ChipLock: Support for Secure Microarchitectures

Taeho Kgil, Laura Falk, Trevor Mudge

{tkgil,laura,tnm}@eecs.umich.edu
Advanced Computer Architecture Lab
University of Michigan

Abstract

The increasing need for security has caused system designers to consider placing some security support directly at the hardware level. In fact, this is starting to emerge as an important consideration in processor design, because the performance overhead of supporting security in hardware is usually significantly lower than a complete software solution. In this paper, we investigate integrating some security support into hardware. We show that security support can be added at some acceptable cost in area and performance. We propose a processor extension called ChipLock. It provides hardware security support for a mostly untrusted operating system to ensure the integrity and confidentiality of all computational results. ChipLock's modular design can be easily integrated into existing hardware platforms with only slight modification to the operating system. ChipLock includes a built-in hardware Key Manager that supports symmetric key assignment, and a read-only-memory, TrustROM, that executes secure hardware routines. The software required is a small trusted portion of the operating system called Trust-Code. We modeled ChipLock's architecture on a full system simulator and showed that, for SPEC2000 benchmarks, it adds about an average of 20% to the execution time, primarily from cryptographic and verification latencies. In addition, layout studies show an area cost of about 8 mm^2 in 180 nm technology. This translates to an area overhead of 5% ~ 15% depending on the processor type.

1 Introduction

Traditionally, security has been handled in software. The widespread demand for security support means that it is becoming worthwhile to move some of the functionality into hardware. Several papers [1], [2] have proposed hardware support for security. The continued advances in integrated circuit technology now mean it is possible to implement security routines in hardware that out perform the corresponding software solutions. Intel and Microsoft's Trusted Computing Platform Alliance (TCPA), ARM's TrustZone and Texas Instruments's Wireless Security are just a few examples. These solutions provide both software and hardware security that protects the developer's intellectual properties (IP) and works to eliminate malicious user attacks.

We have designed a secure architecture called *Chip-Lock*. It is not tied to a particular architecture and so can be viewed as an extension that can be inserted into existing or future microprocessors. In *ChipLock* we chose to assume that a very small part of the operating system is trusted. We call this part *TrustCode*. Trusted code exists in an on-chip ROM to interface with *Trust-Code*. We call this *TrustROM*. By creating *TrustCode* and separating it from the operating system, we guard against certain software attacks such as tampering and eavesdropping. We also decided that certain critical cryptographic software routines needed to be implemented in hardware to make them tamper proof. *Our goal is to ensure the integrity and confidentiality of all computational results. We accomplish this goal by ensuring against binary and shared library tampering during execution.* We assume the microprocessor and the storage device is trusted. Others have shown methods to protect storage devices, see [18] for example. *ChipLock* focuses on the microprocessor side.

ChipLock is designed to support a high level of security in hardware. There is a performance and area cost. The performance cost is small compared to a software solution. The next section describes our security model. Section 3 describes the microarchitectural support to implement this model. Section 4 describes how we evaluate the cost of security support. The results of evaluation are presented in Section 5 followed by conclusions in Section 6.

2 Security Model

The primary assumptions we made when designing *ChipLock* were that only the processor and *TrustCode* were trusted. Everything else is untrusted. Therefore, this implies all code and data moved off the processor during context switches and interrupts are encrypted. Similar to [2] we maintain a strict, tamper free level of

on-chip security. However, unlike [2], we try to minimize the changes in our processor since the hardware components related to security management are not directly related to normal processor execution. Temporary information moved to memory is also secured by means of the cryptographic/verification engines. Any content leaving or entering the microprocessor must have an assigned key to protect it from outside attacks.

The protocol is as follows. When a binary is brought into memory for execution, *ChipLock* obtains a hash of that binary. For security purposes a hash is an algorithm, which when applied to a data set takes an arbitrary input size and produces an output of fixed size. When implemented correctly, the hash can be used to verify the integrity of computer data. *ChipLock* uses the hash to detect unauthorized changes in the binary during execution. We chose to use a hash instead of marking the memory as read only to allow for dynamic code generation and modification. During execution, the processor assigns temporary symmetric keys to encrypt and decrypt the data as it moves to and from memory. The temporary keys are refreshed periodically. Temporary keys are removed at the end of execution or during conflict misses in the key manager's table. The key manager is discussed in Section 3.4. Context switches are not considered the end of a full binary execution and therefore, no temporary keys are removed during that time.

Although we do not discuss it in this paper, we assume the storage device has a secure microcontroller. This microcontroller is necessary for interacting with *ChipLock's* architecture. This requirement makes it important for the device driver of the storage device's microcontroller to be part of the trusted part of the operating system or embedded in *ChipLock*.

