Software Engineering for Security: a Roadmap

Premkumar T. Devanbu
Department of Computer Science
University of California,
Davis, CA, USA 95616
devanbu@cs.ucdavis.edu

Stuart Stubblebine
CertCo
55 Broad Street, Suite 22
New York, NY USA 10004
stuart@stubblebine.org

ABSTRACT
Is there such a thing anymore as a software system that doesn't need to be secure? Almost every software-controlled system faces threats from potential adversaries, from Internet-aware client applications running on PCs, to complex telecommunications and power systems accessible over the Internet, to commodity software with copy protection mechanisms. Software engineers must be cognizant of these threats and engineer systems with credible defenses, while still delivering value to customers. In this paper, we present our perspectives on the research issues that arise in the interactions between software engineering and security.

Keywords
Security, Software Engineering, Copy protection, Watermarking.

1 BACKGROUND

Just about every software system deployed today must defend itself from malicious adversaries. Modern society is critically dependent on a wide range of software systems. Threats from a software security breach could range from the very mild (such as the defeat of copy protection in a video game) to the disastrous (such as intrusion into a nuclear power plant control system). With the advent of the Internet, and increasing reliance on public packet-switched networks for e-commerce, telecommuting, etc., the risks from malicious attacks are increasing. Software system designers today must think not only of users, but also of adversaries. Security concerns must inform every phase of software development, from requirements engineering to design, implementation, testing, and deployment.

At the same time, changes in software development practices and software architectures have opened new opportunities for applying security engineering. Techniques such as cryptography and tamper-resistant hardware can be used to build trust in software tools and processes. These opportunities arise from the fact that software systems are no longer monolithic single-vendor creations. Increasingly, systems are complex, late-bound assemblages made up of commercial, off-the-shelf (COTS) elements and even mobile code. COTS offers great savings over custom-written software. However, COTS vendors, seeking to protect intellectual property, usually will sell components as binaries, without source code or design documentation. Software developers are thus faced with the risks of constructing systems out of unknown black box components. The late introduction of mobile code into applications is another concern. Recent research has shown how cryptographic techniques such as interactive proofs and fair random coin flips, as well as security technologies such as tamper-resistant hardware can be used by the software practitioner to address these concerns.

These and other interactions between software engineering and security engineering give rise to several fascinating research challenges and opportunities. These are the subject of this paper. We have structured the paper roughly along the lines of the waterfall model, beginning with requirements and moving on through later lifecycle activities, ending with deployment and administration.

2 REQUIREMENTS AND POLICIES

Security, like beauty, is in the eye of the beholder. A public library will clearly have a different view of computer security than will a central clearing house for interbank transactions. The specific security requirements of a particular installation can only be determined after careful consideration of the business context, user preferences, and/or defense posture. The TCSEC [3] Glossary defines *security policy* as "....the set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information". A *security requirement* is a manifestation of a high-level organizational policy into the detailed requirements of a specific system. We will

This is a pre-print of paper to appear ICSE 2000 special volume on the "Future of Software Engineering". This paper is under frequent revision until the conference. Please pass on the URL rather than copying this paper.

loosely (ab)use the term "security policy" below to refer to both "policy" and "requirement", to reflect current usage in the security and software engineering research community.

Security policies are complementary to the normal, or *functional* requirements of a system, such as the features that the customer would require. They are a kind of *non*-functional requirement, along with such aspects as performance and reliability. Favored methods for requirements engineering such as use cases [49] do not typically include security concerns as an integral part of requirements engineering. Though some security concerns are addressed during the requirements engineering stage, most security requirements come to light only *after* functional requirements have been completed. As a result, security policies are added as an afterthought to the standard (functional) requirements.

Security Models and Policies: a brief survey
Early formulations of security policies were concerned with *mandatory access control* (MAC). In MAC, objects have associated security classifications (such as secret, top-secret, etc.) and subjects may access[1] them only if they have an appropriate classification. The enforcement system prevents any other type of access. This is in contrast with *discretionary* access control (DAC), whereby access restrictions are based on the identity of the user, and any process and/or groups to which they belong. DAC is "discretionary" in the sense that subjects with a certain access permission can pass that permission to another subject.

This early work represented the first clear formulation of a model for policies that was clear, well-defined, easily implementable. This was followed by the classic Bell & LaPadula [9] multilevel security model, where objects were considered to be readable and writable. Under this model, each subject and object are assigned a security level. Subjects can only read objects at levels below them, and write to objects at levels above them. The central innovation here was that in addition to being clear, well-defined and implementable, this policy also allowed one to show that information never *trickled down*. If this policy was implemented correctly, it would be impossible (if subjects could only gain information from reading objects) for information at a higher level of security classification to leak down to a lower channel. This early work has been followed by many researchers (see for example, [42, 50, 59]), who have contributed such innovations in policy models. These efforts have led to a broad and rigorous understanding of security needs. The *breadth* enables us to capture the security requirements of a wide range of applications; the *rigor* lets us clearly (often formally) characterize the implications of these requirements.

The different *models* [59] for security policies discussed above find their expression in *policy languages*, which may be thought of as specification languages for expressing security requirements. There are several languages, as described in [72, 45, 81, 46, 34, 82]. Policy languages are implemented by *enforcement mechanisms*. There are a wide range of approaches [55, 34, 72] for policy enforcement.

The selection of an appropriate security policy and model is best done early in a product's lifecycle. The challenge is to integrate security requirements analysis with the standard requirements process.

Challenge: Unifying security with systems engineering. A central challenge in systems engineering is to develop product plans which optimize the use of limited project resources (time, funds, personnel). Competitive software vendors must utilize these resources to deliver the most value to customers as early as possible. In mature markets, competitive software vendors already exhibit quite sophisticated product positioning practices: systems engineers choose carefully from a variety of possible features, and deploy those most in demand and most likely to maximize and front-load revenue.

Functional requirements are thus being handled in a rational manner. Unfortunately, security requirements have not typically received the same type of careful analysis. Designing a "truly" secure system (i.e., defending from all credible threats) is too expensive. In practice, limited development resources force compromises. Currently, these compromises are made on an ad-hoc basis, mostly as an afterthought. We strongly support the view that systems engineering must be unified with security engineering. Just as systems engineers analyze and select market-critical features, security engineers must develop applicable threat models, and select those security measures that are most needed for market success. Available resources can then be deployed to build the right combination of customer features and security measures.

