The TrustNo 1 Cryptoprocessor Concept

Markus Kuhn — kuhn@cs.purdue.edu — 1997-04-30

Abstract: Cryptoprocessors feature an on-chip block cipher hardware between the cache and the bus interface. Code and data are decrypted on-the-fly while being fetched from RAM and are encrypted while being written into RAM. Even someone with full physical access to the printed circuit board cannot observe the executed cleartext software and its data structures. Cryptoprocessors have been used for many years as microcontrollers in security sensitive applications like financial transaction terminals. This paper explores the hardware, firmware, operating system, and key management mechanisms necessary in order to apply the cryptoprocessor concept in multitasking operating system workstations.

Introduction

Technical protection of software against unauthorized execution is usually performed by binding the usability of the software to a piece of hardware that only the legitimate owner of a software license has available. Commonly used mechanisms include small tamper-resistant plug-in modules ("dongles") that implement a cryptographic challenge-response authorization protocol, recording media with a special formatting that the operating system I/O functions can detect but not reproduce, and unique workstation serial numbers hardwired into the firmware that application programs can check. These software license protection mechanisms are today widely used in applications with a high risk of software piracy such as computer games or very expensive special purpose software (e.g., CAD, expert systems, computer security tools). These techniques do not prevent the unauthorized reverse engineering and modification of software, and the protection mechanism itself can relatively easily be deactivated by a skilled programmer who analyzes the machine code and removes the protection functions. They also do not prevent system-wide cryptographic keys and certificates that might have to be stored in the software against observation and modification.

In order to not only prevent the execution of a program on other machines, but to protect the entire software code from any access, we require a security perimeter that keeps unauthorized reverse engineers from observing the memory and the execution of instructions. Solutions range from putting the whole computer into a locked room to storing the entire protected software in a single chip, as this is done in smartcard applications. Both extremes have their drawbacks: few people can use a computer application if the hardware is locked away, and only very tiny applications fit into a single chip. An intermediate approach was demonstrated in IBM's μABYSS project [8,9]. Here, the security perimeter protects a single printed circuit board inside a workstation. The operating system and cryptographic keys are stored in battery buffered static RAM chips that are located on the same board as the CPU, the system bus, the harddisk controller, a real-time clock, and a battery. The board is surrounded from all sides by an alarm mechanism that consists of a dense multi-layer winding pattern of a pair of fine wires, which is embedded in hard opaque epoxy resin. Any attempt to open the security package will trigger the alarm mechanism and wipe out the software and encryption keys stored in the battery buffered RAM. The software is stored on external mass storage devices in encrypted form and is decrypted by the operating system when loaded into RAM. Various cryptographic protocols allow the installation of new software and the transfer of software to other systems while the number of usable copies is under the control of the software vendor and the owner of the hardware gets never access to the cleartext software.

The concept of a bus-encryption microprocessor that encrypts both the data and address bus and stores only encrypted values in external memory for the protection of software was first described by Best [1,2,3,4]. In this approach, the security perimeter is reduced from the entire circuit board as with μABYSS to just the CPU chip. Software is stored in encrypted form in external RAM chips and mass storage devices, and it exists in decrypted form only inside the CPU chip in the cache and instruction decoder logic. The encryption key is stored in a protected register inside the CPU that is designed such that the key is destroyed by attempts to read it out using even sophisticated microelectronics testing equipment. The protected register is usually a static RAM that is buffered by an external battery and that keeps the key value throughout the entire lifetime of the system.

Bus encryption processors have commercially successfully been used for almost a decade in the form of 8-bit microcontrollers such as the Dallas Semiconductor

DS5000 series. They have been applied for instance in credit card terminals, automatic teller machines, pay-TV access control devices, and communication encryption modules.

A highly interesting new application field of bus encryption microprocessors that has not yet been studied are the main processors in powerful personal computers and workstations with virtual memory management and multitasking operating systems, including up-coming set-top boxes for multimedia entertainment applications. We will have a closer look at the possible design of such a processor in the remainder of this paper.

