Remote Auditing of Software Outputs Using a Trusted Coprocessor

Bruce Schneier John Kelsey

Counterpane Systems, 101 E Minnehaha Parkway, Minneapolis, MN 55419
{schneier,kelsey}@counterpane.com

Abstract

A cryptographic coprocessor is described for certifying outcomes of software programs. The system for certifying and authenticating outputs allows a third party who trusts the secure components of the system to verify that a specified program actually executed and produced a claimed output.

keywords. *Key words:* Authentication, digital signatures, secure coprocessor, smart cards

1 Introduction

We present an application of digital signatures [2]. Through a cryptographic coprocessor [4,13,11] — here called an “Authenticator” — software can certify particular outputs. Software can use this capability to allow the Authenticator to certify that some specified outputs or outcomes of the software have actually been achieved. These protocols can be implemented on a variety of hardware designs using any of several digital signature algorithms: RSA [9], ElGamal [3], DSA [8], etc.

Normally, if a person claims that he has performed some task or achieved some result with a software program it can be difficult to verify that this actually occurred. The output itself may be displayed (printed or photographed), but this is not always reliable evidence of the process that was followed to produce that result. Our Authenticator can produce a digitally signed statement which securely and reliably attests to the actual process of the program. The Authenticator has access to the internal operation of the software (as will be described in detail), and its ability to produce secure and untamperable output allows this authentication functionality.

An application of this capability would be the certification of a high score in some game software. Perhaps the manufacturer wishes to set up a promotion where prizes are offered to people who complete a game or achieve a specified high score. Such promotions are difficult to offer at present due to the difficulty of fairly verifying claims of success. The Software Authenticator would allow those players who have actually met the goals to certify their results and win the prize, while preventing cheaters from sending in doctored output or copying output from true winners and claiming it as their own.

The same system has another, independent, capability: software metering. The Authenticator can be used to check aspects of a program's functioning related to the amount of time spent or number of uses. As with an electric meter, the Authenticator records information about how much the software has been used, in a form which the end user cannot tamper with. At some regular interval (probably every month) the Authenticator is read by a remote connection to a central computing service. The central computer then bills the user for his actual usage of the software during the month just as with other utilities. Several variations on this general scheme will be described using the same overall system configuration.

2 System Configuration

There are three main hardware components to the authentication system: the Computer, the Authenticator, and the Data Source. Generally, these will be connected in one of the following two configurations:

$$\text{Authenticator} \longleftrightarrow \text{Computer} \longleftrightarrow \text{Data Source}$$

$$\text{Computer} \longleftrightarrow \text{Authenticator} \longleftrightarrow \text{Data Source}$$

The Computer is the main computing unit owned by the end user, which runs the main part of the software programs he uses. It may be a general purpose computer such as a PC, or it may be more specialized, such as a dedicated game playing unit or TV-based computer entertainment system. It will have some input and output capabilities, typically including a video/sound display and possibly a printer for output, with input ranging from simple joystick systems to full keyboard and/or video capability. The Computer will include RAM and usually ROM memory, and may also have non-volatile memory such as a disk drive or Flash RAM. For most applications the Computer must have a modem or other network connection to allow it to communicate with the Central Computer.

The Authenticator is a small piece of hardware enclosed in a tamper-resistant casing which includes CPU and memory. The Authenticator is a computer in itself, but generally a less powerful one than the main Computer in the system. It will have much less RAM and its CPU will probably be less powerful. The Authenticator must include some non-volatile memory as described above, some RAM, and some ROM which holds the program which implements its basic functionality. The non-volatile RAM of the Authenticator will hold the cryptographic keys used for communication and authentication. The

Authenticator will also require a hardware random-number source to be used for initializing keys.

This paper assumes that a software program can be divided into two components–one running on the Authenticator and the other running on the Computer–such that the Authenticator can determine the output of the entire software program.

The Data Source represents the place from which software programs which will be run on the metering system come. Depending on the configuration, this may be a local disk drive or CD-ROM, a game cartridge, or a remote computer connected by telephone line or computer network connection.

