A Capability-Based Transparent Cryptographic File System

Frank Graf
Institute for Graphic Interfaces
Seoul, Korea
frank.graf@igi.re.kr

Stephen D. Wolthusen
Department of Computer Science
Gjøvik University College
Gjøvik, Norway
swolthusen@ieee.org

Abstract

Data on the file system in mobile internetworked working environments are exposed data to a number of threats ranging from physical theft of storage devices to industrial espionage and intelligence activities. This paper describes a fully transparent, capability-based file system security mechanism for use in heterogeneous computing environments with emphasis on the implementation on the Microsoft Windows NT/XP family of operating systems. This mechanism can provide confidentiality and integrity protection for on- and off-line use through modular cryptographic means and is interoperable between several operating system platforms.

1. Introduction

Requirements for both industrial and governmental environments increasingly demand that individuals be mobile and collaborate both locally and across larger geographical regions. Together with the significant increases in productivity resulting from mobility and almost permanent internetworking, a number of threats also gain in significance, particularly in the area of storage and the exchange of data and information. While the protection of network environments through firewalls and cryptographic means such as virtual private networks have both become part of most current operating systems (OS). These can frequently be configured to permit a certain level of interoperability across operating systems and organizational boundaries. This is not necessarily the case for storage and file systems. While a number of cryptographic protection mechanisms for file systems exist (cf. section 6), these are typically limited in one or more aspects that were identified as crucial for both addressing the pertinent threats and providing a seamless and unencumbered operating environment for users. The following provides the requirements and criteria used in the design and development of a file system security mechanism; several threats identified as the basis of this are described later in this section.

Security Policy Support Most COTS operating systems are limited in their ability to support dynamic and flexible security policies beyond discretionary and (in some Unix derivatives) role-based access control mechanisms. Moreover, such controls provide enforcement at the level of OS data structures (and their respective external representations in storage systems) and cannot be dynamically configured to reflect changes in configurations such as collaborative relationships or the ingress and egress of parties from coalition environments. To support policies that cannot be implemented using intrinsic OS mechanisms and which also can provide identical semantics across multiple platforms (see below), a mechanism orthogonal to standard controls is therefore required. Such a mechanism can e.g. provide enforcement for mandatory security policy elements while leaving intact the expected semantics for any user-defined discretionary security controls.

Interoperability Any storage security control mechanism must provide interoperability at several layers both within an individual host system and in interactions with other hosts and storage subsystems. For a given system, controls must not affect APIs and behavior of any interface expected by both users and application programs operating within the confines of any currently active security policy. Moreover, controls must be capable of enforcing policies regardless of the file system type. This requirement has become increasingly relevant given removable media such as memory sticks; such devices frequently use file systems that cannot be replaced (e.g. because of hard-coded file system structures) or which must be retained to ensure interoperability between different OSes. Finally, most non-trivial environments require the seamless internetworking of multiple platforms, whether through exchange of removable storage media such as CD-R or through network file systems and storage area networks. Security controls must therefore be capable of transparently handling all types

of file systems that may be directly attached in the form of fixed or removable media as well as network attached storage and other storage attached through network file systems.

Usability Regardless of the security policies to be enforced and the potential for conflicts between policy and expected or desired user behavior, the actual implementation mechanism must conform closely to expected behavior for unprotected operation. Users should therefore not be required to take explicit steps to conform with security policy or to make use of the security mechanisms (e.g. confidentiality and integrity protection) unless absolutely necessary (e.g. in case of authentication). This also implies that user-visible interfaces must be minimized and hence that all policy enforcement must occur without requiring changes and emendations to the application programming interfaces used. Similarly, since file system properties (e.g. file system or media types) are transparent to users, it is also necessary for security mechanisms to retain this transparency. This also applies to storage locations within file systems; the use of separate container volumes with fixed or pre-defined capacity would break the transparency of mechanism.

Threats to storage mechanisms can be categorized into on-line threats, i.e. occurring while a trusted OS is providing mediation to storage resources, and off-line threats where no trusted mediation of access to storage resources occur.

On-line threats For the purposes of this paper, only software-based on-line threats are considered. These include:

Impersonation Attackers may gain access to credentials of authorized users and can perform any action on files for which the legitimate user is authorized.

