Digital Rights Management of Code: The SecModule Framework

Jason W Kim
Department of Computer Science
Drexel University
jkim@cs.drexel.edu

Technical Report DU-CS-05-02
Department of Computer Science
Drexel University
Philadelphia, PA 19104
February, 2005

Digital Rights Management of Code: The SecModule Framework

Key Words: Digital Rights Management Software Licensing Library Module Code Distribution Security Encryption Trust Management

Jason W Kim
Department of Computer Science
Drexel University
Philadelphia, PA 19104
`jkim@cs.drexel.edu`

Abstract

Having reliable security in systems is of the utmost importance. However, the existing framework of writing and distributing code in the form of libraries and/or modules does a very poor job of keeping track of who has access to what code and who can call what function.

The status-quo is insufficient for a variety of reasons. As the amount of code written that represents some kind of a rights-protected entity increases, we need a systematic, easily adopted framework for designating who has access to what code, and under which conditions.

The goal of this project is to develop the basis for an *infrastructure and protocol for the formalization of* access *to software.* We develop a novel, formal framework for building secured libraries and modules. This framework, called SecModule, allows developers to create libraries of code (possibly in conjunction with accompanying source code) that are session-managed. An encrypted authentication process is required for starting a session and to receive a handle, through which the code in the library is invoked by the accessor. A Variety of controls can be enforced upon the behavior of the handle for custom tuning of security policies. This framework is the basis for future advances in secure systems design and implementation.

1 Introduction

Three distinct kinds of attacks against software[3] are of interest in maintaining software security. These include making illegal copies, reverse engineering of important code (such as in a shared library) and tampering with a prepackaged entity to extract important content (such as movies). Collberg also differentiates two additional scenarios, whether the protection being deployed is against a malicious program that we want to execute safely, or whether we are interested in protecting our code that executes in malicious hosts.

Clearly, the lion's share of those interested in protecting intellectual property are concerned with protecting our code and data that executes in a possibly malicious host somewhere out in the wilderness. Much work already has been done[17, 19, 6, 12, 15] that provides various levels of protection in different contexts.

Until now, the phrase "Digital Rights Management" has been broadly used to mean the design and use of some kind of a trust management system (e.g. KeyNote[2]) for the purpose of managing access to a resource. Our contribution, SecModule, is a systematic approach to rights management for *software*. The access rights in question would be whether an entity p is allowed to execute some function f_i held secure in the module m.

Providing countermeasures against all three kinds of attack is the goal of our work. The scenario however is a little bit different. Because we must presume the existence of a rights management system, the malicious entity that we are protecting against is the client process p which requests access to a secured resource m which exist in a host that features the SecModule framework.

Adding in authentication requirements to the calling of functions in a library is a very simple concept that is long overdue. Its very intuitive especially when thinking of computation as a protected resource.

This work attempts to answer the following questions:

1. Which encryption methods are appropriate for the contents of the library to block unauthorized access? How are the different segments of the library encrypted?

2. What is the protocol for accessing the secured library?

3. What are the ways to generate a secure "handle" which will allow only processes that has successfully shown its valid credentials to allow to call or invoke functions within the library? Will this work with dynamic linking only? or is it possible to do this for statically linked libraries as well?

4. How are the risks of misusing the access handle minimized? That is, is it possible to limit the handle to be usable only by a validated process? In other words, only processes that have successfully authenticated should be given the ability to invoke the library method, and the handle must be valid only for a specific process. Reverse engineering should also be made very difficult - within the limitations of running on an unsecured hardware.

5. What are the performance implications of having encrypted libraries and function calls?

Question 1 is the easiest to answer - practically all secret key systems as well as public key cryptosystems that have survived the test of time, (i.e. in use with KeyNote[2]) can be adopted for use in SecModule.

Note that without some kind of hardware assistance, it is impossible to guarantee that an encrypted object remain tamper-proof (except for certain very specialized operations[13]), especially if it is possible to observe the decryption process via a hardware monitor or emulator. The task of the encryption, then is to discourage all but the most determined attackers.

Thankfully, much of the work needed for answering question 2 has been already been done by the KeyNote Trust Management System[2]. We simplify by disallowing delegation of authority. That is, only credentials signed by the originator of the module will be accepted as valid. The full decentralized chaining of credentials was seen as unnecessary for purely local access to locally installed libraries. In other words, a SecModule m can only be used by the principals that have a directly signed credential from m's originator. This limitation is enforced by the SecModule system.