One system component which is not shown in the diagrams above is the centralized computer system which communicates with the Authenticator (possibly via the Computer). This server will be referred to as the Central Computer (or CC). A communications link must exist for the Authenticator to talk to the Central Computer at regular intervals in order to transmit authenticated output information. In many ways this is similar to Chaum's electronic database with an "observer" chip [1].

As shown in the diagrams above, software programs flow from the Data Source into the Computer/Authenticator system. This flow is essentially one directional, although in some circumstances (such as if the Data Source is a disk drive or a computer on the Internet) data may flow in the opposite direction during program execution or at other times. (Another example of this model is a satellite-to-PC system: lots of continuous bandwidth down, and a small trickle up via a dial-up line.) But for the unique features of the metering and authentication system, the Data Source can be thought of as a read-only source of programs to execute.

In addition, the diagrams both show a two-way link between Authenticator and Computer. This link is active during most phases of the protocol, and although the amount of data to be sent across the link is normally not large, it is necessary during software execution that the link latency (the time needed to get a short message across) is small. The Computer and the Authenticator work in close cooperation during program execution and so their communication must not introduce noticeable delays.

In the protocol descriptions and other discussion below, the Authenticator will be referred to as though it is capable of performing actions which only the Computer can do, such as accessing the Data Source in the first diagram above, or storing data in the Computer's non-volatile memory. It is understood that in such situations it is actually a cooperation between the Computer and the Authenticator which occurs, with the Computer performing these functions at the request of the Authenticator.

The two diagrams differ in whether the Data Source connects to the computer directly, as shown in the first diagram, or whether it connects via the intermediary of the Authenticator, as shown in the second. Example configurations for the first case might include a general-purpose computer, with the Authenticator in the form of a smart card, PCMCIA card (now called a PC Card), parallel- or serial-port dongle, or internal expansion card. A dedicated videogame machine, like a Sega Saturn, which had an expansion port or slot where the Authenticator could plug in would also fit this configuration.

Examples of the second configuration would include systems where the Authenticator is built into a special device to access the Data Source. For example, the Authenticator could be built into a custom CD-ROM drive which would then be able to use special CD's customized for the metering and authentication system. Alternatively, the Data Source could be a remote computer reached via the Internet or some other computer network, and the Authenticator would be built into a special modem used to access the data. Another example of this configuration would build the Authenticator into a pass-through box, like a Game Genie, which would plug into a cartridge slot on a game machine and allow cartridges to plug into the Authenticator and be the Data Source.

The functional differences between the two configurations shown are not significant. The choice will usually be dictated by other considerations in terms of the expansion capabilities of the system as to where the best place is to attach the Authenticator. If all else is equal, there may be some small advantage to the second configuration, where the Data Source passes through the Authenticator en route to the Computer. One option for the use of the system is for the Data Source to store the whole software program in encrypted form, both the secure part which will run on the Authenticator and the insecure part which will run on the computer. If this option is used, then passing the data through the Authenticator will allow it to be conveniently decrypted as it is loaded into the memory of the Computer. With the other configuration this is still possible, but the data would have to be transferred from the Computer to the Authenticator and back in order to be decrypted, requiring more data transfers and so taking more time to load the program into memory. As will be discussed below, the security increase provided by this option is only marginal so it would not normally be the determining factor in selecting which configuration option to use.

3 Applications

Certifying High Scores: One application of the output authentication system is high score verification in the context of a dedicated game system, or entertainment software on a general purpose computer. This could be used as part of a contest, or continual high score rankings could be maintained and people could gain bragging rights by seeing how they ranked against other players around the world. This could open up possibilities where some of the characteristics of online games, specifically the element of competition against other players, would be available in the context of home game systems. Action, strategy, and puzzle games could all be made more fun and exciting if successful players could demonstrate their achievements publicly.

Metering: If the authenticated output were how long an application was running or how much data an application displayed or processed, then the system can be used as a software meter [12]. This could be used as a "pay-for-play" arcade system for home video game machines, a "pay-for-use" or "pay-for-feature" system for personal computers, or a "pay-per-page" system for database programs. The meter could be installed on the much-hyped Internet terminal, allowing users to pay only for the time they use the machine; enhancements could even allow metering of World Wide Web pages.

