# Beyond Full Disk Encryption: Protection on Security-Enhanced Commodity Processors*

Michael Henson and Stephen Taylor

Thayer School of Engineering, Dartmouth College

**Abstract.** Modern computer systems exhibit a major weakness in that code and data are stored in the clear, unencrypted, within random access memory. As a result, numerous vulnerabilities exist at every level of the software stack. These vulnerabilities have been exploited to gather confidential information (e.g. encryption keys) and inject malicious code to overcome access controls and other protections. Full memory encryption (FME) would mitigate the vulnerabilities but the CPU-memory bottleneck presents a significant challenge to designing a usable system with acceptable overheads. Recently, security hardware, including encryption engines, has been integrated on-chip within commodity processors such as the Intel i7, AMD bulldozer, and multiple ARM variants. This paper describes on-going work to develop and measure a clean-slate operating system – Bear – that leverages on-chip encryption to provide confidentiality of code and data. While Bear operates on multiple platforms, memory encryption work is focused on the Freescale i.MX535 (ARM Cortex A8) using its integrated encryption engine.

**Keywords:** Memory encryption, data in use, security-enhanced commodity processors, secure microkernel, mobile platform security.

## 1 Background and Threat Model

Current operating system designs have sought to utilize a *static* base of trust and extend trust into software through deliberate layering [Arbaugh et al. 1997]. Modern computer systems, even those protected by full disk encryption (FDE) [Brink 2009], exhibit a major weakness in that code and data are stored in the clear, unencrypted, within *memory*. These sensitive details are not only available to applications; they are known to persist in multiple unexpected locations (kernel and application), for longer than traditionally thought, even after an application exits [Chow et al. 2004], [Dunn et al. 2012], [Tang et al. 2012]. Unfortunately, this invalidates basic security assumptions rendering it possible to gather confidential information, including encryption keys, passwords, and other sensitive information that can be used to undermine trust [Halderman et al. 2008], [Boileau 2006], [Steil 2005], [Henson and Taylor, 2012]. To exacerbate the problem, memory vulnerabilities extend to *every level of the software*

* This material is based on research sponsored by the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-09-1-0213.

M. Jacobson et al. (Eds.): ACNS 2013, LNCS 7954, pp. 307–321, 2013.
© Springer-Verlag Berlin Heidelberg 2013

*stack* and the opportunities for exploitation extend well beyond physical attack to include remote attacks over the Internet: techniques have evolved that allow malicious code to be injected into device drivers, operating system kernels, and user processes.

To exploit memory vulnerabilities, numerous attack vectors have been developed. In a cold boot attack, for example, memory is frozen using a refrigerant and then removed from the computer. It is then quickly placed into a specially designed system that reads out its content, targeting encryption keys and other sensitive information. This particular attack has recently been shown to be applicable to smart phone devices as well as traditional desktops via the forensic recovery of scrambled telephones (FROST) operating system [Muller et al. 2012]. Besides capturing the encryption key, FROST was used to capture other code and data to include photos, websites visited, e-mails, contact lists, networking credentials and complete ELF binaries. Another particularly effective attack, bus-snooping/injecting, allows information to be captured or inserted via the bus lines between system components [Boileau 2006].

The threat model for this work involves an adversary gaining physical access to a computer system with sufficient resources and motivation (e.g. criminal and point of sale systems or government sponsored attacker and mobile military systems) to pursue the vulnerabilities mentioned above. For example, the smart phone of a diplomat may be confiscated for a period of time while transiting through airport security. Methods of physical access may be used to capture memory and/or disk contents for offline analysis with the sole purpose of the attack being data exfiltration. In another example, an unmanned aerial system (UAS) might be captured and control programs reverse engineered to enable the attack of other similar systems.

In contrast to research on intrusion detection, our research group is focused on exploring methods to *increase attacker workload*, undermining surveillance, forensics and persistence while *reducing the attack surface*. This paper focuses on one such method -- *memory encryption* – explored within the context of a modern microkernel.

## 2 Related Work

In effect, the increasing adoption of full disk encryption (FDE) has pushed the vulnerabilities associated with persistent data on disk down into the next level of the memory hierarchy, which has proven equally vulnerable. The key concept by which vulnerabilities were mitigated on disk was encryption: encrypting the disk provided confidentiality preventing access to sensitive information. By migrating the same solution down into RAM, it may be possible to circumvent similar attacks at this lower level of the memory hierarchy. This constrains the *boundary* available to an attack to lie at the *processor itself*, presenting a barrier that, in most cases, cannot be defeated without mechanical or electrical destruction of the processor chip. Attacks on the device are possible, for example, by etching away the chip walls with acid to reveal internal bus lines, or electromagnetic and differential power analyses [Pope 2008], [Kocher et al. 1999]. These approaches clearly increase the attacker workload by at least an order of magnitude, require expert knowledge, and cannot be exploited remotely over a network [Suh et al. 2007]. Moreover, while tamper resistant

mechanisms are already available that significantly increase the barrier to entry [Chari et al. 1999], protecting circuits from invasive and side-channel attacks is an open research area.