Furthermore, we specialize "software" as modules of invokable functions protected by encryption. Towards that end, we address questions 3 and 4 in the next section (2). We discuss related technologies such as code obfuscation and water-marking in section 3. Question 5 is addressed in the section (4). We conclude with lessons learned and possible future research.

2 Generating a Secure Handle

We refine question 3 posed above in section (1) by mentioning several related issues that impact the formalization of access rights to software.

1. What is required of the programming environment for creating and using linkable libraries that are secured against unauthorized access?

2. What is required of the operating system in order for such libraries (e.g. SecModule compliant libraries) to function properly, with its guarantees intact?

3. What are the requirements for the hardware on which 2 runs?

Item 1 is beyond the scope of this work, and can not be discussed in full detail here. Suffice to say, after a relatively thorough review of existing systems[8] we conclude that *none* of the readily available application development environments and programming languages (as well as those that aren't readily available!) have the necessary tools for generating, and allowing use of a secured, access controlled chunks of executable code, in a secured environment. Specifically, the traditional model of directly using arbitrarily generated chunks of executable code as an invokable or linkable library is no longer acceptable in the context of requiring access validation for the rights to invoke code. Java[7] does have some intentions of having a secured programming environment, providing a protected virtual machine "sand-box" but the authentication aspects of its libraries are incomplete. Furthermore, Java's protection model is geared towards the safe execution of possibly malicious programs (i.e. applets), and thus is not directly applicable to our problem domain.

Although it can be argued that having login access to the machine is an implicit guarantee of access to code, it is too coarse a grain - as it allows access to all libraries that happen to be readable by the user, regardless of whether it make sense for this user to be able to invoke functions in that library.

Ideally, in order for true access authentication guarantees to be valid, the programming environment and the operating system must be very tightly coupled together. Execution of code as well as the linking process (dynamic and static) has to be subject to airtight access controls. Thus, the operating system must control access to resources which *generate* executable code! This implies that even the rights to access the compiler must be under some kind of a trust management system in order for the framework to be considered complete.

For now, lets ignore this complication and assume, a-priori, that some how the code generation facility is under the control of such a trust management system, and that the credentials generated are stored securely - and that no other way to generate and invoke executable code exists in this hypothetical system.

In such a system, the guarantees enforced by the compiler's access policy becomes an axiom from which the rest of the security guarantees naturally flow. Furthermore, the programming language itself must be able to provide level of guarantees about behavior of programs written in it (i.e. trust management of access to C++ and/or assembly language which allow unrestricted access to memory through pointers, is not likely to be a useful basis for building secure systems). However, as of yet, OS's with such guarantees do not exist. Therefore, we must take into account the vulnerabilities in existing software development paradigms, and existing operating systems.

In the next section (2.1), we concern ourselves with certain countermeasures against possible exploits that *may exist now*, but would *not* exist in this hypothetical OS mentioned earlier. In other words, given that most modern existing OS's are written in C/assembly with no trust management for compiler resources, our countermeasures exist for foiling theft of code assuming SecModule is implemented in an existing OS, such as OpenBSD. Additional details regarding the minimal amount of support necessary from the OS Kernel is also discussed later in section 4.

Existing operating systems (for example, the OpenBSD variant of UNIX) have come a long way towards providing an out-of-the-box secure environment (as opposed to requiring expert reconfiguration after install), but even then, the concept of a root privilege class which can disable or even work around security policies that a *local user* should not have control over can be an anathema to security policy implementors.

2.1 Algorithm

A process p runs, and during its execution, p requests access to some function f_i in contained in SecModule m. Initially, all images in m remain securely encrypted. Once the request has been successfully processed, the SecModule system provides to p a handle h which allows access from p, and only p to f_i. The last criteria, to enforce that p and only p is allowed to access to f_i is ensured by the following:

The handle h, is a "co-process" that is started upon request for access to m. The actual dispatch to f_i in m is via an indirect call, managed by the OS Kernel.

The handle process h maps the appropriate memory pages onto itself, and also has the ability to *unmap* the images of m from its address space according to the security policy criteria. The simplest policy is to allow access to m for the lifetime of p. Other policies may be implemented with this scheme.

One drawback to SecModule is that it loses much of the space savings benefits that traditionally come with dynamic linking – direct sharing of a single executable image among multiple processes is no longer possible.