Distributed Key Search: Another possible application of output certification could arise in the context of distributed computing. Several research groups are working on systems to harness the massive collective computational power of the Internet and apply it to hard problems. One example which has already had some success is in factoring large numbers. Another area which has been explored is exhaustive search for cryptographic keys. If these research systems could be commercialized so that people were paid for letting their computers be used for problems like these, it could make available a huge new source of computer cycles. But one problem with these ideas is the issue of verifying that each participant actually performed the computational work he had agreed to. Some kinds of applications are self-checking, such as graphics rendering, but others, such as the key search or factoring examples, may legitimately end up with no successful outputs. In those cases a user could cheat by simultaneously allocating his computers to multiple projects and report no results to all of them, collecting more pay than he is entitled to. The authenticated output system can fix this problem, by making sure that even if the output is null, that that is the legitimate result of running the software. People would be required to present authenticated output from their program runs if they expect to get paid for their compute cycles. This technology can reduce cheating and thereby bring the whole approach closer to economic feasibility.

4 System Overview

4.1 Signature and Trust Issues

One issue relating to any form of authenticated output is what the trust relationships are between the Authenticator and those who view the output. In our system, the output is authenticated by the Authenticator. This means that those who trust the Authenticator and are in a position to verify its signatures are the ones who will be able to trust and accept the authentication. This will include most particularly the Central Computer, who shares a key with the Authenticator and who, if symmetric cryptography is used for the authentication, is the only entity (other than the Authenticator itself) that can verify the signatures.

In some applications the authenticated output needs to be verified by other parties. The CC and its affiliated organizations can verify that data is accurately certified by an Authenticator which is part of the authentication system, and then provide additional public-key signatures on that data. This second-order certification can use a widely known and respected public key and is suitable for wide distribution and acceptance.

4.2 Communications

Unlike software metering, output authentication can be designed to have very modest data transmission requirements between the Authenticator and the CC. This raises the possibility that it could be used even in a low-bandwidth environment where no direct electronic link exists between the Authenticator and the CC, but in which the information is displayed on the screen by the Authenticator and the user manually transfers it to the CC, say by calling the CC on the telephone and entering the data on his telephone keypad. Similarly data could be returned from CC to Authenticator by alphanumeric data being provided over the phone from the CC (via voice synthesis) and entered into the Authenticator through the regular input device, and keyboard or even a simple joystick interface.

4.3 Authenticator/Computer Interface

In order for the Authenticator to be in a position to authenticate the output of the program, it must be able to know that the output actually did occur. This means that the Authenticator must be intimately involved in the calculation of the output, such that even if the part of the program running in the insecure Computer is tampered with, the Authenticator is able to know whether a given output actually occurred or not. This may require a larger fraction of the program execution to occur on the Authenticator than in the case of the metered software application.

5 Protocols

Protocols will be described for two basic cases, the first being an electronic connection between the Authenticator and the Central Computer, and the second being the simple case described above where all such communication is via the human user of the system. This second system will be referred to as the "low bandwidth" case, and probably consists of a human on the telephone, entering characters read to him over the telephone into the computer, and typing digits from the computer screen into the telephone.

1. Initialization of the Authenticator. This is used when the Authenticator is first activated, to generate keys and communicate them to the CC. For the low-bandwidth case it may be preferable to have the Authenticator's unique secret key calculated at the factory and programmed into its ROM at manufacturing time, recorded in the CC's database.
2. Adding a New Program. This protocol is used when the user has acquired a new software program which requires information from the CC in order to run. As with the metering application, some programs may be runnable without any new information from the CC but others will require keying information to be acquired. The low-bandwidth case will require the use of programs which do not require interaction with the CC and so this protocol will not be used in that case.
3. Starting Authenticated Software. This protocol involves the Authenticator, Computer, and Data Source. It describes how these components interact at the time an authenticated software program is loaded and execution begun.
4. Authenticate Output. This is the main protocol of the system, used when a program produces some output which the user wants to have authenticated.

Each program is generally encrypted using a different key, unique to that program. There are some advantages to using a single key to encrypt a large number of programs, but there is some security risk in doing so, since that key would be more valuable than others and if it were somehow exposed the set of programs which use it would all become insecure. So it is expected that in most cases programs will be encrypted using a unique key.

