

Design and Characterization of a Hardware Encryption Management Unit for Secure Computing Platforms

Anthony J. Mahar*, Peter M. Athanas*, Stephen D. Craven*, Joshua N. Edmison*, Jonathan Graf†

*Department of Electrical and Computer Engineering
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061

Email: {amahar, athanas, scraven, jedmison}@vt.edu

†Secure Computing Group

Luna Innovations, Inc.

Blacksburg, VA 24060

Email: jgraf@lunainnovations.com

Abstract—Software protection is increasingly necessary for uses in commercial systems, digital content distributors, and military systems. The Secure Software (SecSoft) architecture is one of several architectures attempting to provide hardware enforced software protection. The advantage of the SecSoft architecture is increased software protection without alteration to the fundamental interaction of software, operating systems, and developer tools. This work presents the design, implementation, and performance analysis of a fast, flexible hardware Encryption Management Unit (EMU) for such architectures. The EMU provides selective decryption of software instructions residing in page-sized sections of memory, without modification to the core processor. Results are derived from a FPGA based prototype platform containing different cryptographic routines operating with a standard processor.

I. INTRODUCTION

Security has remained an important role in software since the beginning of digital computing. Computers are commonly used to perform operations on sensitive data where the data itself and the computational instructions must be safe guarded. Early devices operating on sensitive information were physically isolated, and protection required physical security. However, as computing devices became more pervasive and interconnected, the need for secure computing in insecure locations by untrusted users became necessary.

Attempts to provide secure processors were initially successful. Stand-alone processors [1] that ran a single application and encrypted all external memory transactions were thought to be secure for some time. These machines did not require much processing power, but instead required extreme integrity and confidentiality of software instructions and data. As physical anti-tamper mechanisms protected the chip, attacks were limited to software in external memory and buses. Recently, certain attacks have been demonstrated on these devices [2].

Currently, there is an increasing demand for software protection in modern computing systems [3]. Modern software protection can require methods for maintaining confidentiality and integrity of software during distribution, in external memories, and during execution. There are a number of architectures that exist in this field of research that attempt to offer the full set of software protections, such as tamper resistance and confidentiality. However, their implementations [4] [5] [6] often require trust in parts of software to function, or require fundamental changes to the way an operating system and applications interact.

The inherited trust model used by most systems [7] has already been shown [8] as a fairly weak solution. If a trusted module is compromised, the rest of the system trusting that module can no longer guarantee protection. These problems are also compounded by the lack of confidentiality.

Other systems that rely less on trusted software components typically require fundamental changes [6] to the way software operates, and how the processor and operating system interact with the software. These systems may provide software protection, but due to the fundamental changes and incompatibilities with current software models they are not easily incorporated into modern computer systems.

The goal of the Secure Software (SecSoft) architecture is to provide increased software protection while maintaining standard development and computing models. The proposed architecture achieves increased software protection through simple extensions to the processor memory management unit (MMU) and operating system page handling routines. The developer flags and encrypts desired pages of instructions in the executable, which are then distributed to systems that will run the application.

Only when the appropriate credentials and conditions

are met will the instructions be decrypted. Secure instructions remain encrypted from the point of distribution until they enter the processor core. If credentials are not met, then trust is not required of the operating system or any other software module, which increases security and ease of integration.

This paper presents the hardware unit responsible for the secure status, key selection, and decryption of the processor instruction stream. The Encryption Management Unit (EMU) supports all secure functionality defined by the Secure Software architecture in hardware. Although the EMU prototype is presently targeted for the PowerPC processor architecture, its design is flexible for use with most contemporary processor architectures.

Very few of the secure architecture alternatives in this field have been implemented or prototyped. The TCGA TPM [7] has received backing from industry leaders, but remains flawed in tamper protection and confidentiality. The SecSoft architecture is of the minority to have been implemented. This provides an ideal solution for analyzing the actual performance impact of the secure extensions, on a standard processor and multi-tasking operating system. A benchmark was created and used to analyze the instruction fetch performance of the extensions.

This paper is organized as follows. Section II provides background information on relevant software protections and the SecSoft architecture. Section III details the design of the Encryption Management Unit. Section IV models to the effect of the EMU with respect to application type and system loads. Section V describes the benchmark created to analyze the EMU. Section VI provides results and analysis from the synthetic benchmark. Section VII indicates potential direction of this work. Lastly, Section VIII concludes this work with a summary of the contributions of this work.

II. BACKGROUND

This section provides a background required for the proper context of this work. Important areas include definitions of primary threats to software, a description of the SecSoft architecture, systems with similar functionality to the Encryption Management Unit, and methods for benchmarking secure software systems.