2.2 Additional Security Measures

For adding deterrence value, each executable image f_i of a function within the secured library m is memory mapped into a unique portion of the handle process h's address space (i.e. the base address A_{pi} that f_i is mapped to in h is randomly determined), such that the chances of the image of f_i being mapped to the same virtual address range on any two executions of the application is arbitrarily close to zero.

Furthermore, the image of f_i must be modified in such a way that valid results are obtained if and only if the code in f_i is executed with the same the starting address A_{pi} determined during the handle generation process. Thus, when SecModule libraries are being built, the system must be able to produce images that are effectively identical to classic absolute addressed objects. Whether this is achieved with actual absolute addressing, or via relative offsets which are generated at mapping time is not relevant. The SecModule system also generates a different mapping for each client, so that the same module m which is accessed by different clients have a unique base address for each mapped item for each client p.

In addition to the pseudo-random placement of actual executable pages of f_i at random locations, the function f_i itself can be broken up in to an arbitrarily many number of sub fragments (which may differ for each unique client p_i) that are all addressed absolutely, such that even if the actual images are stolen, they'd only function if mapped to an executable at the exact same addresses as they were when they were initially mapped. This makes reverse engineering of images that make up f_i more tedious.

It is also important to point out that the level of deterrence supplied by SecModule is inversely proportional to the risk, or likelihood that the raw, unmapped image of f_i be captured before they are mapped onto unique addresses in h. If it is possible to steal an unprocessed image of f_i before it is mapped, then the deterrence value of SecModule is precisely zero. Therefore, each function f_i in m must be decrypted separately from the rest of the functions in m. Furthermore, additional security is gained by splitting up f_i into multiple fragments which are processed separately.

Tamper-proofing schemes (discussed in section 3) can be applied to each image f_i separately to discourage crude attempts at reverse engineering. Obfuscation techniques can also used to confuse the randomly mapped addresses. This may be of most use when each f_i is also broken up into multiple fragments as well.

Additional deterrence value is gained if we do not assume a specific calling convention for invoking libraries (i.e. all arguments are passed on the stack). For example, [8] actively prevent the specification of a calling convention for functions - generating specialized versions of a function for each unique invocation signature. So for example, calling f_i with the arguments `(16, "foobar")` would likely generate a completely different binding if the call were made with say `(int_var_eq_17, string_points_to_foobar)`.

Although a common calling convention for functions is of practical importance, in certain situations (such as run time generation, and invocation of machine executable code in [8, 9], and when trying to make sure that the authenticated process p and *only* p is allowed to call some function f_i), allowing arbitrary calling conventions is a useful technique to possess.

However, because SecModule is implemented in an OS with a defined calling convention for functions (OpenBSD), and because the actual code body of f_i is mapped onto the handle h, not the requester p, the easiest way to get arguments and return values is through a shared memory page between p and h. This means that f_i can not return arbitrary data structures directly, nor directly modify arbitrary data in p's address space.

3 Related Work

A purely software based protection of software is considered impossible[18]. Despite this, several research efforts seem to have demonstrated that certain level of deterrence value can be achieved by related methodologies of *tamper-proofing*, *obfuscation* and *water-marking*. In this section, we briefly discuss tamper-proofing and obfuscation techniques, which hopefully provide a certain level of resistance against reverse engineering and theft of executable code.

Watermarking refers to the embedding of a possibly secret message within an object, such as a copyright message within a picture or music sound track. In the ideal case, these embedded messages should be difficult to remove, and should not disturb the object's functionality. For example, in a picture, the embedded water-mark should not change how the picture is perceived. And in a program, the water-mark should not change the the results of the execution nor slow it down. Water-marks for programs that do not depend upon static data (i.e. "dynamic water-marks") can can also be given a degree of tamper resistance[16]. In SecModule, water-marks are not used directly.

Tamper-proofing is a class of techniques for producing executables that are resistant to modification attacks. That is, through various methods (such as examining the the raw image of the executable), tamper-proofing techniques attempt to provide a certain level of guarantee that an executing process is able to either verify that the executable image from which the process started from is identical to its original state, or give assurances that the program will malfunction if modified.

Obfuscation is a group of related techniques designed to confuse human comprehension of the actual methodology in which the obfuscated process carries out its actions. We only discuss automated obfuscation of machine executable code. The techniques used for the obfuscation of source code are not directly relevant to our work and are not discussed here.