Challenge: Unifying security and system models. Software engineers use models early in the life cycle to improve the quality of artifacts such as requirements documents. Attention to quality in the early in the life cycle of a project (e.g., requirements, design) leads to defect detection and avoidance. It is well-known that such defects, if undetected, can propagate downstream, where the costs of detection and removal are greatly amplified. The trend has been to use high-level, object-oriented models (such as UML) early in the life cycle to support requirements analysis and design activities [49]. Modern requirements modeling and object-oriented design methods begin with a careful analysis of the ontology of the application domain. A model of domain ontology is first constructed, and this drives the rest of the requirements process [12, 62, 67]. This approach has been found useful in practice, and is widely used in industry, especially in the design of information systems, e.g., using the Fu-

[1]The word "access" (read, write, open, close, connect, etc) is used to denote the act of a subject performing an action on an object.

sion [15] methodology. Modeling is also useful for reverse engineering. Tools have been built to extract models from a legacy system [48, 23]. Such models can be useful for maintenance or for re-engineering.

So far, however, security modeling and policy work has been largely independent of system requirements and system models. Typically, system requirements and design are done first, and security is added as an afterthought. There has also been lack of interaction between researchers working on requirements modeling and design modeling (e.g., in the UML community) and security policy researchers[2]. Clearly, there is much to be gained by developing processes and tools to unify security policy development into the system development process, specifically by making using of system models when designing security policies. One attractive approach is to adopt and extend standards such as UML to include modeling of security related features such as privacy, integrity, access control, etc. There are several advantage that could accrue from this:

- Unified design of systems and security policies.
- Modularity (through encapsulation), compactness and reuse (through inheritance) in policy representation.
- Leverage of existing standards-based tools for design and analysis (forward engineering) activities, as well as for analysis of legacy code (reverse engineering) activities.

A primary challenge here is to extend the syntax and semantics of standards such as UML to address security concerns. We believe that this presents an opportunity for software engineering researchers. If we can develop tools and processes to help unify the design of systems and security policies, the result will surely be systems that more effectively meet business requirements in a more secure fashion.

Readers interested in object-oriented modeling are also reffered to the companion paper on the future of OO modeling [32].

3 ARCHITECTURE AND DESIGN OF SECURE SYSTEMS

Re-Engineering for Security

Software designers have long recognized the need to incorporate non-functional considerations such as performance and reliability into software design processes. It is well understood that adding performance and reliability requirements into software architectures after the fact is difficult or impossible.

Sadly, the situation with security-oriented non-functional requirements is not not as advanced: very often, security is an afterthought. This typically means that policy enforcement mechanisms have to be shoehorned into a preexisting design. This leads to serious (sometimes impossible) design challenges for the enforcement mechanism and the rest of the system. The best resolution to this problem is to refine requirements and design processes to bring an earlier focus on security issues.

There are other reasons (besides poor planning) that security is not a factor in initial systems design. The advent of networking and open standards often provide new business reasons to re-engineer internal legacy systems (which operated within secure intra-nets) for operation over the open Internet. In such cases, there is no alternative to adding security to a system after the fact. There are several problems that arise here, resulting from different types of architectural mismatch [38]. For example, data or protocol incompatibilities may render it difficult to make a legacy system's services available via standard protocols such as HTTP[85] or IIOP[68], or in standard data formats such as XML [85]. These problems are important, and deserve attention. However, this issue not directly related to security, so we do not discuss it further, except to point to related work on retargetability, wrappers, and wrapper-generation tools.

Challenge: Legacy Security Mismatches. From a security perspective, the most serious problem is one of mismatch between the security framework in the legacy system and the security framework of the target standard protocol. For example, Unix systems and CORBA have different security policies and enforcement mechanisms. Unix authentication is based on user-password authorization. CORBA uses Kerberos-based [66] authentication [69]. The Unix file system uses the well-known access control based on user, group, and everyone else. CORBA access control is more flexible, based on *credentials* that are owned by a CORBA client, and *service controls* which encapsulate the access control policy of the related CORBA servant. These difference greatly complicate systems where principals can authenticate themselves with either mechanism (Unix and CORBA) and use either UNIX or CORBA services.

Consider the task of making the services of a particular Unix application, $\mathcal{A}$, available via a CORBA object. If a particular login, is not permitted to use $\mathcal{A}$, we certainly should also make sure that the same user cannot invoke $\mathcal{A}$'s services through CORBA. Unfortunately, there is no simple way to ensure this.

Mancoridis [78] suggests the use of wrappers and sandboxes that enforce Unix-style policies. In general, such mechanisms need to be made very flexible, allowing ad-

[2]Much effort has been directed recently at mobile code systems, where the security concerns mostly burden the mobile-code host, and the system development concerns are with the applet developer; however, in general this is not true, and design of the policy and the system could be unified.

ministrators to formulate their own policies to suit their needs, and thus use the full power of the CORBA security system in offering secure services. Fraser *et al*[37] propose a sophisticated, flexible wrapper mechanism that can surround COTS components and enforce policies applicable to the system hosting the COTS components.

The rational linkage between various security frameworks is a growing area of concern. Customers demand "single sign-on", where by a user authenticates herself once using a specific mechanism and then gains access in a uniform manner to different services (perhaps on different platforms). Developing uniform policies and enforcement mechanisms for a group of services that span different platforms is a research challenge.

Challenge: Separating the Security "Aspect." The central problem in modifying the security aspects of a legacy system is the difficulty of identifying the code that is relevant to security, changing it, and integrating the changes back into the system. A promising new approach to constructing systems with evolvable security features is suggested by a confluence of two lines of research—work on aspect-oriented programming [52], and work on architectural connectors [73, 5].

Aspect-oriented programming is an approach to simplifying software evolution. The idea is that some aspects of code are naturally modular, such as data storage, which can be placed in a database. Others (usually nonfunctional requirements) such as performance, and distribution are scatterred throughout the code. Changing the way a system is distributed, for example, would involve the difficult task of identifying and changing scattered code concerned with location, connection, distribution, etc. Aspect-oriented programmers seek to isolate these fragments centrally for ease of comprehension and maintenance, thus achieving the classic goal of "separation of concerns." The task of re-scattering them back into the code prior to compilation is automated using program transformers, known as *aspect weavers*. With this strategy, an aspect such as distribution is isolated and easier to change. The other line of research arises in software architecture, where researchers study two major elements of architectures: *components*, which form the centers of computation in the system, and *connectors*, which are the loci of *interaction* between components. This is a conceptual distinction that is made at the design level—of course at the code level, no such such distinction exists. This view, however, is useful non only for design-time activities such as performance modeling, but also during programming activities, especially for program understanding. The notion that connectors could encapsulate various aspects of *interactions* [5, 71] suggests a promising line of research.