Subversion Subversion can occur both at the level of user privilege and at the system level; the latter is discussed here under the heading of privilege escalation. By inducing authorized users or processes to perform actions in the interest of the attacker or of the attacker's choosing, attackers can perform any action on files for which the legitimate user is authorized [14, 2].

Privilege escalation In addition to performing actions as part of subversion attacks (e.g. by editing privileged executable code or data), attackers may gain access to privileged execution environments or memory and storage accessible only to the OS. In such a case the complete behavior of the OS can be compromised as well as any and all data processed by the given system.

It should be noted that mechanisms such as the one described in this paper does not provide protection from the privilege escalation threat category described above. However, a capability-based supplemental security control mechanism as discussed in section 3 (as well as a policy mechanism, cf. section 2) can provide mandatory security controls which at least partially counter impersonation and subversion threats when coupled with strong authentication mechanisms [9, 8].

Off-line threats This category includes both attacks occurring locally while outside the control of a mediating operating system (e.g. in case the operating system is not active) and remotely. These include:

Removable media Removable media, including external hard disks, flash memory, and hybrid devices (e.g. portable audio players, digital cameras) can be easily acquired by an attacker or be inadvertently misplaced by authorized users. Similar considerations also apply to portable computers.

Network file systems Storage over network file systems may, depending on the trust status and security controls of the file system used, also be considered as off-line storage even though their use is transparent to an operating system using such file systems on-line.

Network attached storage All considerations for network file systems also apply to network attached storage; however, NAS systems need not be part of a consolidated trust domain.

Storage area networks While typically part of a consolidated trust domain, SAN systems may distribute the content of a logical file system onto a dispersed set of physical devices which may expose the physical media to handling by attackers.

Foreign system access Host systems may be operated using a foreign operating system or may have storage media removed for access by such a system. The resulting opportunities for unauthorized access are identical to those described above for removable media.

In each of these threat environments, the system can be compromised since attackers can read and manipulate file contents as well as file system structures. Such manipulation may also include the insertion of subversion components for later use (e.g. in case partial encryption or strong authorization is used by the operating system under attack). Also, the complete behavior of the OS (if a storage medium containing OS components or other trusted objects is affected) can be compromised as well in the process. Given e.g. removable media and network-based storage, requirements for effective security for countering off-line threats are increasingly adequate not only for sensitive areas but for the protection of confidentiality and integrity of baseline systems. File-based transparent encryption can mitigate this threat.

2. Background

While the security mechanisms for file systems described here can be used independently, a number of related threats cannot be addressed adequately with storage and file system based mechanisms alone. Most of these threats are addressed as part of a comprehensive security policy definition and enforcement framework (COSEDA) providing for the definition of arbitrary security policies for both local and networked environments including coalition environments and covering all external interfaces beyond file systems, including network and device interfaces such as USB, serial, parallel, and FireWire ports.

Together with the mechanisms described in this paper, security controls can therefore ensure that activity involving devices and communication external to a given host can be mediated at the most appropriate abstraction level [24]. For example, the use of an IEEE 1394 interface to connect two host systems (e.g. one under Microsoft Windows XP and an Apple iPod device under the Linux OS). On connecting systems, the plug-and-play subsystems will perform bus enumeration and identify system and device type classifications. Communication can now be mediated or terminated based on policy decisions at the bus device driver level [25]. However, particularly for interfaces such as IEEE 1394 this may not be an appropriate level of granularity, since the interface can be used as a network connection, e.g. using the TCP/IP protocols. In this case, another abstraction layer providing interception at the network protocol stack can perform policy-based mediation (in addition to host OS firewalling mechanisms) [22].

However, if the same link is to be used for sharing file systems (here, e.g. by the Linux iPod providing CIFS shares via Samba services), the file system layer provides the appropriate semantics instead of the preceding two layers of abstraction. This results in increased efficiency in enforcing security policies since the semantics of each layer can be taken into account in policy decisions and can also provide for more expressive policies that otherwise could not be formulated or would be excessively complex. The security policy mechanism of the COSEDA program is fully described in [24]; it permits the formulation of arbitrary security policies over abstract model systems, both formulated in a first order formal theory. Given abstraction mechanisms based on a lattice structure identified by formal concept analysis of the model system and the ability to use automated reasoning techniques to deduce policy decisions within the framework, policies can be formulated at appropriate abstraction levels (e.g. staff groups and project documents) which are then used to derive concrete actions and requirements at lower levels (e.g. network and file system activities within an OS). The formal model is mapped onto target OSes by way of an interpretation of the formal theory; this typically requires an intermediate step of creating a homomorphous model for each target system.