Although the concept of *memory encryption* has been actively researched for over three decades, it has yet to be used at the core of operating system designs to provide confidentiality of code and data [Henson and Taylor 2012]. The literature on memory encryption is largely concerned with three core approaches based on hardware enhancements [Lie et al. 2000], [Rogers et al. 2005], [Su et al. 2009], operating system enhancements [Chhabra et al. 2011], [Chen et al. 2008], [Peterson 2010], and specialized industrial applications [Dallas 1997], [Arnold and Doorn 2004], [Steil and Domke 2008]. Unfortunately, almost all of the hardware and operating system enhancements have only been implemented through simulation or emulation, and as a result, the claims have yet to be validated and quantified on practical systems. The few processors that implement memory encryption are characterized by low speeds and small addressable memory (<=16 bits) at use in low throughput (e.g. point-of-sale, set top TV access, etc.) applications or specialized gaming systems.

Recently, security hardware, including encryption engines, has been integrated within commodity processors such as the Intel i7, AMD bulldozer, and multiple ARM variants; however, systems developers have yet to embrace these specialized, often vendor-specific, features [Vasudevan et al. 2011]. Little practical experimentation has been conducted and the improvements in security and performance have yet to be quantified [Henson and Taylor 2012]. While this new hardware has not been used to protect an entire system, there are examples of its use to protect particular applications. Several papers have highlighted approaches to mitigate attacks on FDE. For example, Tresor [Muller et al. 2011], aims to protect the FDE key by storing it only inside the CPU and performing encryption/decryption within that boundary. Unfortunately, this technique is inadequate since it is possible to recover the key via a DMA injection attack on unprotected memory [Blass and Robertson 2012]. In another example, memory vulnerabilities were used to undermine the memory encryption protections of the Xbox 360. In the original Xbox, the key was stored in plaintext and transmitted across the southbridge bus. The key was captured in a bus-snooping attack, which led to compromise of the gaming system and to the subsequent growth of the Xbox mod-chip industry [Steil 2005]. In the updated Xbox 360, memory encryption is used to protect against such attacks; however, it appears that the process stack is not encrypted and this has led to another successful compromise [Steil and Domke 2008].

Unfortunately, little work has been performed to explore the trade space of using security enhanced commodity processors to implement *full memory encryption* (FME): encrypting all components of a process – stack, heap, code and data. Although more recent processors make memory encryption less costly, it remains unclear if FME is viable for everyday use or is limited to constrained tactical applications. In past ME work, overhead has been measured at the coarse granularity of an entire process without regard to process sub-components. The relationship between the overhead costs and security gains for encrypting particular process components needs to be understood (e.g. is there a particular component that can be protected with

low overhead yet holds high value code/data). This work is the first to implement ME on a commodity processor, thereby allowing investigation of the low-level implementation details and the cost/security tradeoffs at sub-process component granularity.

Memory vulnerabilities are common in systems ranging from servers and standard desktops to mobile computing devices (e.g. smart phones, tablets, laptops, etc.). However, usage patterns toward the mobile end of the spectrum may exacerbate the situation since many users of smart phones rarely reboot these systems maintaining them in an "always on" fashion [Karlson et al. 2009]. In fact, in a study of the Android operating system, 6 out of 14 applications permanently maintained their passwords in RAM. Additionally, mobile devices are more likely to be lost or stolen providing physical access to possible adversaries. In NYC, for example, 49% of the population has experienced mobile phone theft and/or loss [Tang et al. 2012]. Mobile devices, such as Android based smart phones, are beginning to be used in forward deployed military areas. These phones are loaded with information such as local maps, objectives, and blue force tracker (friendly unit) locations. Unfortunately, these phones (and other devices such as remotely piloted airframes with similar embedded processors) could easily fall into enemy hands. In fact, a recent U.S. Air Force document entitled Air Force Cyber Vision 2025 highlights the need for trust-based techniques to protect captured mobile devices in adversarial territory against reverse engineering efforts [United 2012]. While ME should be considered for both standard desktop and mobile devices, the work described here targets the ARM Cortex A8 which is common to many smart phones and tablets, including Apple's iPhone 3GS and 4, iPad first generation, iPod touch 3rd and 4th generations, and Samsung Galaxy Tablet to name a few.

## 3 Approach

The approach described in this paper is to implement memory encryption within a clean-slate microkernel design – *Bear* – leveraging security-enhanced commodity processors to ensure that code and data *never appear in the clear outside the processor chip boundary* as shown in Figure 1. The motivation for a "from scratch" kernel rests on the desire to conduct experiments in the context of a minimalist, secure microkernel. The design separates core functions into protected layers typical of modern microkernel designs such as MINIX [Tannenbaum and Woodhull 2006]. Monolithic operating systems, such as Linux and Windows, contain millions of lines of code and have a large runtime footprint providing ample opportunity for exploitation. In addition, they rarely enforce protections and allow device drivers direct access to kernel-space. In contrast, the Bear system used in this research involves approximately 3000 lines of code, with a runtime footprint of less than 50Kbytes on the ARM A8, making it an ideal platform to explore the tradeoffs involved in memory encryption in the presence of a small attack surface. All potentially compromised device drivers are executed in user-space, where they are non-deterministically regenerated to refresh trust and undermine persistence. Versions of the system operate on 64-bit Intel X86-based multi-core blade servers and ARM M3, A8, and A9 processors. On 64-bit

systems MULTICs style protections are strictly enforced through paging structures to increase attacker workload; these added protection techniques are not used in the experiments described here in order to quantify the baseline overheads independently.

