Chapter 7

Hardware Encryption Modules

Wherein cryptlib is applied to provide security and encryption services via embedded hardware modules.

1. Problems with Crypto on End-user Systems

The majority of current crypto implementations run under general-purpose operating systems with a relatively low level of security, alongside which exist a limited number of smart-card assisted implementations which store a private key in, and perform private-key operations with, a smart card. Complementing these are an even smaller number of implementations which perform further operations in dedicated (and generally very expensive) hardware.

The advantage of software-only implementations is that they are inexpensive and easy to deploy. The disadvantage of these implementations is that they provide a very low level of protection for cryptovariables, and that this low level of security is unlikely to change in the future. For example Windows NT provides a function `ReadProcessMemory()` which allows a process to read the memory of (almost) any other process in the system (this was originally intended to allow debuggers to establish breakpoints and maintain instance data for other processes [1]), allowing both passive attacks such as scanning memory for high-entropy areas which constitute keys [2] and active attacks in which a target processes' code or data is modified to provide supplemental functionality of benefit to a hostile process. This type of modification would typically be performed by obtaining the target processes' handle, using `SuspendThread()` to halt it, `VirtualProtectEx()` to make the code pages writeable, `WriteProcessMemory()` to modify the code, and `ResumeThread()` to restart the processes' execution (these are all standard Windows functions and don't require security holes or coding bugs in order to work). By subclassing an application such as the Windows shell, the hostile process can receive notification of any application (a.k.a. "target") starting up or shutting down, after which it can apply the mechanisms mentioned previously. A very convenient way to do this is to subclass a child window of the system tray window, yielding a system-wide hook for intercepting shell messages [3]. Another way to obtain access to other processes' data is to patch the user-to-kernel-mode jump table in a processes' Thread Environment Block (TEB), which is shared by all processes in the system rather than being local to each one, so that changing it in one process affects every other running process [4]. Sometimes it isn't even necessary to perform heuristic scans for likely keying information, for example by opening a handle to WINLOGON.EXE (the Windows logon process), using `ReadProcessMemory()` to read the page at 0x10000, and scanning for the text string `lMprNotifyPassword=` it's possible to obtain the current user's password, which isn't cleared from memory by the logon process [5].

Although the use of functions like `ReadProcessMemory()` requires Administrator privileges, most users tend to either run their system as Administrator or give themselves equivalent privileges since it's extremely difficult to make use of the machine without these privileges. In the unusual case where the user isn't running with these privileges, it's possible to use a variety of tricks to bypass any OS security measures which might be present in order to perform the desired operations. For example by installing a Windows message hook it's possible to capture messages intended for another process and have them dispatched to your own message handler. Windows then loads the hook handler into the address space of the process which owns the thread which the message was intended for, in effect yanking your code across into the address space of the victim [6]. Even simpler are mechanisms such as using the HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs key, which specifies a list of DLLs which are automatically loaded and called whenever an application uses the USER32 system library (which is automatically used by all GUI applications and many command-line ones). Every DLL specified in this registry key is loaded into the processes' address space by USER32, which then calls the DLL's `DllMain()` function to initialise the DLL (and, by extension, trigger whatever other actions the DLL is designed for).

A more sophisticated attack involves persuading the system to run your code in ring 0 (the most privileged security level usually reserved for the OS kernel) or, alternatively, convincing the OS to allow you to load a selector which provides access to all physical memory (under Windows NT, selectors 8 and 10 provide this capability). Running user code in ring 0 is possible due to the peculiar way in which the NT kernel loads. The kernel is accessed via the int 2Eh call gate, which initially provides about 200 functions via NTOSKRNL.EXE but is then extended to provide more and more functions as successive parts of the OS are loaded. Instead of merely adding new functions to the existing table, each new portion of the OS which is loaded takes a copy of the existing table, adds its own functions to it, and then replaces the old one with the new one. To add supplemental functionality at the kernel level, all that's necessary is to do the same thing [7]. Once your code is running at ring 0, an NT system starts looking a lot like a machine running DOS.

Although the problems mentioned so far have concentrated on Windows NT, many Unix systems aren't much better. For example the use of `ptrace` with the PTRACE_ATTACH option followed by the use of other `ptrace` capabilities provides similar headaches to those arising from `ReadProcessMemory()`. The reason why these issues are more problematic under NT is that users are practically forced to run with system Administrator privileges in order to perform any useful work on the system, since a standard NT system has no equivalent to Unix's `su` functionality and, to complicate things further, frequently assumes that the user always has Administrator privileges (that is, it assumes it's a single-user system with the user being Administrator). While it's possible to provide some measure of protection on a Unix system by running crypto code as a daemon in its own memory space under a different account, under NT all services run under the single System Account so that any service can use `ReadProcessMemory()` to interfere with any other service [8]. Since an Administrator can dynamically load NT services at any time and since a non-administrator can create processes running under the System Account by overwriting the handle of the parent process with that of the System Account [9], even implementing the crypto code as an NT service provides no escape.

1.1 The Root of the Problem

The reason why problems like those described above persist, and why we're unlikely to ever see a really secure consumer OS is because it's not something which most consumers care about. One recent survey of Fortune 1000 security managers showed that although 92% of them were concerned about the security of Java and ActiveX, nearly three quarters allowed them onto their internal networks, and more than half didn't even bother scanning for them [10]. Users are used to programs malfunctioning and computers crashing (every Windows NT user can tell you what the abbreviation BSOD means even though it's never actually mentioned in the documentation), and see it as normal for software to contain bugs. Since program correctness is difficult and expensive to achieve, and as long as flashiness and features are the major selling point for products, buggy and insecure systems will be the normal state of affairs [11]. Unlike other Major Problems like Y2K (which contain their own built-in deadline), security generally isn't regarded as a pressing issue unless the user has just been successfully attacked or the corporate auditors are about to pay a visit, which means that it's much easier to defer addressing it to some other time [12]. Even in cases where the system designers originally intended to implement a rigorous security system employing a proper TCB, the requirement to add features to the system inevitably results in all manner of additions being crammed into the TCB as application-specific functionality starts migrating into the OS kernel. The result of this creep is that the TCB is neither small, nor verified, nor secure.

An NSA study [13] lists a number of features which are regarded as "crucial to information security" but which are absent from all mainstream operating systems. Features such as mandatory access controls which are mentioned in the study correspond to Orange Book B-level security features which can't be bolted onto an existing design but generally need to be designed in from the start, necessitating a complete overhaul of an existing system in order to provide the required functionality. This is often prohibitively resource-intensive, for example the task of reengineering the Multics kernel (which contained a "mere" 54,000 lines of code) to provide a minimised TCB was estimated to cost $40M (in 1977 dollars) and was never completed [14]. The work involved in performing the same kernel upgrade or redesign from scratch with an operating system containing millions or tens of millions of lines of code would make it beyond prohibitive.

At the moment security and ease of use are at opposite ends of the scale, and most users will opt for ease of use over security. JavaScript, ActiveX, and embedded active content may be a security nightmare, but they do make life a lot easier for most users, leading to comments from security analysts like "You want to write up a report with the latest version of Microsoft Word on your insecure computer or on some piece of junk with a secure computer?" [15], "Which sells more products: really secure software or really easy-to-use software?" [16], "It's possible to make money from a lousy product […] Corporate cultures are focused on money, not product" [17], and "The marketplace doesn't reward *real* security. Real security is harder, slower and more expensive, both to design and to implement. Since the buying public has no way to differentiate real security from bad security, the way to win in this marketplace is to design software that is as insecure as you can possibly get away with […] users prefer cool features to security" [18].

One study which examined the relationship between faults (more commonly referred to as bugs) and software failures found that one third of all faults resulted in a mean time to failure (MTTF) of more than 5,000 years, with somewhat less than another third having a MTTF of more than 1,500 years. Conversely, around 2% of all faults had a MTTF of less than five years [19]. The reason for this is that even the most general-purpose programs are only ever used in stereotyped ways which exercise only a tiny portion of the total number of code paths, so that removing (visible) problems from these areas will be enough to keep the majority of users happy. This conclusion is backed up by other studies such as one which examined the behaviour of 30 Windows applications in the presence of random (non-stereotypical) keyboard and mouse input. The applications were chosen to cover a range of vendors, commercial and non-commercial software, and a wide variety of functionality including word processors, web browsers, presentation graphics editors, network utilities, spread sheets, software development environments, and assorted random applications such as Notepad, Solitaire, the Windows CD player, and similar common programs. The study found that 21% of the applications tested crashed and 24% hung when sent random keyboard/mouse input, and when sent random Win32 messages (corresponding to events other than direct keyboard- and mouse-related actions), *all* the applications tested either crashed or hung [20].

Even when an anomaly is detected, it's often easier to avoid it by adapting the code or user behaviour which invokes it ("don't do that, then") because this is less effort than trying to get the error fixed[1]. In this manner problems are avoided by a kind of symbiosis through which the reliability of the system as a whole is greater than the reliability of any of its parts [21]. Since most of the faults which will be encountered are benign (in the sense that they don't lead to failures for most users), all that's necessary in order for the vendor to provide the perception of reliability is to remove the few percent of faults which cause noticeable problems. Although it may be required for security purposes to remove every single fault (as far as is practical), for marketing purposes it's only necessary to remove the few percent which are likely to cause problems.

In many cases users don't even have a choice as to which software they can use, if they can't process data from Word, Excel, PowerPoint, and Outlook and view web pages loaded with JavaScript and ActiveX, their business doesn't run, and some companies go so far as to publish explicit instructions telling users how to disable security measures in order to maximise their web-browsing experience [22]. Going beyond basic OS security, most current security products still don't effectively address the problems posed by hostile code such as trojan horses (which the Bell-LaPadula model was designed to combat), and the systems the code runs on increase both the power of the code to do harm and the ease of distributing the code to other systems.

This presents a rather gloomy outlook for someone wanting to provide secure crypto services to a user of these systems. In order to solve this problem, we adopt a reversed form of the Mohammed-and-the-mountain approach: Instead of trying to move the insecurity away from the crypto through various operating system security measures, we move the crypto away from the insecurity. In other words although the user may be running a system crawling with rogue ActiveX controls, macro viruses, trojan horses, and other security nightmares, none of these can come near the crypto.