However, since file system security requirements also arise where only lightweight administrative models are feasible or where full policy control over all system components is not appropriate, a mechanism retaining most of the flexibility in providing controls for file systems without the need for centralized policy services and management is desirable. Capability mechanisms as described below provide such a level of control and flexibility and can be integrated into host OSes in such a way that they can operate both on stand-alone systems and in large, heterogeneous, and distributed environments. This combination of host OS security controls for on-line access control and of capability-based policy enforcement for mandatory controls and off-line file system security provides a sufficient balance of controls for most applications operating in a benign threat environment [4, 12, 11].

3. Capability Mechanism

Originally proposed by Dennis and van Horn [6], capabilities have been employed successfully as a lightweight foundation for security controls in several OSes and distributed environments [13, 18]. While simple capability models are generally constructed as 3-tuples consisting of an object identifier, an access rights statement, and a (typically random or derived through a one-way function from the former tuple elements) capability identifier, the purposes of the controls described in this paper require an extended capability model:

Definition 1 *Capabilities are 5-tuples consisting of an object identifier and a set of authorized operations* (i, k, u, r, l) *where i is the unique identifier of the capability, k is the cryptographic key material associated with the capability, u is the identity of the entity to which the capability is assigned, r contains a list of access rights, and l is the expiration time for the capability. Assignment or possession of a capability is necessary and sufficient to gain access to an object identified by the capability within the rights identified by the respective element r. Capabilities* $C = (i, k, u, r, l)$ *satisfy the following security axioms:*

1. *Capabilities cannot be created by unauthorized entities*
2. *Capabilities cannot be modified by unauthorized entities*
3. *Entities can only create capabilities through well-defined interfaces*
4. *Capabilities are only provided to entities whose authorized operations match those described in the capability tuple.*

5. *Capabilities cannot be used to gain access after the expiration time has passed.*

Unlike classical capability systems, identity-based capability systems satisfying definition 1 do not suffer from uncontrolled capability propagation [4] since any propagation is tied to the identity embedded in the capability. Therefore this can be taken into account when creating a new capability for the entity to which (if so permitted by the applicable policy) the access rights are to be extended [9, 8]. A centralized system has no need for capability lifetimes since it can perform online validation. This is not necessarily possible for the systems considered here since protected systems may intermittently be unable to communicate with policy servers. The ability to delegate capability checking provides for such time intervals and also serves as an implicit time bound for capability revocation and renewal. This mechanism is thus suitable both for standalone and distributed systems. Particularly in the latter case, the capability mechanism also provides for a $O(1)$ mechanism to provide authorization independent of the number of nodes within the distributed system, delegating decisions to edge nodes.

Usability and Interoperability In case of file system security mechanisms, the objectives of usability and interoperability are running in parallel to a considerable extent. One of the core principles in usability is conformance to user expectations, which parallels the requirement not to change application behavior and minimal system changes. By performing translation from encrypted to plaintext form at the level of the file system interface, this objective can be met for the host platform. More important, however, is the design decision to encrypt and embed capability information in place as well as to encipher file and directory names in place (see section 4.1). This not only permits the use of arbitrary local and remote file system networks but also the interoperability with other operating system platform implementations. An early prototype of the encryption mechanism also exists for the Sun Solaris (SPARC), SGI Irix (MIPS) and FreeBSD x86 platforms.

4. Implementation Aspects

Core elements of the implementation mechanisms used on the Microsoft Windows NT/XP platform have been described in earlier publications [24, 23]. In addition to administrative functions for generating and maintaining capability lists (which are beyond the scope of this paper), the implementation mechanism consists of two extensions to the base OS. The first, to the identification and authentication subsystem, is described in section 4.1 while the second consists of a file system filter driver intercepting file system operations regardless of file system type (cf. [24]).

