High-Confidence Design for Security

Shiu-Kai Chin

July '99 CACM Issue on Information Warfare

The widespread use of networks makes information security a major concern where the underlying network (e.g., the Internet) is assumed to be insecure. Systems with security requirements typically must operate with a high degree of confidence – they must be highly assured.

The task of designing and building secure systems raises a fundamental question, *how do we know with confidence that our designs will behave securely?* Having confidence in a secure system requires having confidence in the following: the strength of the cryptographic algorithms, the correctness of the hardware and software implementations, and knowing the implementation supports a security model.

This article describes methods that establish confidence that implementations meet their specifications and security requirements. These methods are rigorous in nature. They rely on mathematical logic and are accessible to engineering students at the masters level. As is typical in systems engineering, a variety of methods are used depending on what level of design is being addressed.

Security Properties

The collection of people, keys, processes, or machines who send and receive information and who access information resources (e.g., databases, processors, printers, etc.) are called *principals*. Security properties typically deal with the ability of principals to access information or resources. Key security properties include:

- *privacy or confidentiality*: knowing which principals can read data

- *integrity:* detecting corruption of information
- *authentication:* knowing the identity of a principal or the source of information
- *access control:* restricting the use of a resource to privileged principals
- *non-repudiation:* preventing principals from denying their actions
- *availability of service:* guaranteeing authorized principals always have access to services

(The *Handbook of Applied Cryptography* [1] describes each of these properties in detail).

The means for achieving these properties depends on the collection of security mechanisms that supply security services, the correct implementation of these mechanisms, and how the mechanisms are used.

Security Mechanisms

Some common security mechanisms used to achieve the security properties listed above are:

- *encryption:* hiding information in a reversible way, i.e., for message m, and keys K and K^{-1}: $decrypt(K^{-1}, encrypt(K, m)) = m$, such that it is computationally infeasible to determine the keys K and K^{-1}
- *hash functions:* one-way functions that map messages of arbitrary length to a fixed number of bits where for any hash value $hash(m_1)$, it is computationally infeasible to find another message m_2 such that $hash(m_1) = hash(m_2)$, and find m_1 given $hash(m_1)$
- *digital signatures:* a means to associate information with principals
- *certification:* vouching for the accuracy of information by a recognized authority
- *authorization:* granting permission to do something or act in a particular role

Encryption is used to meet confidentiality requirements. There are two kinds of encryption—symmetric (or secret key) encryption, and asymmetric (or public key) encryption. In symmetric key cryptography, the same key is used as a parameter of a cryptographic function such as DES (Data Encryption Standard) [1] to encrypt or decrypt data. DES is an instance of a Feistel cipher, [1]. Feistel ciphers have a repetitive structure where each stage or *round* of calculations involves (1) splitting the input into two halves (L_{i-1}, R_{i-1}), (2) passing the right input R_{i-1} to the left output L_i, and (3) computing the right output R_i by taking the bitwise exclusive-or of the left input L_{i-1} with a function of the right input R_{i-1} and subkey K_i denoted by $F(R_{i-1}, K_i)$. The function F is a complex function consisting of permuting the order of bits in a block of bits and substituting one block of bits for another. DES uses sixteen rounds with specific functions defining substitutions, permutations, and subkey generation.

Asymmetric key cryptography uses two keys. A *public key* is available to everyone. A *private key* is known only by a single principal. Thus, a private key in some sense uniquely identifies a principal if that key is not compromised.

An example of a public key encryption algorithm is RSA by Rivest, Shamir, and Adleman, [11]. Encryption and decryption are done by exponentiation. If M is the value of a message block, and (e, n) is a public key where e and n are integers, then the encrypted value of M is $M^e \ mod \ n$, i.e., M raised to the power e modulo n. (Recall that $2^3 \ mod \ 7 = 8 \ mod \ 7 = 1 \ mod \ 7$). A decryption key (d, n) is the inverse of an encryption key (e, n), i.e., $(M^e)^d \ mod \ n = M$. The original message is retrieved when the encrypted value is raised to the power d.

Digital signatures are used to meet the needs for integrity, authentication, access control, and non-repudiation. Figure 1 shows one method for accomplishing both encryption for confidentiality and generating digital signatures. A signed and encrypted message *(ciphertext,signature)*, consists of *ciphertext*—the plaintext encrypted with a data encryption key (DEK), and *signature*—the message hash encrypted with a signature key. Recalling the properties of hash functions and public key encryption, the hash value of the plaintext uniquely identifies a message and the signature key uniquely identifies the sender. DES can be used to encrypt the plaintext and RSA can be used to sign the hash value of the plaintext.