In order to begin running such a program, then, it will be necessary for the metering system to acquire the key for that program. This will be done as part of the "Adding a New Program" protocol.

In some cases the convenience of being able to run a program for the first time without any interaction with the CC will be important. In that case the Authenticator must already have a key for that program. This could be handled by having all such programs be encrypted with the same key (or possibly all programs from a given manufacturer encrypted with the same key), and having the CC send that key to all Authenticators during their initialization. At a slight cost in memory the system can be made somewhat more secure by using several different keys for the programs, with the key being chosen based on the Software ID (ID_S), a unique ID associated with each program (discussed in more detail below). In this way the keys will be shared approximately equally across all such immediately-runnable programs, reducing the value of each individual key of this type. In the low-bandwidth implementation it is expected that all programs will be of this type due to the difficulty of acquiring a key for each different program without an electronic connection to the CC.

In the resulting system all programs are of one of two types. They can be immediately-runnable, and hence encrypted using one of the shared keys; or they can requiring interaction with CC before first run, in which case the program is encrypted with a unique key.

5.1 Data Structures

There are many important pieces of data associated with the Authenticator.

Authenticator Keys: The Authenticator requires several keys in order to perform its varied functions, including communicating securely with the Central Computer. Keys are of three types: "secret" keys are those used with conventional cryptosystems such as DES [7]; "private" and "public" keys are those used with public key cryptosystems such as RSA [9].

1. Authenticator's Secret Key (SK_A): Also known to CC. This key is used for secret and authenticated communication between the CC and the Authenticator, possibly via the Computer and/or an insecure communications link. SK_A is normally generated by the Authenticator during the initialization phase, and is then transmitted to the CC in a message secured by PK_{CC}. Alternatively it may be programmed into the Authenticator's ROM at manufacturing time, and recorded into the CC's database at that time. This will be necessary for the low-bandwidth authentication application.
2. PK_{CC}: Central Computer's public key, known to Authenticator. This key is used for initial communications between the Authenticator and the CC before SK_A is created. It is burned into ROM at manufacturing time.
3. SK_{IR}: Secret key for immediately-runnable programs. As described above, it may be desirable to support a class of programs which can be run immediately upon acquisition, without running the Adding a New Program protocol. (This is especially necessary for the low-bandwidth case.) For this to be possible the Authenticator must already store the key for such a program. All such programs share a special ID_S, and the Authenticator will recognize that ID and use the SK_{IR} key to decrypt the program, as described in the Using Authenticated Software protocol. As mentioned above, a variation on this idea would define several ID_S's of the immediately-runnable class, each of which would be associated with a different SK_{IR} key.
4. ID_A: An identification number unique to each Authenticator, burned into ROM at manufacturing time
5. $Table[Software, Key]$: This table has a list of ID_S and $(Software, Key)$ pairs. Each pair contains the key which will be needed to decrypt the encrypted portions of the software with the specified ID_S.

There are also data structures associated with each piece of metered software that comes from the Data Source. Each piece of authenticatable software from the Data Source is divided into three parts. The Software Control Block has information about the software which identifies it. The executable software itself occupies the two remaining parts. Part of the software is designed to run securely on the Authenticator, while part is designed to run in the insecure environment of the Computer.

1. Software Control Block: The Software Control Block has information about the software which will be used by the metering system to run it. The SCB is signed by the private key of the CC, and the Authenticator checks the signature when the software is loaded. SCB fields include the ID_S: This is a unique number identifying this piece of meterable software. Every piece of software and every revision of a software item have unique ID_S's. As discussed in the context of the metering application, there are two general kinds of ID_S's, "program" and "component," distinguishable by their high order bits. Program ID_S's are used to refer to programs as a whole, while Component ID_S's refer to specific features of a program. Only Program ID_S's are necessary for the authentication application.
2. Insecure Software Component: The Insecure Software Component is the bulk of the software program and runs on the Computer. It may be stored in encrypted form in the Data Source, in which case the Authenticator will be responsible for decrypting it at the time the program is loaded into memory. If this decryption step will add unacceptable delay to program loading, this insecure component can be stored unencrypted at only a slight loss of security. Since the memory of the Computer is insecure by definition, a determined attacker can gain access to the plaintext of the Insecure Software Component in any case. So the additional security added by storing it in secure form is limited in value.
3. Secure Software Component: The Secure Software Component runs on the Authenticator itself. It is stored in encrypted form in the Data Source and must be decrypted by the Authenticator as the program is loaded into memory. As will be described below, the Secure Software Component contains software which implements selected but crucial functionality on which the larger body of software in the Insecure Software Component depends. The encryption of the Secure Software Component is the primary feature by which the overall security of the system is maintained. It prevents attackers from replacing this component with software which will authenticate outputs which did not occur.