Microcontrollers contain usually one single application software and often not even a separate operating system. In a multitasking environment, multiple processes that are encrypted by different keys must be handled simultaneously. It is therefore necessary, not to simply use a single key K for the entire address space like in bus-encryption microcontrollers, but to provide for separate keys per process and memory segment. In order to achieve this, the bus encryption logic has to be integrated with the memory management unit.

We will now have a more detailed look at the hypothetical "*TrustNo 1*" security microprocessor design. It is a design for applications, in which we do not even trust the operating system, since it might like any complex software system feature numerous security vulnerabilities and since it might have been modified by our adversary. In addition, we do not trust any system hardware since it is observable by the attacker and might even have been modified, for example for a RAM emulation attack. The concept presented here allows to achieve a highly effective copy protection for valuable software. In addition, security critical software can be protected against manipulations, even if the attacker has full access to the hardware except the internals of the CPU chip itself.

Hardware Foundations

A suitable basis for the design of the *TrustNo 1* would be a processor like the Intel i386, which allows to partition the virtual memory not only into pages but also into segments. There, a memory address consists of a segment selector and an offset. The segment selector identifies a segment descriptor, which is a table entry that describes the accessed memory segments by attributes such as its base address, length, and whether this segment is writable or executable. When accessing an address, the processor gets the segment descriptor identified by the segment selector, compares the offset with the segment length, checks the access permissions, and adds the offset to the base address. This results in what is called the linear address, which will be converted using a page table into a physical address.

Our processor features as protected on-chip memory a key table that can store n segment keys K_i $(0 < i \leq n)$. A key table size of for instance $n = 255$ should normally be sufficient. This key table can only be accessed by a firmware that is also stored in protected on-chip memory. Not even the operating system kernel in supervisor mode has any access to the content of the key table. The operating system can only call firmware functions that access the key table according to the rules described below. The segment descriptors are extended by a new field, which contains the index i of a key K_i found in the key table that is used to decrypt this segment. The special index value $i = 0$ indicates that this segment is not encrypted.

While loading the content of a cache line into the on-chip cache, the memory logic can either transfer the data as it is stored externally, or it can decrypt it. We use a write-back cache, i.e., every write access is first performed into the cache and only at a later time, for instance when the memory occupied by the cache line is needed to hold a different portion of the main memory, the entire cache line will be written back into the external RAM. Each line of the cache is associated with a key index i, which holds the information if $(i \neq 0)$ and with which key K_i this cache line has been decrypted when it was loaded from RAM and has therefore to be encrypted again before being written back to external RAM. When the processor accesses a memory location for which the content is already located in the cache, then it will first check whether the key index in the segment descriptor is equal to the key index associated with the cache line. If this is the case, the processor can immediately access the data in the cache. If the key indices in the segment descriptor and the cache line do not match, then the cache line will be written back to RAM under the key index associated with the old cache line and reloaded under the required new key index in the segment descriptor, which will then be loaded into the key index register associated with the cache line. If the accessed memory location is not present in the cache, then we load the required cache line into the cache while applying the decryption indicated by the segment descriptor key index and update the cache line key index accordingly.

The encryption units operate in electronic codebook mode (ECB) and treat entire cache lines as input and output blocks. The key supplied to the encryption

function is a combination of the key K_i found in the processors key table, and the more significant address bits that identify the cache lines. This way, every cache line is effectively encrypted using a different key. Extending the block size to the entire cache line ensures that an attacker can not perform a cipher instruction search attack and tabulate the entire per byte codebook of the encryption function, as this was possible for cryptoprocessors with a too small block size [10]. The application of stream ciphers are not feasible for cryptoprocessors, as the main memory is accessed in a random access fashion, and not in a predictable linear sequence.