In order to maintain the proper security, *TrustCode* should be isolated in an address space separate from the rest of the operating system, as suggested in [17]. This portion of the kernel directly interacts with *ChipLock's* *TrustROM* so it is necessary to protect it from attacks. The size of *TrustCode* is small. In fact, the majority of the code necessary for communication between the hardware and the operating system is embedded in *ChipLock's* architecture. *ChipLock* functions as a fail-closed system. In other words, if *TrustCode* is corrupted in any way, *ChipLock* will cease to operate. Since the obvious attack is to corrupt *TrustCode*, isolating the trusted portion of the operating system into a separate address space makes it more easy to secure.

We have not addressed any physical attacks based on specialized methods such as power analysis or ultraviolet rays. In addition, we have not considered the existence of any virtual machine in our security model. Virtual machines that precisely reconstruct the hardware architecture allow hackers to simulate attack schemes. Hardware that can be recreated is more vulnerable to attacks because of this property. *ChipLock* has a hardware identifier that takes part in registering the microprocessor, which does provide some barrier to recreating *ChipLock's* hardware exactly.

3 The ChipLock Architecture

Figure 1 illustrates *ChipLock's* architecture. In addition to the microprocessor, *ChipLock* consists of a Crypto Unit, a Verification Unit, a Key Manager, and TrustROM. *ChipLock's* hardware design borrows from the procedures followed in the Secure Sockets Layer (SSL) protocol [6]. The Crypto Unit encrypts and decrypts content coming into and out of the L2 cache. With the exception of regions in memory that may not require encryption, data transactions resulting from load/store instructions and L2 cache misses remain encrypted. Integrity and confidentiality are maintained
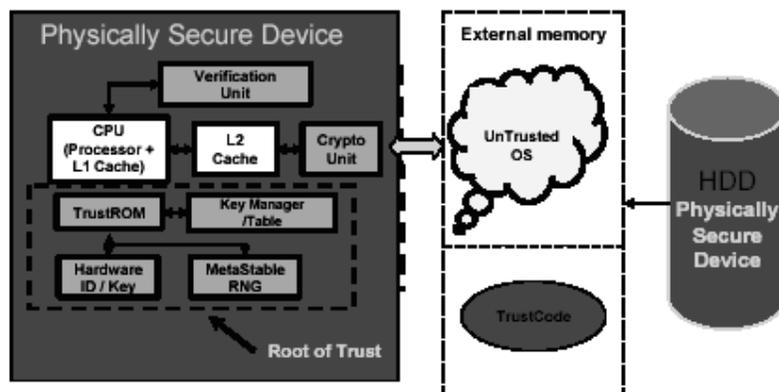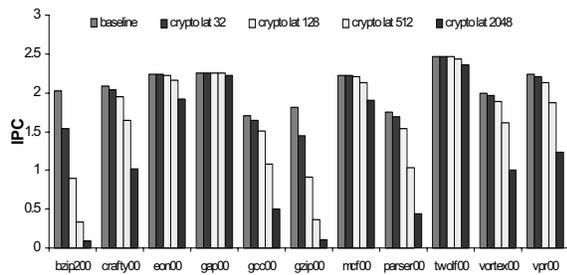

Figure 1. ChipLock Architecture

Figure 2. IPC for various latencies

by the Crypto and Verification Units, respectively. The TrustROM component isolates the region where the microprocessor is executing security related code. Neither software applications nor the operating system can access this code. Symmetric keys are managed on chip by the Key Manager Unit. The Key Manager consists of two separate key tables, one for general keys and one for shared library keys.

In general, when an application loads onto external memory, *ChipLock* observes the system calls made from the operating system. When system calls are made that are related to loading and assigning specific memory addresses, TrustROM is executed without the knowledge of the untrusted portion of the operating system. It then assigns a symmetric temporary key to the memory locations that are required in running the application. For memory allocations that occur during execution, *ChipLock* can also gain control of the microprocessor through the TrustROM in a similar manner. These symmetric keys are stored in the Key Manager. For shared libraries, if the symmetric key already exists in the shared key table, another entry in the Key Manager is allocated with the identical symmetric key after checking the library program's hash. After all this is complete, the processor starts executing the application. On L2 cache misses, the validity of the key is checked. If the key is not valid, *key authentication* is re-initiated. A symmetric key is re-assigned to the corresponding section containing that part of the code and/or data.

3.1 ChipLock and the SSL protocol

ChipLock borrows some of the concepts proposed in SSL. The SSL protocol has been proven to be secure and is widely used in network communications. By leveraging ideas from SSL we sidestep the need to prove our approach is secure. We maintain confidentiality by adopting the cryptographic standards defined in SSL. We also adopted the protocol in generating symmetric keys for these cryptographic algorithms. We call this phase key authentication in ChipLock. Integrity of code/data is done by also adopting the suggested hash

algorithms in SSL.

