No Sugar but all the Taste!
Memory Encryption without Architectural Support

Panagiotis Papadopoulos, Giorgos Vasiliadis, Giorgos Christou,
Evangelos Markatos, Sotiris Ioannidis

FORTH-ICS, Greece
{panpap, gvasil, gchri, markatos, sotiris}@ics.forth.gr

Abstract. The protection of in situ data, typically require solutions that involve different kinds of encryption schemes. Even though the majority of these solutions prioritize the protection of cold data stored on secondary devices, it has been shown that sensitive information like passwords, secrets, and private data can be easily exfiltrated from main memory as well, by adversaries with physical access. As such, the protection of hot data that reside on main memory is equally important.
In this paper, we aim to investigate whether it is possible to achieve memory encryption without any architectural support at a reasonable performance cost. In particular, we propose the first of its kind software-based memory encryption approach, which ensures that sensitive data will remain encrypted in main memory at all times. Our approach is based on commodity off-the-shelf hardware, and is totally transparent to legacy applications. To accommodate different applications needs, we have built two versions of main memory encryption: Full and Selective Memory Encryption. Additionally, we provide a new memory allocation library that allows programmers to manage granular sensitive memory regions according to the specific requirements of each application. We conduct an extensive quantitative evaluation and characterization of the overheads of our software-based memory encryption, using both micro-benchmarks and real-world application workloads. Our results show that the performance overheads due to memory encryption are tolerable in real-world network scenarios, below 17% for HTTP and 27% for HTTPS.

1 Introduction

The theft of sensitive data is an escalating problem. According to a recent study [17], it is estimated that data breaches can cost between $90 and $305 per record exposed, leading to an average cost of around $4.8 million per company per incident. To protect data stored on secondary storage devices, many approaches that provide full disk encryption have been proposed [20, 26]. The majority of these approaches encrypt all data stored on the disk using a secret key that is provided, usually, at boot time. As a result, in case of IT hardware equipment theft, physical attack, or industrial espionage, all corporate and sensitive data stored on the hard disk will be protected.

Besides the protection of (cold) data stored on secondary storage devices, sensitive data can also reside on main memory (*hot data*), where they are typically in clear-text. This permits the launching of memory attacks, and allows

the exploitation of main memory and the exfiltration of data used during execution. More importantly, it is not only servers or desktops that are under threat. According to [17], more than 40% of business users leave their laptops in sleep or hibernation mode when traveling, leaving their private or corporate data, keys or passwords residing in memory unprotected. As a consequence, an adversary is able to retrieve all data from the main memory, along with any stored sensitive data, e.g., session keys, passwords, HTTP cookies, SSL key pairs, gaining this way access to online services, bank accounts or local encrypted hard disks. Some of the typical methods adversaries utilize to steal data from main memory are cold boot attacks [11, 12, 29] and DMA attacks [6, 28].

To overcome these problems, many approaches for memory encryption have been proposed [7, 21]. These approaches integrate several architectural mechanisms to provide encryption and secure context management of data that reside in off-chip memory regions. Although these systems provide strong security guarantees with acceptable performance, their use in practice is limited, as they require hardware support and cannot be directly applied to commodity systems.

In this paper, we design the first to our knowledge software-based memory encryption approach for commodity, off-the-shelf, systems. With our approach, application data are always encrypted in main memory, using a 128-bit AES key, which is randomly generated every time the application is launched to make it resistant against key guessing attacks [18]. To cope with the computational overhead of memory encryption, we leverage the Advanced Encryption Standard Instruction Set for cryptographic operations, which is currently available in the majority of modern microprocessors. Finally, we experimentally quantify the cost of keeping sensitive data secure in practical, real-world scenarios.

To summarize, the main contributions of this work are the following:

1. We present, to the best of our knowledge, the first of its kind, design and implementation for entirely software-based main memory encryption. Our solution can work transparently without any need to modify the application.
2. We provide a library to allow the users perform partial memory encryption enabling them this way to create, at runtime, fine-grained encrypted segments depending on the application requirements.
3. We conduct an extensive quantitative evaluation of software-based main memory encryption for both static and dynamic instrumentation strategies, using both micro-benchmarks and real-world applications.