4.1. Identification and Authentication

The capability architecture requires strong authentication for satisfying the requirements of definition 1; under the Microsoft Windows NT/XP architecture this is accomplished by an extension of the Graphical Identification and Authentication (GINA) mechanism of the base OS. To ensure overall system integrity and to protect against off-line manipulation of the OS and cryptographic file system, a boot protection and system encryption mechanism is used (see [24] for details). The OS mechanism provides for a well-defined key sequence which is non-bypassable; this feature[1] implements a switch to a desktop (i.e. an output rendering with attached processes permitted to perform actions on the display) which cannot be accessed for input or output by application programs that is always triggered by entering the system attention sequence[2]. If triggered by software, the user can also confirm the authenticity of this request by initiating the non-bypassable system attention sequence (SAS), for which the base OS guarantees that it cannot be intercepted or simulated by application programs. Any impostor application displaying an authentication request similar to that of the OS is therefore disconnected from user I/O once the SAS is issued, and only the legitimate request can be displayed. Moreover, the OS also ensures that no application program can intercept communication with the display once the SAS has been issued. This can then be used to complete the authentication and authorization mechanism. The implementation is based on chaining a GINA DLL in the login process after the system login used. This provides flexibility in environments where the base OS I&A mechanism has been replaced by a third party mechanism (single sign on systems or network OSes such as Novell NetWare) and is accomplished through multifactor authentication. In the simplest case, a memory token (e.g. an USB stick) containing key material must be provided prior to authentication, which typically occurs at login time for a given user. In addition to the regular login procedures, a PIN or passphrase is then requested from the user, which is then used to decrypt the capability list assigned to the user based on a symmetric key mechanism. Alternatively, for environments where a public key infrastructure is available, a PKI-based mechanism for retrieving capability material is provided. This mechanism is implemented based on the Microsoft PKI and PC/SC smart card standard and supports both base and extended providers including the Entrust PKI. Moreover, in this case the hybrid-encryted capability list can not only be retrieved from local memory but in networked environments

[1] A requirement that originated in the TCSEC B2 class but which was nevertheless included in the Windows NT design [21].

[2] However, it should be noted that one must still assume that none of the device driver or other kernel components involved in I/O are tainted or compromised; trusted computing extensions to the basic system platform and hardware proposed by the Trusted Computing Group may in the future provide additional assurance for such circumstances.

can also be queried from an LDAP or Microsoft Active Directory server. Regardless of the mechanism for accessing the capability list, only the kernel extension for the cryptographic file system performs user I&A and has access to the decrypted capability list thus associated with the user.

4.2. Enforcement of Capability Properties

The implementation of the capability mechanism described here is based on a minimally modified host OS and therefore cannot make use of capability features in either the OS itself or system hardware [13, 17, 18]. Instead, the mechanism must be enforced primarily through cryptographic means. The implementation of the enforcement mechanism in the form of an upper level file system filter driver provides a natural control flow where the mandatory controls imposed by the capability mechanism are applied first, and discretionary host OS controls[3] applied subsequently, yielding the desired semantics. This layering provides several important benefits. First, it permits the presentation of a completely unchanged API to the upper layers of the OS and hence also to application programs in that operations conforming to the security policy are seen by these layers as in cases where no capability mechanism or encryption were taking place. Second, this mechanism is independent of the file system to which it is attached (for a description of the attachment and system-wide encryption mechanisms used to protect file system integrity see [24]), supporting e.g. local FAT and NTFS file systems but also UDF file systems used for transparent CD-R(W) or DVD-R(W) writing or even custom file systems used by application layer superencryption mechanisms. Moreover, this mechanism is also capable of supporting arbitrary network file systems and redirectors, which is important for interoperability (cf. section 3). For these file systems, regular semantics are retained so that directory structures are maintained and individual files can be handled. Files are individually encrypted and chained with a capability descriptor, while file and directory names themselves can also be encrypted to limit covert channels and inadvertent disclosures. Without the full COSEDA architecture, the enforcement of capability-based access rights is limited to file system operations and objects stored within file systems. Since full control over the life cycle of a file object (e.g. residence in memory) is beyond the scope of the file system in all modern virtual memory based operating systems, the controls are limited to a subset of operations on the file, namely to the creation or opening of file objects, and reading and writing of files. Of primary interest in this process are `IRP_MJ_CREATE` requests. This I/O request packet (IRP) is always issued when a file is accessed for the first time (not just for file creation) by an upper level function of a process; where additional information is needed about a file, a subordinate request in the form of an `IRP_MJ_QUERY_INFORMATION` can provide this data. For operations on existing files, the end of the file is read and it is determined whether this constitutes a capability trailer (CT). This step is necessary since it cannot be determined solely from a file's location whether it was under the control of the capability mechanism. If present, the CT is decoded and the identifier is compared against the capabilities of the user identified in the process of accessing the file. In case of a match, all subsequent read and write operations are transparently encrypted and decrypted, respectively. Upon closing files, an `IRP_MJ_CLEANUP` IRP is sent by the I/O Manager; this request is used to re-compute the CT and affix it to the end of the file in case the file content and size of the file has changed. Similarly, the `IRP_MJ_CLOSE` request must also be processed since the internal bookkeeping tables associated with the file object must be released. For newly created files, the capabilities associated with a file are determined by list structures maintained by the security subsystem detailing which file systems, directories, and files are to be associated with certain capabilities. This requires the creation of a new CT and new entries for internal table entries for file objects; operations are identical to those outlined above for existing objects. Since both mode of operation and cipher used (e.g. AES-192 in CBC mode) can result in padding, and the CT also induces additional space requirements, additional storage must be associated with each file. Moreover, to ensure file system independence, it is not possible to store this data in alternate data streams or separate files since these may not be available on all OS and file system permutations. The amended file sizes themselves must not be revealed to OS levels above the filter driver itself since this would violate the transparency requirement. Thus, the filter driver must adjust the file size reported by underlying file systems in IRPs including `IRP_MJ_QUERY_INFORMATION`, `IRP_MJ_SET_INFORMATION`, `IRP_MJ_DIRECTORY_CONTROL`, and `IRP_MJ_QUERY_DIRECTORY` requests which can be used to adjust and inquire file lengths [15, 16, 24]. To minimize the state that needs to be maintained in each instance, a single VM page (4kBytes) is used for both alignment and for recording the capabilities associated with a given file. This, based on an capability identifier size of 20 bytes permits up to 200 separate capabilities to be associated with an individual file. Remaining space on the page is used up by an (optional) integrity protection value (up to 512 bits) and up to 30 bytes for padding block ciphers as well as the number of capability records. Attackers cannot gain access to a file by changing or adding capability identifiers since this would not affect the (symmetric) key with which the file was