3.2 Cryptographic Unit

The Crypto Unit is intended to encrypt and decrypt data using a symmetric key algorithm. There is no need for us to use asymmetric keys, because symmetric keys are much less expensive there is no multi-agent sharing. These keys are used for encrypting the actual data existing on memory. We have made area estimates of the impact of ChipLock. One set of results in Table 1, shows that the area overhead of adding a Crypto Unit for symmetric keys on chip is quite modest. For example, AES only adds a 5% area overhead for typical high performance processors. Tables 1 and 2 are generated from the Synopsis Design Compiler with synthesizable code provided by [12]. We compared the estimated area and latency with commercial products in [12] and found these were optimistic in terms of performance and pessimistic in terms of area. This is because the delay was relaxed in the interconnect wire model and did not take into account placement and routing difficulties. On the other hand, area was estimated assuming worst case row utilization for placement. For simplicity, we assumed the Crypto Unit would use AES because it is widely used for symmetric key encryption [5]. We assumed this unit can encrypt and decrypt a cache line in 16 rounds. Furthermore, we optimized the AES to reduce latency. This explains the increase in area and smaller critical path/round in Tables 1 and 2. The overall latency is 2.28ns x 16 ~ 36ns, or about 35 cycles for our target system clock. The latency of the Crypto Unit—especially the decryption latency—directly impacts the overall performance of the microprocessor, because cache miss penalties are increased as a result. Figure 2 shows the decrease of Instructions Per Cycle (IPC) in SPEC2000 for various latencies in the Crypto Unit. We varied the latency of the Crypto Unit in a typical out-of-order machine. The latency was added to external memory transactions. As we increase latency, we model the effect of a Crypto Unit that requires less area and power. A latency of 2048 cycles models a Crypto Unit implemented with a microcontroller executing cryptographic software. We can see dramatic decreases in IPC for latencies that are long, especially for high memory traffic benchmarks. It suggests that a system where performance is of importance should minimize decryption and encryption latencies on L2 cache misses or writebacks. With this in mind the shaded rows in Tables 1 and 2 are *ChipLock's* choice for the Crypto and Verification Unit
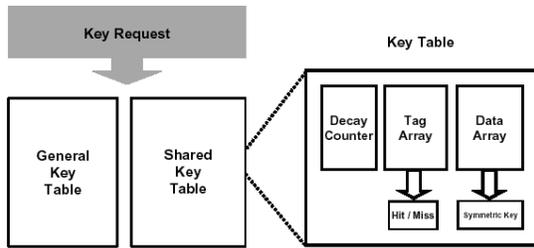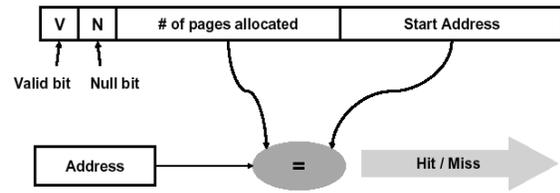
.

Figure 3. (a) Key Manager

(b) Key Tag format

TABLE 1. Area overhead for crypto. & verification units

	Area (mm^2)
AES (Rijndael)	**5.374**
DES (64 bits)	2.370
Blowfish	1.744
TEA (Tiny Encryption Algorithm)	0.708
SHA	**1.011**
CRC	0.173

TABLE 2. Latency for crypto. & verification unit

	Rounds	Critical path delay / round	Total Latency	FO4**
AES (Rijndael)	**16**	**2.28ns**	**36.48ns**	**34.21**
DES (64 bits)	16	2.78ns	44.48ns	41.62
Blowfish	16	12.15ns	194.4ns	181.89
TEA (Tiny Encryption Algorithm)	32	5.21ns	166.72ns	77.99
RSA *	1.67E7	3ns	50ms	44.91
SHA	**80**	**4.02ns**	**321.6ns**	**60.18**
CRC	4	3.88ns	15.52ns	58.08
MD5 *	64	4.88ns	312.32ns	73.05

* From [10],[11]

** FO4 : Fanout of 4 inverter delay

3.3 Verification Unit

The Verification Unit checks for software tampering during execution. If such tampering is detected, it alerts the processor so that the execution can be aborted. Table 2 shows that verification is expensive relative to en/description in terms of performance. This latency is difficult to reduce. The hash itself also becomes a large overhead, in terms of space. The well-known SHA-1 hash algorithm is used in *ChipLock*. By definition, SHA-1 is a stream algorithm that processes a message 512 bits at a time until the end of the message is reached. Since it is desirable to verify whenever a cache block is read, if we use SHA-1 we can streamline the verification process by setting our cache block size to be equal to the hash block size of 512 bits. Any code or data that has been tampered with would result in an unpredictable microprocessor execution state, and at worst, would result in an encrypted core dump that disturbs the operation of the system. However, under no circumstances will vital information be exposed externally in an unencrypted format during the execution time. *ChipLock* can also be modified to support the hash routine in [1] so that verification latency is reduced.