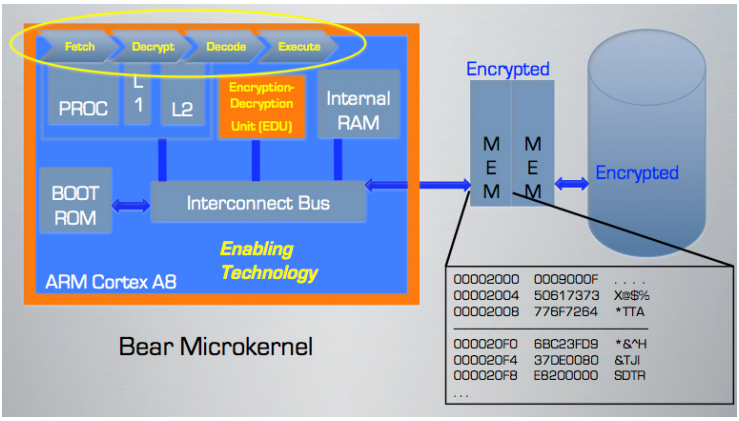


**Fig. 1.** General Approach for Memory Encryption

**Hardware.** ARM licenses the design of the basic processor (e.g. the A8) while various vendors build them with additional functionality. The exploration of memory encryption described here is focused on Freescale's i.MX535 applications processor. Critical components of the processor for this research include the internal RAM (iRAM-128 KB + 16 KB "secure"), symmetric asymmetric hashing and random accelerator (SAHARA), L1/L2 cache (32KB Harvard L1, 256KB L2), and the NEON single instruction multiple data (SIMD) coprocessor. These components are common to other ARM processors that include security hardware. Most of the techniques in the memory encryption literature targeting hardware involve modifying the fetch-decode-execute (FDE) engine to include decryption (fetch-decrypt-decode-execute) while adding encryption acceleration and internal storage space. Without specialized FDDE hardware, data can not be decrypted and placed directly into caches and execution pipelines. This results in a requirement for significant internal space in which to store and operate on sensitive, plaintext information.

SAHARA implements AES, DES and 3DES encryption, MD5, SHA-1, SHA-224, and SHA-256 hashing and hardware based (ring oscillator) random number generation. It also provides its own DMA controller with an AHB bus interface to reduce the interaction/burden on the primary CPU. For AES encryption, SAHARA includes electronic codebook (ECB), cipher-block chaining (CBC), counter (CTR) and counter with CBC-MAC (CCM) modes of operation. Descriptors are used to notify SAHARA of blocks of memory (internal or external) for encryption/decryption. Internal (secure) registers are cleared after a descriptor chain has completed processing to provide for usage by multiple, mutually distrusting processes. Completion of encryption/decryption is signaled via an interrupt. The encryption-decryption unit (EDU), is controlled via a *descriptor chain*, consisting of six 32-bit words. Each bit or group of bits (generally 2-3) are selected to enable the hardware module

(e.g., encryption, authentication, random number generation), algorithm (e.g. RSA, DES), mode of operation (e.g. electronic codebook, cipher block chaining) and other details. A security API was developed to hide proprietary Freescale encryption details and is responsible for building the appropriate descriptor chain in the latest prototype. For example, the following function call:

*EDU('E', 0x000001A0, 0xF8000000, 0x70000000, 0xF801FFFD);*

causes the encryption unit to encrypt (E=encrypt, D=decrypt) a process block of 416 bytes -- the current size of a process descriptor and stack -- from iRAM at location *0xF8000000,* placing the result in external RAM (eRAM) at location *0x70000000*.

For simplicity, a 128-bit AES symmetric key is downloaded via JTAG into iRAM and used for all process encryption. In practice an out-of-channel or standard key distribution scheme would be used in a full system implementation [Mel and Baker 2001]. Several other techniques for key management are described in the memory encryption literature. For example, one scheme generates a new random key at system reset; this key is used to encrypt processes, which are initially stored in plaintext [Chen et al. 2008]. Other work describes the method by which programs are delivered encrypted. Programs developed externally are encrypted using a public key. The private key, stored inside the processor, is used to decrypt the program in iRAM. The program is then re-encrypted with a randomly generated symmetric key to improve encryption performance. Regardless of the key generation and escrow techniques used, the keys are *never* available in eRAM. In the work described in this paper, there is space for storage of many keys whereas several of the approaches to protecting FDE schemes rely on internal registers (e.g. SSE, debug, etc.) limiting storage to a small number of keys [Muller et al. 2011], [Muller et al. 2012].