3.1 Obfuscation

Obfuscation refers to a group of methods of program transformation with the following goals[3] - The transformed program $T(P)$ must have "the same observable behavior" as the original program P. Second, understanding and reverse engineering of $T(P)$ must be more time consuming than that of P. Third, it must be difficult to generate P if given $T(P)$. Lastly, the execution time of $T(P)$ must be as close to P as possible, that is, the time penalty for $T(P)$ must be minimized.

3.2 Tamper-proofing

There are three main categories of tamper-proofing[3], one is to examine the actual binary image and compare it against a known value. Collberg reports that algorithms like MD5[11] can be used to quickly generate a reliable signature of the image itself. Second, the program continually examines the intermediate results. Third, some form of encryption is used for protecting the actual executable.

One example of a workable tamper-proofing system of the third case is by Aucsmith[1]. The program is broken up into individually encrypted portions. These fragments are then decrypted and executed in some in a sequence such that any tampering with the executable will very likely cause software malfunctions.

In our framework, both obfuscation and tamper-proofing techniques can be orthogonally applied to add additional deterrence value. However, the nature of a SecModule in this work is to serve as a securely wrapped container of invokable functions. Radically slowing down the execution of a function (via Aucsmith's algorithm, for example) that is possibly called within a deeply nested loop would cause dramatic loss of performance, which would be detrimental. So a careful balance of deterrence value versus execution performance need to be maintained.

4 Implementation Details

A prototype of SecModule is implemented on top of OpenBSD v3.6 running on an PentiumIII PC. For our test case, we test the cost of the indirect dispatch from the user process p, to the handle process h and back again. We compare against an identical no-op function implemented as a locally running RPC[14, 10] service, as well as the `getpid()` kernel call as a point of reference. Our test machine is described in figure 2 in appendix A. The measurements are detailed in appendix 3. Our initial measurements demonstrate that invoking an unpacked SecModule function is slower than a simple kernel call, but is no more expensive to invoke than a locally served RPC call. Matter of fact, due to the tight coupling between the handle and the client, invoking a SecModule function is roughly 5 to 6 times faster than the identical function being executed via RPC.

Currently, generating a SecModule involves several processing steps after the actual .o files for the functions have been created. Each advertised function in the SecModule is encrypted separately, each with a unique randomly generated key. The steps for invoking a SecModule are initially similar to manually mapping a dynamic library's pages onto a running process (i.e. what `ld.so` does in OpenBSD and other similar OS's).

The SecModule generator takes five arguments. First is the name of the SecModule being generated. The second is its version number. The third argument is the set of .o files and their dependencies (including any system libraries). The fourth argument is a list of advertised functions that the new SecModule advertises. The fifth and final argument is the KeyNote Policy statement that describes the client and the rights that the client has.

The dependencies caused by the advertised list of functions in the object files can be recursively resolved at SecModule creation time. Effectively, this results in a statically relinked library consisting of the object files, with all of their external dependencies statically resolved (i.e. present in the resulting SecModule). This reduces the observable behavior of the SecModule while it is running. The current prototype only works with single closed object file with no unresolved references.

Each symbol in this new aggregate is encrypted separately using a unique key for each entity, and the secret keys are then encoded as part of the module using the SecModule generator's private key. Specifically, each advertised function of the SecModule are encrypted using a unique secret key.

A separate tool chain then registers the SecModule with the kernel, which then keeps track of the registered SecModules. At some point in time, the client process p, with credential c then makes a request to the kernel for access to the SecModule m. The kernel then verifies that c public key is properly listed in m's policy, and that m (consisting of name and version) actually is a registered SecModule. If so, then the kernel starts a new "co-process" h, linked with p and sets up some shared memory pages between p and h. It is important to note that the secret keys that wrap the individual symbols are never revealed to p. Once the SecModules are registered, the secret keys for each encrypted segment in m exist *only* in kernel space.

In our test case, there are two principals, the SecModule implementor and the client. The creation and registration of the SecModule is handled by the same principal. However, in more realistic scenarios, The SecModule m exists in a truly multiuser environment, and there is a third principal, which is the system s that hosts m. In cases like this, s must be a trusted party and the secret keys that protect m are encrypted using s's public key, and is shipped as part of m. In both cases, the operating system which hosts m has to be a trusted party. If this is not the case, then a security prerequisite is not met, and SecModule's guarantees become invalid.