Important security concerns, such as authentication and access control, arise out of interactions between components. Thus security concerns are naturally placed with architectural connectors. Authentication, security policies, and enforcement mechanisms could be considered different aspects of connectors. Aspect weavers could take on the task of integrating the implementation of these aspects with the rest of the system. In this context, security features would be easier to isolate and maintain.

4 SOFTWARE PIRACY & PROTECTION

Software piracy is an enormous challenge to the software business. Most vulnerable are vendors of popular and expensive products such as office suites for commodity desktop machines. When the cost of a legitimate copy of a piece of software approaches the cost of a machine (*e.g.,* about $300 for an office suite upgrade, versus about $700 for the cost of an entire new machine), the incentives for individual consumers to commit piracy are intense. There are also other, more dangerous types of pirates: organized, rogue entities, especially in countries with lax enforcement of copyright laws, who have the resources to pirate thousands or millions of copies. Such entities may even export the pirated copies. From all sources, piracy is now acknowledged to cost in the range of *$15-20 Billion* annually.

Law enforcement measures are essential practical deterrents for piracy. However, these measures must be augmented with technology to keep otherwise honest people honest. While there are various technologies exist to combat piracy, we believe that a key innovation still awaits: *a good model of the economics of piracy.* Without such a model of the adversary, the effectiveness of such technologies cannot be properly evaluated.

Adversary Economics

Consider an entity (individual or organization) considering piracy of a software item. We assume that the software item is needed by the entity. The entity can either buy (for cost C_b) the item, or first hack the copy protection mechanism (cost C_h) and then make n copies (each of value C_c) and bear the very small risk[3] (P_{11}) of getting caught, which carries a large (possibly subjective) cost (C_{11}). The cost C_{11} and prosecution probability P_{11} may vary with n. There may be civil liabilities payable to the software vendor, although suing an individual in a tort lawsuit for uninsured liabilities can be unrewarding. Criminal penalties such as fines and jail terms are possible, though quite unlikely. Given current technology, C_h is usually not high, and (in most cases) copyright enforcement is lax. So the confronted with the reality that

$$n * C_b >> C_h + n * C_c + P_{11}(n) * C_{11}(n)$$

most people and organizations have a strong economic incentive to commit piracy. If the goal was actually sell pirated copies, then with C_b as the profit, the same model as above would apply. One may, however expect that the

[3]We use the subscript 11 for the so-called "eleventh" commandment.

likelihood of getting caught would be greater.

Clearly, one approach is to reduce C_b to zero. This approach, adopted by the free software movement, is remarkably effective at discouraging illegal piracy! Barring this, the technical challenge here is to increase the probability of capture (P_{11}), or increase C_h and C_c. Various approaches are now discussed, and possible attacks are discussed.

Approaches to Protection
There are various technologies that may discourage piracy. They are: hardware and software *tokens*, *water marking*, and *code partitioning*.

Hardware and Software Tokens. Software tokens are the most common technique. Along with the software product, a "license" file is shipped. This file contains information that the product checks every time it is run; if the file is not present, or the information is wrong, the product exits with a license violation error. The information may include information specific to the installation site [51], such as the hardware network card address. With hardware tokens, one actually installs a physical "dongle" on the machine, usually on a serial or parallel port. The software checks for the presence of this token before starting normal function. In both cases (hardware and software), the protection code checks for the presence of a valid token. Another approach is to obtain the token dynamically over a telephone or a network connection [44]. Yet another popular approach is to exploit idiosyncratic physical features specific to a particular digital medium (such as timing anomalies on a specific floppy disk) as a "token" that the protected program will check for; a copy will not have the same features and thus will not run. In all these cases, the basic goal is to raise the cost of breaking the protection mechanism, or C_h.

In all these cases, the applicable attack is to locate the token-checking code and patch around it. A debugger with the ability to set breakpoints can be used for this purpose. Another approach to search the binary for the character string corresponding to the license violation error. Once the code is found, it can be easily patched. The use of names containing substrings such as `license` or `dongle` in routines can make things easier for the attacker. There are some techniques that make it difficult to run a debugger on software; however, there are approaches to getting around these as well. In general, token-checking (or debugger-hindering) code in one or more fixed locations in the code can be found and removed mechanically, once the locations are known. There have been other suggestions [8] which include some level of self-checking, where the program monitors itself to detect tampering of the license-checking mechanisms. Again, one has to find these self-checking codes and disable them. Another approach used here is to add self-destruct code that would destroy all copies of the software (including any code in memory) upon detection of tampering efforts. Self-destruct approaches can be detected and contravened by running the software under full emulation (or a system-level debugger) and observing all system calls.

An entity desirous of making many pirated copies needs to find and remove these checking mechanisms once, and make as many copies as desired. An attack involves finding the checking (including self-checking) code and patching around it. Thus, the cost here is primarily a reverse engineering cost: the greater this cost, the greater the barrier to piracy. Unfortunately, none of the existing proposals based on a token technique come with any guarantees such as a lower bound on this reverse engineering cost. Anecdotal evidence regarding several commercial products that use this approach leads to the discouraging conclusion that these reverse engineering costs are likely to be low in most cases.

Dynamic Decryption of Code. With this approach, the software is stored in encrypted form on some digital media, and is only decrypted prior to execution using an independently stored key. Sometimes, multiply encrypted keys [14] (such as DES key encrypted with an RSA private key) associated with a particular machine are used to protect the software. Some have proposed associating keys with machines during manufacture [54]. Thus, copying the media without the key is useless. There are some complications. The dynamic decryption may add an unacceptable performance overhead. Customers may find it difficult to move the application from one of their machines to another, for perfectly legal reasons (e.g., when retiring one machine).

The applicable attack here would the direct monitoring of the program memory locations to harvest the decrypted code (since the code has to be in memory prior to execution). By exercising all the different features of the software, it should be possible to gather all the code for the system. It might become necessary to write a monitoring program for a specific operating system and platform that harvests the decrypted code as it is being readied for execution. Again, no precise lower bounds on the harvest cost C_h are given, but it seems reasonable to assume that it would be greater than the case of the simple token. This approach is atypical in the software business, and in fact, we not aware of any mass-market products that use it.