**Static Encrypted Processes**. The initial memory encryption proof-of-concept was implemented on the ARM A8 processor, using the Freescale SAHARA encryption engine, with the MMU and cache disabled. In this method, only the code is encrypted, using 128-bit AES symmetric-key encryption, and stored on disk as part of the executable binary. Other process components (data, stack, heap) are never encrypted as they remain within the protected iRAM. A small bootloader stored in internal ROM is responsible for initializing the hardware and loading the microkernel over the JTAG interface directly into iRAM. Next, a shell is bootstrapped using the on-board USART connection to allow programs to be executed. User processes are added to the scheduling queue and executed from iRAM. The microkernel then begins execution by decrypting the user process code and storing it into iRAM. This technique, referred to as *static encrypted processes,* only performs decryption once at code loading and is relevant to embedded systems where processes fit entirely within iRAM [Henson and Taylor 2013]. Measurements detailed in Section 4 quantify the overhead of this approach. Other than the one-time initial decryption cost (dependent upon the size of the process code), there is little evidence of overhead using this method. Since embedded processors are continually increasing on-chip memory, this technique represents an increasingly practical, low-overhead approach to memory encryption.

**Dynamic Encrypted Processes**. A more general case, *dynamic encrypted processes*, occurs where there is sufficient memory pressure (i.e. processes + data are larger than available iRAM) to force processes back to eRAM during execution. Process components include code, data (global/static), stack and heap, and iRAM buffers are created for each. The prototype allows swapping of encrypted processes to eRAM. Process segments are stored in eRAM in encrypted form and brought into iRAM, decrypted, and executed on-demand. Segments are re-encrypted before being sent back to eRAM with the exception of code, which does not change. In the absence of an enabled MMU, this movement of code and data required some virtual memory management (e.g. updating of stack pointers, addresses, program counters, jump addresses, etc.) where all segments of a given type correspond to a single internal buffer. This management was taken care of via modifications to the process creation, context switching and heap allocation routines. Figure 2 illustrates how the prototype encrypts the process control block (PCB) and stack (as one chunk); dynamically allocated memory and code are encrypted separately. The process context switch provides a natural point at which to perform decryption of these segments. Since the prototype does not utilize a paging mechanism, there is no similar point at which to intercede in accesses to global/static data, which are solely controlled by the compiler. Therefore, global/static data currently remains in iRAM.

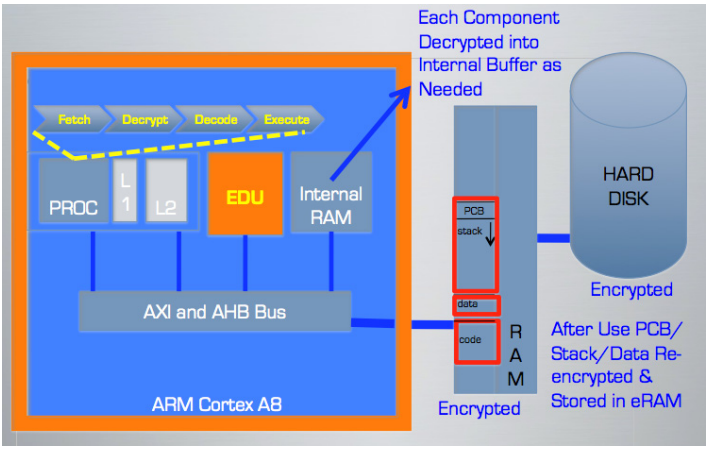


**Fig. 2.** Dynamic Encrypted Processes – Cache Disabled

The PCB-stack and code segments are of predetermined sizes while the size of the heap segments are not known a-priori. Depending on the size of the allocated segment, two alternative approaches are available. If it is small enough to fit within iRAM (after taking into consideration the space occupied by the kernel and other segments) then the whole segment is decrypted and placed within the internal data buffer in a similar fashion to the code and PCB-stack. However, if the segment is too large, then decryption of data on-demand at the size appropriate to the application (or smallest size possible) is used.

For a strenuous test of worst-case heap performance, a radix-2, in-place fast Fourier transform (FFT) based on the Tukey-Cooley algorithm was used to gauge the overhead [Press et al. 1992]. The smallest size for decryption in AES is a block of 16 Bytes. Since the data in each component of the FFT (real and imaginary part) take up one word each (4 Bytes), additional overhead is introduced in order to align the smaller data with the algorithm requirements. Whereas the unprotected version implements a simple swap of two of the real and imaginary components, the protected version must determine the appropriate 16 Byte aligned address to decrypt into the internal buffers for each component. Then the proper half of the 16 Bytes must be identified after which the swap is performed in iRAM, data re-encrypted and stored back to eRAM.

## 4 Measurement

Since the performance degradation of memory encryption results in less likelihood of its use, it is an extremely important factor in the comparison of different schemes. First, the cost of decryption was quantified in terms of total number of cycles for generic data blocks, using the Cortex A8 performance monitors. Next, the total number of cycles required for executing the unprotected system running two simple user processes was measured. Finally, the total number of cycles for protecting the various process segments of the two user processes was measured independently, allowing for the calculation of accumulated overhead (i.e. slowdown). The system runs at 800 MHz, which is used to determine the cycles-per-bit cost of decryption commonly provided in the literature. Each measurement of the context switching segments (PCB-stack and code) is based on averaging the number of cycles for 1000 context switches. The heap data encryption is tested with a single run of the FFT program using a large (128 KB) array.