A. Software Threats

There are two threats to software applications that an effective software protection solution must address. This primarily includes software integrity, also known as tamper resistance or anti-tamper, and confidentiality. The

addition of confidentiality distinguishes *secure* architectures from *trusted* architectures; trusted architectures do not provide confidentiality.

Software confidentiality offers protection from threats such as unauthorized or unlicensed copying of programs, including piracy and the reverse engineering of algorithms. Software confidentiality is also an effective method to protect intellectual property, avoiding public disclosure to the public, competitors, and malicious entities.

Software integrity relates to threats that modify program instructions. Maintaining integrity ensures secure software is running the way developers programmed it to. Furthermore, the user can trust that the program has not been altered beyond the developer's creation. Threats from viruses and malicious users that rely on code modification are prevented.

B. Secure Software Architecture

The purpose of the Secure Software project [9] is to investigate, design, and prototype various security mechanisms when used in several computing architectures, including single and parallel processor environments. The base configuration provides software confidentiality and integrity through protection of a program's instructions. These features are offered without trusting any software module. Furthermore, development, software execution, and operating system models are not critically altered. Standard mechanisms are extended instead.

Most modern computing platforms operate using the concept of virtual paged memory. Pages are fundamental units of memory that the operating system and hardware memory management unit (MMU) recognize. Virtual paged memory allows a program memory space to be virtually represented in the physical system memory space. The translation look-aside buffer (TLB) provides the virtual to physical address mappings. Pages are mapped in and out of memory as necessary, with different pages from different programs populating the physical memory space. This allows many programs to exist in memory at the same time, and allows programs larger than system memory to have a subset of its pages loaded at one time.

It was determined for the Secure Software Project that page-level encryption was the best method for providing software protection from both a hardware and software perspective. Each page in physical memory is associated with a specific key and ancillary data.

First, different encryption keys can be used for different memory pages of the same program, strengthening encryption and preventing replay attacks between pages.

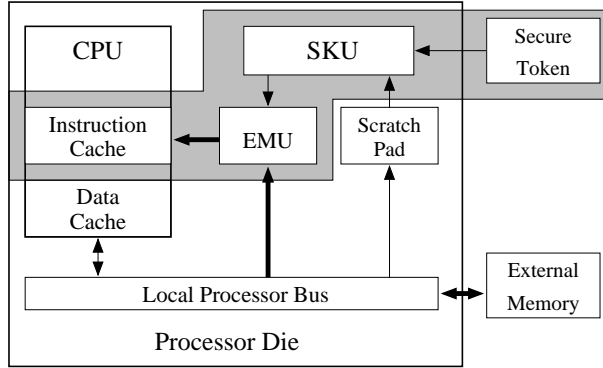


Fig. 1. Base Secure Software Architecture. Shaded area indicates secure components. Thick arrows represent instruction flow.

Second, different secure applications can share memory pages using the same key, while using different keys for non-shared memory. Third, this permits the use of shared, secure libraries, as the libraries can maintain their own security credentials.

Lastly, the mechanisms required to handle page level encryption directly extend operations already existing in the operating system and MMU. The operating system provides on-demand paging, where a page is mapped in the TLB when an application virtual page is not already mapped in. The operations that load the TLB on-demand are extended to also load the page-key mappings in the Secure Software architecture at the same time. No additional event handlers or interrupts are required.

C. Secure Software Components

The three primary areas of development within the Secure Software architecture are as follows.

1) *CPU/Operating System*: The first component is the standard CPU with a memory management unit (MMU) to map virtual program address to the physical memory space. The operating system running on the CPU recognizes security tags and sections within secure executables. The secure executables use the same format and standards as normal executables, but utilize optional sections. Extra information in the secure executable indicate which instruction pages are encrypted, the identifiers of cryptographic keys for the associated pages, and partial security credentials for the keys.

The operating system assigns page to key mappings at the same time it handles the virtual to physical address mappings in the translation look-aside buffer. Keeping these operations together ensures memory fetches on the bus will always reference the most current page encryption information.

Although the operating system associates groups of instructions (pages) with a key, it is only aware of the index of the key, never the value. To establish this relationship between groups of instructions and key pointers, the system requires simple extensions to memory handling mechanisms already used in modern operating systems, specifically the page handling routines. Furthermore, developers can designate secure shared memory and secure dynamic libraries by using the same key for those specific pages and maintaining their individual private keys for all other pages.

2) *Encryption Management Unit*: The second primary component is the Encryption Management Unit (EMU). The EMU maps a physical page to a specific key slot. This extends the MMU behavior of virtual-to-physical page address translation by adding physical page-to-key slot mappings. These mappings are possible through a series of lookup tables. Ancillary data is also supplied directly by the processor and returned along with the search. As instructions are requested from the instruction side interface of the processor the encrypted status and key slot of the page are searched. The EMU uses this result to selectively decrypt an instruction stream.