5.2 Encryption of messages

Where electronic communication is used, all messages between the CC and the Authenticator are encrypted and authenticated. Either public-key or symmetric encryption can be used, although symmetric encryption appears to provide sufficient security for most of the protocols. The encryption used is assumed to be a strong, modern cipher with key sizes in the range of 64 to 128 bits. Examples include IDEA [6], or Blowfish [10]. Public key encryption would most commonly use RSA [9].

Once initialization is complete, CC and Authenticator share SK_A, which can be used with a conventional encryption system to provide for both encryption and authentication. Because the Authenticator has limited access to sources of entropy, and because the total volume of data to be communicated between Authenticator and CC is small, a few hundred bytes per month in typical usage, using SK_A as the key for all communications between the two systems should provide adequate security for this application. In this configuration, messages from the Authenticator are preceded by sending ID_A (a unique identifier specific to the Authenticator and burned into its ROM at manufacturing time) in the clear, allowing the CC to lookup the encryption key used, followed by the message itself encrypted with SK_A. Responses from CC are sent encrypted with SK_A.

All communications between CC and Authenticator are initiated by the Authenticator, with the CC acting as a server. As described above, it is expected that the user will actually initiate such communications rather than having the Authenticator spontaneously issue requests. Messages sent by the Authenticator will include a sequence number which will increment each time a message is sent in that direction. Reply messages from the CC will include that same sequence number. This will allow both sides to detect message replay attacks, in which messages are captured and then replayed at a later time in order to disrupt the protocols.

The packet formats shown below do not include encryption headers or the account and sequence numbers, which are included as described above except where indicated. Each packet begins with an unique identifier value describing the kind of packet it is, and is followed by data as described below.

For the low-bandwidth case, the protocols used are more modest in their communication requirements, and no implicit sequence numbers or encryption are used other than those explicitly called out.

5.3 Initialization of the Authenticator

For the low-bandwidth case, no special protocol is needed at initialization time. SK_A is programmed into ROM at manufacturing time like ID_A and PK_{CC}. This protocol is only used in the case of electronic communication between Authenticator and CC.

This is used when the Authenticator is first activated, to generate keys and communicate them to the CC. Note that at that time the Authenticator has access to PK_{CC}, the Central Computer's public key, and ID_A, its own unique ID. We assume that the Authenticator has access to a good source of random numbers via a hardware random number generator.

Unlike other protocols, these packets are not implicitly encrypted with the keys shared between CC and the Authenticator. Instead, the encryption used is explicitly identified at each step of this protocol.

1. Authenticator generates SK_A, the random key which will be used for communication between itself and the CC.
2. Authenticator creates an Initialization Message block of the following

format:

Initialization Message

ID_A

Current date and time

SK_A

3. Authenticator encrypts the Initialization Message block with SK_A, then encrypts SK_A using PK_{CC} and sends both blocks to CC.
4. CC recovers first SK_A, then the Initialization Message block. It verifies that date and time are approximately current, and records the new ID_A, checking that it has not been used before. It remembers SK_A and associates that value with ID_A.
5. CC creates an Initialization Message Response block of the following form:

Initialization Message Response

6. CC encrypts the Initialization Message Response block under SK_A and sends it back to the Authenticator.
7. Authenticator decrypts and verifies the IM Response block.

5.4 Adding a New Program

This protocol is used when the user has acquired one or more new software programs which require information from the CC in order to run. All messages in this protocol are sent protected by encryption and sequence numbers as described above. In the low-bandwidth case all programs are of the immediately-runnable type and so this protocol is not used in that case.