1.2 Solving the Problem

The FIPS 140 standard provides us with a number of guidelines for the development of cryptographic security modules. NIST originally allowed only hardware implementations of cryptographic algorithms (for example the original NIST DES document allowed for hardware implementation only [23][24]), however this requirement was relaxed somewhat in the mid-1990's to allow software implementations as well [25][26]. FIPS 140 defines four security levels ranging from level 1 (the cryptographic algorithms are implemented correctly) through to level 4 (the module or device has a high degree of tamper-resistance including an active tamper response mechanism which causes it to zeroise itself when tampering is detected). To date only one general-purpose product family has been certified at level 4 [27][28].

Since FIPS 140 also allows for software implementations, an attempt has been made to provide an equivalent measure of security for the software platform on which the cryptographic module is to run. This is done by requiring the underlying operating system to be evaluated at progressively higher Orange Book levels for each FIPS 140 level, so that security level 2 would require the software module to be implemented on a C2-rated operating system. Unfortunately this provides something if an impedance mismatch between the actual security of hardware and software implementations, since it implies that products such as a Fortezza card [29] or Dallas iButton (a relatively high-security device) [30] provide the same level of security as a program running under Windows NT. It's possible that the OS security levels were set so low out of concern that setting them any higher would make it impossible to implement the higher FIPS 140 levels in software due to a lack of systems evaluated at that level.

Even with sights set this low, it doesn't appear to be possible to implement secure software-only crypto on a general-purpose PC. Trying to protect cryptovariables (or more generically critical security parameters, CSP's in FIPS 140-speak) on a system which provides functions like `ReadProcessMemory` seems pointless, even if the system does claim a C2/E2 evaluation. On the other hand trying to source a B2 or more realistically B3 system to provide an adequate level of security for the crypto software is almost

[1] This document, prepared with MS Word, illustrates this principle quite well, having been produced in a manner which avoided a number of bugs which would crash the program.

impossible (the practicality of employing an OS in this class, whose members include Trusted Xenix, XTS 300, and Multos, speaks for itself). A simpler solution would be to implement a crypto coprocessor using a dedicated machine running at system high, and indeed FIPS 140 explicitly recognises this by stating that the OS security requirements only apply in cases where the system is running programs other than the crypto module (to compensate for this, FIPS 140 imposes its own software evaluation requirements which in some cases are even more arduous than the Orange Book ones).

An alternative to a pure-hardware approach might be to try to provide some form of software-only protection which attempts to compensate for the lack of protection present in the OS. Some work has been done in this area involving the obfuscation of the code to be protected, either mechanically [31][]32 or manually [33]. The use of mechanical obfuscation (for example reordering of code and the insertion of dummy instructions) is also present in a number of polymorphic viruses, and can be quite effectively countered [34][35]. Manual obfuscation techniques are somewhat more difficult to counter automatically, however computer game vendors have trained several generations of crackers in the art of bypassing the most sophisticated software protection and security features they could come up with [36][37][38], indicating that this type of protection won’t provide any relief either, and this doesn’t even go into the portability and maintenance nightmare which this type of code presents (it is for these reasons that the obfuscation provisions were removed from a later version of the CDSA specification where they were first proposed [39]). There also exists a small amount of experimental work involving trying to create a form of software self-defence mechanism which tries to detect and compensate for program or data corruption [40][41][42][43], however this type of self-defence technology will probably stay restricted to Core Wars Redcode programs for some time to come. As the final nail in the coffin, a general proof exists which shows that real code obfuscation is impossible [44].

1.3 Coprocessor Design Issues

The main consideration when designing a coprocessor to manage crypto operations is how much functionality we should move from the host into the coprocessor unit. The baseline, which we’ll call a tier[2] 0 coprocessor, has all the functionality in the host, which is what we’re trying to avoid. The levels above tier 0 provide varying levels of protection for cryptovariables and coprocessor operations, as shown in Figure 1. The minimal level of coprocessor functionality, a tier 1 coprocessor, moves the private key and private-key operations out of the host. This type of functionality is found in smart cards, and is only a small step above having no protection at all, since although the key itself is held in the card, all operations performed by the card are controlled by the host, leaving the card at the mercy of any malicious software on the host system. In addition to these shortcomings, smart cards are very slow, offer no protection for cryptovariables other than the private key, and often can’t even protect the private key fully (for example a card with an RSA private key intended for signing can be misused to decrypt a key or message since RSA signing and decryption are equivalent).

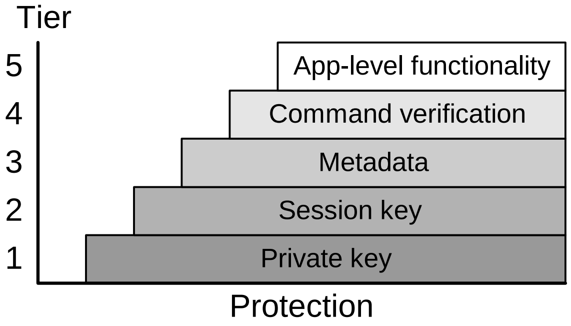

Figure 1: Levels of protection offered by crypto hardware

The next level of functionality, tier 2, moves both public/private-key operations and conventional encryption operations along with hybrid mechanisms such as public-key wrapping of content-encryption keys into the coprocessor. This type of functionality is found in devices such as Fortezza cards and a number of devices sold as crypto accelerators, and provides rather more protection than that found in smart cards since no cryptovariables are ever exposed on the host. Like smart cards however, all control over the

[2] The reason for the use of this somewhat unusual term is because almost every other noun used to denote hierarchies is already in use; “tier” is unusual enough that no-one else has got around to using it in their security terminology.

device’s operation resides in the host, so that even if a malicious application can’t get at the keys directly, it can still apply them in a manner other than the intended one.

The next level of functionality, tier 3, moves all crypto-related processing (for example certificate generation and message signing and encryption) into the coprocessor. The only control the host has over processing is at the level of “sign this message” or “encrypt this message”, all other operations (message formatting, the addition of additional information such as the signing time and signer’s identity, and so on) is performed by the coprocessor. In contrast if the coprocessor has tier 1 functionality the host software can format the message any way it wants, set the date to an arbitrary time (in fact it can never really know the true time since it’s coming from the system clock which another process could have altered), and generally do whatever it wants with other message parameters. Even with a tier 2 coprocessor such as a Fortezza card which has a built-in real-time clock (RTC), the host is free to ignore the RTC and give a signed message any timestamp it wants. Similarly, even though protocols like CSP which is used with Fortezza incorporate complex mechanisms to handle authorisation and access control issues [45], the enforcement of these mechanisms is left to the untrusted host system rather than the card(!). Other potential problem areas involve handling of intermediate results and composite call sequences which shouldn’t be interrupted, for example loading a key and then using it in a cryptographic operation [46]. In contrast, with a tier 3 coprocessor which performs all crypto-related processing independent of the host the coprocessor controls the message formatting and the addition of additional information such as a timestamp taken from its own internal clock, moving them out of reach of any software running on the host. The various levels of protection when the coprocessor is used for message decryption are shown in Figure 2.

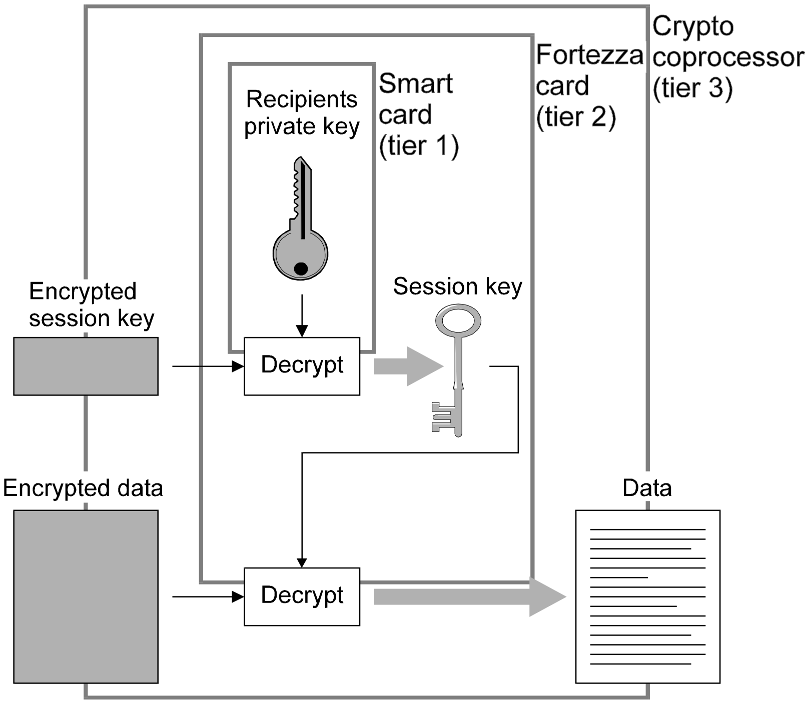

Figure 2: Protection levels for the decrypt operation

Going beyond tier 3, a tier 4 coprocessor provides facilities such as command verification which prevent the coprocessor from acting on commands sent from the host system without the approval of the user. The features of this level of functionality are explained in more detail in Section 4, which covers extended security functionality.

Can we move the functionality to an even higher level, tier 5, giving the coprocessor even more control over message handling? Although it’s possible to do this, it isn’t a good idea since at this level the coprocessor will potentially need to run message viewers (to display messages), editors (to create/modify messages), mail software (to send and receive them), and a whole host of other applications, and of course these programs will need to be able to handle MIME attachments, HTML, JavaScript, ActiveX, and so on in order to function as required. In addition the coprocessor will now require its own input mechanism (a keyboard), output mechanism (a monitor), mass storage, and other extras. At this point the coprocessor has evolved into a second computer attached to the original one, and since it’s running a range of untrusted and potentially dangerous code we need to think about moving the crypto functionality into a coprocessor for safety. Lather, rinse, repeat.

The best level of functionality therefore is to move all crypto and security-related processing into the coprocessor, but to leave everything else on the host.

2. The Coprocessor

The traditional way to build a crypto coprocessor has been to create a complete custom implementation, originally with ASICs and more recently with a mixture of ASICs and general-purpose CPUs, all controlled by custom software. This approach leads to long design cycles, difficulties in making changes at a later point, high costs (with an accompanying strong incentive to keep all design details proprietary due to the investment involved), and reliance on a single vendor for the product. In contrast an open-source coprocessor by definition doesn't need to be proprietary, so it can use existing COTS hardware and software as part of its design, which greatly reduces the cost (the coprocessor described here is one to two orders of magnitude cheaper than proprietary designs while offering generally equivalent performance and superior functionality), and can be sourced from multiple vendors and easily migrated to newer hardware as the current hardware base becomes obsolete.