Watermarking. One approach to discouraging piracy is to embed a secret "watermark" in the software specific to each customer. A pirated copy could thus be traced back to the original customer via the watermark. The effect of this approach is to increase the probability of getting caught, P_{11}. If this could be made large enough, most customers would balk at the risk of exposure and prosecution.

Watermarking is applicable to any type of object, and abstractly, to any *message*. Most research in the area has

been concerned with watermarking visual or audo media [18, 87, 76, 53]. Abstractly, given a message M of N bits, one seeks to embed a smaller message, or a watermark, w of n bits into the original message, as a way of identifying provenance. The goal is that any time this message M is found, the watermark w can be readily extracted, thus establishing the provenance of the message M. An adversary A seeks is to disguise the message M so that the watermark can no longer be extracted. Collberg and Thomborson [16] have recently published a useful analysis of software watermarking techniques. They describe two desirable properties of a watermark: *stealth*, so that it is difficult for A to find it; and *resilience*, so that A cannot remove it without damaging the message M. They divide software watermarking techniques into *static* and *dynamic* approaches. Static watermarking techniques encode the watermarks in static program properties, such as the ordering in the program text of otherwise independent basic blocks. Static program analysis can then reveal the watermark. Dynamic watermarks are embedded in program state, typically in the response to a specific input. Such watermarks are also called "easter eggs." One example might be to encode a number as a particular state of connectivity in a stored graph. The watermark is detected by running the program against this input, and observing the program state. Collberg and Thomborson [16] discuss various types of watermarking techniques, and applicable attacks. Although most of their proposed techniques have attacks, they offer useful intuitions of the difficulties of watermarking programs. They also present several useful metrics for evaluating the effectiveness of watermarks; however, the application of these metrics to known approaches remains an open issue. A knotty technical problem in watermarking software is the wide variety of meaning-preserving transforms that are available to the attacker to confound the watermark. Most the watermarks that have been proposed are masked by the application of program transforms. In domains such as digital imaging and sound, where watermarking techniques have been more successful, such a plethora of meaning-preserving transforms are not available.

As we discussed earlier, the core aim of watermarking is to raise the probability of getting caught, P_{11}. Of course, watermarking says nothing about the actual cost of getting caught, C_{11}. This suggests non-technical means to evade prosecution. A pirate might hire someone difficult to prosecute, such as a juvenile, or a destitute individual in a foreign country to be the "front man" on the original purchase. Another approach might be to pirate copies in a jurisdiction with lax enforcement. In addition, legitimate consumers may have privacy concerns in associating themselves with specific software purchases. Customers may seek to mask purchases via cash or anonymous transactions [56]. If anonymity becomes common in e-commerce, watermarks will do little to discourage pirates.

Code Partitioning. A pirate with access to a bus analyzer and a probe can contrive to harvest any software that is visible in ordinary RAM, even if it is only visible immediately prior to execution. Recognizing this, some inventors recommend placing a portion of the software in inaccessible memory. One early proposal [74] recommends placing just the license-checking part of an application in protected hardware. In this case, the attacker can find the code within the application (which is in unprotected memory) that invokes the protected license-checking code, and patch around it.

To discourage such attempts, it will be necessary to physically protect a more substantial portion of the application. One such approach [80] recommends placing a "proprietary" portion of an application within ROM, leaving the rest in RAM. However, a bus analyzer could simply harvest the addresses and instructions as they were retrieved from the ROM, developing a complete $address \longrightarrow instruction$ map of the ROM, allowing it to be readily copied. To avoid this attack, it is necessary not only to protect the memory storing part of the program, but also the processor executing these instructions, and the memory bus itself. One approach [61] is to relegate the protected part of the program to a remote server administered by a trusted party (perhaps the software vendor). This program component invoked using a remote procedure call. When such a call is received, the caller's identity is checked for valid licensing before it is allowed to proceed. As long as critical functions could be separated into the remote server, pirated copies at unauthorized sites would be inoperative. Performance is certainly an issue with this approach. Perhaps more critically, users might worry about covert channels to the server, and the attendant loss of privacy. These issues can be addressed running the protected components locally, within a tamper-resistant hardware device such as a smart card [86]. The protected part of the software could be shipped encrypted, using the private key associated with the smart card at manufacture, and decrypted and run within the device [58].

A central question with code-partitioning approaches is the criteria that should be used for selecting the portion of the code to be protected. Unfortunately, this issue has remained open.

Sander and Tschudin [77] have proposed a number-theoretic approach for protecting software. Using homomorphisms over rings, one can create encrypted versions of functions to compute polynomials. These encrypted versions are resistant to reverse-engineering; adversaries cannot feasibly decipher the polynomials. However, the results computed by the polynomials are also encrypted, and must be sent back by customers to the vendor for decryption. This approach is so far restricted to the computation of polynomials. In addition, the need to have the final results decrypted by the vendor raises both perfor-

mance and privacy concerns, and may not be suitable for interactive applications. However, the approach is both elegant and provably secure, and the limitations might be ameliorated in the future.

Challenge: Attacker Cost Models. Most of the research described in this section is not based on an economic model of the adversary's behavior. Without such a model, it is difficult to judge the effectiveness of each approach.

For example, consider a token-based approach, augmented with some self-checking code, and perhaps some additional measures to inhibit debugging. The cost here is the one-time reverse engineering cost of identifying and removing these defensive measures. How does one quantify this cost for a particular implementation of the token-based approach? With the dynamic decryption approach, the attack is to monitor, using hardware devices or a debugger, the instruction stream flowing past the CPU and gradually accumulate the code for the entire system in the clear. There are several different approaches to dynamic decryption of code that attempt to complicate the harvesting task, but none of them provide a clear model of adversary cost. The same criticism applies to watermarking approaches. Collberg [16] suggests several measures to evaluate the "resistance" of watermarking techniques to attack. He also discusses applicable attacks. But the actual human cost of removing watermarks still awaits investigation.

In our view, given that adversaries have full access to the hardware and software of the operating platform, the best method to protect software is partitioning, with the protected component executed entirely within a tamper-resistant co-processor. This approach has been known for quite a while; however, proponents have failed to provide a suitable cost model. There are two attacks here: one is to break the physical security of the co-processor [7], and the other is to passively reverse-engineer the contents. The latter task would involve first fully characterizing the behavior of the protected component and re-implementing it. The vendor must ensure that these costs are high. Techniques to partition the software to meet these considerations await development.