The memory manager of the processor has to ensure that read, write, or execution access to data stored in a segment encrypted under a key K_i is only granted if the machine instruction that performed the access—or in the case of execution simply the previously executed instruction—has also been fetched from a segment that is encrypted with K_i. This prevents even the operating system from reading or modifying the cleartext of an encrypted segment and from calling parts of the code in an uncontrolled fashion. Jumps into code in the encrypted segment is only allowed at certain gate locations, for instance only at offset 0. This ensures that the encrypted program has control over under which circumstances parts of it can be called. The firmware of the processor is as well located in encrypted ROM and can be called by the operating system at certain entry points, but cannot be read.

Secure context switches

If during the execution of instructions that have been fetched from an encrypted code segment an interrupt occurs, then the entire state Z of the processor that is related to the current thread of execution including the program counter and the register content will automatically be transfered to an on-chip temporary storage T that is not accessible to the operating system. After this, the registers will be initialized before the interrupt handler of the operating system is called. This way, the processor can resume the previous thread of execution after the interrupt handler has finished, without giving the operating system any chance to see any register values of the process executing in the encrypted segment. After the interrupt handler has finished its job, the processor just restores the processor state Z that had been stored in T. The temporary storage T is actually a small LIFO memory, which allows even interrupts of higher priority to interrupt the encrypted handlers of lower priority interrupts.

In operating systems with preemptive scheduling, it is possible that an interrupt handler will not return control over the processor to the interrupted thread of execution. Therefore, we need a way to store the content of T in a process descriptor table that is maintained by the operating system in a way that does not endanger the confidentiality and integrity of the state of the process. For this purpose, our processor features a special `SAVE_STATE` instruction, which pops the top processor state from the stack T, encrypts it, and stores it at a location that is specified as a parameter of this function. Similarly, a second special instruction `RESTORE_STATE` decrypts the processor state loaded from a specified location and reactivates this state. This way, an interrupt handler can perform a context switch between application processes without getting access to the protected register contents of these processes.

We also need a mechanism for ensuring that a state that has been saved this way can only be reactivated once. Otherwise, the operating system could without agreement of the protected application execute unauthorized loops by simply reactivating the protected process at the same state and aborting it shortly later, for instance due to a page fault. This could be used by an attacker to bring the protected software in an insecure state, or to work around software license enforcement mechanisms that limit the number of executions of certain program functions.

As a counter measure, our processor features an on-chip table with an entry H_j for every protected thread of execution j. This table can also not be accessed directly by the operating system. The `SAVE_STATE` command has an additional parameter j, an index of the currently executed thread. The `SAVE_STATE` instruction generates a random key and stores it in H_j. This key is generated using a cryptographic hardware pseudo random number generator that is continuously spiced by an on-chip noise source in order to ensure a continuous flow of high entropy randomness into the generator state. This random number generator is also available to application software for the generation of cryptographic keys and nonces. `SAVE_STATE` then adds some redundant information like a checksum or just a few zero bytes to the state in T and encrypts both together with the key H_j and stores the result at the specified address in main memory. The `RESTORE_STATE` instruction also gets the thread index j as a parameter, as well as the location of the encrypted state in main memory. It decrypts the stored processor state with H_j, then clears the value H_j, and checks the redundant information in the decrypted state. If

it is intact, the stored processor state will be reactivated and the thread can resume its execution. Since `RESTORE_STATE` deletes the value H_j to which the operating system never had access, the thread cannot be reactivated again from the same state, because without the H_j value, the state will not be reconstructed correctly, which the processor can detect using the redundant information that is part of the state, and therefore the `RESTORE_STATE` instruction will fail.

In order to create new threads of execution, application processes use a special instruction `TRANSFER_STATE` in order to copy the current state to T, but unlike during an interrupt, the registers are not initialized. It can then call an operating system function that will find a new thread index j and will store the state in T using `SAVE_STATE` in a new entry in the thread descriptor table. Similar to the return value of a Unix *fork()* system call, `TRANSFER_STATE` sets a status bit in the saved state that is not set in the actual status bit. This way, the application can distinguish between the old and the new thread.