3.4 Key Manager, Hardware Identifier, and Random Number Generation

The Key Manager is an integral part of *ChipLock* and the most important feature in a trusted platform. *ChipLock* divides the Key Manager into two subcomponents, a general key table and a shared key table. The Key Manager determines key assignment, revocation, and in some cases generates keys for transactions. *ChipLock* uses circuit level techniques that are simple but effective to implement this feature. Figure 3(a) shows a block diagram of the Key Manager. Each table has a tag and data array just like a conventional cache. However, since it must refresh itself from time to time we use a decay counter similar to that in [13] to trigger key invalidation. Figure 3(b) shows the contents inside the tag. The tag for the key table is similar to that of a trace cache. It holds the start address of the assigned key and the number of pages assigned. It is indexed by the program counter and process ID. The null bit exists for software applications that are not encrypted. (Configurable on *ChipLock* and used for legacy software.) If the null bit is set to *zero*, data coming in and going out of the chip is neither decrypted nor encrypted. A counter is required to implement key revocation. *ChipLock* does not use a conventional timer since the size of the counter would be extremely large and randomness is desirable. However, precision is not important, so to save space we chose a decay counter, which uses a capacitor that requires no more space than a single flip-flop. We modeled it with *HSPICE* and noticed it took approximately 10ms to fully discharge a capacitor equivalent in size to a single flip-flop. This can make a decay counter roughly equivalent to a 20 ~ 25 bit counter. For a single flip-flop, we used a standard flip-flop cell provided from the Artisan cell libraries for TSMC 0.18μ. A discharge event updates a relatively

small counter. When the counter overflows, the key assigned to that entry of the key table is revoked or invalidated. The application may continue to operate out of cache. In the meantime a temporary symmetric key is generated. This new symmetric key is then allocated in the key table by the Key Manager. When a L2 cache miss eventually occurs this new key will have to be used for transfers.

In general, an L2 cache miss causes the Key Manager to see if the key exists or is invalid for the specific miss address. The Key Manager broadcasts the address to the general key table and shared key table. A hit or miss results. On a key table miss, the *key authentication* step is performed: a temporary symmetric key is generated during this process (it may already be created as in key revocation above) and the new key is allocated in the Key Manager. The *key authentication* step is also performed on an application load. We differentiate shared keys and general keys by the properties assigned to the page to which the address is pointing. On initializing an allocated memory location, the operating system assigns a property to each memory block. By looking at this property, we determine if this block of memory should be assigned a key that belongs to the general key table or the shared key table.

In addition to the features enumerated above, *Chip-Lock* requires a random number generator (RNG) and a unique hardware key generator. A *hardware key* is a unique ID that is created at startup time. It is important to generate unique hardware keys. A hardware ID/key is used to uniquely identity the microprocessor chip and to prohibit other unauthorized peripherals and storage units and network connections to interact with the microprocessor. This hardware key serves as an identifier which can be equally accessed by anyone who can physically access the microprocessor. We assume hardware key uniqueness is generated from process intra-die silicon process variation—sensors are deployed across the chip to measure the process variation. Although, this might be useful in generating unique hardware keys, it is not acceptable for the random number used in generating a symmetric key. Hence, *ChipLock* requires a separate random number generator for symmetric key creation. Conventional pseudo-random number generators are not good enough. Higher quality hardware-based random number generators rely primarily on fundamental noise in the silicon. Methods using metastability of logic devices have been proposed in [9]. The *ChipLock* architecture assumes these can be employed as an RNG. The hardware key and random number generator are used for assigning keys to applications, metadata, and core dumps.

3.5 TrustROM

TrustROM is an important part of our design. We have extended the instruction set of [3] to include secure symmetric key store, secure key load, secure key allocation, and secure key invalidation. Table 3 summarizes the instructions. Each instruction is explained below. The instruction *Clalloc* is necessary in order to allocate key table entries in the two separate areas, which are general processing and shared libraries. An operand accompanies the call to *Clalloc*, which provides the requested key area. The *Clentr* instruction is used once the key has been allocated and the process is ready to execute code. It uses an index for key look up. Exiting the code execution area is done with *Clexit*. It is used either in a context switch or at the completion of a process. We have separated key and hash checking and created instructions called *Clcheck* and *Clhash*, which allow for finer security control. *Clcheck* and *Clhash* are the only two instructions that will contact the verifier. Allowing only two instructions to talk with the verifier minimizes the ways in which it can be attacked. We want as few instructions as possible accessing the verifier because of the central role it has in legitimizing computations.

The *Clcheck* instruction is used to check the validity of the key in the key table. The *Clhash* instruction is used to determine whether a given software hash matches in the verification unit. When a hash doesn't match, a separate event is called to handle the possible security breach. This instruction is called *Clhnmatch*. The *Clinval* and *Clrclm* instructions are used to invalidate a key and reclaim an invalid entry in the key table, respectively.

