provided uniformly.

- configuration simplification: Ethos uses strong techniques everywhere. (Traditionally, implementations vary cryptographic strength to save computation, but this increases the difficulty of assurance).

- simplification of administration: system administrators provide only policy, determining who can sign what and how the resulting certificates may propagate through the system. This gives administrators adequate control without overwhelming them with options.

Work that remains to be done in Ethos includes vTPM integration, support for external signing devices, performance tuning, code auditing and user interface work. Moreover, this paper focuses on certificates per se; therefore, an encompassing PKI is beyond its scope. We make no claim that Ethos' digital identity system could effectively operate on a large scale without PKI; for example, an Ethos certificate contains a public key but it is PKI that would link this public key to a real world entity. Moreover, the negotiation of trust between two entities is a complex topic [30, 5, 36]. PKI is a difficult but important problem and we are investigating it in parallel [32, 33, 34]. Previous work in PKI [28, 37] influenced the design of our certificates.

Properly handling certificates within a system is a difficult task. It follows that an implementation should be performed after careful consideration and at the appropriate layers within a system. We believe that our design of Ethos makes more protections compulsory and frees both developers and administrators from being exposed to unnecessary complexity. We further believe that this approach will reduce the security bugs found in certificate systems.

[4]http://cloc.sourceforge.net/

7. REFERENCES

[1] A. Arnellos, D. Lekkas, T. Spyrou, and J. Darzentas. A framework for the analysis of the reliability of digital signatures for secure e-commerce. *The electronic Journal for e-commerce Tools & Applications (eJETA)*, 1(4), 2005.

[2] M. Ben-Or, O. Goldreich, S. Micali, and R. L. Rivest. A fair protocol for signing contracts (extended abstract). In *Proceedings of the 12th Colloquium on Automata, Languages and Programming*, pages 43–52, London, UK, 1985. Springer-Verlag.

[3] S. Berger, R. CÃąceres, K. A. Goldman, R. Perez, R. Sailer, and L. Doorn. vTPM: Virtualizing the trusted platform module. In *Proc. of the USENIX Security Symposium*, pages 305–320, 2006.

[4] D. J. Bernstein. Cache-timing attacks on AES. Technical report, The University of Illinois at Chicago, 2005.

[5] E. Bertino, E. Ferrari, and A. C. Squicciarini. Trust-$\mathcal{X}$: A peer-to-peer framework for trust establishment. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):827–842, July 2004.

[6] F. Buccafurri, G. Caminiti, and G. Lax. Fortifying the dalí; attack on digital signature. In *Proceedings of the 2nd International Conference on Security of Information and Networks*, SIN '09, pages 278–287, New York, NY, USA, 2009. ACM.

[7] R. Cox, E. Grosse, R. Pike, D. Presotto, and S. Quinlan. Security in Plan 9. In *Proc. of the USENIX Security Symposium*, pages 3–16, 2002.

[8] P. Dasgupta, K. Chatha, and S. K. S. Gupta. Viral attacks on the DoD common access card (CAC). available at `http://cactus.eas.asu.edu/partha/Papers-PDF/2007/milcom.pdf`, 2009.

[9] E. Gabrilovich and A. Gontmakher. The homograph attack. *Commun. ACM*, 45:128–, February 2002.

[10] H. Gobioff, S. Smith, J. D. Tygar, and B. Yee. Smart cards in hostile environments. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce (EC-96)*, pages 23–28, Berkeley, Nov. 18–21 1996. USENIX Association.

[11] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Usenix Security*, 2008.

[12] C. Herley. So long, and no thanks for the externalities: the rational rejection of security advice by users. In *Proceedings of the 2009 Workshop on New Security Paradigms Workshop (NSPW'09)*, pages 133–144, New York, NY, USA, 2009. ACM.

[13] A. Jøsang and B. AlFayyadh. Robust wysiwys: a method for ensuring that what you see is what you sign. In *Proceedings of the Sixth Australasian Conference on Information Security—Volume 81*, AISC '08, pages 53–58, Darlinghurst, Australia, 2008. Australian Computer Society, Inc.

[14] A. Jøsang, D. Povey, and A. Ho. What you see is not always what you sign. In *in âĂŸThe proceedings of the Australian UNIX User Group*, AUUG'02, pages 4–6, 2002.