1. Authenticator reads new programs' Program Control Block(s) from Data Source, and extracts ID_S for (each) program.
2. Authenticator creates a New Program Message of the following format:

New Program Message

Number of programs requested

ID_S

ID_S

...

3. Authenticator transmits the New Program Message to the CC, encrypted with SK_A.

4. CC looks up the ID_S's for which keys are requested to determine the keys needed to decrypt those programs.
5. CC creates a New Program Message Response of the following format:

    New Program Message Response

    Number of programs

    ID_S, Key

    ID_S, Key

    ID_S, Key

    ...

6. CC securely transmits the New Program Message Response to the Authenticator.
7. Authenticator records the key information for each software program in its $Table[Software, Key]$ structure.

5.5 Starting Authenticated Software

This protocol describes how the Authenticator, Computer, and Data Source interact at the time an authenticated software program is loaded and execution begun. As described above, the software which comes from the Data Source contains a Software Control Block and two executable components, an insecure component which executes on the Computer and a secure component which executes on the Authenticator. At least the secure component is encrypted, and the insecure component may be encrypted as well. Note that in the low-bandwidth case the software will be immediately-runnable.

1. Authenticator reads the Software Control Block from the Data Source and extracts the ID_S for the software.
2. Authenticator determines whether the required key is available to decrypt the program. The key will be found either by looking up the ID_S from the Software Control Block in the $Table[Software, Key]$, or else will be SK_{IR} for immediately-runnable programs (recognized by their ID_S). If the key is not available the Authenticator displays a message informing the user that he needs to add the new program to the current list of software and enable the communication with CC to acquire the needed keys, and the protocol terminates.
3. Authenticator and Computer then read and decrypt the Insecure and Secure Software Components from the Data Source. As noted above, the Secure Software Component will always be encrypted, and the Insecure Software Component may or may not be encrypted, depending on design tradeoffs.
4. Authenticator and Computer then transfer control to the newly read soft-

ware components, Authenticator running the Secure Software Component and Computer running the Insecure Software Component.

5.6 Authenticate Output

This is the principle protocol of the system, used when a program produces some output which the user wants to have authenticated. We assume that the division of software between the secure and insecure components is such that the Authenticator can in fact determine that the specified output actually occurred. For the electronic communication case it works as follows.

1. Authenticator creates an Authenticated Output message of the following form:

   Authenticated Output

   ID_S

   Null terminated text string describing output

2. Authenticator sends the Authenticated Output message to CC, encrypted as usual with SK_A.
3. CC validates that the message correctly decrypts using SK_A and accepts output on that basis. As described above CC may then re-authenticate the output under its own PK_{CC} or perform whatever other actions are appropriate.
4. CC Returns an Authentication Output Response block to confirm that it has accepted the authenticated output. The form is:

   Authenticated Output Response

   ID_S

   Null terminated text string from CC with implementation-specific response.

5. Authenticator displays response from CC.

In the low-bandwidth case a different protocol is used, one suitable to the limited communications bandwidth available if a person is manually transferring the data between the Authenticator and CC via a telephone connection:

1. Authenticator displays program output on the screen, along with its ID_A (possibly just some fraction of the bits of ID_A is shown, enough to narrow down the possibilities to no more than a handful of Authenticators).
2. User dials CC on the telephone and enters this information using his touchtone keypad, as prompted by a recorded voice. (This option is severely limited by the number of digits a user can reasonably be expected to type in.)

3. CC tells the user to enter a specific random challenge string into the Authenticator using the input devices available, a keypad or a joystick interface.
4. Authenticator calculates a cryptographic hash of the program output and challenge string and encrypts it using SK_A, displaying the result.
5. The User enters this result into the CC again using the keypad.
6. CC calculates the hash and encryption on its own and confirms the value entered by the user.

Several variations are possible. To improve reliability, the values which the user is asked to transfer can be padded with some redundancy to allow some level of error correction if he gets a few digits wrong. A variation on the above protocol uses a separate key than SK_A, one which is the same for all Authenticators and is programmed into their ROM and manufacturing time. If this is done there is no need for the Authenticator to display ID_A in step 1 or the user to enter it in step 2. Step 4 can be done by using a keyed one-way hash function instead of an encryption function.