Cost models of possible attacks must consider not only current attacks, but all possible *future* attacks. One way to do this to use reduction techniques[10] that relate the cost of attacks to cryptographic problems thought to be difficult, such as the difficulty of inverting one-way functions or factoring large numbers. Sander & Tschudin [77] is an example of this, although at the moment their results are primarily of theoretical interest, being limited to computing polynomials. Without such strong results, copy protection remains a black art.

In addition to economic models of the cost of attacks on different protection techniques, we need an over-arching model of the entire piracy process. Our piracy cost relation (above, and reproduced below)

$$n * C_b >> C_h + n * C_c + P_{11}(n) * C_{11}(n)$$

is at best a very rudimentary beginning.

5 TRUSTING SOFTWARE COMPONENTS

Much of software development today is largely a matter of *integrating* off-the-shelf components; rarely are new systems built entirely from scratch. Middleware technologies (see also the companion paper specifically on middleware [31]) such as COM and CORBA have given rise to a wide range of components, frameworks, libraries, etc. These are collectively known as commercial off-the-shelf software (COTS). A useful summary of research issues in COTS products are very attractive to developers confronted with ever more stringent requirements of cost, quality and time-to-market. However, the use of these products, particularly in safety-critical systems, is fraught with risk. The procurement policies of the customers of safety-critical systems (utilities, government, etc) have traditionally required software vendors to disclose enough details to evaluate their processes and products for safety. However, these policies are not compatible with current component vendors, who are faced with the risk of intellectual property loss. Recently a committee appointed by the National Research Council (NRC) in the U.S.A (See [70], pp. 71–76) has discussed the reuse of COTS software in nuclear power plants. Their report states (Page 76, first para):

> Dedication of commercial components requires much more information than commercial vendors are accustomed to supplying... Some vendors may be unwilling to provide or share their proprietary information, particularly about development or testing procedures and results of service experience.

Speaking on the same issue, Voas ([83], page 53) states:

> Software components are delivered in "black boxes" as executable objects whose licenses forbid de-compilation back to source code. Often source code can be licensed, but the cost makes doing so prohibitive."

This leaves would-be users of COTS products with two unpleasant choices: forego the use of components, and the attendant cost savings, or live with the risk of using a "black box" component. The COTS vendor also faces a similar challenge: how can she assure her users of the quality of her development process and the resulting product without untoward intellectual property loss? One might call this the *grey-box verification problem.*

Black box Approaches
Voas [83] proposes two complementary approaches: first, test the component *in situ* to make sure it doesn't misbehave, and second, test the system to make sure it can still function even if the component misbehaves. These approaches treat the component as a black box, and employ extensive testing to ensure that the system functions as desired. The significant advantage here is that no additional effort is required by the COTS vendor. In addition, the COTS vendor need not disclose any intellectual property. The additional testing effort is likely to be time-consuming and expensive; however, it will likely contribute towards the overall quality of the entire system, so it is effort well spent. However, if a conscientious COTS vendor has already used stringent testing and verification practices, then the above approach might lead to duplicated effort. In this context, one might seek a *grey box* that might allow the COTS vendor to guardedly disclose enough details of her verification practices to convince a skeptical COTS user, while also protecting much of her intellectual property.

Gray-box Approaches
We have described two [27, 22] approaches: one using interactive cryptographic techniques, and the other relying upon tamper-resistant hardware.

Cryptographic coverage verification. Suppose a COTS vendor has achieved 99% basic-block coverage. This is a significant achievement indicative of a stringent QC process. To convince a user of this, she would typically have to use a third party (trusted by her and the user) to verify the coverage, or she would have to disclose the source code, the tests, and any applicable tools to her customer. In [26, 27], we propose a way in which the customer can provide credible evidence of coverage, while disclosing (in most practical situations) only a few test cases. Essentially, our approach is driven by a fair random process. An unbiased coin flip (say) chooses basic blocks at random, and vendor provides test cases as evidence of coverage of those blocks. The challenges are unpredictable, and the vendor cannot easily cheat. We describe an approach whereby the vendor makes a claim about her level of coverage (say 80%) and each additional challenge lowers the upper-bound on the probability that she is lying. With about 25 challenges, we can reduce the lying probability to about 0.05. The verifiable use of a fair random process (technically, a cryptographically strong pseudo-random number generator [60]) allows the vendor, acting alone, to provide this evidence. Thus any vendor, regardless of reputation, can provide credible evidence of stringent quality control with only a modest additional effort over the cost of coverage testing.

Tamper-resistant Hardware. Suppose a vendor has, with great effort, constructed a formal proof[4] that the component satisfies some important safety property, but that this proof discloses significant details such as loop invariants, datastructure invariants etc. Certainly, the vendor would like the customer to know that this proof exists, without disclosing the proof details. In this context, we suggest the use of tamper-resistant hardware [22] device (*e.g.,* a smart card) comprising a secret private key. Attempts to extract this key would render the device inoperative. We embed a proof *checker* in such a device. Proof checking is well-known to be much simpler and faster thant proof creation. The vendor presents the device with the COTS software (appropriately annotated with invariants, etc.) and the proof; the smart card then processes the component and the proof. Now, using the private key, the smart card signs the component, and a statement that the applicable property has been proven, using its private key. We assume that the corresponding public key is introduced to COTS users with the appropriate certificates. This approach allows the proof to remain undisclosed. The COTS user can reasonably rely on the signature (and his trust in the proof checker and the smart card) as evidence that a correct, complete proof of the desired property indeed exists.

Challenges: more Grey box Approaches. Both the approaches described above are only beginnings, which address only certain aspects of the grey box verification problem. We have developed test coverage verification protocols only for block and branch coverage. Some U.S. government agencies require more stringent coverage criteria (*e.g.,* data-flow based criteria [36]). Our protocols are vulnerable to certain types of attacks, for example attempts by the vendor to boost coverage by including spurious (easily covered) code. Resilience to these attacks is needed.

Cryptographic approaches might also be feasible for some verifying the use of other types of quality control methods, such as model-checking approaches. Fair random choice approaches could perhaps be used to show that a) a particular model is a reasonable, fair abstraction of a system, and b) that the model has the desired properties. The challenge here, again, would be provide "grey box" evidence of these claims, without untoward disclosure.