Given the pair *(ciphertext,signature)*, the plaintext is extracted by decrypting *ciphertext* using the inverse of the DEK, and the integrity and au-

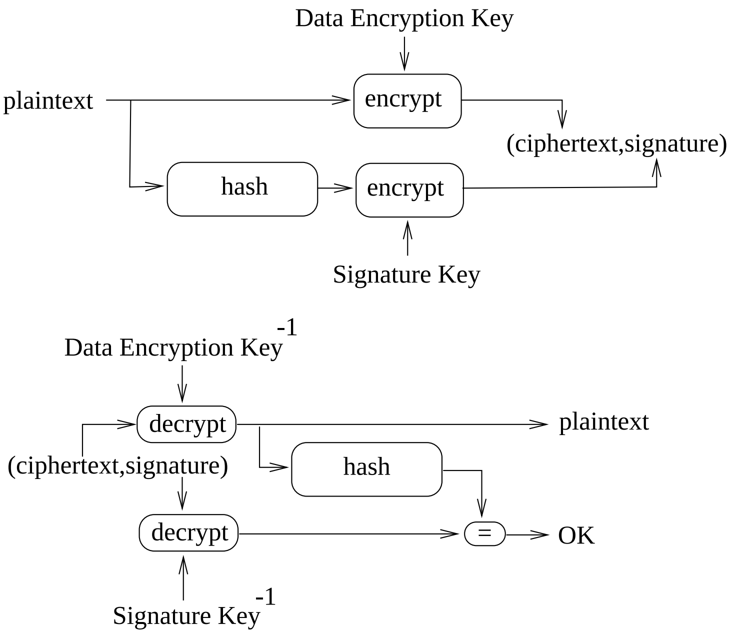

Figure 1: Channels with Encryption and Digital Signatures

thenticity checked by comparing the hash of the decrypted message with the decrypted signature received with the encrypted message. If the message arrives intact, then the owner of the private key (assuming that the private key of the sender has not been compromised) is the only one who could have sent the message.

Establishing Confidence

What gives us confidence in the security of a system and how do we go about establishing that confidence? Considering the security properties and mechanisms described in the previous sections as exemplified by Figure 1, the following questions come to mind.

1. Are the cryptoalgorithms and hash functions good enough?

2. Are the mechanisms, such as those in Figure 1, correctly implemented in hardware and software?

3. Are the mechanisms for handling requests (such as access to resources) properly used to maintain security and trust?

Each of these questions is addressed in the following three sections.

Confidence in Algorithms

Confidence in the underlying cryptographic algorithms requires a detailed analysis of the algorithms themselves using number theory, statistics, and complexity theory. This is to substantiate the expectation that the algorithms are resistant to cryptanalysis used by attackers. An important part of having confidence in an algorithm is whether it has been thoroughly examined by analysts. In most cases, implementers do not create new cryptoalgorithms nor are responsible for assessing their cryptographic strength. Rather, most system designers use cryptoalgorithms as components and are responsible for their correct implementation and use. This moves us to the questions of correct implementation and maintaining trust and security.

Confidence in Implementations

Confidence in the implementation of security mechanisms requires knowing the implementations are correct. Besides the normal use of testing and simulation to check correctness for specific test cases, formal verification using mathematical logic is possible and becoming more practical.

Temporal logic and reasoning tools based on model checking [9], have verified properties of microprocessors and controllers. As model checkers explore the properties of state machines, they are particularly appropriate for verifying properties of controllers and control paths. Example properties include checking for the absence of deadlock and the reachability of certain states. Freedom from deadlock is expressed as "for all states it is possible to make a transition to another state." Eiriksson in [6] used the Cadence Berkeley Labs SMV (Symbolic Model Verifier) model checker [9] to verify the correctness of a one-million gate application specific integrated circuit (ASIC) implementing a cache coherence protocol.

Higher-order logic, logic that allows functions to be parameters of functions, is well-suited for describing and verifying data paths and their specifications. The ability to have functions as parameters means security mechanisms are specified, described, and verified without having to specify the specific hash or encryption function used in a particular instance. The correctness of the mechanisms is established based on the general properties of hash and encryption functions.