In addition to protecting the code/data, we need to protect the metadata or the keys. Following the Horton principle of [6], we included a secure key store and a secure key load instruction, *Clsst* and *Clsld*, respectively. These are necessary because if the keys were obtained by an intruder or otherwise malicious user, then the integrity of the computation and its result would be compromised. In addition to securing the key, we also need to secure the state of the machine during a context switch. For this task, a separate instruction called *Cldump* is defined. It dumps the state for encryption to a designated area in the cryptographic unit. Once encrypted, the cryptographic unit is responsible for handing this information to *TrustCode*, the trusted portion of the operating system, for temporary storage. When that information is needed, *TrustCode* then moves it back to the cryptographic unit where it is decrypted and restored for the processor to reuse.

TABLE 3. **Trusted Instruction Set**

Instruction	Instruction Description
Clentr	enters general execution area
Clinval	invalidates key
Clexit	exit trusted code excution area
Clrclm	Removes key entry in table if marked invalid
Clcheck	checks to see if key in tables is invalid
Clsst	Secure key store
Clsld	Secure key load
Clenc	Encrypt register contents for memory storage
Clsave	Saves encrypted registers to memory
Clrstr	Restore encrypted values to registers from memory
Cldec	Decrypts register contents
Cldump	Dumps state for context switches for encryption
Clalloc Opr	Allocates key table entries used for all three key hashes
Clfkey	Fetches key information
Clinkey	Installs public keys on disk
Clhash	Determines whether hash matches in verification unit
Clhnmatch	Called when hash doesn't match
Cldosat	Returns whether there is a possible denial-of-service attack
Cldoscount	Keeps count on number of times process has requested processor

Encryption and decryption of the registers are handled by the *Clenc* and *Cldec* instructions, respectively. These are necessary to preserve the integrity of the bits in the registers both during execution and when the contents are being prepared to store off-chip. Separate instructions, *Clsave* and *Clrstr*, exists for moving the encrypted register data off-chip and for moving the data from memory back to the register. These are important and basically unchanged from the original design [3].

We have provided a means of secure communication between *ChipLock* and the storage device by establishing an interface between the storage device and its microcontroller. This microcontroller is given the processor's symmetric key upon each key revocation. The volatile memory between the processor and the disk is encrypted. Therefore, the storage device's microcontroller needs the symmetric key to decrypt the data before it is sent back to the storage device.

Now that we have described the instructions, we will provide two examples. The first is loading a legitimate new process and the second is a context switch. When a legitimate new process is executed, *ChipLock* supervises the system calls made during this process. Before handing over control to the untrusted portion of the operating system, a hash is made of the contents of the memory prior to execution. *Clcheck* is then called to check to see if there is a key in the key table. If not, then *Clalloc* Opr is called to allocate space for one, after which a symmetric key is issued and assigned in the general key table for this process. Recall that the symmetric key is refreshed periodically. This helps guard against a replay attack either by a malicious user or by an impatient novice user. *Clentr* is then called so the process can enter the general area for code execution. If this process needs to use a shared library during execution, then the symmetric key from the shared library is obtained via the *Clfkey* instruction. *Clcheck* is then called again to verify that the symmetric key is valid and to check if the key is located in the key table. If not, *Clalloc* Opr is called to allocate an entry.

For our second example, we will continue with the same process which is interrupted by a context switch. What our process has to do is encrypt and save state and registers to memory. Therefore, *Clexit* is called to remove the process from the general code execution area. *Cldump* is called to dump state. *Clenc* is then called to encrypt the contents and *Clsave* and *Clsst* save both the key and state to memory. Let us now consider what happens when this process is returning from the context switch, *Clrstr* is called to retrieve the encrypted information from memory and restore it to the registers. *Cldec* is called to decrypt the contents of the registers. The instruction *Clsld* is then called to load the key so that *Clcheck* can check the validity of the key. *Clalloc* Opr is called to allocate an entry in the key table, then *Clentr* is called to start the execution of the process. The code we have described above is placed in *TrustROM* because it is compact. Again, all keys that are being used are generated from the random number generator.

3.6 Microprocessor

ChipLock requires the microprocessor to execute the additional trusted instructions from the TrustROM. When the trusted instructions are handed off to the processor pipeline, it transitions to secure mode and issues the appropriate control signals. For some trusted instructions, a subroutine that exists off-chip is invoked and executed. In order to distinguish between the new secure mode and the regular mode, the microprocessor requires an extra status bit in the program status register. Extra exception entry slots should also be allocated for security related features.