6 VERIFICATION OF SYSTEMS

U.S. and other military establishments have a long history of building high assurance secure systems. Out of the U.S. effort came the so called "rainbow series" of books including the "orange book" [19] which specified security feature requirements and assurance requirements for the implementation of those features. This effort was largely too costly for use in general purpose systems.

The U.S. government has been forced to move towards using COTS software to meet cost, quality and schedule constraints. The latest evaluation criteria for software is the Common Criteria [1] which has been internationally standardized [2]. Although the evaluation required will be

[4]This situation is analogous to the proof-carrying codes of Necula [64].

more aligned with the needs of commercial systems, it is not clear that that the state of evaluation technology has adequately advanced to make such evaluation practical for most systems and companies.

Traditionally, the high quality expectations on secure systems have led investigators to apply rigorous formal methods to show desirable security properties of computing systems (particularly access control and information flow properties), and of cryptographic protocols particularly showing properties of authentication [13]. Formal methods involve significant human labour, and are expensive. They are usually based on formal specifications, rather than on actual implemented systems; confidence in formal verification is therefore subject to concerns about the fidelity and completeness of the specification with respect to customer expectations, and the relationship of the final implementation to the specification. In addition, implementations are typically much larger and more complex, and in practical languages like C++. All of this renders manual or automated verification of implementations difficult. An applicable engineering approach here is to not attempt to show correctness, but to make use of conservative techniques such as model checking [39, 29, 63, 20] and static analysis to find defects. While such techniques do not guarantee the complete elimination of defects, they can be a useful adjunct to conventional defect testing.

Challenge: Implementation-based verification methods. Automated tools that do a credible job of finding security-related defects in the *implementations* practical systems are needed. One approach is to use model-checkers on abstractions derived automatically from source code [30, 63, 17] to identify states in which potential security vulnerabilities may exist.

Another approach, by analogy with Verisoft [39], is to create "hostile" library versions of popular APIs (such the file system API), which attempt, after applicable API calls, to "simulate" attacks on a program and attempt to expose vulnerabilities. An application can then be re-linked against this "hostile" library and tested to expose security vulnerabilities.

Companion papers on analysis [47] and testing [43] present useful summaries of existing work on testing and verification, as well as important issues for the future. trained on using such techniques.

7 SECURE SOFTWARE DEPLOYMENT

The use of component software within popular desktop applications has exploded. Consider a particular component, say c, produced by a vendor v. This component c may be used in any application: a word-processor, a spread sheet, an e-mail program, or an electronic game. These applications may be created by different vendors, who package and ship their application together with the constituent components. Certainly as time passes, component vendor v continuously updates his component software c to fix defects, and improve functionality, thus creating versions c_1, c_2, etc. Thus it could happen that two needed applications eventually will require different, incompatible versions of the same component: one using c_1, and the other c_3 e.g. Thus arises a common, difficult problem: installing one application may cause another, apparently unrelated application to fail. Most users of personal computers have suffered the consequences of such problems. In addition, personal computer users are often asked to update and reconfigure their systems in response to announced threats (e.g., viruses) and newly discovered vulnerabilities. Such announcements are not infrequent, and the recommended remedies are often beyond the abilities of the unskilled user.

Resolving these problems is very time consuming, and may require a user to spend long hours with technical support personnel or with a help-line and discuss intimate details of the software installed on their PC. This certainly risks an individual's privacy. Worse, a potentially malicious outsider (at the help-line, for example) can induce an unsuspecting user to reveal critical details of the installation at a particular PC. These details could reveal vulnerabilities that could be subsequently exploited to gain access to critical data and services on the user's PC or perhaps even his network.

The task of maintaining a correct, current configuration of software at a machine has been called "post-deployment configuration management" [40] (PDCM). There are now several approaches to automating some aspects of PDCM [57, 41] (see also the companion paper in this volume on configuration management [33]). All of these provide timely distribution of both information about correct configurations, and the software releases themselves. However, there are several security challenges that remain to be addressed.

Secure Configuration Management

There are some security issues [25] which still need to be addressed in PDCM systems: controlled delegation of administration and privacy protection.

Challenge: Controlled Delegation. PDCM is a complex task. Complex application configurations and interdependencies have to be considered in deriving the right configuration. This may involve expertise drawn from within and without the organization. Some sources may be trusted for certain types of expertise, but not for others. In addition, the software itself may be published from different sites, with varying accordances of trust levels. In response to security events, responsibility for an attack may be properly fixed on a previously trusted source, whose privileges may have to be revoked. Indeed, a user may rely upon one or administrators to identify trusted sources of PDCM information and software releases. Administrators or their delegates may source information, until or unless

their privileges have been revoked.

PDCM systems need to provide facilities which can allow and enforce flexible delegation and revocation of PDCM administrative privileges.

Challenge: Privacy Protection. While obtaining information or software releases from sources, the PDCM system must uphold the user's privacy. Of course, notions of privacy may vary from user to user; the user should be able to specify what information at his or her site can be accessible to whom. The PDCM system must be flexible in this regard, but must take responsibility for enforcing such policies. Of course, there are theoretical limits to full enforcement of privacy policies; within this limitation, the PDCM system should make use of the best and most applicable technologies [79, 75] to protect privacy.

8 SECURE COMPUTATIONS, NOT SECURE COMPUTERS

All software systems are potentially error-prone, and may be expected to sometimes produce incorrect results due to defects in the software. Where security is a concern, users may worry that a system has been compromised by attackers, and thus might perform incorrectly. In general, it requires effort to decide whether the the result of a computation is correct. This need for a *test oracle* is recognized as a significant problem for software testers [84]; how can one know that a system responded correctly to a test? If it were possible for a system to produce a *proof of correctness*, along with the computed results, a user (or a tester) can gain solid assurance that the system has functioned correctly, regardless of whether it is under attack. An efficient *proof-checker* can automate the process of checking the proof, thus providing an automated test oracle.

Some early work in this area includes work to check cryptographic operations performed by an untrusted assistant [35] and for secure circuit evaluation [4]. Other approaches are in the direction of using quorum schemes for distributing trust among servers using threshold cryptography [21]. However, these suffer from performance issues associated with quorum schemes.