The EMU supports various cryptographic algorithms and modes through an extensible modular framework. Cryptographic modes, such as direct block or counter modes decryption can be selected based on architectural requirements. The keys stored in the EMU reflect the cryptographic key for a particular page, while ancillary data contains supporting information for certain cryptographic modes, such as page specific base counters for counter mode encryption. Information supplied to the decryption module in the EMU indicates if a page is encrypted in addition to the encryption key and ancillary data if encrypted.

Although the EMU provides software protection, it must also protect other sensitive operations of the Secure Software architecture from the CPU. The EMU does not allow access to keys by the CPU. The CPU can only write to the page-key slot mapping tables, and never has access to the keys stored in the key slots.

3) *Secure Key Management Unit*: The third primary component of the Secure Software Project is the Secure Key Management Unit (SKU). This device is responsible for generating keys used within the Secure Software architecture. A user maintains a physical secure token containing their partial credentials, which can be specific to the user or a group of users. The executable partial credentials are supplied by the operating system at run time. The physical processor may also contain a processor specific partial credential.

The SKU uses the application, user, and possibly processor credentials to generate the key. The key is inserted into a specific slot within the EMU key table. The slot location is supplied by the operating system along with the key generation request. All partial credentials used in key generation must be valid for a correct key to be created.

The SKU resides on the same physical die as the processor, however it is isolated from direct access by CPU. A small scratch pad memory exists for communication between the CPU and SKU for key generation. This isolates the SKU and SKU micro-architecture from the CPU. Similarly, the secure token is accessed through a secure slave interface, and does not directly expose any part of the SKU.

D. Implementation

Implementation requires a standard processor and operating system to demonstrate full support with modern architectures. The prototype used in this work utilizes a Xilinx Virtex-II Pro FPGA [10] with an internal hard core IBM PowerPC 405 embedded processor [11]. This processor is surrounded by reconfigurable logic.

The Xilinx Embedded Development Kit is used to create a complete System on Chip (SoC) containing essential soft cores for internal peripherals. These cores include memory controllers, serial interfaces, PCI bridges, in addition to the instantiations of the SecSoft EMU and SKU. This chip is mounted on the Xilinx ML-310 embedded development board, realizing a complete platform running secure applications under a full version of Linux and XFree86 X-Windows.

III. DESIGN

Based on the requirements of the Secure Software architecture, there are a number of primary functional units that are visible. These units are considered primary as they represent abstract units of functionality necessary to achieving the requirements of the system. These high level modules are depicted along with their interaction in Figure 2.

A. Bus Modifier / Bridge Unit

The Bus Modifier / Bridge unit contains the processing necessary for active modification of instruction requests and returned instruction. It is located directly between the CPU instruction side bus interface and the local processor bus. Internally, the unit provides a Criteria for determining encrypted status, appropriate key, and ancillary data is

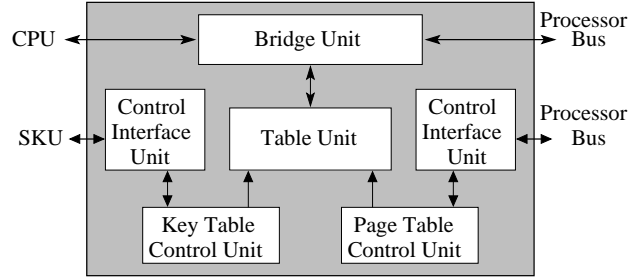


Fig. 2. EMU Primary Functional Units

queried from the Table Unit using the requested page as search criteria.

Although the EMU supports many cryptographic algorithms and block modes through the modular construction, the prototype has been developed using direct block and counter mode decryption modes and using 128-bit block cipher algorithms such as DES and AES [12] [13]. Because the decryption block size is commonly 128-bit, consideration must be given for processor bus protocols that do not operate on 128-bit blocks. Minimizing latency is also a critical aspect to this design.

With respect to the prototype, the local processor bus is the IBM Processor Local Bus (PLB). The instruction side interface of the PowerPC 405 uses a strict subset of this protocol, specifically 64-bit, 128-bit, and 256-bit block modes transferred in 64-bit data beats. Furthermore, the unit supports target-word-first which allows a particular 64-bit word within larger sized and aligned blocks to be sent first, out of order.

Although cryptographic algorithms that operate on fixed block sizes are interchangeable, such as DES and AES operating on the same block size, the block mode using the algorithm can behave very differently. Because of this, the Bridge Unit provides a framework for a linear chain of modifier components. Each component contains a proprietary CPU-side and memory-side interface, where the CPU directed side mates directly with the memory directed side. The Bridge Unit translates from the actual CPU and memory bus and provides these proprietary interfaces internally.