**Static Encrypted Processes**. To quantify decryption speed, generic data was used as the data itself is of no consequence to decryption overhead. The average number of cycles for decrypting chunks of eRAM ranging from 16 Bytes (the smallest size possible) to 128 KB was measured in order to determine performance of the EDU in AES 128 mode. These results are directly applicable to the implemented *static encrypted processes*: Recall that the cost for protecting processes in that technique is the one-time cost of decryption of code. The results of the decryption tests are shown in Table 1 below. The overhead associated with initializing the EDU (key expansion, etc.) is approximately 8096 cycles (as shown in the first row of the table). For the other rows, the cycles per bit cost of decryption is calculated by dividing the approximate cycles by the number of bits decrypted. For example, decrypting a chunk at the smallest possible size of 16 Bytes results in a cost of approximately 71.5 cycles per bit (9152 cycles/16*8). As the decryption chunk increases the overhead remains constant so that the measure of cycles per bit decreases (better performance). This trend is shown graphically in Figure 3 below. After 4KB, the improvement in cycles per bit is reduced dramatically. The ARM Cortex A8 architecture supports page sizes of 4 KB,

64 KB, 1 MB, and 16 MB. These measurements suggest that decryption overhead may be about the same whether 4 KB or larger page sizes are selected in future implementations. They also suggest that any granularity less than 4KB (e.g. a cache line of 64 Bytes) is sub-optimal.

**Table 1.** Overhead for Decryption of Various Sizes (Chunks) of Memory

| Data Size in Bytes | Average Cycles | Std Dev | Cycles per bit |
|---|---|---|---|
| Overhead | 8096 | 40 | N/A |
| 16 | 9152 | 65 | 71.5 |
| 32 | 9664 | 60.9 | 37.7 |
| 64 (Cache line) | 10496 | 384.5 | 20.5 |
| 128 | 11712 | 55.4 | 11.4 |
| 256 | 14208 | 590.7 | 6.9 |
| 512 | 19776 | 376.3 | 4.8 |
| 1024 | 30080 | 577.2 | 3.7 |
| 2048 | 50688 | 578.2 | 3.1 |
| 4096 (Page size) | 91776 | 578.7 | 2.8 |
| 8192 | 181632 | 401.8 | 2.77 |
| 16384 | 355584 | 716.8 | 2.71 |
| 32768 | 702720 | 566.2 | 2.68 |
| 65536 | 1397184 | 560.3 | 2.66 |
| 131072 | 2785792 | 658.7 | 2.66 |

ARM processors are targeted for operations in constrained space and power environments. It is likely because of this that the performance of the EDU on the Cortex A8 is slow relative to figures presented in the memory encryption literature (which tends to target X86 processors). In AEGIS [Suh et al. 2007], a single AES unit is estimated at 86,655 gates. Yet, AEGIS is demonstrated with an OR1200 soft core in FPGA with a total size of approximately 60,000 gates (meaning the AES unit is 144% of the original core size). Recall that encryption hardware has been added to other processors such as Intel's i5 and i7 and AMD bulldozer chipsets. Intel's advanced encryption standard-new instructions (AES-NI) provide a significant speedup over both software and ARM hardware-enhanced encryption. The authors of this paper ran an implementation of TrueCrypt's encryption algorithm benchmark test on a MacBook Pro with an Intel i7 dual-core, 2.66 GHz CPU. Using a 5 MB buffer in RAM, the throughput averages 202 MB/s without AES-NI support, and 1 GB/s with it – approximately 119 cycles for 64 Bytes. This represents an improvement of 88 times over the 10,496 cycles measured on the i.MX535 (as shown above). While x86 based processors do not tend to include user accessible iRAM, the combination of improved decryption performance and large caches in those systems might enable some form of memory encryption protection. Intel has recently filed a patent for processors incorporating memory encryption, perhaps indicating a move toward support in commodity processors [Gueron et al. 2013].