4 Evaluating the Impact of ChipLock

We employ two simulation modes. The first uses *SimpleScalar*[8] with a model of the Alpha 21264 architecture to study single process activity. The latency of the cryptographic and verification units are modeled. Generally, most security transactions are sequential and hard

to parallelize. Therefore, we added all latencies together for performance accounting purposes. Features of *Chip-Lock* that are dependent on virtual memory were not modeled in this mode. In particular, this mode was used to get approximate comparisons between software and hardware implementations. We ran simulations of both hardware and software implementations of the Crypto Unit and the Verification Unit to generate comparisons. As one would expect they confirmed that ChipLock's security implementations were faster than software counterparts. Figure 2 was also generated in by this mode.

The second simulation mode employed the M5 simulator [7] to run experiments. M5 is a cycle accurate simulator intended for multi-processor and network workloads, which made it ideal to create an environment where one wants of observe the behavior of an operating system executing on a processor. In particular, we could account for the effects of the interaction between *TrustROM, TrustCode,* and the remaining untrusted operating system.

The operating system that runs on M5, and used for our simulations, is tru64 UNIX. We modified M5 by adding the necessary security features from our architecture. We traced the security related events that interacted with *ChipLock* and executed *ChipLock's* components. SPEC2000 Benchmarks were used as workloads that ran on the tru64 UNIX. We forked multiple SPEC2000 processes and measured the overhead versus the number of processes.

TABLE 4. Configuration for experiments

Architectural Parameters	Specifications
Clock Frequency	1GHz
L1 - I, DCaches	64KB, 2-way, 32B line
L2 Caches	1MB, 4-way, 64B line
L1 latency	2 cycles
L2 latency	10 cycles
Memory latency	80 cycles
I / D TLB	4 way 128 entries
AES latency	32 cycles
SHA latency	160 cycles
Key authentication latency	1,000,000 cycles
Operating System	Tru64 Alpha UNIX

Table 4 shows the configuration on our simulations. A simple in-order Alpha processor is used to reduce execution time when running the M5 simulator. Therefore, the effect of multiple issue and out-of-order execution is neglected in M5. Each unit of *ChipLock*, described in section 3, is modeled as a separate security component in M5. Each unit is described in terms of size and latency. When we execute M5, it first goes through the

booting process similar to a desktop machine. M5 has virtual devices for full system implementation. Virtual hardware devices are initialized, which are the hard-disk drive and network driver. The operating system disk image is loaded onto the simulated memory. After initialization, a script created in one of the files on the simulated hard-disk drive is executed. This script executes additional scripts to run the SPEC2000 benchmarks. The SPEC2000 benchmarks also exist on the simulated hard-disk image. We ran simulations for 2, 3, 4 user level SPEC2000 benchmark processes. We used unique identifiers for the benchmark groups we ran. Table 5 shows the assigned names of each benchmark group.

TABLE 5. Assigned names for benchmark groups

Benchmark group	Assigned name
bzip, crafty	BC2
crafty, eon	CE2
eon, gap	EG2
gap, gcc	GG2A
gcc,gzip	GG2B
gzip,mcf	GM2
mcf,parser	MP2
parser,perlbmk	PP2
perlbmk,twolf	PT2
twolf, vortex	TV2
vortex, vpr	VV2
bzip, crafty, eon	BCE3
crafty, eon, gap	CEG3
eon, gap, gcc	EGG3
gap, gcc, gzip	GGG3
gcc, gzip,mcf	GGM3
gzip,mcf, parser	GMP3
mcf, parser, perlbmk	MPP3
parser, perlbmk, twolf	PPT3
perlbmk, twolf, vortex	PTV3
twolf, vortex, vpr	TVV3
bzip, crafty, eon, gap	BCEG4
crafty, eon, gap, gcc	CEGG4
eon, gap, gcc, gzip	EGGG4
gap, gcc, gzip, mcf	GGGM4
gcc, gzip,mcf, parser	GGMP4
gzip, mcf, parser, perlbmk	GMPP4
mcf, parser, perlbmk, twolf	MPPT4
parser, perlbmk, twolf, vortex	PPTV4
perlbmk, twolf, vortex, vpr	PTVV4

5 Results

5.1 Hardware vs. Software Comparison

We use the *SimpleScalar* simulation environment to compare hardware and software implementations of the
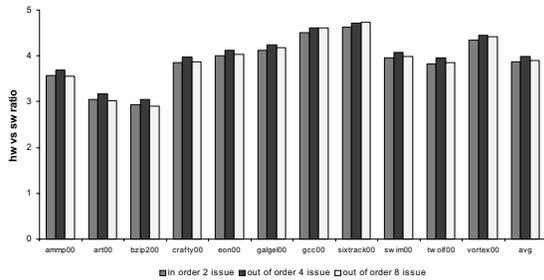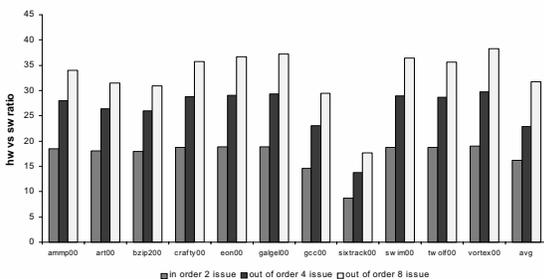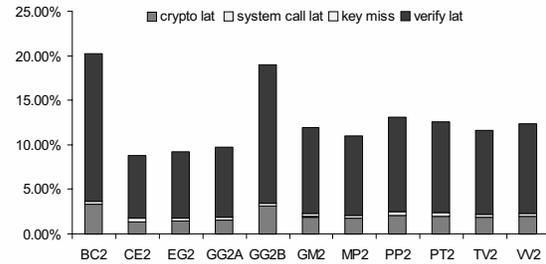
Figure 4. hardware vs. software AES