2 Our approach

The goal of our approach is to secure *hot data* of running processes by deploying main memory encryption without any architectural support. To do so, we use code instrumentation to ensure that any process' data will be stored in main memory, encrypted at all time. This way, sensitive data residing in main memory or moving among the different components of an untrusted domain, will always be protected against prying eyes.

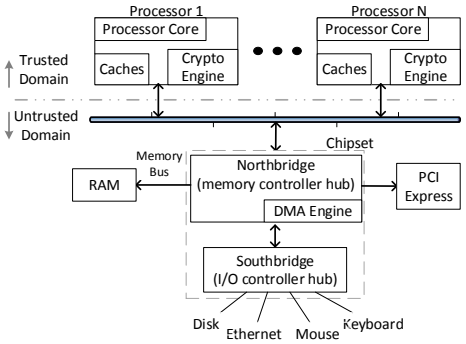


Fig. 1: Data are always encrypted when residing in main memory or moving between the different components of the untrusted domain.

2.1 Threat model

In this section, we describe the classes of physical hardware attacks within our threat model, and additional threats that fall outside the scope of our paper.

In-Scope Threats. We are concerned of adversaries that have physical access to the victim's system where sensitive information is stored, and that the machine can be exposed to physical hardware attacks: or DMA attacks.

Cold Boot Attacks. In a *cold boot attack* [11,12,29], the data remanence effect of RAM is exploited by the adversary to extract the data from the memory. There are two ways of achieving this: (i) an attacker can freeze the RAM modules using a refrigerant [22] which then physically remove from the victim's device and inserts them into a device that is capable to read the contents of the RAM; (ii) an attacker can perform a *warm boot* by running specific attack tools, and retrieve the contents of the residual memory [8]. In this type of side channel attack, the attacker is able to retrieve encryption keys and sensitive data from a running operating system even when the user session is locked. As has been shown in [27], modern SRAM chips can retain about 80% of their data for up to a minute at temperatures above -20°C.

DMA Attacks. This type of attacks leverage the ability of a DMA interface to allow a peripheral to directly access arbitrary memory regions, and read memory contents without any supervision from the processor or the OS. More specifically, an attacker can program a DMA-capable peripheral to manipulate the DMA controller and read sensitive data stored in memory [24,28]. This type of attack can be carried out over different IO buses, such as the Firewire, PCI, PCI Express or Thunderbolt.

Out-of-Scope Threats. Apart from the above attacks, obviously there are many more threats for the data residing in memory, that fall outside the scope of this paper.

Memory disclosure attacks. This type of attacks aim to compromise the software, accessing this way possible secrets and passwords. Such attacks exploit a software vulnerability to install malicious code. Although this type of attacks are quite common and important to consider, this paper focuses on attacks that do not rely on running compromised software.

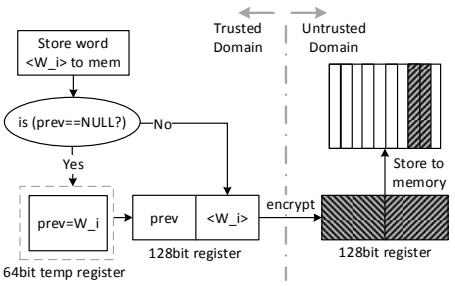


Fig. 2: Subsequent store instructions have words encrypted as a bundle in the same block and are then stored on main memory.

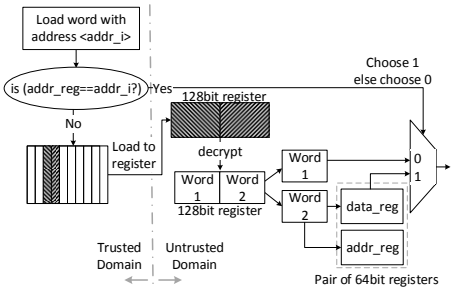


Fig. 3: For sequential memory accesses, the block is decrypted once and the 2nd word is retrieved directly from the register instead of re-decrypting the same block.