For system calls to the operating system, we use a special `SUPERVISOR_CALL` instruction, which like in a classical processor design switches into a special privileged context of the operating system kernel, and which in addition also hides the current processor state from the operating system similar to what happens during an interrupt. A major difference between an interrupt and the execution of a `SUPERVISOR_CALL` is that the latter does not initialize all register values. This is the responsibility of the application software, since some of the registers will be used to pass parameters to the system call. The transfer of larger pieces of data can be performed using unencrypted data segments to which both the application and the operating system kernel have unrestricted access.

Key management

Let us assume, we want to cryptographically enforce a software license that will the buyer of a software not allow to use it concurrently on more than one processor. The software vendor V sells a usage license for the software S to be used on the processor P. In order to do this, V has to generate for S a distribution key K_S and distribute S only encrypted by K_S. The encryption algorithm is the same algorithm that is implemented in the bus encryption unit of the processor type for which S has been designed. In order to allow the customer to use the software S on her processor P, the vendor V has to make K_S available to P in a secure way that will not allow the buyer or anyone else to execute S on another processor in addition.

This key transfer problem is an application for asymmetric or public key cryptosystems, i.e. encryption and decryption functions where the key information is generated as a pair (K, K^{-1}), such that data that has been encrypted with K (the public key) can only be decrypted with K^{-1} (the secret key) or vice-versa and such that finding a matching K^{-1} for a given K is extremely difficult. We will denote the result of encrypting or decrypting the message M with the key K by $\{M\}_K$.

One fundamental assumption in this scheme is that the software vendor V who distributes protected software for the *TrustNo 1* processor trusts the manufacturer C of this processor in that this manufacturer will not support or participate in application program reverse engineering and software piracy. Like with most security systems, the user has to rely on the trustworthiness of the system manufacturer and we have to hope that market forces will keep untrustworthy manufacturers out of doing business for a long time. The chip producer C of the security processor could always easily get access to any software protected under this scheme if he wants.

First, C generates its public and secret key pair (K_C, K_C^{-1}) and makes sure that K_C is well known at all participating software vendors. Each software vendor can decide, which chip manufacturer C is trustworthy in a configuration list of its distribution system.

The manufacturer has generated for any individual *TrustNo 1* security processor P an asymmetric key pair (K_P, K_P^{-1}), as well as a secret key K'_P for a symmetric cryptosystem. The additional symmetric key is used because currently known symmetric cryptosystems are much more efficient in key length and throughput than currently known asymmetric systems. The key K_P is publicly known, i.e. the operating system can call a firmware function of the processor in order to get a copy of it, and the processor manufacturer could even print it as a sort of serial number on the chip package. In contrast, K_P^{-1} and K'_P are stored in a well-protected security register inside the processor that is designed to be destroyed by any attempt to open the processor package [7]. Not even the operating system or any other non-firmware instruction can access these secrets. Apart from the processor's public key K_P, the operating system can also request from the processor a certificate $\{K_P\}_{K_C^{-1}}$ that has been precomputed by the manufacturer and is stored in the firmware. It can be used to proof to the software vendor that K_P is really the public key of a processor that has been generated by the trusted

processor manufacturer C and not by some intruder.

The vendor V generates for his software product S a single key K_S for the bus encryption algorithm and makes the encrypted software $\{S\}_{K_S}$ commonly available, for instance downloadable over public network servers or on cheap portable storage media (CD-ROM). Since S has been encrypted, nobody can use it directly. If the owner of processor P wants to purchase a usage license for S, then she has to send to the vendor of S her processor key K_P and the manufacturer certificate $\{K_P\}_{K_C^{-1}}$. Then she pays for the software and gets in return the encrypted key $\{K_S\}_{K_P}$.