A low-level step towards this direction is the use of secure data structures. In this scenario, a processor off-loads storage of data onto an untrusted co-processor. In this context, it is necessary to verify that the data returned by the co-processor is correct. Approaches have been reported for off-loading RAMs [11], secure stacks and queues [28], and other linked data structures [6]. At a higher level, one would like "application servers" that provide a useful service to generate "proof carrying answers" that come with a proof of correctness. One example of this is Necula & Lee's certifying compiler [65]. Given a source program, the compiler's certifier either produces a proof of type safety (of the resulting binary) or a counter example illustrating a potential violation. As another example, consider third-party publication of databases, whereby the contents of a database are published by an untrusted entity, who takes over the task of responding to queries from a customer. In this context, we would like the publisher to provide compact proofs of correctness along with answers. In [24], we show how to construct proofs of correctness that are linear in the size of the answer to the query.

9 CONCLUSION

The security of software-controlled systems has become critical to normal everyday life. The timely, effective and correct construction of these systems is an ongoing, difficult challenge faced by software engineers. In this paper, we have outlined some of the important open issues faced by researchers in software engineering and security as we engineer security-critical systems for the next millennium.

REFERENCES

[1] COMMON CRITERIA VERSION 2.1, see `http://csrc.nist.gov/cc`.

[2] Working group 3—Security Criteria. Technical report, International Standards Organization (ISO), Joint Technical Committee 1—Information Technolgy, Subcommittee 27—Security Techniques.

[3] TCSEC: Department of defense trusted computer system evaluation criteria. Dept. of defense standard, Department of Defense, Dec 1985.

[4] M. Abadi and J. Feigenbaum:. Secure circuit evaluation. *Journal of Cryptology*, 2(1), 1990.

[5] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*. IEEE Computer Society, May 1994.

[6] N. M. Amato and M. C. Loui. Checking linked data structures. In *Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing (FTCS)*, 1994.

[7] R. Anderson and M. Kuhn. Tamper resistance—a cautionary note. In *Second Usenix Electronic Commerce Workshop*. USENIX Association, November 1996.

[8] D. Aucsmith. Tamper resistant software: An implementation. In *Hiding Information Hiding, First International Workshop*, volume 1174, pages 317–333. Springer-Verlag, 1996.

[9] D. Bell and L. LaPadula. Secure computer system unified exposition and multics interpretation. Technical Report MTR-2997, MITRE, Bedford, MA, 1975.

[10] M. Bellare. Practice-oriented provable security. In G. D. E. Okamoto and M. Mambo, editors, *Proceedings of First International Workshop on Information Security (ISW 97)*, volume 1396 of *LNCS*. Springer Verlag, 1997.

[11] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Noar. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994.

[12] A. Borgida, S. J. Greenspan, and J. Mylopoulos. Knowledge representation as the basis for requirements specifications. *IEEE Computer*, 18(4):82–91, 1985.

[13] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. Technical Report SRC Research Report 39, Digital Equipment Corporation, Feb. 1989. Revised February 1990.

[14] B. J. Chorley and *et al.* Software protection device. United States Patent 4,634,807, 1987.

[15] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1994.

[16] C. Collberg and C. Thomborson. Software watermarking: Models and dynamic embeddings. In *Symposium on principles of Programming Languages*, 1999.

[17] J. C. Corbett. Constructing compact models of concurrent java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, March 1998.

[18] S. Craver, N. Memon, B. L. Yeo, and M. M. Yeung. Resolving rightful ownership with invisible watermarking techniques: Limitations, attacks and implications. *IEEE Journal on Selected Areas in Communications*, 16(4):573–586, May 1998.

[19] Department of Defense. Trusted Computer System Evaluation Criteria Orange book, 1985.

[20] Z. Dang and R. A. Kemmerer. Using the astral model checker to analyze mobile ip. In *Proceedings of the 21st International Conference on Software Engineering*, pages 133–142, 1999.

[21] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In *Advances in Cryptology—CRYPTO*, 1989.

[22] P. Devanbu, P. Fong, and S. Stubblebine. Techniques for trusted software engineering. In *Proceedings of the 20th International Conference on Software Engineering*, 1998.

[23] P. Devanbu and W. Frakes. Extracting formal domain models from existing code for generative reuse. *ACM Applied Computing Review*, 1997.

[24] P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine. Authentic third-party data publication. (`http://seclab.cs.ucdavis.edu/~devanbu/authdbpub.pdf`), 1999.

[25] P. Devanbu, M. Gertz, and S. Stubblebine. Security for automated, distributed configuration management. In *Proceedings, ICSE 99 Workshop on Software Engineering over the Internet*, 1999.

[26] P. Devanbu and S. Stubblebine. Cryptographic Verification of Test Coverage claims. In *Proceedings, Fifth ACM/SIGSOFT Conference on Foundations of Software Engineering*, 1997.

[27] P. Devanbu and S. Stubblebine. Cryptographic verification of test coverage claims. *IEEE Transactions on Software Engineering*, 1999. Accepted to appear.

[28] P. Devanbu and S. G. Stubblebine. Stack and Queue Integrity on Hostile Platforms. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, California, May 1998.

[29] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings, Second ACM/SIGSOFT Conference on Foundations of Software Engineering*, 1994.

[30] M. B. Dwyer and J. Hatcliff. Slicing software for model construction. In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, 1999.

[31] W. Emmerich. Software engineering for middleware: a roadmap. In A. Finkelstein, editor, *"The Future of Software Engineering", Special Volume published in conjunction with ICSE 2000*, 2000.

[32] G. Engels and L. Groenewegen. Object-oriented modeling: a roadmap. In A. Finkelstein, editor, *"The Future of Software Engineering", Special Volume published in conjunction with ICSE 2000*, 2000.

[33] J. Estublier. Software configuration management: a roadmap. In A. Finkelstein, editor, *"The Future of Software Engineering", Special Volume published in conjunction with ICSE 2000*, 2000.

[34] D. Evans and A. Twyman. Flexible policy-directed code safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.

[35] J. Feigenbaum. Encrypting problem instances, or, can you take advantage ofsomeone without having to trust him. In *Advances in Cryptology—CRYPTO*, 1986.

[36] P. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, August 1988.

[37] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, 1999.

[38] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch, or, why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*. IEEE Computer Society, May 1995.

[39] P. A. Godefroid. Model checking for programming languages using verisoft. In *Proceedings, POPL 97*, 1997.

[40] R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf. An architecture for post-development configuration management in a wide-area network. In *17th International Conference on Distributed Computing Systems*, May 1997.

[41] R. S. Hall, D. Heimbigner, and A. L. Wolf. A cooperative approach to support software deployment using the software dock. In *International Conference on Software Engineering*, May 1999.