Side-channel Attacks. Such type of attacks aim to extract sensitive information by exploiting physical properties (like timing information or power consumption) of the cryptographic implementation. These attacks usually have a limited accuracy and require a relatively high level of sophistication, especially when the attacker cannot run arbitrary code on the device, therefore they fall beyond the threat model of this paper.

Sophisticated Physical Attacks. It is hard to defend against every type of physical attacks. Indeed, there are several Advanced Persistent Threats (APTs), usually deployed for corporate espionage, intelligence stealing from governmental or military infrastructures etc., which under specific circumstances, can achieve severe data breaches. However, such attacks require specialized equipment and can often take several months even when carried out by a skilled attacker.

2.2 Main assumptions

Our main assumptions, which are in line with the related literature [13], are that the processor provides a secure region, within which sensitive information can reside. As we see in Figure 1, all components outside of the processor are assumed to be vulnerable, including RAM and its interconnections, like the data and memory bus, I/O devices, etc. Additionally, we assume a trusted kernel in the target system. This is a reasonable assumption, keeping in mind that an adversary capable of controlling the kernel can cause more significant damage than just eavesdropping sensitive data. The core idea of main memory encryption related techniques is to avoid potential data breaches, and make any adversary with the above properties unable of observing, deleting, replacing or modifying any piece of data existing in a victim system.

3 Main Memory Encryption

In this section, we present our technique for main memory encryption, in order to secure the data of running applications. Specifically, we show how we instrument the load and store operations with encryption and decryption instructions.

The instrumentation of load and store instructions can be implemented in two ways, either (i) *statically*: by instrumenting the specific memory access instructions, or (ii) *dynamically*: by running the corresponding binary executable on top of a dynamic instrumentation tool.

The static instrumentation of the binary executable offers better performance, however requires the static instrumentation of all linked shared libraries as well. On the other hand, dynamic instrumentation is able to handle complex run-time code manipulation cases, such as dynamically generated (JIT), obfuscated or self-modifying code. As such, even though dynamic instrumentation has an extra performance overhead (as we will see in Section 4), it is considered more flexible and supports both shared libraries and run-time generated code.

3.1 Full memory encryption (FME)

An important design decision, when applying memory encryption, is how to encrypt the memory data. In 64-bit architectures memory operations operate up to 64-bit words. However, the AES algorithm operates in block units, where each block is a minimum of 128 bits. Hence, during each memory operation we need to collect nearby 128-bit aligned data. This is accomplished by making use of two `xmm` registers, one as a load buffer and one as a store buffer. In case of multiple encryptions, this register helps us temporarily keep data until the next store instruction targeting near data is issued. The sequential store instructions, as can be seen in Figure 2, will get a couple of words encrypted in the same block and then the block will proceed to be stored in the main memory. In case of decryption, this register allows us to *pre-fetch* data during sequential memory accesses. This way, as seen in Figure 3, when a process loads a word and then loads its very next one, it will retrieve it directly from the register instead of decrypting again the same block. This solution has the additional benefit of hiding decryption latency when consecutive words are accessed.

We note that it could be possible to use even larger blocks (> 128 bits). Such approaches may benefit from less number of performed encryptions and decryptions, which would improve the performance of programs that exhibit cache locality and reduce their overall execution time. However, it would also require quite extensive buffering, which would result to massive utilization of registers. The reason behind this is that the data will need to be in the registers for an unknown period of time, until they reach the proper size of the block. More importantly, applications will have performance gains solely in the case of sequential data accesses, while in the case of random memory accesses, a large part of the decrypted data will remain underused and be quickly evicted from the registers. Using the above encryption scheme, all data placed in the memory is encrypted, however they are still not well-protected. Given that each block is encrypted separately, an attacker is able to identify identical ciphertext blocks that yield identical plaintext blocks, after scanning the entire memory. These unprotected data patterns allow trivial attacks available in the adversary's toolchest even in the single-snapshot scenario of the cold boot attack. To remedy this issue, we use a stream cipher encryption mode of operation instead of block

cipher. The challenge of such an approach, in our case, is that applications may need to randomly access non-sequential single blocks that need to be decrypted separately. To obtain this random access property during decryption, we employ the CTR mode of operation by using a per-session random nonce and a per-block counter. This way, we turn the block ciphers into a stream cipher, eliminating the potential appearance of patterns.