(a) 2 process benchmarks

Figure 5. hardware vs. software SHA-1

(b) 3 process benchmarks

(c) 4 process benchmarks

Figure 6. Breakdown of overhead

Crypto Unit and the Verification Unit. Figures 4 and 5 shows the run time ratios for the spec2000 benchmarks. As expected software implementations are considerably slower. This trend will be even more pronounced for high-end out-of-order microprocessors due to the lack of parallelism in cryptographic and verification algorithms.

5.2 Verification Unit & Crypto Overhead

To study the verification and crypto overhead we employ the M5 simulation environment. For the verification unit we reduce overhead and latency by assuming that any writeback or update to hash values can be queued. Verification is initially done for all L2 cache misses. Figure 6 breaks out the overhead for *ChipLock* running in a full system. Typically, an L2 cache miss rate for SPEC2000 benchmarks is very small (less than 1%), however the long latencies required by verification can still cause the processor to spend a significant amount of execution time on verification. Fortunately, the latency can be hidden if infrequent verification occurs (infrequent L2 misses) or verification is done in parallel with cache miss events. Accurate prefetches may also hide the latency [15]. However, prefetching when frequent concurrent L2 cache misses occurs may not help reduce latency.
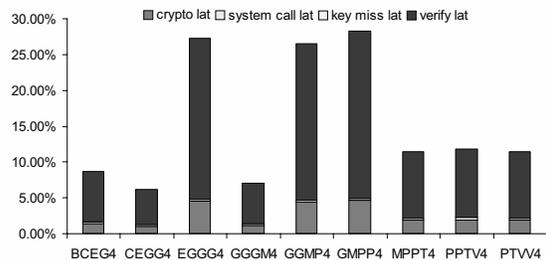
The Crypto Unit is also a major contributor to overhead. For our simulations, we again assumed that encryption writebacks are queued so that their latency is zero. The Crypto Unit is accessed along with the Verifi-

cation Unit. However, unlike verification, reducing the frequency is not admissible for the Crypto Unit. *ChipLock* cannot selectively decrypt incoming code and/or data. Therefore, a fast response to an L2 cache read miss would be unmistakable. The only exception would be to create a faster cryptographic engine. Prefetching would also become a important factor in decrypting incoming cache blocks [15].

5.3 Key Management Overhead

When applications are loaded into the external memory, symmetric keys are assigned to protect the data. For system calls that are related to memory mapping, *ChipLock* intercepts these calls and executes the trusted instructions, assigning keys and storing them in the Key Manager. Figure 7 shows the key table miss rate for various benchmarks. The key table miss rate is not as significant, in terms of overhead, as the L2 cache miss rate.
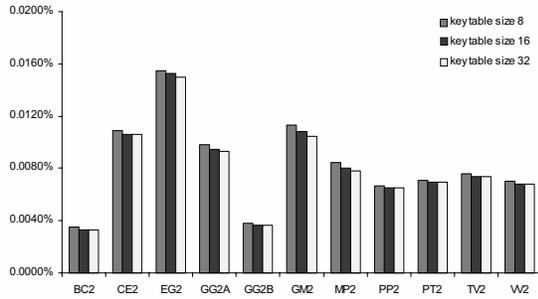
Figure 7. Key Table miss rate vs table size
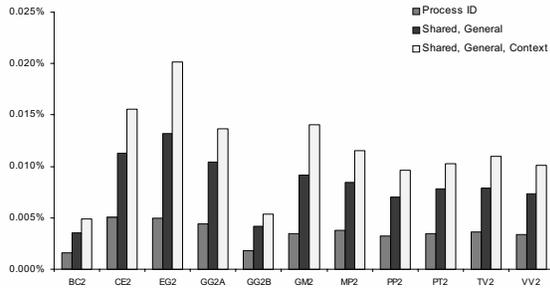

Figure 8. Key Table miss rate vs granularity

This also applies to various architectural events, since spatial locality is maintained for a lengthy period of time. We measured various key assignments, such as per contiguous memory, per library, and metadata. Figure 8 shows, in general, that even with a fine grain assignment, key management is not a main contributor to overhead. Keys are assigned to processes, temporary context machine states, shared libraries, and separate memory maps as shown in the right-most bar of Figure 7. We see a minimal increase in terms of overall misses. Recalling that *ChipLock* does key allocation and checks for L2 cache misses, and system calls, this shows that potentially more processor cycles could be used in *key management*. Furthermore, in terms of overall execution time, the contribution of *key management* is not significant compared to that of the Verification and Crypto Units.