The coprocessor requires three layers, the processor hardware, the firmware which manages the hardware (for example initialisation, communications with the host, persistent storage, and so on) and the software which handles the crypto functionality. The following sections describe the coprocessor hardware and resource management firmware on which the crypto control software runs.

2.1 Coprocessor Hardware

Embedded systems have traditionally been based on the VME bus, a 32-bit data/32-bit address bus incorporated onto cards in the 3U (10×16cm) and 6U (23×16cm) Eurocard form factor [47]. The VME bus is CPU-independent and supports all popular microprocessors including Sparc, Alpha, 68K, and x86. An x86-specific bus called PC/104, based on the 104-pin ISA bus, has become popular in recent years due to the ready availability of low-cost components from the PC industry. PC/104 cards are much more compact at 9×9.5cm than VME cards, and unlike a VME passive backplane-based system can provide a complete system on a single card [48]. PC/104-Plus, an extension to PC/104, adds a 120-pin PCI connector alongside the existing ISA one, but is otherwise mostly identical to PC/104 [49]

In addition to PC/104 there are a number of functionally identical systems with slightly different form factors, of which the most common is the biscuit PC shown in Figure 3, a card the same size as a 3½" or occasionally 5¼" drive, with a somewhat less common one being the credit card or SIMM PC roughly the size of a credit card. A biscuit PC provides most of the functionality and I/O connectors of a standard PC motherboard, as the form factor shrinks the I/O connectors do as well so that a SIMM PC typically uses a single enormous edge connector for all its I/O. In addition to these form factors there also exist card PC's (sometimes called slot PC's), which are biscuit PC's built as ISA or (more rarely) PCI-like cards. A typical configuration for a low-end system is a 5x86/133 CPU (roughly equivalent in performance to a 133 MHz Pentium), 8-16MB of DRAM, 2-8MB of flash memory emulating a disk drive, and every imaginable kind of I/O (serial ports, parallel ports, floppy disk, IDE hard drive, IR and USB ports, keyboard and mouse, and others). High-end embedded systems built from components designed for laptop use provide about the same level of performance as a current laptop PC, although their price makes them rather impractical for use as crypto hardware. To compare this with other well-known types of crypto hardware, a typical smart card has a 5MHz 8-bit CPU, a few hundred bytes of RAM, and a few kB of EEPROM, and a Fortezza card has a 10 or 20MHz ARM CPU, 64kB of RAM and 128kB of flash memory/EEPROM.

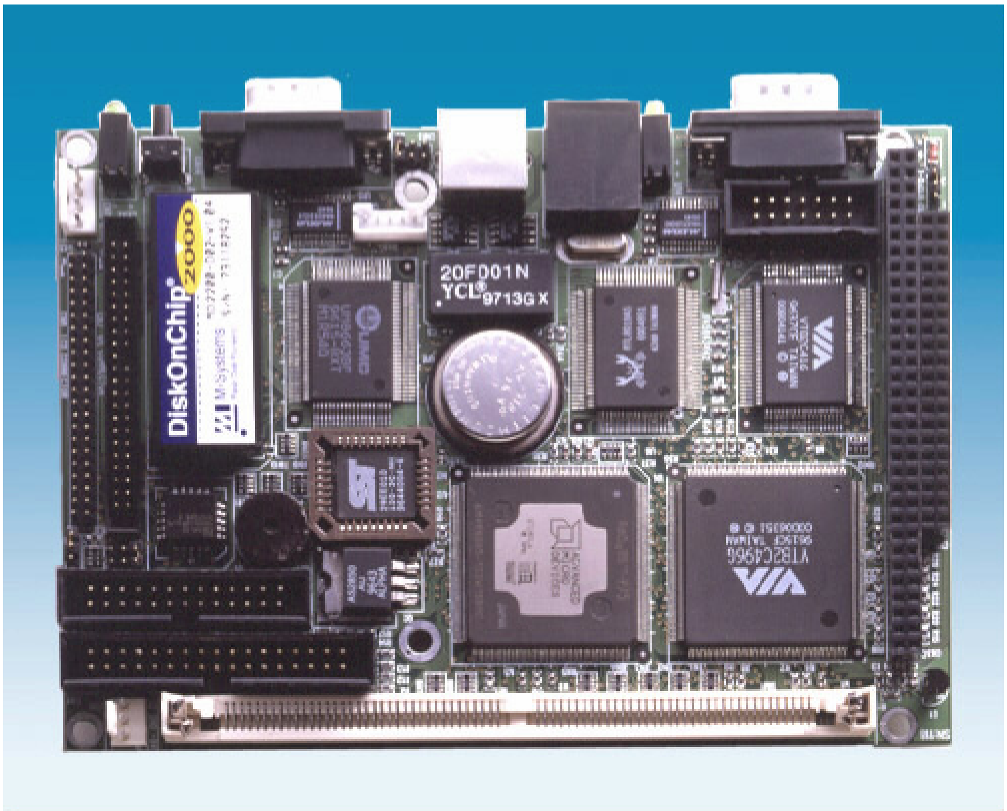

Figure 3: Biscuit PC (life-size)

All of the embedded systems described above represent COTS components available from a large range of vendors in many different countries, with a corresponding range of performance and price figures. Alongside the x86-based systems there also exist systems based on other CPU's, typically ARM, Dragonball (embedded Motorola 68K), and to a lesser extent PowerPC, however these are available from a limited number of vendors and can be quite expensive. Besides the obvious factor of system performance affecting the overall price, the smaller form factors and use of exotic hardware such as non-generic-PC components can also drive up the price. In general the best price/performance balance is obtained with a very generic PC/104 or biscuit PC system.

2.2 Coprocessor Firmware

Once the hardware has been selected the next step is to determine what software to run on it to control it. The coprocessor is in this case acting as a special-purpose computer system running only the crypto control software, so that what would normally be thought of as the operating system is acting as the system firmware, and the real operating system for the device is the crypto control software. The control software therefore represents an application-specific operating system, with crypto objects such as encryption contexts, certificates, and envelopes replacing the user applications which are managed by conventional OS's. The differences between a conventional system and the crypto coprocessor running one typical type of firmware-equivalent OS are shown in Figure 4.

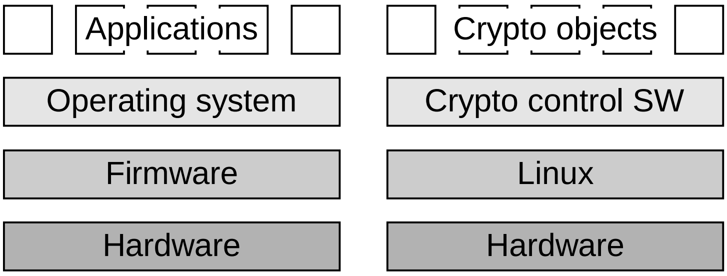

Figure 4: Conventional system vs. coprocessor system layers

Since the hardware is in effect a general-purpose PC, there's no need to use a specialised, expensive embedded or real-time kernel or OS since a general-purpose OS will function just as well. The OS choice is then something simple like one of the free or nearly-free embeddable forms of MSDOS [50][51][52] or an open source operating system like one of the x86 BSDs or Linux which can be adapted for use in embedded hardware. Although embedded DOS is the simplest to get going and has the smallest resource requirements, it's really only a bootstrap loader for real-mode applications and provides very little access to most of the resources provided by the hardware. For this reason it's not worth considering except on extremely low-end, resource-starved hardware (it's still possible to find PC/104 cards with 386/40's on them, although having to drive them with DOS is probably its own punishment).

A better choice than DOS is a proper operating system which can fully utilise the capabilities of the hardware. The only functionality which is absolutely required of the OS is a memory manager and some form of communication with the outside world. Also useful (although not absolutely essential) is the ability to store data such as private keys in some form of persistent storage. Finally, the ability to handle multiple threads may be useful where the device is expected to perform multiple crypto tasks at once. Apart from the multithreading, the OS is just acting as a basic resource manager, which is why DOS could be pressed into use if necessary.

Both FreeBSD and Linux have been stripped down in various ways for use with embedded hardware [53][54]. There's not really a lot to say about the two, both meet the requirements given above, both are open source systems, and both can use a standard full-scale system as the development environment — whichever one is the most convenient can be used. At the moment Linux is a better choice because its popularity means there's better support for devices such as flash memory mass storage (relatively speaking, as the Linux drivers for the most widely-used flash disk are for an old kernel while the FreeBSD ones are mostly undocumented and rather minimal), so the coprocessor described here uses Linux as its resource management firmware. A convenient feature which gives the free Unixen an extra advantage over alternatives like embedded DOS is that they'll automatically switch to using the serial port for their consoles if no video drivers and/or hardware are present, which enables them to be used with cheaper embedded hardware which doesn't require additional video circuitry just for the one-off setup process. A particular advantage of Linux is that it'll halt the CPU when nothing is going on (which is most of the time), greatly reducing coprocessor power consumption and heat problems.

2.3 Firmware Setup

Setting up the coprocessor firmware involves creating a stripped-down Linux setup capable of running on the coprocessor hardware. The services required of the firmware are:

- Memory management
- Persistent storage services
- Communication with the host
- Process and thread management (optional)

All newer embedded systems support the M-Systems DiskOnChip (DOC) flash disk which emulates a standard IDE hard drive by identifying itself as a BIOS extension during the system initialisation phase (allowing it to install a DOC filesystem driver to provide BIOS support for the drive) and later switching to a native driver for OS's which don't use the BIOS for hardware access [55]. More recently systems have begun moving to the use of compact flash cards which emulate IDE hard drives due to their popularity in digital cameras and somewhat lower costs then DOCs. The first step in installing the firmware involves formatting the DOC or compact flash card as a standard hard drive and partitioning it prior to installing Linux. The flash disk is configured to contain two partitions, one mounted read-only which contains the firmware and crypto control software, and one mounted read/write with additional safety precautions like `noexec` and `nosuid`, for storage of configuration information and encrypted keys.