For example, in the integrity checking portion of the mechanism in Figure 1, the pair *(ciphertext,signature)* is deemed *OK* if there is a match between the hash of the decrypted ciphertext and the decrypted signature.

The properties of the integrity check depend on the properties of hash and decryption functions. If the decryption functions invert the corresponding encryption functions, then the integrity check is true if and only if the received message is equivalent to the message that was sent, where equivalence is defined by having the same hash value. Specification and verification of integrity checking at the abstract level and lower levels using specific message structures is possible using higher-order logic [7] and automated tools [4].

To give a sense of how lower-level implementation descriptions are related to higher-level behavioral specifications, consider the addition operation. What must be proved is the implementation at the bit level corresponds to correct arithmetic-level behavior given a particular mapping from bit vectors to integers (e.g., consider the unsigned interpretation of bit vectors).

For example, say we have a 4-bit adder $Add(In_1, In_2, Sum)$ that adds In_1 and In_2 to get Sum. An example at the bit level is $0011+0101 = 1000$. At the arithmetic level, if we use an unsigned interpretation of 4-bit words, we have $3 + 5 = 8$, where $value(0011) = 3$, $value(0101) = 5$, and $value(1000) = 8$. If all input and output bit vectors allowable by adder $Add(In_1, In_2, Sum)$ satisfy the relationship $value(In_1) + value(In_2) = value(Sum)$, then Add satisfies the arithmetic specification of addition for unsigned numbers.

Relating low-level implementations to higher-level specifications (such as gate-level implementations to arithmetic behavior) relates models or descriptions at differing levels of abstraction. This is done at a variety of levels. For example, Tahar and Kumar have verified, using higher-order logic, that the pipelined DLX RISC processor implementation satisfies its instruction set architecture as seen by users, [12].

While the above examples have a hardware flavor to them, software verification is becoming practical as well, especially in the area of model checking of software. Examples using model checking for software include checking a telephone switching application using Lucent Technology's VeriSoft tool [10], and checking safety-critical systems using the Naval Research Laboratory's SCR (Software Cost Reduction) tools [3].

Confidence in Security

The cryptographic algorithms and security mechanisms described in the previous sections establish that messages and data were encrypted and signed using a key K. Consider the case where the mechanisms in Figure 1 are implemented correctly and the pair *(ciphertext,signature)* is deemed *OK*. If *ciphertext* decrypts to statement s and the key used to sign *hash(s)* is K, then we can say $K\ says\ s$

A critical question is *whom does K represent?* This question raises a number of issues that extend beyond cryptographic strength of algorithms and correct implementation of hardware and software. These questions include:

1. What is the basis for associating keys with principals?

2. Who are the recognized authorities that vouch for principals, their information, and their privileges?

3. How do we decide if we should honor a request to access a resource such as a file?

The above questions point to a need to reason about trust relationships, authentication, and authority. This is particularly important for distributed systems where system components are geographically distant.

Generally, the above questions are handled as follows:

1. Keys are associated with principals using *certificates*. Certificates document the association of public keys with principals. They are unforgeable as they are cryptographically signed by the certification authority issuing the certificate. An example is the X.509 certificate standard, [8].

2. Certain principals and their keys are explicitly recognized as having the authority to certify the validity of associating keys with principals.

3. Access to resources is determined by consulting an access control list after authenticating who is making a request and in what role they are making the request.

The challenge is to have a means to reason consistently about who is making requests. One way to address this need is to create specialized logical systems to reason about trust, authentication, and authority. These systems have a specialized language and inference rules for reasoning about principals and who they speak for. The advantage of having such a system is it maintains logical consistency when making decisions on authentication and access control. Of course, the issue of whether the inference rules themselves make sense for the application at hand must be examined closely.

One example is the logic created by Lampson, Abadi, and Burrows to reason about authentication in distributed systems, [2]. In this system, all communication is done using *channels* where a channel is an implementation of what is shown in Figure 1. When a statement is decrypted and its integrity is verified, the channel has made the statement K *says* s.

Certificates are signed statements saying that a key *speaks for* a principal. (Note: in this logic, principals are people, processes, or keys.) A certificate has the form K_{CA} *says (*K_A *speaks for* A*)*, where CA is a certificate authority and A is a principal whose public key is K_A. This formula is interpreted as the statement K_A *speaks for* A is signed by K_{CA}. If it is accepted that K_{CA} *speaks for* CA, (i.e., K_{CA} is CA's key), where CA is a recognized certification authority, then we conclude that K_A *speaks for* A, i.e., K_A is A's key. As a simple illustration of how the logic of Lampson, Abadi, and Burrows is used, we go through the chain of reasoning that enables principal B to conclude K_A is A's key.