5.4 Putting it all Together

Figure 6 shows the break down of execution overhead for our benchmarks. Most of the security related executions are sequential and hard to parallelize due to the security protocol we follow. This implies that the Verification and the Crypto Unit play a significant role in contributing to the overhead of the system. We can also conclude that this overhead would have more of an impact on multi-issue out-of-order machines. Looking at the overall impact of *key management*, assuming the

latency to be 1,000,000 cycles as shown in Table 4, we see that the execution time of *key management* is small when compared to its latency. Comparing the benchmarks, we can see a significant increase in verification time for benchmark groups that have a high L2 cache miss latency. In some cases, the verification time takes approximately 25% of the overall execution time. Another trend we observe from Figure 6, is the increase in overhead as the number of processes for security related functions increase. We can clearly see an increase in the verification time on MPP3 compared to MP2 resulting in the increase of processes sharing an L2 cache. Additional misses are incurred thereby increasing verification and decryption.

6 Conclusion

We have proposed hardware support for security in a microprocessor with on-chip cache called *ChipLock* that provides confidentiality and integrity. The salient concepts from a verified security protocol, SSL, were used to design *ChipLock* and ensure security. An efficient Key Manager was implemented in hardware which maintained symmetric keys with fine granularity. Assigning unique keys while maintaining dynamic *key management* is an important feature in *ChipLock*. Trusted instructions were added to the conventional instruction set. These are necessary to maintaining security when there are transactions between the mostly untrusted operating system and the trusted architecture. They interact with the small trusted portion of the operating system. We have simulated each component of our security system on M5, evaluated various instances of our benchmarks, and measured the impact on chip area and performance. We have shown from the results that some overhead is associated with these added security features. We have also suggested prefetch techniques to reduce performance degradation. We have noted that certain tradeoffs must be made in order to maintain a certain level of security.

7 References

[1] E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, Efficient Memory Integrity Verification and Encryption for Secure Processors, Proceedings of the 36th International Symposium on Microarchitecture (MICRO), December 2003

[2] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural
Support for Copy and Tamper Resistant Software. In

Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX). November, 2000. Pp 169--177

[3] D. Lie, C. Thekkath, and M. Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. Proceedings of the nineteenth ACM symposium on Operating systems principles table of contents Bolton Landing, NY, USA October, 2003

[4] TCPA Alliance, http://www.trustedcomputing.org/

[5] Advanced Encryption Standard (AES), [FIPS 197]

[6] D. Wagner and B. Schneier, Analysis of the SSL 3.0 Protocol, The Second USENIX Workshop on Electronic Commerce Proceedings, USENIX Press, November 1996, pp. 29-40

[7] N. Binkert, E. Hallnor, and S. Reinhardt. Network-Oriented Full-System Simulation using M5 Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW), February 2003

[8] T. Austin, E. Larson, and D. Ernst. SimpleScalar: an Infrastructure for Computer System Modeling, IEEE Computer, 35 (2), February 2002.

[9] D. Kinniment, E. Chester, Design of an on-chip random number generator using metastability. ESSCIRC 2002. Proceedings of the 28th European Solid-State Circuit Conference. Univ. Bologna, Italy, pp.595-598, September, 2002

[10] MD5, http://www.sci-worx.com/internet/designobjects/do_list/handouts/cryptography/VC01C2.008/VC01C2.008HO.pdf

[11] RSA,http://www.sci-worx.com/internet/homepage_internet_e.html?/internet/designobjects/do_list/handouts/cryptography/VC01C2.002/rsa.htm

[12] OPENCORES.ORG, http://www.opencores.org

[13] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In Proc. ISCA-28, July 2001

[14] The MOSIS Service, http://www.mosis.org

[15] W. Lin, S. Reinhardt, and D. Burger. Designing a Modern Memory Hierarchy with Hardware Prefetching, IEEE Transactions on Computers, 50(11):1202-1218, Nov. 2001

[16] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Silicon Physical Random Functions. In Proceedings of the Computer and Communication Security Conference, May 2002

[17] B. Chen and R. Morris. Certifying Program Execution with Secure Processors. 9th Workshop on Hot Topics in Operating Systems (HotOS IX), May 2003

[18] J. Strunk, G. Goodson, M. Scheinholtz, C. Soules, and G. Ganger, Self-Securing Storage: Protecting Data in Compromised Systems. Appears in Proc. of the 4th Symposium on Operating Systems Design and Implementation, 2000