[3] For Microsoft Windows NT/XP, these are ACL entries applied either directly or through Active Directory's group policy mechanism

encrypted. Depending on the mode of operation and use of the integrity protection mechanism, an attacker may cause changes to ciphertext resulting in undesirable changes to plaintext. Integrity protection requires computation of a cryptographic hash over the entire length of the file prior to use; while current computer systems provide adequate speed for cryptographic hash algorithms[4], the storage I/O required and subsequent modification of the file system cache can have significant performance implications in case of larger files. Capabilities can be created and revoked centrally (use of the mechanism on an individual system can be considered a degenerate case) and are transmitted to edge devices over a trusted channel (as noted in section 4.1, this can occur through shared secrets or through hybrid encryption where a PKI is available). Capabilities are issued in the form

$$i, u, E_{k_u^i}(u, r, l, k \oplus R_i), S_A(H\{E_{k_u^i}(u, r, l, k \oplus R_i)\})$$

such that H is a cryptographic hash function that is preimage and second preimage secure as well as collision-resistant. S is a digital signature algorithm used to both tie the identity of a user or group to a capability identity and to provide authenticity of the issued capabilities. The index i is a number that merely serves as an index and to link CTs to capabilities; unlike in sparse capability architectures, it is not part of the protective mechanism itself. The construct $E_{k_u^i}$ is an encryption mechanism specific to a user or to a group with a key that can be specific to the index i but must be specific to the user or group u; this may in the simplest case be a password-based symmetric cipher but is typically a hybrid algorithm tied to a user or group. R is a secret random number indexed by the capability number and shared in advance by the capability issuer and validating edge nodes and serves to mask the key material to users. This construct assumes that k provides randomization and masking of the hash function input in case asymmetric signature mechanisms are used. In addition, a simple signed revocation list is also published periodically by the capability issuer. In ascertaining the rights of a user to a given object, each CT index obtained from the object is matched against the list of revoked capabilities, then against those of the given user. In case of the first match, the capability mechanism must decrypt $E_{k_u^i}(u, r, l, k \oplus R_i)$ and verify the validity of the signature $S_A(H\{E_{k_u^i}(u, r, l, k \oplus R_i)\})$, verifying the match between the user identity stored in the capability with the user identity obtained from the trusted computing base (TCB). Subsequently, the expiry time of the capability l must be matched against the current timestamp and the list of rights r against the desired operation. If any of these steps fail, the next capability from the user or object list are matched until all possibilities are exhausted. In

[4]with moderate optimization, SHA-1 reaches 68 MB/s on a 2 GHz Intel P4 system whereas SHA-256 reaches 44 MB/s.