The result of these interfaces allow simple insertion of new modules into the modification chain, switching locations in the chain, and direct replacement of a single module. Future capabilities are easily added to the EMU without affecting other modification units or the rest of the EMU. This modification chain is shown in Figure 3.

For analytical purposes, two decryption blocks were created to support two different block decryption modes:

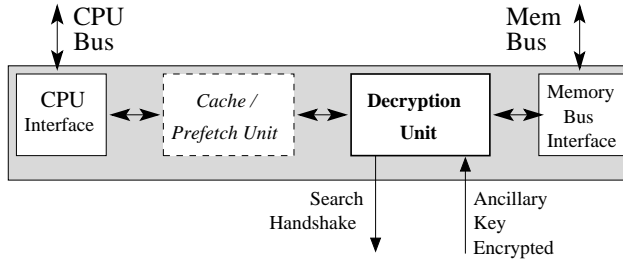


Fig. 3. Bridge Unit. Components feature standard interlocking interfaces allowing easy insertion and arrangement.

direct block and counter mode decryption. The underlying cryptographic algorithm often requires only a key and data, and returns the result at a later time. All algorithms used in the prototype maintain the same block size and, therefore, can be interchanged independent of the block mode. However, direct block and counter modes required separate instantiations. Although these units require Table Unit search interfaces, they share the exact same external interface and are directly swappable.

1) *Direct Block Decryption Unit*: The flow for direct block decryption begins with the CPU instruction request. This request is scaled up to a 128-bit transaction if the request is for 64-bits, to provide a full block for decryption. The request is then forwarded to the memory bus arbiter. When the bus acknowledges the transaction address, the base page of the address is supplied to the Table Unit to query encrypted status.

Even though the address request may be valid before arbiter acknowledgment, it is necessary to wait until the address has been acknowledged before querying the Table Unit. The CPU is capable of aborting before address acknowledgment. The encrypted status of the page must be returned before the first data beat arrives. This ensures the encrypted status of returned data is known without incurring additional latency from searching operations.

If the page is not encrypted, the instructions are sent directly to the CPU bus interface. One exception is with single beat transfers. These transfers have been scaled up, and are first buffered before returning the requested single beat from the larger block.

When the page is determined to be encrypted, the memory transfer is first completely buffered. The unit contains a 256-bit buffer and words are inserted into the appropriate buffer location as they arrive. The prototype features dual 128-bit cryptographic cores that process each half of a 256-bit in parallel. Once input buffering is complete, the cryptographic cores initiate decryption and return the result in an output buffer when complete. The

requested words are finally returned back to the CPU.

2) *Counter Mode Decryption Unit*: Counter mode block encryption and decryption operates by taking a non-repeating value, encrypting it using a key, and XORing the result against a specific block of data. This is achieved in the Secure Software architecture through the use of base and offset values.

The counter is a combination of a base page counter, stored as ancillary data in the tables, and the offset address into the page of the requested transaction. The base page counter ensures no values are repeated between pages, and the offset address component ensures there are no repeated portions within the same page.

As with block mode, the status look-up from the Table Unit begins with the acknowledgment of address during the arbitration cycle. Once the search is complete and the status indicates encrypted, a 256-bit counter specific to the 256-bit aligned region of the page is encrypted using the retrieved key, ancillary data, and the offset address. When encryption of the counter is complete the result is stored in a 256-bit XOR map.

As instructions are returned from memory they enter a FIFO along with their associated address within the transfer block. If the transaction is not encrypted, the words are removed from the FIFO in the sequence they arrived. Otherwise, any instructions will wait in the FIFO until counter encryption is complete. Once encryption is complete, instructions waiting in the FIFO or entering the FIFO are XORed against the corresponding word within the XOR map before forwarding to the CPU.

B. Table Unit

The Table Unit contains several small memories necessary for storing and searching page-to-key and page-to-ancillary data mappings into one module. The front end of the search path begins with a memory to associate page addresses with a *page index*, similar to a TLB entry.

To allow reuse of the same key for multiple pages, a secondary memory is used to return the key slot a particular page entry requires. Similarly, a parallel memory is used to return the ancillary data associated with the page entry. Finally, a key table is used to store the keys used by the architecture. The encrypted status flag indicates when the key slot pointer does *not* point to an address of zero.