The operating software running on P now has a copy of the encrypted software $\{S\}_{K_S}$ as well as the purchased license key $\{K_S\}_{K_P}$. In order to install the software, the operating system sends $\{K_S\}_{K_P}$ to the protected firmware of P, which will then calculate $\{\{K_S\}_{K_P}\}_{K_P^{-1}} = K_S$. This asymmetric cryptosystem application is quite time consuming and $\{K_S\}_{K_P}$ is a relatively large (e.g., 128 bytes) long data packet. Therefore we perform this calculation not for every invocation of S, but only once during installation. The firmware hands back to the operating system the more compact (e.g., 10 bytes) and quickly decryptable key $\{K_S\}_{K'_P}$, which can be stored together with $\{S\}_{K_S}$ on the mass storage device.

In a real implementation, $\{S\}_{K_S}$ would not simply be a single big chunk of binary data that has been entirely encrypted with K_S, but it would be a file format with a cleartext header that contains information about the number, lengths, and types of segments needed by S, as well as for every preloaded code and data segment the with K_S encrypted content. When S is to be started, the operating system opens according to the information in the file header of S the required new memory segments in non-encrypted mode and fills them with the corresponding contents of $\{S\}_{K_S}$. Then it selects a free key index i and uses a firmware call with the parameters i and $\{K_S\}_{K'_P}$ in order to load K_S into the on-chip key table entry K_i. Then, the operating system writes the key index i into the segment descriptors of S, and from this point on, the processor can see the cleartext of S in those segments, but only while executing commands inside these segments.

Before we can start the new process, the operating system also has to create a new thread table entry. We have to find a free entry H_j in the processor's thread key table, in which we will store the key that will encrypt the register state at the next thread context switch. The thread table of the operating system contains apart from the usual entries also a field for storing j as well as a location for storing the encrypted processor state.

After all these preparations, we can call the software S with suitable register values over the allowed entry gates in the code segment.

Revocation of licenses

We can even implement with little additional cost a mechanism that allows the revocation of software licenses, or their transfer to other processors. For this, each processor P needs a "black list", a small on-chip write-once memory table, into which the owner of P can put via some firmware function an entry for every software S which she does not want to use any more on P. In order to revoke a software license, we give the license $\{K_S\}_{K_P}$ that we have originally purchased from the vendor V to a revocation firmware routine, which will calculate the shorter key $\{K_S\}_{K'_P}$ and will store it in the black list. In addition, this firmware routine will erase any key table entry $K_i = K_S$ that might still be there.

Each time, when the firmware receives a data packet $\{K_S\}_{K'_P}$ in order to set a new K_i entry in the key table to K_S, it will first check, whether this data packet has not yet already been added to the black list. In this case, it will prevent that K_S can be written into the processor key table. Since the black list is an unerasable memory, entered licenses can never be removed again without destroying the processor. This means that the processor P will never again execute any software that has been encrypted with K_S, and the owner of P has effectively terminated the usage license for S. The firmware has a function that generates a signed certificate $B = \{h(K_S)\}_{K_P^{-1}}$ for any entry $\{K_S\}_{K'_P}$ in the blacklist, where h is a cryptographic one-way hash function. The owner of the processor can now send B together with K_P back to the software vendor V. V will find in his customer database the license agreement for the processor P identified by K_P. By calculating $\{B\}_{K_P}$ and verifying that indeed $\{B\}_{K_P} = h(K_S)$, the vendor can be sure that the processor P has rendered the software license unusable. V can now for instance stop charging a monthly usage fee, or he can offer to his customer a new license $\{K_S\}_{K_Q}$ for her new processor Q if the customer has bought a new computer and wants to use the software now on the new machine instead.