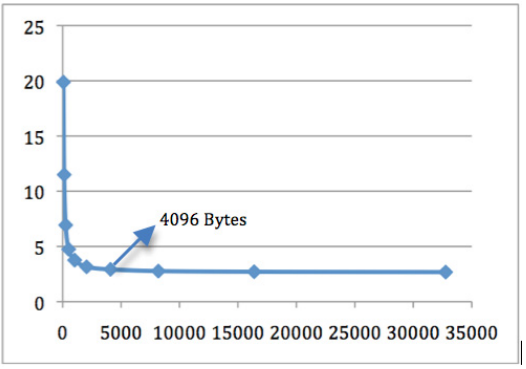


**Fig. 3.** Graph of Cycles/bit Vs. Number of Bytes Decrypted (64 B through 32 KB)

**Dynamic Encrypted Processes**. The measurements for protecting the PCB-stack and code are shown below in Table 2. The system schedules two simple processes in a round-robin fashion and for these measurements the scheduling quantum was set to 200 milliseconds, resulting in approximately 300 context switches per minute. The process behavior has nothing to do with the costs of protection since the costs are incurred during the context switch, not process execution. As in previous experiments, all measurements are averaged over 1000 context switches. The unprotected context switch routine averages approximately 20 microseconds as shown in the first row of the table. The overhead for protecting the segments is fairly large: a factor of approximately 2.9 to protect the PCB-stack and 3.4 times for both the PCB-stack and code when compared to the unprotected context. However, this cost is only incurred on average 300 times per minute. Thus the total overhead per minute is about 14,700 microseconds (.0147 seconds) giving ~1.5 seconds of overhead after 100 minutes of execution. This indicates that context and code protection are viable even without the benefit of the MMU and cache. While the size of the context and code were fixed for these experiments (416 and 672 Bytes respectively) the results from Table 1 suggest that larger component sizes (e.g. 4 KB page size) would more effectively hide the cost of the EDU initialization overhead.

**Table 2.** Overhead for PCB-Stack and Code Protection

| Component within Context Switch | Average Cycles | Std Dev | Execution Time @ 800 MHz | Overhead |
|---|---|---|---|---|
| Unprotected | 16064 | 70 | 20 us | N/A |
| PCB-Stack | 47296 | 682 | 59 us | 2.9 |
| Code | 23800 | 400 | 30 us | 1.5 |
| PCB-Stack & Code | 54976 | 856 | 69 us | 3.4 |

Table 3 shows the overhead of decryption of data in the FFT problem with 128 KB arrays holding the real and imaginary data components. Since 128 KB was too large to fit into iRAM, on-demand decryption was implemented at the size that most closely approximates data accesses (16 Bytes). The cycles per bit cost of decryption is

large at the 16 Byte size (~71.5). In summary, about 17.2 billion cycles were required to execute the unprotected FFT. Providing encryption protection during the bit reversal (first half of the FFT) only requires an additional 3.2 billion cycles (20.4 billion cycles total). Encrypting all data for the entire FFT operation requires approximately 20 billion additional cycles (37.2 billion cycles total): resulting in a slow down of approximately 2.2 times over the unprotected execution. Memory accesses in the FFT problem are pathological, providing a thorough (worst-case) evaluation of our memory encryption approach.

**Table 3.** Overhead for Data Protection in FFT Function

| FFT Data Structure @ 128 KB | Average Cycles | Std Dev | Execution Time @ 800 MHz | Overhead |
|---|---|---|---|---|
| Unprotected | 17269347514 | 70 | 21.6 s | N/A |
| Bit Reversal Only | 20438649003 | 682 | 25.5 s | 1.2 |
| Fully Protected | 37197691328 | 400 | 46.5 s | 2.2 |

In reality, most mobile processor packages include SIMD cores, such as the NEON processor to optimize algorithms like the FFT. Mobile system use tends to be characterized by applications such as chat, e-mail, and those displaying spatial/temporal locality (e.g. photo viewing). It is reasonable to believe that the performance on these more typical workloads will be considerably improved even without optimization of the on-demand decryption techniques used in this work.

It is important to understand the performance characteristics of the worst-case (on-demand decryption) scenario where decryption overhead is added directly to memory access time. It was anticipated that performing memory encryption without the benefit of the MMU and cache (including prefetching etc.) would yield excessively large overheads. While this was the case for the FFT data-structure access, PCB-stack and code protection were surprisingly efficient. Further, the slowdown for the FFT (2.2x) is considerably less than that reported in the simulation results of a similar technique that took advantage of caching mechanisms but lacked encryption hardware. In that work, slowdowns of 2.53x and 8.5x were measured when utilizing a 4 KB page with a 256 KB and 64 KB L2 cache respectively [Chen et al. 2008].

While the use of MMU/cache will make the system more closely approximate those of smart-phones, there are many examples where the techniques already developed in this work could be applicable. For example, many devices at the lower end of the embedded spectrum, including the large number of *smart electric meters* recently deployed, tend not to include MMU's [McLaughlin et al. 2010].

## 5 Future Work

A valuable next step is to take advantage of the ARM Cortex A8's MMU and cache. However, effectively modifying the fetch-decode-execute cycle requires a way to decrypt pages brought on-chip before they are loaded into the cache. The A8 architecture includes a built-in preload engine (PLE) that can be used to move data to and

from the L2 cache under software control [ARM]. This engine will be used to load the cache with decrypted data and instructions, with iRAM continuing to act as a workspace and extension to L1/L2 cache. Additionally, the NEON SIMD coprocessor is tightly coupled with the L2 cache, which may provide another method for update. Enabling the cache (and other optimization mechanisms such as prefetching) should provide significant improvement over the current decrypt-on-demand prototype.

While there are many current requirements for a from-scratch microkernel (especially in the military), this work can be expanded to incorporate currently popular operating systems. A Bear *microvisor*, quite similar to the microkernel, has already been developed. Efforts are currently under way to enable the NetBSD (5.0.1) operating system to boot on top of the microvisor, protecting it with Bear's security mechanisms. After experimenting with MMU-enabled memory encryption, the techniques will be added to the microvisor's capabilities. The microvisor can then be used on future ARM hardware (supporting virtualization) to boot mobile operating systems (e.g. Android).