The firmware consists of a basic Linux kernel with every unnecessary service and option stripped out. This means removing support for video devices, mass storage (apart from the flash disk and floppy drive), multimedia devices, and other unnecessary bagatelles. Apart from the TCP/IP (or similar protocol) stack needed by the crypto control software to communicate with the host, there are no networking components running (or even present) on the system, and even the TCP/IP stack may be absent if alternative, more low-level means of communicating with the host (explained in more detail further on) are employed. All configuration tasks are performed through console access via the serial port, and software is installed by connecting a floppy drive and copying across pre-built binaries. This both minimises the size of the code base which needs to be installed on the coprocessor, and eliminates any unnecessary processes and services which might constitute a security risk. Although it would be easier if we provided a means of FTP'ing binaries across, the fact that a user must explicitly connect a floppy drive and mount it in order to change the firmware or control software makes it much harder to accidentally (or maliciously) move problematic code across to the coprocessor, provides a workaround for the fact that FTP over alternative coprocessor communications channels such as a parallel port is tricky without resorting to the use of even more potential problem software, and makes it easier to comply with the FIPS 140 requirements that (where a non-Orange Book OS is used) it not be possible for extraneous software to be loaded and run on the system. Direct console access is also used for other operations such as setting the onboard real-time clock, which is used to add timestamps to signatures. Finally, all paging is disabled, both because it isn't needed or safe to perform with the limited-write-cycle flash disk, and because it avoids any risk of sensitive data being written to backing store, eliminating a major headache which occurs with all virtual-memory operating systems [56].

At this point we have a basic system consisting of the underlying hardware and enough firmware to control it and provide the services we require. Running on top of this will be a daemon which implements the crypto control software which does the actual work.

3. Crypto Functionality Implementation

Once the hardware and functionality level of the coprocessor have been established, we need to design an appropriate programming interface for it. An interface which employs complex data structures, pointers to memory locations, callback functions, and other such elements won't work with the coprocessor unless a complex RPC mechanism is employed. Once we get to this level of complexity we run into problems both with lowered performance due to data marshalling and copying requirements and potential security problems arising from inevitable implementation bugs. A better way to handle this is to apply the forwarder-receiver model shown in Figure 5, which takes cryptlib function calls on the local machine and forwards them to the coprocessor, returning the results to the local machine in a similar manner.

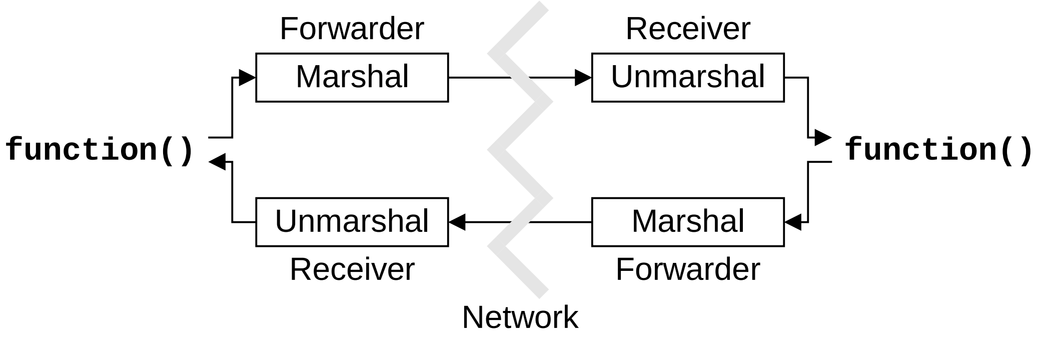

Figure 5: Coprocessor communication using the forwarder-receiver model

The interface used by cryptlib is ideally suited for use in a coprocessor since only the object handle (a small integer value) and one or two arguments (either an integer value or a byte string and length) are needed to perform most operations. This use of only basic parameter types leads to a very simple and lightweight interface, with only the integer values needing any canonicalisation (to network byte order) before being passed to the coprocessor. A coprocessor call of this type, illustrated in Figure 6, requires only a few lines of code more than what is required for a direct call to the same code on the host system. In practice the interface is further simplified by using a pre-encoded template containing all fixed parameters (for example the type of function call being performed and a parameter count), copying in any variable parameters (for example the object handle) with appropriate canonicalistion, and dispatching the result to the coprocessor. The coprocessor returns results in the same manner.

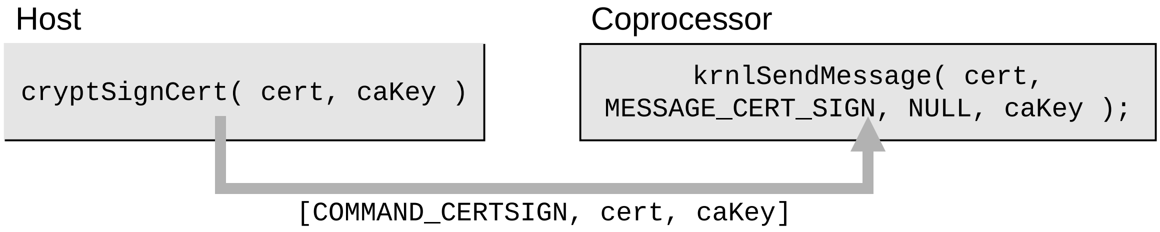
Host
cryptSignCert(cert, caKey)
Coprocessor
krnlSendMessage(cert,
MESSAGE_CERT_SIGN, NULL, caKey);
[COMMAND_CERTSIGN, cert, caKey]

Figure 6: Command forwarding to the coprocessor

The coprocessor interface is further simplified by the fact that even the local cryptlib interface constitutes a basic implementation of the forwarder-receiver model in which both ends of the connection happen to be on the same machine and in the same address space, reducing the use of special-case code which is only required for the coprocessor.

3.1 Communicating with the Coprocessor

The next step after designing the programming interface is to determine which type of communications channel is best suited to controlling the coprocessor. Since the embedded controller hardware is intended for interfacing to almost anything, there are a wide range of I/O capabilities available for communicating with the host. Many embedded controllers provide an Ethernet interface either standard or as an option, so the most universal interface uses TCP/IP for communications. For card PCs which plug into the host's backplane we should be able to use the system bus for communications, and if that isn't possible we can take advantage of the fact that the parallel ports on all recent PCs provide sophisticated (for what was intended as a printer port) bidirectional I/O capabilities and run a link from the parallel port on the host motherboard to the parallel port on the coprocessor. Finally, we can use more exotic I/O capabilities such as USB and similar high-speed serial links to communicate with the coprocessor. By using (or at least emulating via a sockets interface) TCP/IP over each of these physical links, we can provide easy portability across a wide range of interface types.

The most universal coprocessor consists of a biscuit PC which communicates with the host over Ethernet (or, less universally, a parallel port). One advantage which an external, removable coprocessor of this type has over one which plugs directly into the host PC is that it's very easy to unplug the entire crypto subsystem and store it separately from the host, moving it out of reach of any covert access by outsiders [57] while the owner of the system is away. In addition to the card itself, this type of standalone setup requires a case and a power supply, either internal to the case or an external wall-wart type (these are available for about $10 with a universal input voltage range which allows them to work in any country). The same arrangement is used in a number of commercially-available products, and has the advantage that it interfaces to virtually any type of system, with the commensurate disadvantage that it requires a dedicated Ethernet connection to the host (which typically means adding an extra network card), as well as adding to the clutter surrounding the machine.

The alternative option for an external coprocessor is to use the parallel port, which doesn't require a network card but does tie up a port which may be required for one of a range of other devices such as external disk drives, CD writers, and scanners which have been kludged onto this interface alongside the more obvious printers. Apart from its more obvious use, the printer port can be used either as an Enhanced Parallel Port (EPP) or as an Extended Capability Port (ECP) [58]. Both modes provide about 1-2 MB/s data throughput (depending on which vendor's claims are to be believed) which compares favourably with a parallel port's standard software-intensive maximum rate of around 150 kB/s and even with the throughput of a 10Mbps Ethernet interface. EPP was designed for general-purpose bidirectional communication with peripherals and handles intermixed read and write operations and block transfers without too much trouble, whereas ECP (which requires a DMA channel which can complicate the host system's configuration process) requires complex data direction negotiation and handling of DMA transfers in progress, adding a fair amount of overhead when used with peripherals which employ mixed reading and writing of small data quantities. Another disadvantage of DMA is that its use paralyses the CPU by seizing control of the bus, halting all threads which may be executing while data is being transferred. Because of this the optimal interface mechanism is EPP. From a programming point of view, this communications mechanism looks like a permanent virtual circuit which is functionally equivalent to the dumb wire which we're using the Ethernet link as, so the two can be interchanged with a minimum of coding effort.

To the user, the most transparent coprocessor would consist of some form of card PC which plugs directly into their system's backplane. Currently virtually all card PCs have ISA bus interfaces (the few which support PCI use a PCI/ISA hybrid which won't fit a standard PCI slot [59]) which unfortunately doesn't provide much flexibility in terms of communications capabilities since the only viable means of moving data to and from the coprocessor is via DMA, which requires a custom kernel-mode driver on both sides. The alternative, using the parallel port, is much simpler since most operating systems already support EPP and/or ECP data transfers, but comes at the expense of a reduced data transfer rate and the loss of use of the parallel port on the host. Currently the use of either of these options is rendered moot since the ISA card PCs assume they have full control over a passive-backplane-bus system, which means they can't be plugged into a standard PC which contains its own CPU which is also assuming that it solely controls the bus. It's possible that in the future card PCs which function as PCI bus devices will appear, but until they do it's not possible to implement the coprocessor as a plug-in card without using a custom extender card containing an ISA or PCI connector for the host side, a PC104 connector for a PC104-based CPU card, and buffer circuitry in between to isolate the two buses. This destroys the COTS nature of the hardware, limiting availability and raising costs.

The final communications option uses more exotic I/O capabilities such as USB (and occasionally other high-speed serial links such as FireWire) which are present on newer embedded systems, these are much like Ethernet but have the disadvantage that they are currently rather poorly supported by operating systems targeted at embedded systems.

The discussion so far has looked at the communications mechanism either as an interface-specific one or an emulated TCP/IP sockets interface, with the latter being built on top of the former. Although the generic sockets interface provides a high level of flexibility and works well with existing code, it requires that each device and/or device interface be allocated its own IP address and creates extra code overhead for providing the TCP/IP-style interface. Instead of using the standard AF_INET family, the sockets interface could implement a new AF_COPROCESSOR family with the address passed to the `connect()` function being a device or interface number or some similar identifier which avoids the need to allocate an IP address. This has the disadvantage that it loses some of the universality of the TCP/IP interface, which by extension makes it more difficult to perform operations such as direct device-to-device communications for purposes such as load balancing. Another advantage of the TCP/IP interface is that it's possible to apply existing cryptlib security mechanisms to the interface so that, for example, one coprocessor could talk directly to another over an SSL-protected link which would keep the communications secure even if the host which was handling them was compromised. Another possibility (covered in more detail in section 4.3) is that this interface frees the coprocessor from having to be located in the same physical location as the host or coprocessor which it's communicating with it.