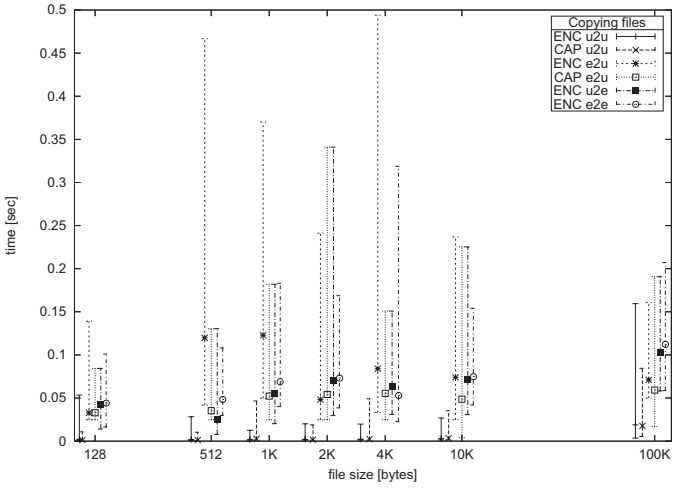

Figure 1. Copy operations

case of success, the key is decrypted and unmasked with the index-specific masker R_i and used for encryption and decryption of the given file. In writing the file's CT list, the mechanism can subsequently update entries by removing expired entries and inserting new entries obtained from the capability issuer. It should be noted that the multiple-match mechanism provides a relatively elegant mechanism for chaining limited-lifetime capabilities and for modifying access rights, typically some of the more problematic aspects of a capability architecture. This mechanism also replicates to a limited extent the dynamic policy-based security controls provided by the full COSEDA architecture. The ciphers and algorithms used (here: SHA-1, RSA-1536, and AES-192) are fully modular and can be replaced with functional equivalents where warranted by threat assessments and cryptographic requirements.

5. Experimental Results

Both capability-based systems and on-line encryption mechanisms have long been associated with performance concerns. For capability systems, the significant operations in this process are the checking of capability rights and their revocation since these are the operations occurring with the highest frequency. The following viewgraphs from experimental (unoptimized and heavily instrumented code) demonstrate the relative cost of capability-based encryption to be very moderate. The graphs compare baseline operations (no encryption/decryption) to non-capabilty encryption (denoted "ENC") and capability-based encryption (denoted "CAP"). Since the operations in question are occurring asynchronously, the differences between minimum and maximum time required are significant, but the mean shift can be seen to be very limited. It should be noted that

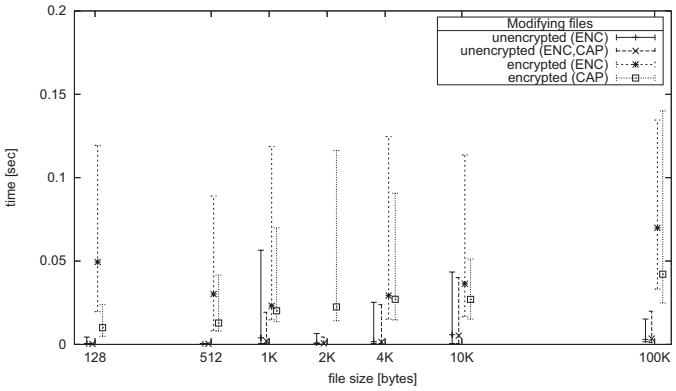

Figure 2. Modifying operations

the speed of the encryption mechanism itself is in this case heavily dominated by the unoptimized encryption implementation used in this case, but is irrelevant for the purposes of comparing the non-capabilty encryption and capability-based encryption mechanisms. Figure 1 illustrates the time differences for copying data from one path to another for varying file sizes (in the key, "u" and "e" are shorthands for unencrypted and encrypted paths), while figure 2 shows time differences for modifying data in situ. In each case, the test was performed for files of different sizes and repeated 100 times. Range bars are indicative of the strong influence on caching and concurrent operations of the OS.