4.1 Security Implications for the Operating System

In this section we answer a question that was implicitly stated in the prior section.

Why is the code body of f_i mapped to the handle h instead of the requester process p?

The answer is simple. With the limitation that C, assembly or some derivation thereof, is used to develop the application that spawns the in-memory process p, there can be *no trust* placed on any memory portion directly under the control of p

Assuming that the user who owns p received the credentials legitimately, the requirement for allowing access to m is still there. So the obvious solution is to control access to each call to f_i through a kernel level call, to get around the restriction that p can not directly access the code body of f_i. The arguments are copied over to the shared page between p and h. Then p invokes f_i indirectly by invoking a new kernel method `smod_invoke(h, func)` which will then verify that p did provide the proper credentials (a boolean level check, as the physical KeyNote processing would have already been done by the time h was generated), and passes control over to h which will execute f_i on p's behalf.

This abstracted function call is not necessary if the OS and language itself did not allow arbitrary formulation of addresses and jumps. But as no such OS exists yet, we are forced to accept this slowdown in order to increase the level of deterrence.

The steps taken above with respect to randomizing the mapping addresses, tamper-proofing and/or obfuscating the actual images are all towards deterring brute force theft of the image by either emulation and/or hardware aided debugging.

In summation, the minimal set of changes needed in the OS are as follows:

1. Several new kernel level calls with the associated user level wrappers. See Figure 1 in Appendix A. The several `void *` arguments are pointers to structures that contain the needed arguments, i.e. additional information about the bodies themselves - generated through external tools.

2. Processes no longer generate a core image when they crash. Certainly no Handle process should!

3. `ptrace()` and related kernel calls must not allow tracing of any processes associated with the handle.

4. System libraries that are critical to security must not be available as dynamically linkable objects. In a system which feature a compiler, allowing

for the run time sharing of critical code facilitates for run time replacement, or the proverbial "Trojan horse" attack where the application inadvertently uses modified routines that give the attacker the results it wants. Ideally, KeyNote, Crypto, and SSL libraries (as well as their dependencies!) should be changed such that spoofing of these critical modules can not happen for a pre-existing application. Moving them into the kernel is one solution.

5. As always, extreme care must be taken when choosing the pseudo-random keys for the symmetric cipher that actually encrypts the bulk of m.

4.2 Supported Licensing Schemes

Our prototype implementation directly supports several basic licensing schemes. Although the actual license specification language (KeyNote, in our case) can support a larger set of possible licensing schemes, the complexity of dealing with (more to the point, *enforcing*) arbitrary schemes prevents them.

The first licensing scheme is the most basic - it allows unlimited access to m. The handle process h exits when p does. Kernel support is required for "co-exit" of two processes.

The second licensing scheme is a time limited one, where the license specifies a closed region of time in which a process is allowed access to m. For now (pun intended), the handle process uses the current system time as reported by the hardware. A better scheme would be to check a credentialled networked time source. Time limitations (again, pun intended) prevented its implementation.

Because of OS support from OpenBSD for secure communications via SSL, a third variant scheme would not be too difficult to implement. This scheme is a "just-in-time" verification - in other words, the handle h communicates with the vendor to receive last minute "go-ahead." This scheme can be used to enforce any number of stateful transactions between the client and the vendor, such as "pay-per-play" However, practical considerations prevented its implementation for the time being. These concerns are discussed in the next section.

4.3 Drawbacks in KeyNote

The stated goal of KeyNote is to support arbitrary licensing schemes. Unfortunately, KeyNote is unable to directly work with some types that are likely to be useful when expressing and enforcing many different licensing terminologies, such as time. Currently, the only way to formulate and compare timestamps within KeyNote's assertion syntax is to represent them as stringified integers, like "20041123" for November 23, 2004. This is suboptimal because it is possible to represent nonsensical timestamps such as "17761762" for the month 17, day 62 of year 1776 in the actual assertions themselves.

In other words, the computational state space of KeyNote assertions are based upon three primitives - integers, floats and strings - that do not always meet the design criteria for formulating real world licenses. Using ill-suited primitives is one of the great sins of software engineering. It can lead to bugs and potential exploits of vulnerabilities.