The primary element is a Content Addressable Memory, CAM, which provides an extremely low latency reverse map from page address to page table slot. The resulting table index provides the address into the page-slot and page-ancillary memories. Finally, the result

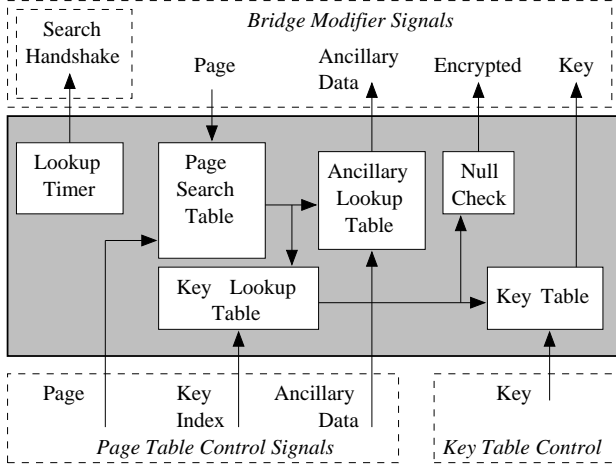


Fig. 4. Table Unit. Search pathways.

from the page–key slot look–up is addressed into the key table to retrieve the key. This is depicted in Figure 4.

To load the tables with appropriate information there are two dedicated configuration interfaces. The first interface, the page table control interface, is attached to the CPU bus and loads the page–key slot and page–ancillary data mappings. The other interface, the key table configuration interface, attached to the SKU bus and is used to load the keys into the key slots. Both configuration interfaces are write–only. Although the CPU and SKU access an interrelated set of tables, the CPU cannot write or read keys from the key table, and the SKU cannot write or read from the page mapping tables.

C. Page Table Control Unit

The Page Table Control Unit provides the control register set and logic for the CPU to interface with the Table Unit. The unit contains registers for storing the page address to search in the tables, an index to store the page mappings, the key slot the page address is mapped to, and 128–bits of ancillary information associated with the page. Writing to the page index register acts as a trigger, which in turn writes the contents of the page address, key slot, and ancillary data registers into the Table Unit at the location specified by the page index register.

D. Key Table Control Unit

Similar to the Page Table Control Unit, the purpose of the Key Table Control Unit is to enable the SKU to load keys into specific slots within the Table Unit. Individual words of the key are written to registers in the Key Table Control Unit. Writing to a key index register initiates a

transaction of the contents of all key registers to the Table Unit at desired slot contained in the key slot register.

E. Control Interface Unit

The Key Table Control Unit and Page Table Control Unit both use the same generic interface to their respective controlling devices. The Control Interface Unit translates this generic interface to the actual bus interface of the controlling device. This allows various units to be constructed and used specifically for a target configuration, without altering the functionality of the rest of the EMU. In the prototype, both the CPU and SKU communicate to the EMU over the On–Chip Peripheral Bus (OPB), therefore an OPB Control Interface Unit was generated.

IV. MODELING

When evaluating the application performance effect of an Encryption Management Unit there are two aspects that should be considered. First, the overall delay resulting from using EMU must be modeled. Once this is sufficiently formulated, it can be incorporated into additional models representing the relative effect of the EMU on applications.

The majority of delay resulting from the EMU can result from buffering and decryption. This penalty over a system without an EMU only occurs during instruction cache misses. To isolate the effect of the EMU, the total execution time is separated into two parts, shown in figure 1. t_{busy} reflects time the system is not stalled on instruction fetches. This includes waiting on system calls, processing data, running instructions from cache, and more. $t_{stalled}$ reflects time stalled on an instruction fetch.

$$T_{Total} = t_{busy} + t_{stalled} \quad (1)$$

$$\%T_{stalled} = \frac{t_{stalled}}{t_{total}} \quad (2)$$

Equation 3 expresses the total time stalled on cache misses for the base system. M_{App} reflects the number of application cache misses. L_{Mem} reflects the average latency of the cache miss. Memory latency can vary slightly from application to application due to pre–fetch and speculative execution behavior.

$$t_{Stalled} = M_{App} \times L_{Mem} \quad (3)$$

Equation 3 is modified to account for differences between encrypted and unencrypted fetch latencies. In the Secure Software architecture, an application can have encrypted and unencrypted pages. Time stalled in an encrypted environment is shown in Equation 4. M_{Enc} and M_{Unenc}

are the number of encrypted and unencrypted cache misses respectively.

$$t_{StalledEnc} = M_{Enc}(L_{Mem} + L_{EMU}) + M_{Unenc}L_{Mem} \quad (4)$$

In the prototype implementation, the EMU behaves purely as a delay element to the instruction stream. There are no performance enhancing features within the EMU to reduce latency from the base system. Therefore, the EMU directly adds a fixed latency dependent on decryption mode and algorithm. This results in Equation 5, which represents the total execution time as a function of EMU latency L_{EMU} , memory latency L_{Mem} , and application stalled behavior.

$$\begin{aligned} T_{TotalEnc} &= t_{busy} + t_{StalledEnc} \quad (5) \\ &= t_{busy} + M_{Enc}(L_{Mem} + L_{EMU}) + \\ &\quad M_{Unenc}L_{Mem} \end{aligned}$$

With the effect of the EMU framed within total execution time, a new model can be generated to show the relative change in performance by the EMU with an encrypted application. The relative effect is an important metric for understanding the performance change caused by the EMU in the context of the application type and system load. Application types, which are important to this modeling, vary on two primary parameters: time in the process wait queue and instruction locality.