6. Related Work

Security controls at the file system level are provided by virtually all general-purpose OSs; however, the limitations of file system protection only via data structures maintained by the operating system have long been obvious. Mechanisms which provide off-line enforcement of security policies or access control mechanisms generally incorporate encryption mechanisms; while a large number of specialized container volume type systems exist, these do not meet the transparency requirement that is paramount to usability of such security controls. Most documented systems are based on the Unix OS family; however, the underlying principles documented there can generally be transferred to other OS families. One of the earliest transparent mechanisms is the Cryptographic File System (CFS) for the Unix OS family. In this approach, users individually associated key material with certain directories. The files in these directories including the entries representing the pathname components were transparently encrypted and decrypted with the specified key without further user intervention after setting the key in such a way that plaintext is never stored on a disk or sent to a remote (NFS) file server. The implementation relied on the redirection of file systems through the mechanism used for NFS external to the OS kernel and user space processes for communicating key and directory information to this subsystem from individual users [3]. This was later migrated and integrated into the Linux OS by in the TCFS system [5]. This approach has the advantage of not being limited to specific devices attached to a given node as well as being able to discern among multiple users of the same system, problems that are immanent to simple volume encryption systems. The Secure File System (SFS) [10] represents another mechanism for providing cryptographic security to Unix-based systems at the user level based on earlier work on the UFO user-space file system extension mechanisms [1]. Unlike the previously described systems, SFS permitted the coordinated use of multiple client systems where individual users were authenticated using smart cards and access control was accomplished using a central server, the Group Server which maintained access control lists. Another approach, based on the observation that the processing capabilities of storage devices are reaching levels previously associated with general purpose computer systems [20, 7] was pursued in the NASD system, which embeds an OS within storage devices communicating via RPC or translations of RPC to NFS and includes survivability and intrusion detection mechanisms in the form of journaling and version retention for operations. The approaches for adding security functionality to the Unix operating system pursued by CFS, TCFS, and SFS suffer from severe performance penalties due to the cost of changing between privileged and unprivileged modes of operation. An alternative to this approach for Unix-based systems was proposed in the form of an interface at the VFS/VNode level but was not incorporated into mainstream systems [19]; however, a direct manipulation of the modular file system is feasible for most Unix variants [27, 26].

7. Conclusions

This paper has presented a transparent and usable security control that can be added seamlessly to standard off-the-shelf operating systems. Transparent policy enforcement is provided both on- and off-line through cryptographic mechanisms based on a capability architecture that is interoperable across multiple heterogeneous OS platforms and is independent of specific file system features. We demonstrated that the performance impact of using a capability-based rights architecture is limited relative to the impact for encryption and decryption, although the encryption mechanisms used in this case were heavily instrumented and not at all optimized for performance. Future work includes the integration of capability-based rights mechanisms into a distributed security architecture that not only encompasses file systems but also ensures that access to other interfaces is protected by a similar approach to efficient and inobtrusive mandatory security controls.

Acknowledgments Parts of this work was performed at Fraunhofer-IGD, Darmstadt, Germany. Parts of this work were supported by the Korean MIC (Ministry of Information and Communication) Grant (A1100-0401-0143).

References

[1] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. Extending the Operating System at the User Level. In USENIX Association, editor, *Proceedings of the 1997 USENIX Annual Technical Conference*, pages 77–90, Berkeley, CA, USA, June 1997. USENIX.

[2] E. Anderson III. A Demonstration of the Subversion Threat: Facing a Critical Responsibility in the Defense of Cyberspace. Master's thesis, Naval Postgraduate School, Monterey, CA, USA, 2002.

[3] M. Blaze. A Cryptographic Filesystem for Unix. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 9–16, Fairfax, VA, USA, Nov. 1993. ACM Press.

[4] W. E. Boebert. On the Inability of an Unmodified Capability Machine to Enforce the *-Property. In *Proceedings of the 7th DoD / NBS National Computer Security Conference*, pages 291–293, Gaithersburg, MD, USA, Sept. 1984.

[5] G. Cattaneo, L. Catuogno, A. D. Sorbo, and P. Persiano. The Design and Implementation of a Transparent Cryptographic File System for UNIX. In USENIX, editor, *Proceedings of the 2001 USENIX Annual Technical Conference (FREENIX Track)*, pages 199–212, Boston, MA, USA, June 2001. USENIX.