[42] M. A. Harrison, W. L. Ruzzo, and J.D. Ullman. Protection in operating systems. *Comunications of the ACM*, 19(5), 1976.

[43] M. J. Harrold. Testing: a roadmap. In A. Finkelstein, editor, *"The Future of Software Engineering", Special Volume published in conjunction with ICSE 2000*, 2000.

[44] M. E. Hellman. Software distribution system. United States Patent 4,658,093, 1987.

[45] J. A. Hoagland, R. Pandey, and K. Levitt. Security policy specification using a graphical approach. Technical Report CS-98-3, University of California, Dept. of Computer Science, Davis, California, July 1998.

[46] M. Hurley and M. Zurko. The ADAGE policy language `http://www.camb.opengroup.org/ri/secweb-/adage/index.htm`.

[47] D. Jackson and M. Rinard. Reasoning and analysis: a roadmap. In A. Finkelstein, editor, *"The Future of Software Engineering", Special Volume published in conjunction with ICSE 2000*, 2000.

[48] D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. In *Proceedings of the 1999 international conference on Software engineering*, Los Angeles, CA, May 1999.

[49] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse : Architecture Process and Organization for Business Success*. Addison Wesley, 1997.

[50] A. K. Jones, R. J. Lipton, and L. Snyder. A linear-time algorithm for deciding subject–object security. In *Proc. of the 17th Annual Foundations of Computer Science*. IEEE Press, 1976.

[51] B. S. Joshi. Computer software security system. United States Patent 4,688,169, 1987.

[52] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in LNCS. Springer-Verlag, 1997.

[53] J. Kilian, F. T. Leighton, L. R. Matheson, T. G. Shamoon, R. E. Tarjan, and F. Zane. Resistance of digital watermarks to collusive attacks. Technical Report TR-585-98, Princeton University, Computer Science Department, July 1998.

[54] S. Kubota. Microprocessor for providing software protection. United States Patent 4,634,807, 1991.

[55] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine specification*. Addison Wesley, Reading, Mass., USA, 1996.

[56] S. Low, N. F. Maxemchuk, and S. Paul. Anonymous credit cards and its collusion analysis. *IEEE Transactions on Networking*, Dec. 1996.

[57] Marimba, Inc. Castanet product family, 1998. `http://www.marimba.com/datasheets/castanet-3_0-ds.html`.

[58] S. M. Matyas and J. Osias. Code protection using cryptography. United States Patent 4,757,534, 1988.

[59] J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley & Sons, 1994.

[60] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[61] A. Monden. A secure keyed program in a network environment. In *Proceedings of the Twentieth International Conference on Software Engineering*, 1998.

[62] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing knowledge about information systems. *ACM Transactions on Office Information Systems*, 8(4):325–362, October 1990.

[63] G. Naumovich, G. S. Avrunin, and L. A. Clarke. Data flow analysis for checking properties of concurrent java programs. In *International Conference on Software Engineering*, 1999.

[64] G. Necula. Proof-carrying code. In *Proceedings of POPL 97*. ACM SIGPLAN, 1997.

[65] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the '98 Conference on Programming Language Design and Implementation*, 1998.

[66] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9), 1994.

[67] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In A. Finkelstein, editor, *"The Future of Software Engineering", Special Volume published in conjunction with ICSE 2000*, 2000.

[68] OMG. The common object request broker architecture (CORBA) `http://www.omg.org/`, 1995.

[69] OMG. The SECURITY service `http://www.omg.org/homepages/secsig`, 1995.

[70] Committee on Application of Digital Instrumentation & Control Systems to Nuclear Power Plant Operations and Safety. *Digital Instrumentation and Control Systems in Nuclear Power Plants—Safety and Reliability Issues–Final Report*. National Academy Press (Board on Energy and Environmental Systems, National Research Council), 1997.

[71] R. Pandey, V. Akella, and P. Devanbu. Support for system evolution through software composition. In *ICSE '98 International Workshop on the Principles of Software Evolution*, 1998.

[72] R. Pandey, R. Olsson, and K. Levitt. Policy-driven runtime support for secure execution of user code in extensible kernels. (`http://seclab.cs.ucdavis.edu/~pandey/ariel.html`).

[73] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, October 1992.

[74] A. Pickholz. Software protection method and apparatus. United States Patent 4,593,353, 1986.

[75] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communication Special Issue on Copyright and Privacy Protection*, 1998.

[76] J. Ó. Ruanaidh, H. Petersen, A. Herrigel, S. Pereira, and T. Pun. Cryptographic copyright protection for digital images based on watermarking techniques. *Theoretical Computer Science*, 226(1–2):117–142, Sept. 1999.

[77] T. Sander and C. F. Tschudin. On software protection via function hiding. In *Information Hiding*, pages 111–123. Springer-Verlag, 1998.

[78] T. S. Souder and S. Mancoridis. A tool for securely integrating legacy systems into a distributed environment. In *Working Conference on Reverse Engineering (WCRE)*, Atlanta, GA, October 1999.

[79] S. Stubblebine, P. Syverson, and D. Goldschlag. Unlinkable serial transactions: Protocols and applications. *ACM Transactions on Information and System Security*, 2(4), November 1999.

[80] L. J. Tolman and A. J. Etstrom. Anti-piracy system using separate storage and alternate execution of selected public and proprietary portions of computer programs. United States Patent 4,646,234, 1987.

[81] V. Ungureanu and N. Minsky. Unified support for heterogeneous security policies in distributed systems. In *7th USENIX Security Symposium*, San Antonio, Texas, 1998.

[82] V. Varadharajan and T. Hardjono. Security model for distributed object framework and its applicability to CORBA. In S. K. Katsikas and D. Gritzalis, editors, *Information Systems Security – Facing the information society of the 21st century, Proceedings of the 12th International Information Security Conference IFIP SEC'96*, pages 452–463, Samos, Greece, May 1996. Chapman & Hall.

[83] J. M. Voas. Certifying off-the-shelf software components. *IEEE Computer*, 31(6), 1998.

[84] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[85] WWW-3C. The World-Wide Web Consortium `http://www.w3c.org/`, 1999.

[86] B. Yee and D. Tygar. Secure coprocessors in electronic commerce applications. In *Proceedings of The First USENIX Workshop on Electronic Commerce*, New York, New York, July 1995.

[87] M. M. Yeung. Digital watermarking. *Communications of the ACM*, 41(7):30–33, July 1998.