Time in the operating system wait queue can result from multitasking context switching and system calls, among other reasons. Instruction locality represents how effective an application is at keeping its instructions in lower level cache, which also depends on hardware cache size and replacement strategy. In one extreme, an entire program can fit into cache, incurring stall latency only on the initial fetch. In the other extreme, instructions will always be replaced in cache after their use, requiring additional fetches from memory for the same instructions.

High code locality and high wait queue usage increase the time busy relative to $t_{stalled}$. If $t_{busy} \gg t_{stalled}$ for an unencrypted application, then there will be little difference in performance with increases to $t_{stalled}$ when encrypted. Likewise, if $t_{stalled} \gg t_{busy}$, then small increases to $t_{stalled}$ due to the EMU will have a larger effect on the total execution time.

Amdahl's equations for speed-up provides a foundation for evaluating the absolute effect of the EMU relative to application types and system loads. As the EMU adds latency, the corresponding slow-down, shown in Equation 6, is used instead. In the Secure Software system this slow down is shown in Equation 6.

$$SlowDown = \frac{T_{new}}{T_{old}} = \frac{T_{TotalEnc}}{T_{Total}} \quad (6)$$

If the application is entirely encrypted, Equation 6 can be significantly reduced. As all instructions of an application become encrypted, M_{Enc} becomes M_{App} , and M_{Unenc} converges to 0. The equation reduces to 7.

SlowDown

$$= \frac{t_{busy} + M_{Enc}(L_{Mem} + L_{EMU}) + M_{Unenc}L_{Mem}}{t_{busy} + M_{App}L_{Mem}} \quad (7)$$

$$= 1 + \frac{L_{EMU}}{L_{Mem}} \times \frac{M_{Enc}L_{Mem}}{t_{busy} + M_{Enc}L_{Mem}} \quad (8)$$

V. BENCHMARKING

To support analysis of the Secure Software architecture a benchmark to measure latency from the instruction side of the processor was required. Most research in the field of software protection use the SPEC CPU benchmarks [14] to relate the effect of their hardware against various application types. Unfortunately, the nature of this application based benchmark does not quantify the performance impact for use in mathematical models, leaving these models without validation and latency assumptions not verified. There are also indications [15] that SPEC CPU does not provide a sufficient load on the instruction side bus to properly evaluate performance of instruction side devices.

One of the leading benchmarks in memory subsystem latency measurement is LMBench's *lat_mem_rd* benchmark [16] [17]. The benchmark begins by establishing a circular linked list within an increasing size of memory, where each pointer is separated by a certain stride. The last element in the list circulates back to the first. Elements point backward to reduce likelihood of pre-fetching by the processor. For each size of memory, the pointers are walked for a number of iterations to ensure the execution time is sufficiently within the resolution of the timer. Walking the list involves a single instruction iterated over many times, which loads the value at a memory location into the register that held the memory location, and repeats.

For small memory areas, most of the list elements remain in cache. As the area grows, cache can no longer hold the list elements, which results in external memory fetches from external memory. In each case, removing the time spent executing instructions and dividing by the number of strides performed will result in the average latency for traversing a particular size of memory. These results produce visible "plateaus" of latency with increasing area indicating the access latency of various memory subsystems.

Unfortunately, while *lat_mem_rd* provides a validated solution for measuring latency of memory access, it does

so using the data side bus of the processor. To enable this measurement, a new benchmark, *iBench*, was created using similar methods as *lat_mem_rd*. Instead of using a circular linked list, static memory pages were created during compile containing conditional branch instructions set stride length apart. Multiple executable files were created for each memory size. Using hardware counter registers, the branch loop would be executed many times over using single instruction calls as before. The loop would conditionally return to the calling function after the iteration counter had completed.

In this manner, timing and latency calculations could remain similar to *lat_mem_rd*. The *iBench* benchmark was used to compare its calculations of latency on the instruction side bus against LMBench’s calculation for memory fetch latency on the data side bus. As the PowerPC 405 cache units operate using similar cache performance, the results were expected to be the same for both benchmarks. In fact, there was less than 0.25% error in the numbers obtained from *iBench* compared to *lat_mem_rd*.

VI. RESULTS

The purpose of characterizing the EMU is to determine the difference between encrypted and unencrypted executables running on a platform. Although the EMU handles both encrypted and unencrypted executables, a reference system running without the EMU will also be measured to provide a base profile.