The logic has (among others) the following inference rules where P_1 and P_2 are any principals and s is any statement:

1. If *(*P_1 *speaks for* P_2*)* and *(*P_1 *says* s*)*, then P_2 *says* s
2. If P_1 *says (*P_2 *speaks for* P_1*)*, then P_2 *speaks for* P_1

Assume that principal B has received the certificate K_{CA} *says (*K_A *speaks for* A*)*. If B assumes (1) CA *speaks for Anyone* and (2) K_{CA} *speaks for* CA, then B can conclude K_A *speaks for* A using the above two rules, the two assumptions, and the certificate.

The chain of reasoning is as follows. Using the first rule, the second assumption, and the received certificate, B concludes CA *says (*K_A *speaks for* A*)*. Using the first assumption, B concludes CA *speaks for* A, and using the first rule again concludes that A *says (*K_A *speaks for* A*)*. Finally, using the second rule, B determines K_A *speaks for* A. If our authentication and access control mechanisms act consistently with the inference rules, then we can have confidence in the security of the system, assuming the security model as reflected by the inference rules is appropriate for the application.

The logic of Lampson, Abadi, and Burrows contains many rules for reasoning about delegation of authority, principals acting in roles, and authentication with multiple certification authorities. The logic was used to design an authentication system as part of Digital Equipment Corporation's Taos operating system, [5]. Taos featured a distributed file system that authenticated principals and their privileges.

Conclusions

Having confidence that a system is secure requires having confidence in the cryptographic algorithms, confidence that the security mechanisms are implemented properly, and that the mechanisms themselves support a security model. While the mathematical analysis necessary to reason about the cryptographic strength of algorithms may be beyond most system engineers, the type of mathematical logic used to analyze the correctness of implementations and trust relationships is not. In fact, these reasoning methods are taught to engineering students at the masters level. Formal reasoning and tools are being adopted by industry, particularly by integrated circuit manufacturers.

This paper was not comprehensive in its treatment of all security concerns and formal reasoning techniques. For example, the analysis of covert channels was not addressed. Nevertheless, the methods discussed in this paper do cover a wide variety of concerns that must be addressed to establish confidence. These reasoning methods and tools can and should be part of the repertoire of designers of secure systems.

References

[1] Alfred J. Menzies, Paul C. von Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, New York, 1996.

[2] Butler Lampson, Martin Abadi, Michael Burroughs, and Edward Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[3] C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj. SCR: A Toolset for Specifying and Analyzing Software Requirements. In *Proceedings 10*

Annual Conference on Computer-Aided Verification (CAV'98), Vancouver Canada, 1998.

[4] Dan Zhou and Shiu-Kai Chin. Verifying Privacy Enhanced Mail Functions with Higher Order Logic. *DIMACS Series in Discrete Mathematics*, 38:11–20, 1998.

[5] Edward Wobber, Martin Abadi, and Mike Burroughs Butler Lampson. Authentication in the TAOS Operating System. Report 117, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, December 1993.

[6] Asgeir Th. Eiriksson. The Formal Design of a 1M-gate ASICs. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design*, number 1522 in Lecture Notes in Computer Science, pages 49 – 63. Springer-Verlag, 1998.

[7] M.J.C. Gordon. A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subramanyam, editors, *VLSI specification, verification and synthesis*. Kluwer, 1987.

[8] ITU-T. Information Processing Systems — Open Systems Interconnection — The Directory Authentication Framework. ITU-T Recommendation X.509. 11/93. Also ISO/IEC 9594-8.

[9] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.

[10] Patrice Godefroid, Robert S. Hanmer, and Lalita Jategaonkar Jagadeesan. Systematic Software Testing using VeriSoft: An Analysis of the 4ESS Heart-Beat Monitor. *Bell Labs Technical Journal*, 3(2), April – June 1998.

[11] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications ACM*, 21(2):120 – 126, February 1978.

[12] Sofiene Tahar and Ramayya Kumar. A Practical Methodology for the Formal Verification of RISC Processors. *Formal Methods in System Design*, 13(2):159 – 225, 1998.