Correctly reasoning about time can be a critical component of licensing schemes, and not allowing for its direct expression can be a serious lack in KeyNote. The workaround of course is for the application and all KeyNote assertions to express time in GMT epoch format i.e. express time as the number of seconds since 1970. Regrettably, most humans (even cryptographers) will likely find this not very readable, thus KeyNote's claim of "human readable"[2] assertions falls flat on its face.

Additional primitives that are likely to be useful, such as sets, or at the very least bitwise operations are also missing from KeyNote. Much of the complexity necessary for real world licenses are distilled into comparison expressions between variables of the three supported primitive types, and it is up to the application developer to retrofit the license terms in this manner.

KeyNote's implementation also has a general problem beyond missing primitives. It is difficult for the KeyNote application to ask questions about the assertions provided to the application. For example, there does not seem to be an easy way for a KeyNote application to receive the list of action variables in use in an actual KeyNote assertion. Therefore, in order to guarantee proper processing of *any* supported license type, *all* authentications must generate *all* possible action variables, even if certain steps were not required for verifying a specific license.

This means that in order to process simple "on-off" license, the handle must actually generate the values for all possible variables that are supported, *including* time, as well as contacting the vendor via SSL to get the final "OK" only to time out 5 minutes later because that particular vendor hasn't a clue why we're trying to connect them. The only workaround (besides delving into the internal KeyNote data-structures) is for the SecModule system to keep a separate map of vendors and their supported action variables, which is an inelegant solution.

5 Conclusions

In this work, we have shown a framework that allows the formalization of access to the rights to execute code in a trust management environment. Because SecModule currently exists in a system (OpenBSD) that does not feature the axiomatic guarantees for access control of the rights to generate executable code, our design includes several countermeasures against possible attacks and exploits that would not be needed in a (as of yet hypothetical) operating system that has a workable policy management of access to resources that create executables and libraries.

SecModule must rely upon the following facts.

- The keys wrapping the important contents of m must not be revealed to the client process p. This is guaranteed by the registration process which takes m and registers it into the kernel.

- The OS must protect h's address space from the client p. In some ways, this is beyond our control. We can only hope that the host system s (OpenBSD in our case) has no exploitable weaknesses such that client, given the full compilation capabilities available in s is unable to formulate an exploit to snoop in h, nor (more importantly) gain system privileges.

For our test implementation, we specialized the problem of the trust management of access to software as secured access to linkable, executable modules. However, in principle, our novel framework, paired with a sufficiently well managed software development environment (and accompanying OS) can be applied to the problem of managing rights to all software as a whole.

As an example, visualizing code (binary or otherwise) as a protected resource does not necessarily have to be tied to purely commercial venues, nor does it have to be tied strictly to actual executable code. For example, The GNU Public License[4, 5] presents a framework in which source code licensed under the GPL, and all derivatives thereof always remains under the GPL. Currently, the GPL is enforced in an ad-hoc manner (if at all), because the actual text of the source code is distributed as a context-free sequence of characters. Within our framework, GPL'd source code can be distributed as a SecModule such that all derivative works also remain under the GPL, and such policy level enforcement can be guaranteed by the SecModule system, given sufficient axiomatic guarantees from the OS.

Supporting such licensing schemes also entails a sophisticated trust management language. We expect that the trust management systems (TMS) as a whole will undergo evolution as more communities adopt its principles and come up with novel ways of taxing the TMS's expressive nature.

References

[1] David Aucsmith. Tamper resistant software: An implementation. In *Information Hiding*, pages 317–333, 1996.

[2] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. RFC2704: The KeyNote Trust-Management System Version 2, 1999.

[3] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proffing, and obfuscation: tools for software protection. *IEEE Trans. Softw. Eng.*, 28(8):735–746, 2002.

[4] The Free Software Foundation. The GNU General Public License, Version 2. `http://www.gnu.org/licenses`, June 1991.

[5] The Free Software Foundation. The GNU Lesser General Public License, Version 2.1. `http://www.gnu.org/licenses`, February 1999.

[6] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[7] James Gosling Ken Arnold and David Holmes. *The Java Programming Language*. Addison-Wesley, 2000.

[8] Jason W Kim. *The META4 Programming Language*. PhD thesis, Lehigh University, 2002.

[9] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, Department of Computer Science, 1992.

[10] Sun Microsystems. RPC: Remote Procedure Call Protocol specification: Version 2, 1988.

[11] Ron Rivest. RFC1321: The MD5 Message Digest Algorithm, 1999.