Latency measurements for the five profiles were measured using *iBench* and are shown in Figure 5. Each cycle on the processor bus is 10 ns long, and applications strictly use four 64-bit word cache lines. The single beat 64-bit and double beat 128-bit modes are only used during the kernel boot–strap phase and their use is not indicative of actual application performance.

Analyzing the latency plot demonstrates that results are close to what was expected. The first interesting result is the slightly higher latency for unencrypted executables using the EMU, compared to the base profile without an EMU. This difference is due to the the lack of support for address pipelining on the processor bus, evident in both block mode and counter mode decryption configurations. Additionally, 10 ns of delay from counter mode over direct block mode results from instructions always passing through the FIFO unit in counter mode.

The difference in latency between encrypted and unencrypted executables on the EMU enabled system are as expected. The decryption unit uses cryptographic algorithms with a typical non–pipelined 12–cycle latency. For block mode, the pre–buffering before decryption adds

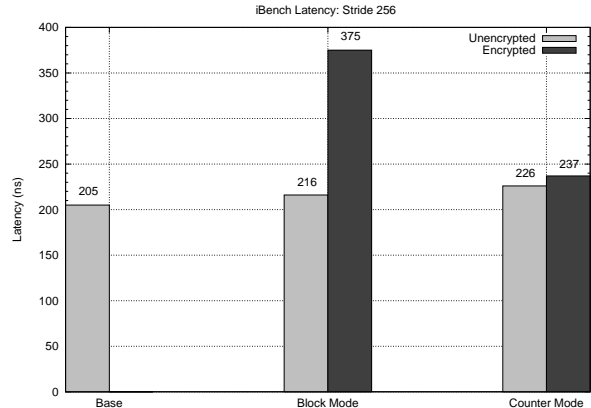


Fig. 5. Latency measurements for configuration profiles

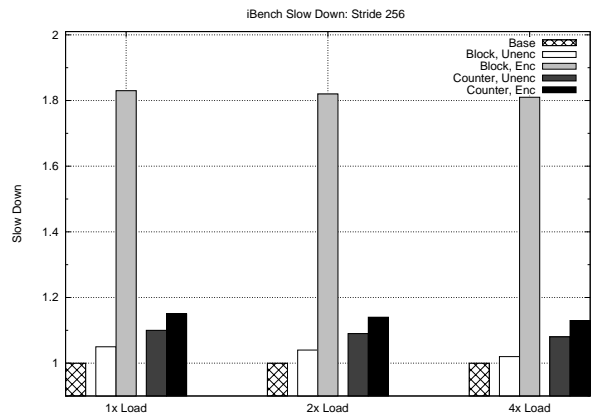


Fig. 6. Slow down for various profiles under several system loads

four additional cycles, resulting in the expected 16–cycle additional latency beyond unencrypted mode.

The EMU operating in counter mode also demonstrates expected results. For the prototype, the memory require requires 12 cycles to fetch data after arbitration, plus the one cycle for FIFO entry. For encrypted operation, two cycles for encryption status, key, and counter look–up, and 12 cycles for counter encryption. This results in one additional cycle beyond unencrypted execution.

Figure 6 shows the slow down for each profile with external memory fetches. 1x load corresponds to the measurements used in Figure 5. Each load indicates how many instances of the benchmark were run (multi–tasked) at the same time. An important factor is that the slow down decreases, although not drastically, as load increases.

One application is spending more time waiting on other applications. This is true for encrypted and unencrypted modes alike. However, proportionately less time is stalled on encrypted instructions, and more time busy pro-

cessing (unencrypted) context switches for the encrypted applications.

These results also verify the models derived in Section IV. For 1x load, it is a safe assumption that the benchmark is stalled approximately 100% of the time. L_{Enc} and L_{Mem} are substituted for the latencies in Figure 5. Memory latency is defined as the time spent fetching from unencrypted external memory, discounting the cache access time. Encryption latency is the time spent fetching and decrypting from external memory, less the time spent for the corresponding unencrypted fetch.

For block mode, Equation 8 results in a calculated slow down of 1.79 under 1x load, compared with a measured slow down of 1.83. For counter mode, Equation 8 results in a calculated slow down of 1.14, compared with a measured slow down of 1.15. The error for both methods is less than 2.2%, further confirming the assertions of Equations 7 and 8. Measured slow down for both profiles is based on actual execution time.

VII. FUTURE DIRECTIONS

Future directions for this work include features specific to the EMU and the general SecSoft architecture. The primary direction would provide support for data side encryption and decryption. This would allow additional research and validation into data protection in addition to instruction protection. Data protection requires stream encryption in addition to decryption, and possibly using more advanced transfer techniques such as burst mode or direct memory access (DMA).

VIII. CONCLUSIONS

Providing secure software protections through extensions to standard hardware and software mechanisms will allow easier adoption of this technology in modern computing environments. The SecSoft project defines such an architecture. This work presents the successful design and implementation of an Encryption Management Unit to support software protections in hardware for the Secure Software Project.