In case a customer has accidentally written a value $\{K_S\}_{K'_P}$ into the black list, then it is not sufficient if the vendor simply issues a new license $\{K_S\}_{K_P}$, as this would be exactly the same now invalid license

that she knew already before. Instead, the vendor would have to reencrypt the entire software for this customer with a new key K'_S and redeliver the entire (potentially huge) software package. In order to avoid this problem, a real implementation could insert another symmetric key layer between the K_S and K_P encryption of S, such that P uses K_P^{-1} in order to decrypt a license key K_L, which is then again used to decrypt $\{K_S\}_{K_L}$. This way, only $\{K_L\}_{K'_P}$ will be entered into the black list, and the next license will have a different K_L value and the software will not have to be completely reencrypted.

In case a processor P breaks down before it can generate license revocation certificates for all programs licensed for it, the owner has to send the defect chip back to the manufacturer C, who will verify the serial number printed on the package, destroy the chip, and issue a death certificate back to the former owner that has been signed with K_C^{-1}. The customer can then forward this death certificate to all software vendors in order to confirm that the processor has been destroyed and that thereby the old license for P has been terminated. This way, the vendors can issue a free second license for the replacement processor. As a useful side effect, this mechanism ensures that processor manufacturers get back a large number of defect processors, which allows them to recycle material and to perform detailed quality control studies in order to extend the lifetime and reliability of their processors and provide exact statistical time-to-failure estimates for customers.

Application fields and outlook

The software protection concept that has been outlined here might on first glance look pretty complicated, since in order acquire a software data packets have to be exchanged with a server of the software vendor. However, with the Internet and various online service providers, an infrastructure for the electronic delivery of software to customers is emerging, in which for electronic payment there have anyway cryptographic data packets to be exchanged with the software vendors. In these applications, exchanging the license data won't create any additional hassle.

Whether a security processor design like the here presented *TrustNo 1* concept will get a significant market share does not only depend on technical considerations. So far, hardware manufacturers have often profited from the large number of available pirate copies of software, since this increased availability of application software makes the hardware platform much more attractive. Especially in the private consumer market, the wide availability of numerous freely copyable software can be a significant factor for choosing a certain system architecture. Therefore, it is quite possible that many hardware manufacturers do not have a particularly high interest in a powerful software protection mechanism. First main applications for bus encryption processors in workstations might therefore very well be security critical applications with secret algorithms and data that have to be protected against criminal manipulation and industrial espionage.

References:

[1] Best, R. M.: *Preventing Software Privacy with Crypto-Microprocessors*, Proceed. IEEE Spring COMPCON 80, San Francisco, California, February 25–28, 1980, pp. 466–469.

[2] Best, R. M.: *Microprocessor for Executing Enciphered programs*, U.S. Patent No. 4 168 396, September 18, 1979.

[3] Best, R. M.: *Crypto Microprocessor for Executing Enciphered Programs*, U.S. Patent No. 4 278 837, July 14, 1981.

[4] Best, R. M.: *Crypto Microprocessor that Executes Enciphered Programs*, U.S. Patent No. 4 465 901, August 14, 1984.

[7] *Security Requirements for Cryptographic Modules*, FIPS PUB 140-1, Federal Information Processing Standards Publication, National Institute of Standards and Technology, U.S. Department of Commerce, January 11, 1994.

[8] Weingart, Steve H.: *Physical Security for the μABYSS System*, Proc. 1987 IEEE Symposium on Security and Privacy, April 27–29, 1987, Oakland, California, IEEE Computer Society Press, pp. 52–58.

[9] White, Steve R.; Comerford, Liam: *ABYSS: A Trusted Architecture for Software Protection*. Proc. 1987 IEEE Symposium on Security and Privacy, April 27–29, 1987, Oakland, California, IEEE Computer Society Press, pp. 38–51.

[10] Kuhn, Markus: *Sicherheitsanalyse eines Mikroprozessors mit Busverschlüsselung*, diploma thesis, Lehrstuhl für Rechnerstrukturen, University of Erlangen-Nürnberg, Erlangen, July 1996.