6 Dividing the Software

Crucial to the success of this system is the ability to divide the software into two components: one running on the Computer and the other on the Authenticator. Currently we have only prototyped one system in this manner — a computer game — but our experiences have shed some light on how to accomplish this in the general case.

It is essential that both components must be designed from the outset to work together. We envision this as a distributed application running on two processors; arbitrary software running on the Computer cannot be certified by the Authenticator. It is extremely difficult to retrofit "conventional" software to run on this type of system.

The system works best when the Authenticator runs a portion of software that is, at the same time, essential for the successful execution of the application and difficult to reverse-engineer from its inputs and outputs. One way to subvert this system is to reverse-engineer the Authenticator code and then run it as another process on the Computer. For our game software, we choose routines related to survival — number of lives remaining, ammunition, time left, etc. — to secure in our Authenticator.

Finally, the amount of security is necessarily tied to the value of the application being secured. If the Authenticator is being used to secure a low-value application, only a small portion of software is required to run on the Authenticator. If it is being used to secure a high-value application, more of the software should run on the Authenticator.

7 Conclusions

Real-world applications of cryptography often only require the simplest of algorithms and protocols. We have shown how to take the simple notion of digital signatures and, by combining it with the notion of a secure processor, create a robust application for authenticating the outputs of software. Protocols such as these will most likely play a large role in electronic commerce application.

Further research is required in dividing software into secure and insecure components in such a way that if the Authenticator executes only the secure components, then the Authenticator can determine that specified output for the whole software has actually occurred.

8 Acknowledgments

The authors would like to thank James Jorasch, Jay Walker, and the CARDIS program committee for their helpful comments. This work was originally done under contract with Walker Digital, Inc. — U.S. and international patents pending — and presented at CARDIS '96 [5].

References

[1] D. Chaum and T. Pedersen, Wallet Databases with Observers, in Advances in Cryptology — CRYPTO '92 (Springer-Verlag, 1993) 89–105.

[2] W. Diffie and M. Hellman, New Directions in Cryptography, IEEE Transactions on Information Theory, IT-22 (1976) 644-654.

[3] T. ElGamal, A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms, IEEE Transactions on Information Theory, IT-31 (1985) 469-472.

[4] L.C. Guillou, M. Udon, and J.-J. Quisquater, The Smart Card: A Standardized Security Device Dedicated to Public Key Cryptography, in G. Simmons, ed., Contempory Cryptology: The Science of Information Integrity (IEEE Press, 1992) 561-613.

[5] J. Kelsey and B. Schneier, Authenticating Outputs of Computer Software Using a Cryptographic Coprocessor, in Proceedings 1996 CARDIS, Smart Card Research and Advanced Applications, (Amsterdam, 16–18 September 1996) 11–24.

[6] X. Lai, J. Massey, and S. Murphy, Markov Ciphers and Differential Cryptanalysis, in Advances in Cryptology — CRYPTO '91 (Springer-Verlag, 1991) 17–38.

[7] National Bureau of Standards, NBS FIPS PUB 46, Data Encryption Standard, National Bureau of Standards, U.S. Department of Commerce, Jan 1977.

[8] National Institute of Standards and Technologies, NIST FIPS PUB 186, Digital Signature Standard, U.S. Department of Commerce, May 1994.

[9] R. Rivest, A. Shamir, and L. Adleman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Communications of the ACM, 21 (1978) 120-126.

[10] B. Schneier, Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish), in R. Anderson, ed., Fast Software Encryption, Cambridge Security Workshop Proceedings (Springer-Verlag, 1994) 191–204.

[11] B. Schneier, Applied Cryptography, 2nd Edition (John Wiley & Sons, 1996).

[12] B. Schneier and J. Kelsey, A Peer-to-Peer Software Metering System, in The Second USENIX Workshop on Electronic Commerce (USENIX Association, 1996), pp. 279–286.

[13] B. Yee and J.D. Tygar, Secure Coprocessors in Electronic Commerce Applications, in The First USENIX Workshop on Electronic Commerce (USENIX Association, 1995) 155-170.