The EMU maintains the SecSoft goal of remaining compatible with many modern computing architectures. In addition to demonstrating that the EMU can work with a modern architecture, it was designed to allow the implementation to be extended and directly reused with other architectures. The definition of a cohesive set of modules allows specific sets of functionality in the EMU to be tailored to target platform, without requiring modification to the remaining system.

Although performance of the EMU is heavily dependent on the cryptographic routine and mode, the EMU itself must still maintain minimal latency. The prototype provides a very efficient solution for handling page mapping and key retrievals. Encrypted status and results are available on the second cycle after transaction acknowledge, well before instructions are returned from memory.

Although the EMU serializes any address pipe-lining by the processor, it is still possible for other platforms to invest the additional hardware necessary for this support. Counter mode encryption was shown to produce an extremely efficient two or three cycles of overhead. Similarly, block mode saw a slow down of only 1.83.

The models created in this work provide methods to calculate the absolute and relative effects of the EMU. Two equations were derived for calculating slow down of an encrypted executable. This permits the ability to determine slow down depending on different sets of information known about the application under test. The same solution was also determined in [18], which aids in validation of these models. Benchmarking the system demonstrated the models were accurate within 2.2% of actual results.

A new benchmark was created to test the EMU and to fill a significant void in benchmarking within the secure software field of research. Emulating the methods used by proven data-side memory latency benchmarks, the developed instruction-side benchmark, iBench, achieved less than 0.25% error in comparison for memory fetch latency. With the accuracy of the benchmark established, the benchmark was able to characterize the EMU through latency measurements of several platform configurations, and remain useful for model validation.

REFERENCES

- [1] R. Best, "Preventing software piracy with crypto-microprocessors," in *Proceedings of the IEEE Spring COMPCON 80*, San Francisco, CA, February 1980, pp. 466-469.
- [2] M. G. Kuhn, "Cipher instruction search attack on the bus-encryption security microcontroller ds5002fp," *IEEE Trans. Comput.*, vol. 47, no. 10, pp. 1153-1157, 1998.
- [3] Business Software Alliance and IDC, "Second annual BSA and IDC global software piracy study," May 2005.
- [4] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Aegis: architecture for tamper-evident and tamper-resistant processing," in *ICS '03: Proceedings of the 17th Annual International Conference on Supercomputing*. New York, NY, USA: ACM Press, 2003, pp. 160-171.
- [5] W. Shi, H.-H. S. Lee, C. Lu, and M. Ghosh, "High speed memory centric protection on software execution using one-time-pad prediction," Georgia Institute of Technology, Atlanta, GA, Tech Report GIT-CERCS-04-27, July 2004.
- [6] D. Lie, C. A. Thekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," in *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating*

Systems Principles. New York, NY, USA: ACM Press, 2003, pp. 178–192.

- [7] The Trusted Computing Platform Alliance, 2005, www.trustedcomputinggroup.org.
- [8] A. Huang, “Keeping secrets in hardware: the Microsoft Xbox (TM) case study,” Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, MA, Tech Report AIM-2002-008, May 2002, mITLCS-TR-872.
- [9] J. Edmison, “Secure software platform: Achieving software security via COTS augmentation,” June 2005, to be presented at ERSAC 2005.
- [10] *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet. Data Sheet 83*, Xilinx, March 2005.
- [11] *PowerPC 405 Processor Block Reference Guide. User Guide 18*, Xilinx, August 2004.
- [12] V. Fischer and M. Drutarovsky, “Two methods of rijndael implementation in reconfigurable hardware,” in *CHES '01: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*. London, UK: Springer-Verlag, 2001, pp. 77–92.
- [13] U.S. National Institute of Standards and Technology, “Specification for the advanced encryption standard,” *Federal Information Processing Standards Publication 197*, November 2001.
- [14] Standard Performance Evaluation Corporation, “SPEC CPU 2000,” 2005, www.spec.org.
- [15] M. J. Charney, “Prefetching and memory system behavior of the SPEC95 benchmark suite,” *IBM Journal of Research and Development*, vol. 41, no. 3, February 1997.
- [16] L. McVoy and C. Staelin, “Imbench: Portable tools for performance analysis,” *Proceedings of the 1996 USENIX Technical Conference*, pp. 279–295, January 1996.
- [17] C. Staelin, “Imbench – an extensible micro-benchmark suite,” HP Laboratories, Israel, Tech Report HPL-2004-213, December 2004.
- [18] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, “Architectural support for copy and tamper resistant software,” in *ASPLOS-IX: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM Press, 2000, pp. 168–177.

This material is based on work supported by the United States Air Force under Contract Number FA8650-04-C-8005. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Air Force.