Since we're using Linux as the resource manager for the coprocessor hardware, we can use a multithreaded implementation of the coprocessor software to handle multiple simultaneous requests from the host. After initialising the various cryptlib subsystems, the control software creates a pool of threads which wait on a mutex for commands from the host. When a command arrives, one of the threads is woken up, processes the command, and returns the result to the host. In this manner the coprocessor can have multiple requests outstanding at once, and a process running on the host won't block whenever another process has an outstanding request present on the coprocessor.

3.2 Coprocessor Session Control

When cryptlib is being run on the host system, the concept of a user session doesn't exist since the user has whatever control over system resources are allowed by their account privileges. When cryptlib is being used in a coprocessor which exists independently from the host, the concept of a session with the coprocessor applies. This works much like a session with a more general-purpose computer except that the capabilities available to the user are usually divided into two classes, those of a security officer or SO (the super-user- or administrator-equivalent for the coprocessor) and those of a standard user. The SO can perform functions such as initialising the device and (in some cases) performing key loading and generation actions but can't actually make use of the keys, while the user can make use of the keys but can't generally perform administrative actions.

The exact details of the two roles are somewhat application-specific, for example the Fortezza card allows itself to be initialised and initial keys and certificates to be loaded by the SO (in the circles where Fortezza is used the term is site security officer or SSO), after which the initial user PIN is set which automatically logs the SO out. At this point the card initialisation functions are disabled, and the SO can log in again to perform maintenance operations or the user can log in to use the card to sign or encrypt data. When logged in as SO it's not possible to use the card for standard operations, and when logged in as user it's not possible to perform maintenance operations [60]. The reason for enforcing this sequence of operations is that it provides a clear chain of control and responsibility for the device, since the card is moved into its initial state by the SO who started with a pristine (at least as far as the FIPS 140 tamper-evident case is able to indicate) card into which they load initial values and then hand the device on to the user. The SO knows (with a good degree of certainty) that they have an untampered card, and initialises it as required, after which the user knows that they have an initialised card which was configured for them by the SO. This simplified version of the Fortezza life cycle (the full version has a more fine-grained number of states) is shown in Figure 7.

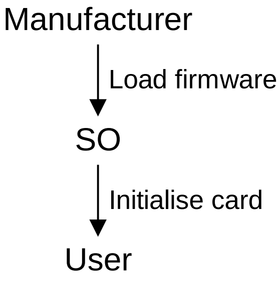

Figure 7: Fortezza card life cycle

A similar function is played by the SO in the Dallas iButton (in iButton terms the crypto officer), who is responsible for initialising the device and setting various fixed parameters, after which they set the initial user PIN and hand the device over to the user. At the point of manufacture the iButton is initialised with the Dallas Primary Feature Set which includes a private key generated in the device when the feature set was initialised. The fixed Primary Feature Set allows the SO to initialise the device and allows the user to check whether the SO has altered various pre-set options. Since the Dallas Primary key is tied to an individual device and can only sign data under control of the iButton internal code, it can be used to guarantee that certain settings are in effect for the device and to guarantee that a given user key was generated in and is controlled by the device. Again, this provides a trusted bootstrap path which allows the user and relying parties to determine with a good degree of confidence that everything is as it should be.

An even more complex secure bootstrap process is used in the IBM 4758. This is a multi-stage process which begins with the layer 0 miniboot code in ROM. This code allows (under carefully controlled conditions) layer 1 miniboot code to be loaded into flash memory, which in turn allows layer 2 system software to be loaded into flash, which in turn loads and runs layer 3 user applications [27][61]. The device contains various hardware-based interlocks which are used to protect the integrity of each layer, during the boot process each boot phase advances a ratchet which ensures that once execution has passed through layer *n* to a lower-privileged layer *n* + 1, it can never move back to layer *n*. As execution moves into higher and higher layers, the capabilities which are available become less and less, so that code at layer *n* + 1 can no longer access or modify resources available at layer *n*. An attempt to reload code at a given layer can only occur under carefully controlled conditions either hardcoded into or configured by the installer of the layer beneath it. A normal reload of a layer (that is, a software update with appropriate authorisation) will leave the other data in that layer intact, an emergency reload (used to initially load code and for emergencies such as code being damaged or non-functional) erases all data such as encryption keys for every layer from the one being reloaded on up. This has the same effect as the Fortezza multi-stage bootstrap where the only way to change initial parameters is to wipe the card and start again from scratch. Going beyond this, the 4758 also has an extensive range of authorisation and authentication controls which allow a trusted execution environment within the device to be preserved.

As discussed in a previous chapter, cryptlib’s flexible security policy can be adapted to enforce at least the simpler Fortezza/iButton-type controls without too much trouble. At present this area has seen little work since virtually all users are working with either a software-only implementation or a dedicated coprocessor under the control of a single user, however in future work the implications of multiuser access to coprocessor resources will be explored. Since cryptlib provides native SSL/TLS and ssh capabilities, it’s likely that multiuser access will be protected with one of these mechanisms, with per-user configuration information being stored using the PKCS #15 format [62] which was originally designed to store information in smart cards and which is ideally suited for this purpose.

3.3 Open vs. Closed-source Coprocessors

There are a number of vendors who sell various forms of tier 2 coprocessor, all of which run proprietary control software and generally go to some lengths to ensure that no outsiders can ever examine it. The usual way in which vendors of proprietary implementations try to build the same user confidence in their product as would be provided by having the source code and design information available for public scrutiny is to have it evaluated by independent labs and testing facilities, typically to the FIPS 140 standard when the product constitutes crypto hardware (the security implications of open source vs. proprietary implementations have been covered exhaustively in various fora and won’t be repeated here). Unfortunately this process leads to prohibitively expensive products (thousands to tens of thousands of dollars per unit) and still requires users to trust the vendor not to insert a backdoor or to accidentally void the security via a later code update or enhancement added after the evaluation is complete (strictly speaking such post-evaluation changes would void the evaluation, but vendors sometimes forget to mention this in their marketing literature). There have been numerous allegations of the former occurring [63][64][65], and occasional reports of the latter.

In contrast, an open source implementation of the crypto control software can be seen to be secure by the end user with no degree of blind trust required. The user can (if they feel so inclined) obtain the raw coprocessor hardware from the vendor of their choice in the country of their choice, compile the firmware and control software from the openly-available source code, and install it knowing that no supplemental functionality known only to a few insiders exists. For this reason the entire suite of coprocessor control software is made available in source code form for anyone to examine, build, and install as they see fit.

A second, far less theoretical advantage of an open-source coprocessor is that until the crypto control code is loaded into it, it isn’t a controlled cryptographic item as crypto source code and software aren’t controlled in most of the world. This means that it’s possible to ship the hardware and software separately to almost any destination (or source it locally) without any restrictions and then combine the two to create a controlled item once they arrive at their destination (like a two-component glue, things don’t get sticky until you mix the parts).

4. Extended Security Functionality

The basic coprocessor design presented so far serves to move all security-related processing and cryptovariables out of reach of hostile software, but by taking advantage of the capabilities of the hardware and firmware used to implement it, it’s possible to do much more. One of the features of the cryptlib architecture is that all operations are controlled and monitored by a central security kernel which enforces a single, consistent security policy across the entire architecture. By tying the control of some of these operations to features of the coprocessor, it’s possible to obtain an extended level of control over its operation as well as avoiding some of the problems which have traditionally plagued this type of security device. While this isn’t a panacea (there are too many ways to get at sensitive information which don’t require any type of attack on the underlying cryptosystem or its implementation [66]), the measures help close some of the more glaring holes.

4.1 Controlling Coprocessor Actions

The most important type of extra functionality which can be added to the coprocessor is extended failsafe control over any actions it performs. This means that instead of blindly performing any action requested by the host (purportedly on behalf of the user), it first seeks confirmation from the user that they have indeed requested that the action be taken. The most obvious application of this mechanism is for signing documents where the owner has to indicate their consent through a trusted I/O path rather than allowing a rogue application to request arbitrary numbers of signatures on arbitrary documents. This contrasts with other tier 1 and 2 processors which are typically enabled through user entry of a PIN or password, after which they are at the mercy of any commands coming from the host. Apart from the security concerns, the ability to individually control signing actions and require conscious consent from the user means that the coprocessor provides a mechanism required by a number of digital signature laws which recognise the dangers inherent in systems which provide an automated (that is, with little control from the user) signing capability.

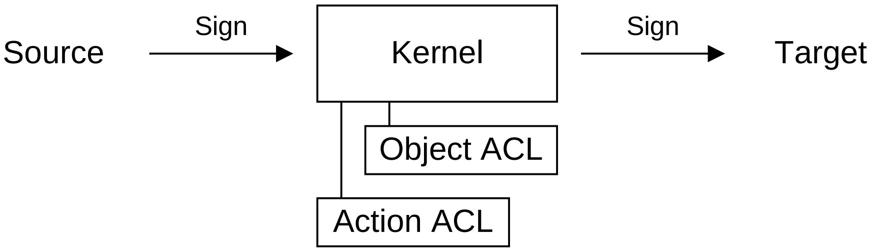

Figure 8: Normal message processing

The means of providing this service is to hook into the cryptlib kernel’s sign action and decrypt action processing mechanisms. In normal processing the kernel receives the incoming message, applies various security-policy-related checks to it (for example it checks to ensure that the object’s ACL allows this type of access), and then forwards the message to the intended target, as shown in Figure 8. In order to obtain additional confirmation that the action is to be taken, the coprocessor can indicate the requested action to the user and request additional confirmation before passing the message on. If the user chooses to deny the request or doesn’t respond within a certain time, the request is blocked by the kernel in the same manner as if the object’s ACL didn’t allow it, as shown in Figure 9. This mechanism is similar to the command confirmation mechanism in the VAX A1 security kernel, which takes a command from the untrusted VMS or Ultrix-32 OSs running on top of it, requests that the user press the (non-overridable) secure attention key to communicate directly with the kernel and confirm the operation (“Something claiming to be you has requested *X*. Is this OK?”), and then returns the user back to the OS after performing the operation [67].