## 6 Conclusions

This paper describes a clean-slate operating system design that leverages security-enhanced commodity processors to ensure that code and data *never appear in the clear outside the processor chip boundary*. By utilizing the SAHARA security hardware of the Freescale i.MX535 processor, the system provides memory encryption with various granularities of a process. The current work utilizes on-demand decryption whereby the overhead for decrypting code and data is added directly to the fetch-decode cost. In this way, an upper bound on the overhead associated with memory encryption is established. The experimental overhead associated with the protection of process PCB-stack and code is surprisingly small.

Few operating system developers have taken advantage of the new security hardware available in many commodity processors. There are various projects that utilize some aspects of this hardware, for example, to protect the key in FDE. Since sensitive data is left in memory for relatively long periods of time, it is logical to conclude that the protections afforded "data at rest" on disk should also apply to memory. By forcing an attacker to rely on brute-force attacks against encrypted memory (or other relatively difficult attacks on the chip itself) we seek to *increase attacker workload* enough to dissuade or delay the attack, allowing for mission completion (or protection of user information). The overhead displayed in the work described here suggests this protection is feasible today with security-enhanced commodity processors. While the concept of memory encryption has existed for over three decades, there are still no general-purpose, commercial-off-the-shelf solutions integrated with secure operating systems. Unfortunately, while full disk encryption seems to be the state-of-the art, it is insufficient for the protection of systems holding sensitive information.

**Notice.**The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The

views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

## References

Arbaugh, A., Farber, D., Smith, J.: A secure and reliable bootstrap architecture. In: Proceedings of the 1997 IEEE Symposium on Security and Privacy (SP 1997). IEEE Computer Society, Washington, DC (1997)

Arnold, T., Doorn, L.: The IBM PCIXCC: a new cryptographic coprocessor for the IBM eserver. The IBM Journal of Research and Development, 120–126 (2004)

Blass, E., Robertson, W.: TRESOR-HUNT: Attacking CPU-Bound Encryption. In: Proceedings of the 28th Annual Computer Security Applications Conference (December 2012)

Blunden, B.: The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System. Jones and Bartlett Publishers, Inc., USA (2009)

Anderson, R., Kuhn, M.: Tamper resistance – a cautionary note. In: Proceedings of the Second USENIX Workshop on Electronic Commerce, vol. 2, pp. 1–11 (1996)

Barrantes, E., Ackley, D., Forrest, S., Palmer, T., Sefanovic, D., Zovi, D.: Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In: Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003), pp. 281–289 (October 2003)

Boileau, A.: Hit by a Bus: Physical Access Attacks with Firewire. Presented at Ruxcon (2006)

Brink, D.: Full-disk encryption on the rise. Aberdeen Research Group Report (September 2009)

Casey, E., Fellows, G., Geiger, M., Stellatos, G.: The growing impact of full disk encryption on digital forensics. Digital Investigation 8, 129–134 (2011)

Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 398–412. Springer, Heidelberg (1999)

Chhabra, S., Rogers, B., Solihin, Y., Prvulovic, M.: SecureMe: a hardware-software approach to full system security. In: Proceedings of the International Conference on Supercomputing (ICS) (May 2011)

Chhabra, S., Solihin, Y.: i-NVMM: a secure non-volatile main memory system with incremental encryption. In: Proceedings of the International Symposium on Computer Architecture (ISCA) (June 2011)

Chen, X., Dick, R., Choudary, A.: Operating system controlled processor-memory bus encryption. In: Proceedings of DATE (2008)

Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., Rosenblum, M.: Understanding data lifetime via whole system simulation. In: Proceedings of the USENIX Security Symposium (August 2004)

Cortex-A Series Programmer's Guide, Version: 2.0, `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0013b/index.html`

Dallas Semiconductor. Secure microcontroller data book. Dallas (1997)

Duc, G., Keryell, R.: CryptoPage: an efficient secure architecture with memory encryption, integrity and information leakage protection. In: Proceedings of the Annual Computer Security Applications Conference, ACSAC (2006)

Gueron, S., Savagaonkar, U., McKeen, F., Rozas, C., Durham, D., Doweck, J., Mulla, O., Anati, I., Greenfield, Z., Maor, M.: Method and apparatus for memory encryption with integrity check and protection against replay attacks. WO patent number 2013002789 (January 3, 2013)

Halderman, J., Schoen, S., Heninger, N., Clarkson, W., Paul, W., Calandrino, J., Feldman, A., Appelbaum, J., Felten, E.: Lest we remember: cold boot attacks on encryption keys. In: Proceedings of the USENIX Security Symposium (February 2008)

Hennessy, J., Patterson, D.: Computer Architecture, 4th edn. A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco (2006)

Henson, M., Taylor, S.: Memory Encryption: A Survey of Existing Techniques. Submitted to ACM Computing Surveys (July 2012), Available as Thayer Technical Report TR13-001 at `http://thayer.dartmouth.edu/tr/reports`