[12] Tomas Sander and Christian F. Tschudin. On software protection via function hiding. In *Proceedings of the Second International Workshop on Information Hiding*, pages 111–123. Springer-Verlag, 1998.

[13] Tomas Sander and Christian F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security*, pages 44–60, 1998.

[14] R. Srinivasan. RFC1831: RPC: Remote Procedure Call Protocol Specification Version 2, 1995.

[15] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171. ACM Press, 2003.

[16] Clark Thomborson, Jasvir Nagra, Ram Somaraju, and Charles He. Tamper-proofing Software Watermarks. In *Proceedings of the Second Workshop On Australasian Information Security, Data Mining and Web Intelligence, and Software Internationalisation*, pages 27–36. Australian Computer Society, Inc., 2004.

[17] G. Wroblewski. General method of program code obfuscation, 2002.

[18] Boaz Barak; Oded Goldreich; Russell Impagliazzo; Steven Rudich; Amit Sahai; Salil P. Vadhan; Ke Yang. On the (im)possibility of obfuscating programs. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2001.

[19] Xiangyu Zhang and Rajiv Gupta. Hiding program slices for software security. In *Proceedings of the international symposium on Code generation and optimization*, pages 325–336. IEEE Computer Society, 2003.

A Figures

```
301 STD { int sys_smod_find(const char *name, int version); }
302 STD { int sys_smod_open(int m_id, \
void *cred, int cred_size); }
;; sys_smod_session_info() is ONLY for the handle process,
;; that is, the handle process started by sys_smod_open()
303 STD { int sys_smod_session_info(void * sinfo); }
;; sys_smod_handle_info() is ONLY for the client process
;; that is, the client process started by sys_smod_open()
304 STD { int sys_smod_handle_info(void *hinfo); }
;; allows multiple versions
305  STD { int sys_smod_add(void *smodinfo) ; }
;;
306  STD { int sys_smod_remove(int m_id, void *credential, \
                          int credential_size); }
;; For testing purposes, it is sufficent for
;; sys_smod_invoke() to be implemented as a userland function.
;; It will eventually become a kernel call at some point in time.
;; But its here now to reserve a number.
307     STD            { int sys_smod_invoke(int h_id, int f_id) ; }
```

Figure 1: Necessary Additions to the OpenBSD Kernel for Implementing Sec-Module

```
OpenBSD 3.6 (sys) #69: Tue Jan 25 03:52:35 EST 2005
    root@base.cs.drexel.edu:/home/jkim/sys
cpu0: Intel Pentium III ("GenuineIntel" 686-class, 512KB L2 cache) 599 MHz
cpu0: FPU,V86,DE,PSE,TSC,MSR,PAE,MCE,CX8,SEP,MTRR,PGE,MCA,CMOV,PAT,PSE36,MMX,FXSR,SSE
real mem  = 536440832 (523868K)
avail mem = 482570240 (471260K)
pcib0 at pci0 dev 7 function 0 "Intel 82371AB PIIX4 ISA" rev 0x02
pciide0 at pci0 dev 7 function 1 "Intel 82371AB IDE" rev 0x01: DMA, channel 0 wired to compa
wd0 at pciide0 channel 0 drive 0: <IBM-DPTA-372730>
wd0: 16-sector PIO, LBA, 26105MB, 53464320 sectors
wd0(pciide0:0:0): using PIO mode 4, Ultra-DMA mode 2
atapiscsi0 at pciide0 channel 1 drive 0
scsibus0 at atapiscsi0: 2 targets
cd0 at scsibus0 targ 0 lun 0: <SAMSUNG, CD-ROM SC-140B, d005> SCSI0 5/cdrom removable

CLOCK_TICK_PER_SECOND is 100
```

Figure 2: Abbreviated Test System Information

	Number of Calls/Trial	Total Number of Trials
GETPID	1,000,000	100
SYSCALL(20)	1,000,000	100
RPC	100,000	100
SECMODULE	100,000	100

	STDEV(TICKS)	AVG(TICKS)	ticks/call	mSEC/call
GETPID	0.58	65.84	0.00006584000	0.65840000000
SYSCALL(20)	0.67	64.75	0.00006475000	0.64750000000
RPC	16.33	626.97	0.00626970000	62.69700000000
SECMODULE	7.38	98.18	0.00098180000	9.81800000000

Figure 3: Performance Comparisons