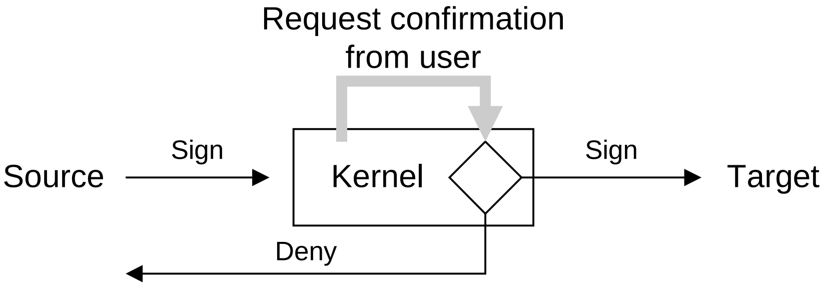

Figure 9: Processing with user confirmation

The simplest form of user interface involves two LEDs and two pushbutton switches connected to a suitable port on the coprocessor (for example the parallel port or serial port status lines). An LED is activated to indicate that confirmation of a signing or decryption action is required by the coprocessor. If the user pushes the confirmation button, the request is allowed through, if they push the cancel button or don’t respond within a certain time, the request is denied.

4.2 Trusted I/O Path

The basic user confirmation mechanism presented above can be generalised by taking advantage of the potential for a trusted I/O path which is provided by the coprocessor. The main use for a trusted I/O path is to allow for secure entry of a password or PIN which is used to enable access to keys stored in the coprocessor. Unlike typical tier 1 devices which assume that the entire device is secure and therefore can afford to use a short PIN in combination with a retry counter to protect cryptovariables, the coprocessor makes no assumptions about its security and instead relies on a user-supplied password to encrypt all cryptovariables held in persistent storage (the only time keys exist in plaintext form is when they’re decrypted to volatile memory prior to use). Because of this, a simple numeric keypad used to enter a PIN isn’t sufficient (unless the user enjoys memorising long strings of digits for use as passwords). Instead, the coprocessor can optionally make use of devices such as PalmPilots for password entry, perhaps in combination with novel password entry techniques such as graphical passwords [68]. Note though that, unlike a tier 0 crypto implementation, obtaining the user password via a keyboard sniffer on the host doesn’t give access to private keys since they’re held on the coprocessor and can never leave it, so that even if the password is compromised by software on the host, it won’t provide access to the keys.

In a slightly more extreme form, the ability to access the coprocessor via multiple I/O channels allows us to enforce strict red/black separation, with plaintext being accessed through one I/O channel, ciphertext through another, and keys through a third. Although cryptlib doesn’t normally load plaintext keys (they’re generated and managed internally and can never pass outside the security perimeter), when the ability to load external keys is required FIPS 140 mandates that they be loaded via a separate channel rather than the one used for general data, which can be provided for by loading them over a separate channel such as a serial port (a number of commercial crypto coprocessors come with a serial port for this reason).

4.3 Physically Isolated Crypto

It has been said that the only truly tamperproof computer hardware is Voyager 2, since it has a considerable air gap (strictly speaking a non-air gap) which makes access to the hardware somewhat challenging (space aliens notwithstanding). We can take advantage of air-gap security in combination with cryptlib’s remote-execution capability by siting the hardware performing the crypto in a safe location well away from any possible tampering. For example by running the crypto on a server in a physically secure location and tunnelling data and control information to it via its built-in ssh or SSL/TLS capabilities, we can obtain the benefits of physical security for the crypto without the awkwardness of having to use it from a secure location or the expense of having to use a physically secure crypto module (the implications of remote execution of crypto from countries like China or the UK (with the RIPA act in force) with keys and crypto being held in Europe or the US are left as an exercise for the reader).

Physical isolation at the macroscopic level is also possible due to the fact that the cryptlib separation kernel has the potential to allow different object types (and, at the most extreme level, individual objects) to be implemented in physically separate hardware. For those requiring an extreme level of isolation and security, it should be possible to implement the different object types in their own hardware, for example keyset objects (which don’t require any real security since certificates contain their own tamper protection) could be implemented on the host PC, the kernel (which requires a minimum of resources) could be implemented on a cheap ARM-based plug-in card, envelope objects (which can require a fair bit of memory but very little processing power) could be implemented on a 486 card with a good quantity of memory, and encryption contexts (which can require a fair amount of CPU power but little else) could be implemented using a faster Pentium-class CPU. In practice though it’s unlikely that anyone would consider this level of isolation worth the expense and effort.

4.4 Coprocessors in Hostile Environments

Sometimes the coprocessor will need to function in a somewhat hostile environment, not so much in the sense of it being exposed to extreme environmental conditions but more that it will need to be able to withstand a larger than usual amount of general curiosity by third parties. The standard approach to this problem is to embed the circuitry in some form of tamper-resistant envelope which in its most sophisticated form has active tamper response circuitry which will zeroise cryptovariables if it detects any form of attack.

Such an environmental enclosure is difficult and expensive to construct for the average user, however there exist a variety of specialised enclosures which are designed for use with embedded systems which are expected to be used under extreme environmental conditions. A typical enclosure of this form, the HiDAN system[3], is shown in Figure 10. This contains a PC104 system mounted on a heavy-duty aluminium-alloy chassis which acts as both a heatsink for the PC and provides a substantial amount of physical and environmental protection for the circuitry contained within it.

[3] HiDAN images copyright Real Time Devices USA, all rights reserved.

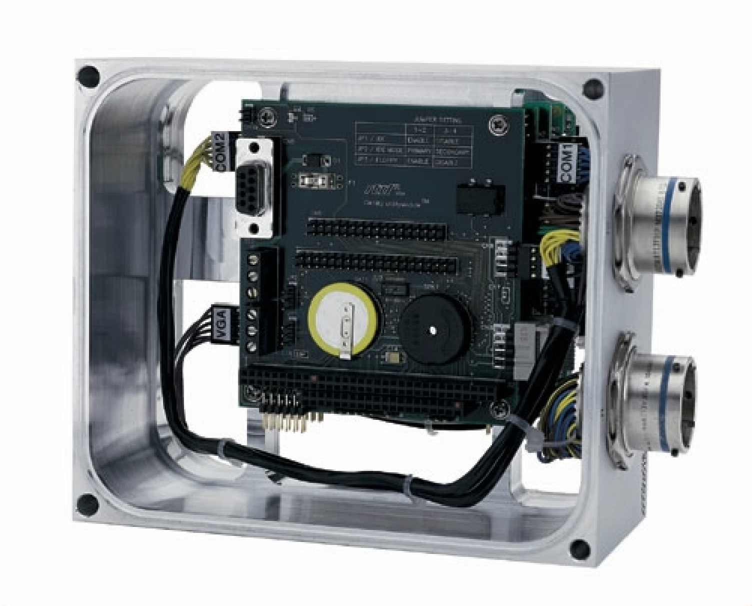

Figure 10: HiDAN embedded PC internals (image courtesy RTD)

This type of enclosure provides a high degree of shielding and isolation for the internal circuitry, with a minimum of 85dB of EMI shielding from 10-100MHz and 80dB of shielding to 1GHz, sufficient to meet a number of TEMPEST emission standards. All I/O is via heavily-shielded milspec connectors, and the assembly contains a built-in power supply module (present in the lower compartment) to isolate the internal circuitry from any direct connection to an external power source. As Figure 11 indicates, the unit is constructed in a manner capable of withstanding medium-calibre artillery fire.

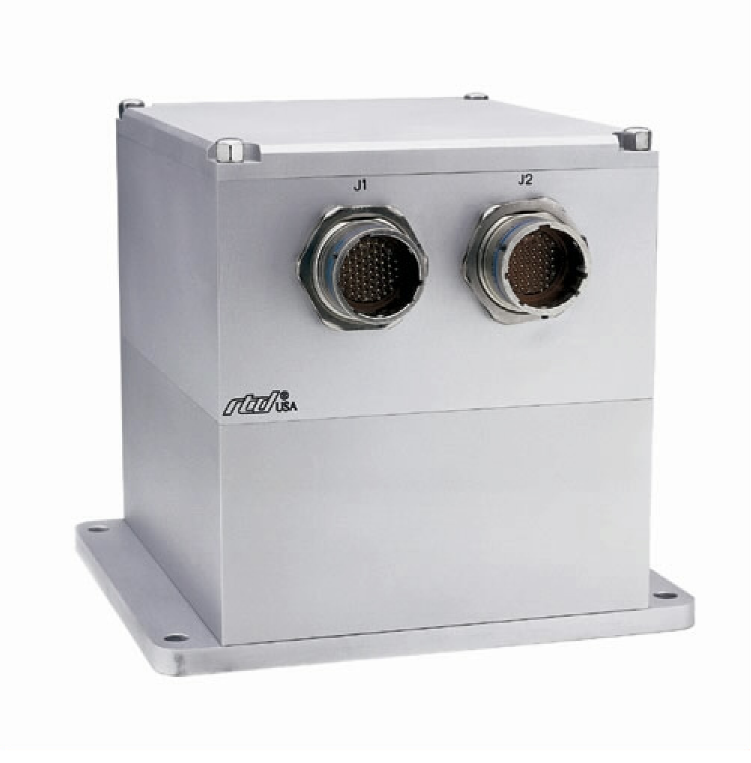

Figure 11: HiDAN embedded PC system (image courtesy RTD)

This type of enclosure can be easily adapted to meet the FIPS 140 level 2 and 3 physical security requirements. For level 2, “the cryptographic module shall provide evidence of tampering (e.g., cover, enclosure, and seal)” (section 4.5.1) and “the cryptographic module shall be entirely contained within a metal or hard plastic production-grade enclosure” (section 4.5.4), requirements which the unit more than meets (the EMI shielding includes a self-sealing gasket compound which provides a permanent environmental seal to a tongue-and-groove arrangement once the case is closed).

For level 3, "the cryptographic module shall be encapsulated within a hard potting material (e.g., a hard opaque epoxy)" (section 4.5.3), which can be arranged by pouring a standard potting mix into the case before it is sealed shut.