[6] J. B. Dennis and E. C. van Horn. Programming Semantics for Multiprogrammed Computations. *Communications of the Association for Computing Machinery*, 9(3):143–155, Mar. 1966.

[7] G. R. Ganger and D. F. Nagle. Enabling Dynamic Security Management of Networked Systems via Device-Embedded Security. Technical Report CMU-CS-00-174, Carnegie Mellon University School of Computer Science, Pittsburgh, PA, USA, Dec. 2000.

[8] L. Gong. A Secure Identity-Based Capability System. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy (SOSP '89)*, pages 56–63, Oakland, CA, USA, May 1989. IEEE Computer Society.

[9] L. Gong. On security in capability-based systems. *ACM Operating Systems Review*, 23(2):56–60, Apr. 1989.

[10] J. Hughes, M. O'Keefe, C. Feist, S. Hawkinson, J. Perrault, and D. Corcoran. A Universal Access, Smart-Card-Based Secure File System. In *Proceedings of the Atlanta Linux Showcase*, Atlanta, GA, USA, Oct. 1999.

[11] P. A. Karger. *Improving Security and Performance for Capability Systems*. PhD thesis, University of Cambridge Computer Laboratory, University of Cambridge, Wolfson College, UK, Mar. 1988. Technical Report UCAM-CL-TR-149.

[12] P. A. Karger and A. J. Herbert. An Augmented Capability Architecture to Support Lattice Security and Traceability of Access. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy (SOSP '84)*, pages 2–12, Oakland, CA, USA, May 1984. IEEE Computer Society.

[13] H. M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, MA, USA, 1984.

[14] P. A. Myers. Subversion: The Neglected Aspect of Computer Security. Master's thesis, Naval Postgraduate School, Monterey, CA, USA, 1980.

[15] R. Nagar. *Windows NT File System Internals: A Developer's Guide*. O'Reilly & Associates, Sebastopol, CA, USA, 1997.

[16] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals*. Microsoft Press, Bellevue, WA, USA, 4th edition, 2005.

[17] R. D. Sansom, D. P. Julin, and R. F. Rashid. Extending a Capability Based System into a Network Environment. In W. Kosinsky, J. Garcia-Luna, and F. Kuo, editors, *Proceedings of the ACM SIGCOMM Conference on Communications Architectures & Protocols (SIGCOMM '86)*, pages 265–274, Stowe, VT, USA, Aug. 1986. IEEE Computer Society.

[18] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a Fast Capability System. *ACM Operating Systems Review*, 33(5):170–185, Dec. 1999. Proceedings of the 17th Symposium on Operating Systems Principles (17th SOSP'99).

[19] G. C. Skinner and T. K. "Stacking" Vnodes: A Progress Report. In USENIX, editor, *Proceedings of the Summer 1993 USENIX Conference*, pages 161–174, Seattle, WA, USA, June 1993. USENIX.

[20] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, pages 165–180, San Diego, CA, USA, Oct. 2000. USENIX.

[21] United States Department of Defense. *DoD 5200.28-STD: Department of Defense (DoD) Trusted Computer System Evaluation Criteria (TCSEC)*, 1985.

[22] S. Wolthusen. Tempering Network Stacks. In *Proceedings of the NATO RTO Symposium on Adaptive Defense in Unclassified Networks*, Toulouse, France, Apr. 2004. NATO Research and Technology Organization.

[23] S. Wolthusen. Molehunt: Near-line Semantic Activity Tracing. In *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop, United States Military Academy*, West Point, NY, USA, June 2005. IEEE Press.

[24] S. D. Wolthusen. *A Model-Independent Security Architecture for Distributed Heterogeneous Systems*. Logos Verlag, Berlin, Germany, 2003.

[25] S. D. Wolthusen. Goalkeeper: Close-In Interface Protection. In *Proceedings 19th Annual Computer Security Applications Conference (ACSAC'03)*, pages 334–341, Las Vegas, NV, USA, Dec. 2003. IEEE Press.

[26] E. Zadok. *FiST: A System for Stackable File System Code Generation*. PhD thesis, Columbia University, New York, NY, USA, May 2001.

[27] E. Zadok, I. Badulescu, and A. Shender. CryptFS: A Stackable VNode Level Encryption File System. Technical Report CUCS-021-98, Columbia University Department of Computer Science, New York, NY, USA, June 1998.