[15] K. Kain, S. W. Smith, and R. Asokan. Digital signatures and electronic documents: a cautionary tale. In *Proceedings of the IFIP TC6/TC11 Sixth Joint Working Conference on Communications and Multimedia Security*, pages 293–308, Deventer, The Netherlands, 2002. Kluwer, B.V.

[16] G. Kent and B. Shrestha. Unsecured SSH—the challenge of managing SSH keys and associations. White paper, SecureIT, 2010.

[17] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Crypto'99*, pages 388–397. Springer-Verlag, 1999.

[18] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computing Systems (TOCS)*, 10(4):265–310, 1992.

[19] W. J. Lynn III. Defending a new domain. *Foreign Affairs*, September 2010.

[20] J. Marchesini, S. W. Smith, and M. Zhao. Keyjacking: the surprising insecurity of client-side SSL. Technical report, Dartmouth College, 2004.

[21] T. S. Messerges, E. A. Dabbish, and R. H. Sloan. Examining smart-card security under the threat of power analysis attacks. *IEEE Trans. Comput.*, 51(5):541–552, 2002.

[22] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos authentication and authorization system. Technical report, MIT, 1987.

[23] B. Neuman. Proxy-based authorization and accounting for distributed systems. In *Distributed Computing Systems, 1993., Proceedings the 13th International Conference on*, pages 283 –291, may 1993.

[24] R. Nielsen. Observations from the deployment of a large scale PKI. In *Proceedings of the 4th Annual PKI R&D Workshop*, 2005.

[25] R. Pike. System software research is irrelevant, Aug. 2000.

[26] M. Radhakrishnan and J. A. Solworth. Quarantining untrusted entities: Dynamic sandboxing using LEAP. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, pages 211–220. ACSA, Dec. 2007.

[27] R. Rivest, A. Shamir, and L. Adleman. On digital signatures and public key cryptosystems. *Communications of the ACM (CACM)*, 21:120–126, 1978.

[28] R. L. Rivest and B. Lampson. SDSI — a simple distributed security infrastucture. Technical report, MIT, Apr. 1996.

[29] J. H. Saltzer and M. D. Schroeder. The protection of information in computer system. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[30] K. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobson, H. Mills, and L. Yu. Requirements for policy languages for trust negotiation. In *Third International Workshop on Policies for Distributed Systems and Networks (POLICY 2002)*, 2002.

[31] S. Sinclair and S. W. Smith. PorKI: Making user PKI safe on machines of heterogeneous trustworthiness. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, pages 419–430. IEEE Computer Society, 2005.

[32] J. A. Solworth. What can you say? and what does it mean? In *Workshop on Trusted Collaboration*. IEEE, 2006.

[33] J. A. Solworth. Beacon certificate push revocation. In *Computer Security Architecture Workshop (CSAW'08)*, pages 31–48, Oct. 2008. available at `http://www.rites.uic.edu/~solworth/solworth08bcpr.pdf`.

[34] J. A. Solworth. Instant revocation. In *EuroPKI'08*, pages 31–48, June 2008. available at `http://www.rites.uic.edu/~solworth/solworth08instantRevocation.pdf`.

[35] P. Syverson. A taxonomy of replay attacks [cryptographic protocols]. In *Computer Security Foundations Workshop VII, 1994. CSFW 7. Proceedings*, pages 187 –191, jun 1994.

[36] U.M.Mbanaso, G. Cooper, D. Chadwick, and A. Anderson. Obligations of trust for privacy and confidentiality in distributed transactions. *Internet Research*, 19(2):153–173, January 2009.

[37] H. Wang, S. Jha, T. W. Reps, S. Schwoon, and S. G. Stubblebine. Reducing the dependence of SPKI/SDSI on PKI. In D. Gollmann, J. Meier, and A. Sabelfeld, editors, *11th European Symposium on Research in Computer Security (ESORICS'06)*, volume 4189 of *Lecture Notes in Computer Science*, pages 156–173. Springer, 2006.

[38] Wikipedia. Key size. `http://en.wikipedia.org/wiki/Key_size`, 2011.

[39] T. Ylonen. SSH—secure login connections over the Internet. In *Proc. of the USENIX Security Symposium*, pages 37–42, San Jose, California, 1996.

[40] J. Zhou and D. Gollman. A fair non-repudiation protocol. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 55–61, may 1996.