5. Crypto Hardware Acceleration

So far the discussion of the coprocessor has focused on the security and functionality enhancements it provides, avoiding any mention of performance concerns. The reason for this is that for the majority of users the performance is good enough, meaning that for typical applications such as email encryption, web browsing with SSL, and remote access via ssh, the presence of the coprocessor is barely noticeable since the limiting factors on performance are set by network bandwidth, disk access times, modem speed, bloatware running on the host system, and so on. Although never intended for use as a special-purpose crypto accelerator of the type capable of performing hundreds of RSA operations per second on behalf of a heavily-loaded web server, it is possible to add extra functionality to the coprocessor through its built-in PC104 bus to extend its performance. By adding a PC104 daughterboard to the device, it's possible to enhance its functionality or add new functionality in a variety of ways, as explained below (although the prices quoted for devices will change over time, the price ratios should remain relatively constant).

5.1 Conventional Encryption/Hashing

Implementing an algorithm like DES, which was originally targeted at hardware implementations, in a field-programmable gate array (FPGA) is relatively straightforward, and hash algorithms like MD5 and SHA-1 can also be implemented fairly easily in hardware by implementing a single round of the algorithm and cycling the data through it the appropriate number of times. Using a low-cost FPGA, it should be possible to build a daughterboard which performs DES and MD5/SHA-1 acceleration for around $50. Unfortunately a number of hardware and software issues conspire to make this non-viable economically. The main problem is that although DES is faster to implement in hardware than in software, most newer algorithms are much more efficient in software (ones with large, key-dependent S-boxes are particularly difficult to implement in FPGAs because they require huge numbers of logic cells, requiring very expensive high-density FPGAs). A related problem is the fact that in many cases the CPU on the coprocessor is already capable of saturating the I/O channel (ethernet/ECP/EPP/PC104) using a pure software implementation, so there's nothing to be gained by adding expensive external hardware (all of the software-optimised algorithms run at several MB/s whereas the I/O channel is only capable of handling around 1MB/s). The imbalance becomes even worse when any CPU faster than the entry-level 5x86/133 configuration is used, since at this point any common algorithm (even the rather slow triple DES) can be executed more quickly in software than the I/O channel can handle. Because of this it doesn't seem profitable to try to augment software-based conventional encryption or hashing capabilities with extra hardware.

5.2 Public-key Encryption

Public-key algorithms are less amenable to implementation in general-purpose CPUs than conventional encryption and hashing algorithms, so there's more scope for hardware acceleration in this area. We have two options for accelerating public-key operations, either using an ASIC from a vendor or implementing our own version with an FPGA. Bignum ASICs are somewhat thin on the ground since the vendors who produce them usually use them in their own crypto products and don't make them available for sale to the public, however there is one company who specialise in ASICs rather than crypto products who can supply a bignum ASIC (it's also possible to license bignum cores and implement the device yourself, this option is covered peripherally in the next section). Using this device, the PCC201 [69], it's possible to build a bignum acceleration daughterboard for around $100.

Unfortunately, the device has a number of limitations. Although impressive when it was first introduced, the maximum key size of 1024 bits and maximum throughput of 21 operations/s for 1024-bit keys and 74 operations/s for 512-bit keys compares rather poorly with software implementations on newer Pentium-class CPU's, which can achieve the same performance with a CPU speed of around 200MHz. This means that although one of these devices would serve to accelerate performance on a coprocessor based on the entry-level 5x86/133 hardware, a better way to utilise the extra expense of the daughterboard would be to buy the next level up in coprocessor hardware, giving somewhat better bignum performance and accelerating all other operations as well as a free side-effect (the entry level for Pentium-class cards is one containing a 266MHz Cyrix MediaGX, although it may be possible to put together an even cheaper one using a bare card and populating it with an AMD Duron 750, currently selling for around $30). A second disadvantage of the

PCC201 is that it’s made available under peculiar export control terms which can make it cumbersome (or even impossible) to obtain for anyone who isn’t a large company.

An alternative to using an ASIC is to implement our own bignum accelerator with an FPGA, with the advantage that we can make it as fast as required (within the limits of the available hardware). Again, there is the problem that much of the published work in the area of bignum accelerator design is by crypto hardware vendors who don’t make the details available, however there is one reasonably fast implementation which achieves 83 operations/s for 1024-bit keys and 340 operations/s for 512-bit keys using a total of 6,700 FPGA basic cells (configurable logic blocks or CLBs) [70]. The use of such a large number of CLBs requires the use of very high-density FPGAs, of which the most widely-used representative is the Xilinx XC4000 family [71]. The cheapest available FPGA capable of implementing this design, the XC40200, comes with a pre-printed mortgage application form and a $2000-$2500 price tag (depending on speed grade and quantity), providing a clue as to why the design has to date only been implemented on a simulator. Again, it’s possible to buy an awful lot of CPU power for the same amount of money (an equivalent level of performance to the FPGA design is obtainable using (in early-2000 prices) about $200 worth of AMD Athlon CPU [72]).

This illustrates a problem faced by all hardware crypto accelerator vendors, which may be stated as a derivation of Moore’s law: AMD and Intel can make it faster cheaper than you can. In other words, putting a lot of effort into designing an ASIC for a crypto accelerator is a risky investment because, aside from the usual flexibility problems caused by the use of an ASIC, it’ll be rendered obsolete by general-purpose CPUs within a few years. This problem is demonstrated by several products currently sold as crypto hardware accelerators which in fact act as crypto handbrakes since, when plugged in or enabled, performance slows down.

For pure acceleration purposes, the optimal price/performance tradeoff appears to be to populate a daughterboard with a collection of cheap CPUs attached to a small amount of memory and just enough glue logic to support the CPU (this approach is used by nCipher, who use a cluster of ARM CPUs in their SSL accelerators [73]). The mode of operation of this CPU farm would be for the crypto coprocessor to halt the CPUs, load the control firmware (a basic protected-mode kernel and appropriate code to implement the required bignum operation(s)) into the memory, and restart the CPU running as a special-purpose bignum engine. For x86 CPUs, there are a number of very minimal open-source protected-mode kernels which were originally designed as DOS extenders for games programming available, these ignore virtual memory, page protection, and other issues and run the CPU as if it were a very fast 32-bit real-mode 8086. By using a processor like a Duron which contains 128K of onboard level 1 cache (running at the full CPU speed), the control code can be loaded initially from slow, cheap external memory but will execute from cache at full speed from then on. Each of these dedicated bignum units should be capable of ~500 512-bit RSA operations per second, with the added benefit that they can run standard applications when they’re not acting as crypto accelerators.

Unfortunately the use of commodity x86 CPUs of this kind has several disadvantages. The first is that they are designed for use in systems with a certain fixed configuration (for example SDRAM, PCI and AGP busses, a 64-bit bus interface, and other high-performance options) which means that using them with a single cheap 8-bit memory chip requires a fair amount of glue logic to fake out the control signals from the external circuitry which is expected to be present. The second problem is that these CPU’s consume significant amounts of power and dissipate a large amount of heat, with current drains of 10-15A and dissipations of 20-40W being common for the range of low-end processors which might be used as cheap accelerator engines. Adding more CPUs to improve performance only serves to exacerbate this problem, since the power supplies and enclosures designed for embedded controllers are completely overwhelmed by the requirements of a cluster of these processors. Although the low-cost processing power offered by general-purpose CPU’s appears to make them ideal for this situation, the practical problems they present rules them out as a solution.

A final alternative is offered by digital signal processors (DSPs), which require virtually no external circuitry since most newer ones contain enough onboard memory to hold all data and control code, and don’t expect to find sophisticated external control logic present. The fact that DSPs are optimised for embedded signal-processing tasks makes them ideal for use as bignum accelerators, since a typical configuration contains two 32-bit single-cycle multiply-accumulate (MAC) units which provide in one instruction the most common basic operation used in bignum calculations. The best DSP choice appears to be the ADSP-21160, which consumes only 2 watts and contains built-in multiprocessor support allowing up to 6 DSPs to be combined into one cluster [74]. The aggregate 3,600 MFLOPS processing power provided by one of these clusters should prove sufficient (in its integer equivalent) to accelerate bignum calculations.

5.3 Other Functionality

In addition to pure acceleration purposes, it's possible to use a PC104 add-on card to handle a number of other functions. The most important of these is a hardware random number generator (RNG), since the effectiveness of the standard entropy-polling RNG using by cryptlib is somewhat impaired by its use in an embedded environment. A typical RNG would take advantage of several physical randomness sources (typically thermal noise in semiconductor junctions) fed into a Schmitt trigger with the output mixed into the standard cryptlib RNG. The use of multiple independent sources ensures that even if one fails the others will still provide entropy, and feeding the RNG output into the cryptlib PRNG ensures that any possible bias is removed from the RNG output bits.

A second function which can be performed by the add-on card is to act as a more general I/O channel than the basic LED-and-pushbutton interface described earlier, providing the user with more information (perhaps via an LCD display) on what it is they're authorising.

6. Conclusion

This chapter has presented a design for an inexpensive, general-purpose cryptlib-based cryptographic coprocessor which is capable of keeping crypto keys and crypto processing operations safe even in the presence of malicious software on the host which it is controlled from. Extended security functionality is provided by taking advantage of the presence of trusted I/O channels to the coprocessor. Although sufficient for most purposes, the coprocessor's processing power may be augmented through the addition of additional modules based on DSPs which should bring the performance into line with considerably more expensive commercial equivalents. Finally, the open-source nature of the design and use of COTS components means that anyone can easily reassure themselves of the security of the implementation and can obtain a coprocessor in any required location by refraining from combining the hardware and software components until they're at their final destination.

7. References

[1] "Inside Windows NT", Helen Custer, Microsoft Press, 1993.

[2] "Playing Hide and Seek with Stored Keys", Nicko van Someren and Adi Shamir, 22 September 1998, presented at Financial Cryptography 1999.

[3] "Monitoring System Events by Subclassing the Shell", Eric Heimburg, *Windows Developers Journal*, **Vol.9**, **No.2** (February 1998), p.35.

[4] "Windows NT System-Call Hooking", Mark Russinovich and Bryce Cogswell, *Dr.Dobbs Journal*, January 1997, p.42.

[5] "Win NT 4.0 UserId and Password available in memory", Russ Osterlund, posting to the ntbugtraq mailing list, message-ID `C12566CD.00485E7F.00@ZurichNotes.com`, 1 December 1998.

[6] "In Memory Patching", Stone / UCF & F4CG, 1998

[7] "A *REAL* NT Rootkit, Patching the NT Kernel", Greg Hoglund, *Phrack*, **Vol.9**, **Issue 55**.