Henson, M., Taylor, S.: Attack Mitigation through Memory Encryption of Security Enhanced Commodity Processors. In: Hart, D. (ed.) The Proceedings of the 8th International Conference on Information Warfare and Security (ICIW 2013), pp. 265–268 (March 2013)

i.MX53 Multimedia Applications Processor Reference Manual, `http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=IMX53QSB&fpsp=1&tab=Documentation_Tab`

Karlson, A.K., Meyers, B.R., Jacobs, A., Johns, P., Kane, S.K.: Working overtime: Patterns of smartphone and PC usage in the day of an information worker. In: Tokuda, H., Beigl, M., Friday, A., Brush, A.J.B., Tobe, Y. (eds.) Pervasive 2009. LNCS, vol. 5538, pp. 398–405. Springer, Heidelberg (2009)

Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)

Kgil, T., Falk, L., Mudge, T.: ChipLock: support for secure microarchitectures. ACM Sigarch 33(1) (March 2005)

Kuhn, M.: Cipher instruction search attack on the bus-encryption security microcon-troller DS5002FP. IEEE Transactions on Computing 47, 1153–2257 (1998)

Lee, M., Ahn, M., Kim, E.: I2SEMS: interconnects-independent security enhances shared memory multiprocessor systems. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT (2007)

Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., Horowitz, M.: Architectural support for copy and tamper resistant software. In: Proceedings of the 9th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 168–177 (2000)

McLaughlin, S., Podkuiko, D., Delozier, A., Miadzverzhanka, S., McDaniel, P.: Embedded firmware diversity for smart electric meters. In: Proceedings of the 5th USENIX Workshop on Hot Topics in Security, HotSec 2010, Wshington, DC, USA, August 1-8 (2010)

Mel, H., Baker, D.: Cryptography Decrypted. Addison-Wesley, Upper Saddle River (2001)

Muller, T., Freiling, F., Dewald, A.: TRESOR runs encryption securely outside RAM. In: Proceedings of the 20th USENIX Conference on Security (2011)

Müller, T., Taubmann, B., Freiling, F.C.: TreVisor: OSIndependent Software-Based Full Disk Encryption Secure Against Main Memory Attacks. In: Bao, F., Samarati, P., Zhou, J. (eds.) ACNS 2012. LNCS, vol. 7341, pp. 66–83. Springer, Heidelberg (2012)

Nagarajan, V., Gupta, R., Krishnaswamy, A.: Compiler-assisted memory encryption for embedded processors. In: HiPPEAC, pp. 7–22 (2007)

Peterson, P.: Cryptkeeper: improving security with encrypted RAM. In: Proceedings of the IEEE International Conference on Technologies for Homeland Security (HST), pp. 120–126 (November 2010)

Press, W., Teukolsky, S., Vetterling, W., Flannery, B.: Numerical Recipes in C, 2nd edn. Cambridge University Press, Cambridge (1992)

Rogers, B., Solihin, Y., Prvulovic, M.: Memory predecryption: hiding the latency overhead of memory encryption. ACM SIGARCH Computer Architecture News 33(1), 27–33 (2005)

Rogers, B., Prvulovic, M., Solihin, Y.: Efficient data protection for distributed shared memory multiprocessors. In: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT) (September 2006)

Shi, W., Lee, H., Ghosh, M., Lu, C.: Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems. In: Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques, PACT (2004)

Steil, M.: 17 mistakes Microsoft made in the Xbox security system. In: Proceedings of the 22nd Chaos Communication Congress (2005)

Steil, M., Domke, F.: The Xbox 360 Security System and its Weaknesses (August. 2008), Google TechTalk available at
http://www.youtube.com/watch?v=uxjpmc8ZIxM

Su, L., Martinez, A., Guillemin, P., Cerdan, S., Pacalet, R.: Hardware mechanism and performance evaluation of hierarchical page-based memory bus protection. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE (2009)

Suh, G., O'Donell, C., Devadas, S.: Aegis: a single-chip secure processor. IEEE Design and Test of Computers 24(6), 570–580 (2007)

Suh, G., Clarke, D., Gassend, B., Dijk, M., Devadas, S.: Efficient memory integrity verification and encryption for secure processors. In: Proceedings of the 36th International Symposium on Microarchitecture (2005)

Tanenbaum, Woodhull: Operating Systems: Design and Implementation. Prentice-Hall (2006)

Tang, Y., Ames, P., Bhamidipati, S., Bijlani, A., Geambasu, R., Sarda, N.: CleanOS: Limiting mobile data exposure with idle eviction. In: OSDI (2012)

United States Air Force Cyberspace Science and Technology Vision. AF/ST TR 12-01 (December 2012),
http://www.globalsecurity.org/security/library/policy/usaf/cybervision2025_afd-130327-306.pdf

Vasudevan, A., Owusu, E., Zhou, Z., Newsome, J., McCune, J.: Trustworthy execution on mobile devices: what security properties can my mobile platform give me? Carnegie Mellon University CyLab Technical Report 11-023 (November 2011)

Yan, C., Rogers, B., Englender, D., Solihin, Y., Prvulovic, M.: Improving cost performance and security of memory encryption and authentication. In: Proceedings of the 33rd International Symposium on Computer Architecture (June 2006)