[8] "Design a Windows NT Service to Exploit Special Operating System Features", Jeffrey Richter, *Microsoft Systems Journal*, **Vol.12**, **No.10** (October 1997), p.19.

[9] "A Programming Fusion Technique for Windows NT", Greg Hoglund, SecurityFocus.com forum, guest feature `http://www.securityfocus.com/templates/-forum_message.html?forum=2&head=2137&id=557`, 14 December 1999.

[10] "Securing Java and ActiveX", Ted Julian, Forrester Report, *Network Strategies*, **Vol.12**, **No.7** (June 1998).

[11] "Death, Taxes, and Imperfect Software: Surviving the Inevitable", Crispin Cowan and Castor Fu, *Proceedings of the ACM New Security Paradigms Workshop '98*, September 1998.

[12] "User Friendly, 6 March 1998", Illiad, 6 March 1998, `http://www.userfriendly.org/cartoons/archives/98mar/19980306.html`.

[13] "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments", Peter Loscocco, Stephen Smalley, Patrick Muckelbauer, Ruth Taylor, S.Jeff Turner,

and John Farrell, *Proceedings of the 21st National Information Systems Security Conference*, (formerly the National Computer Security Conference), October 1998, CDROM distribution.

[14] "The Importance of High Assurance Computers for Command, Control, Communications, and Intelligence Systems", W. Shockley, R. Schell, and M.Thompson, *Proceedings of the 4th Aerospace Computer Security Applications Conference*, December 1988, p.331.

[15] Jeff Schiller, quoted in *Communications of the ACM*, **Vol.42**, **No.9** (September 1999), p.10.

[16] "Software Security in an Internet World: An Executive Summary", Timothy Shimeall and John McDermott, *IEEE Software*, **Vol.16**, **No.4** (July/August 1999), p.58.

[17] "Formal Methods and Testing: Why the State-of-the-Art is Not the State-of-the-Practice", David Rosenblum, *ACM SIGSOFT Software Engineering Notes*, **Vol21**, **No.4** (July 1996), p.64.

[18] "The Process of Security", Bruce Schneier, *Information Security*, **Vol.3**, **No.4** (April 2000), p.32.

[19] "Optimizing Preventive Service of Software Products", Edward Adams, *IBM Journal of Research and Development*, **Vol.28**, **No.1** (January 1984), p.2.

[20] "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing", Justin Forrester and Barton Miller, *Proceedings of the 4th USENIX Windows Systems Symposium*, August 2000.

[21] "How Did Software Get So Reliable Without Proof?", C.A.R.Hoare, *Proceedings of the 3rd International Symposium of Formal Methods Europe (FME'96)*, Springer-Verlag Lecture Notes in Computer Science No.1051, 1996, p.1

[22] "How to bypass those pesky firewalls", Mark Jackson, in *Risks Digest*, **Vol.20**, **No.1**, 1 October 1998.

[23] "Data Encryption Standard", FIPS PUB 46, National Institute of Standards and Technology, 22 January 1988.

[24] "General Security Requirements for Equipment Using the Data Encryption Standard", Federal Standard 1027, National Bureau of Standards, 14 April 1982.

[25] "Data Encryption Standard", FIPS PUB 46-2, National Institute of Standards and Technology, 30 December 1993.

[26] "Security Requirements for Cryptographic Modules", FIPS PUB 140, National Institute of Standards and Technology, 11 January 1994.

[27] "Building a High-Performance Programmable, Secure Coprocessor", Sean Smith and Steve Weingart, *Computer Networks and ISDN Systems*, **Issue 31** (April 1999), p.831.

[28] "Building the IBM 4758 Secure Coprocessor", Joan Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, Sean Smith, and Steve Weingart, *IEEE Computer*, **Vol.34**, **No.10** (October 2001), p.57.

[29] "Fortezza Program Overview, Version 4.0a", National Security Agency, February 1996.

[30] "iButton Home Page", `http://www.ibutton.com`.

[31] "A Tentative Approach to Constructing Tamper-Resistant Software", Masahiro Mambo, Takanori Murayama, and Eiji Okamoto, *Proceedings of the ACM New Security Paradigms Workshop '97*, September 1997.

[32] "Evaluation of Tamper-Resistant Software Deviating from Structured Programming Rules", Hideaki Goto, Masahiro Mambo, Hiroki Shizuya, and Yasuyoshi Watanabe, *Proceedings of the 6th Australian Conference on Information Security and Privacy (ACISP'01)*, Springer-Verlag Lecture Notes in Computer Science No.2119, 2001, p.145.

[33] "Common Data Security Architecture", Intel Corporation, 2 May 1996.

[34] "The Giant Black Book of Computer Viruses (2nd ed)", Mark Ludwig, American Eagle Publications, 1998.

[35] "Understanding and Managing Polymorphic Viruses", Symantec Corporation, 1996.

[36] "Fravia's Page of Reverse Engineering", `http://www.fravia.org`.

[37] “Phrozen Crew Official Site”, `http://www.phrozencrew.com/index2.htm`.

[38] “Stone’s Webnote”, `http://www.users.one.se/~stone/`.

[39] “Common Security: CDSA and CSSM, Version 2”, CAE specification, The Open Group, November 1999.

[40] “The Human Immune System as an Information Systems Security Reference Model”, Charles Cresson Wood, *Computers and Security*, **Vol.6**, **No.6** (December 1987), p.511.

[41] “A model for detecting the existence of software corruption in real time”, Jeffrey Voas, Jeffery Payne, and Frederick Cohen, *Computers and Security*, **Vol.12**, **No.3** (May 1993), p.275.

[42] “A Biologically Inspired Immune System for Computers”, Jeffrey Kephart, *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, MIT Press, 1994, p.130.

[43] “Principles of a Computer Immune System”, Anil Somayaji, Steven Hofmeyr, and Stephanie Forrest, *Proceedings of the 1997 New Security Paradigms Workshop*, ACM, 1997, p.75.

[44] “On the (Im)possibility of Obfuscating Programs (Extended Abstract)”, Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yung, *Proceedings of Crypto 2001*, Springer-Verlag Lecture Notes in Computer Science No.2139, 2001, p.1.

[45] “Common Security Protocol (CSP)”, ACP 120, 8 July 1998.

[46] “Cryptographic API’s”, Dieter Gollman, *Cryptography: Policy and Algorithms*, Springer-Verlag Lecture Notes in Computer Science No.1029, July 1995, p.290.

[47] “The VMEbus Handbook”, VMEbus International Trade Association, 1989.

[48] “PC/104 Specification, Version 2.3”, PC/104 Consortium, June 1996.

[49] “PC/104-Plus Specification, Version 1.1”, PC/104 Consortium, June 1997.

[50] “EZ Dos Web Site”, `http://members.aol.com/RedHtLinux/`.

[51] “The FreeDOS Project”, `http://www.freedos.org`.

[52] “OpenDOS Unofficial Home Page”, `http://www.deltasoft.com/opendos.htm`.

[53] “PicoBSD, the Small BSD”, `http://www.freebsd.org/~picobsd/picobsd.html`.

[54] “Embedded Linux”, `http://www.linuxembedded.com/`.

[55] “DiskOnChip 2000: MD2200, MD2201 Data Sheet, Rev.2.3”, M-Systems Inc, May 1999.

[56] “Secure Deletion of Data from Magnetic and Solid-State Memory”, Peter Gutmann, *Proceedings of the 6th Usenix Security Symposium*, July 1996.

[57] “Physical access to computers: can your computer be trusted?”, Walter Fabian, *Proceedings of the 29th Annual International Carnahan Conference on Security Technology*, October 1995, p.244.

[58] “IEEE Std.1284-1994: Standard Signaling Method for a Bi-Directional Parallel Peripheral Interface for Personal Computers”, IEEE, March 1994.

[59] “PCI-ISA Passive Backplace: PICMG 1.0 R2.0”, PCI Industrial Computer Manufacturers Group, 10 October 1994.

[60] “Interface Control Document for the Fortezza Crypto Card, Revision P1.5”, National Security Agency, 22 December 1994.

[61] “Application Support Architecture for a High-Performance, Programmable Secure Coprocessor”, Joan Dyer, Ron Perez, Sean Smith, and Mark Lindemann, *Proceedings of the 22nd National Information Systems Security Conference* (formerly the National Computer Security Conference), October 1999, CDROM distribution.

[62] “PKCS #15 v1.1: Cryptographic Token Information Syntax Standard (draft)”, RSA Laboratories, 4 May 2000.

[63] “Verschlüsselt: Der Fall Hans Buehler”, Res Strehle, Werd Verlag, Zurich, 1994.

[64] “No Such Agency, Part 4: Rigging the Game”, Scott Shane and Tom Bowman, *The Baltimore Sun*, 4 December 1995, p.9.

[65] “Wer ist der befugte Vierte? Geheimdienste unterwandern den Schutz von Verschlüsselungsgeräten”, *Der Spiegel*, No.36, 1996, p.206.

[66] “Beyond Cryptography: Threats Before and After”, Walter Fabian, *Proceedings of the 32nd Annual International Carnahan Conference on Security Technology*, October 1998, p.97.

[67] “A Retrospective on the VAX VMM Security Kernel”, Paul Karger, Mary Ellen Zurko, Douglas Bonin, Andrew Mason, and Clifford Kahn, *IEEE Transactions on Software Engineering*, **Vol.17**, **No.11** (November 1991), p.1147.

[68] “The Design and Analysis of Graphical Passwords”, Ian Jermyn, Alain Mayer, Fabian Monrose, Michael Reiter, and Aviel Rubin, *Proceedings of the 8th Usenix Security Symposium*, August 1999.

[69] “Pijnenburg Product Specification: Large Number Modular Arithmetic Coprocessor, Version 1.04”, Pijnenburg Custom Chips B.V., 12 March 1998.

[70] “Modular Exponentiation on Reconfigurable Hardware”, Thomas Blum, MSc thesis, Worcester Polytechnic Institute, 8 April 1999.

[71] “XC4000XLA/XV Field Programmable Gate Arrays, v1.3”, Xilinx, Inc, 18 October 1999.

[72] “Apache e-Commerce Solutions”, Mark Cox and Geoff Thorpe, ApacheCon 2000, March 2000.

[73] nCipher, `http://www.ncipher.com`.

[74] “ADSP-21160 SHARC DSP Hardware Reference”, Analog Devices Inc, November 1999.