Handling system calls. It is quite often for applications to perform specific operations that only the kernel has the privilege to execute. For instance, hardware-related operations (e.g. accessing a hard disk drive), or communication with integral kernel services, such as process scheduling. The request of such privileged applications (i.e. system call) usually is followed by application user data and parameters that need to be passed to the kernel. In our case, all of the data passed from user to kernel space are encrypted. As a consequence, after extracting the calling process `pid`, the kernel obtains the proper process key (see Section 3.4 below), and decrypts the parameters before and respectively, encrypts any results after executing the system call. There are system calls that are so frequently used from user-space applications, that can dominate the overall performance. To avoid the expensive performance penalty of system calls and context-switches, the kernel uses a virtual dynamic shared object (`vDSO`) mechanism. In particular, selected kernel space routines (e.g. `gettimeofday(2)`) are mapped into the address space of user-space applications by the kernel, enhancing thus the performance of these applications. Given that there is no switch to the kernel space, in our case, `vDSO` is treated like any other shared library object: by having its store and load instructions cryptographically instrumented.

Signals and non-local jumps. Another case we need to take into account is signals. When a signal arrives at an application, the used registers and the processor's state must be stored for the execution to smoothly continue afterwards from the current state. That said, in our case, the specific registers may contain sensitive data that we cannot risk to be spilled in memory in plaintext. To overcome this, we modified the kernel by using the proper process key, to encrypt their contents before saving them to `sigcontext` structure. When the specific execution continues, the loaded values are decrypted before restored back to the registers. In a similar way, we deal with the case of non-local jumps (i.e. `setjmp/longjmp`). Specifically, in case of `setjmp`, the data from the utilized general purpose registers are being encrypted before stored in a jump buffer in memory. On the other hand, in case of `longjmp` the data are decrypted after restored from the jump buffer to the registers right before the application jumps to the return address set by the `setjmp`.

Handling context switches. Typically the CPU loads data at run-time in its registers in order to perform its computations. When context switch evictions take place, all the previously used data from the registers are moved onto the stack, which resides in main memory. Considering that there are cases where this data may be sensitive, sensitive information may find its way unencrypted on main memory, if these evictions are left unhandled. In our case, these evictions may swap out to sensitive states of AES stored in `XMM` registers, even though

they were implemented to run solely on CPU. To remedy this, we modified the kernel's typical context switch procedure to encrypt the content of `XMM` registers before they get evicted and decrypt them after the process is switched back. We achieve this by encrypting and decrypting the contents, right before and right after `FXSAVE` (i.e. store to register) and `FXRSTOR`(i.e. restore from register) instructions respectively.

3.2 Selective memory encryption (SME)

Having all memory encrypted provides the best protection for all applications. However, our experiments in Section 4 show that the overhead, in terms of performance, can increase significantly. To lower the performance overhead of FME, it is feasible to encrypt only the memory regions that contain secret or sensitive data. Indeed, such approach could result in much lower overheads during execution, proportional to the size of the data that need to be protected from memory attacks. Unfortunately, though, the exact location of sensitive data in memory is very difficult to be known in advance. Instead, it will require the developer to define the exact memory regions, the sensitive data will later reside in. One solution would be to use `#pragma` directives to provide additional information about which variables will be encrypted at compile time. However, this would restrict memory encryption to static variables only and do not offer much flexibility to the developer. To address this issue, we implement a secure memory allocator, namely s_malloc to dynamically allocate arbitrary size of memory from the heap. In order to have an integral number of blocks, the memory is allocated in multiples of 128 bits. Any data written in this portion of memory allocated with this allocator will always be encrypted. To achieve this, s_malloc taints the memory regions it allocates to ensure that the corresponding memory addresses have to be encrypted or decrypted when accessed accordingly. For instance, during load operations we can determine if the loaded data originate from s_malloc and need to be first decrypted before being read. s_malloc keeps a structure for each allocation to note the starting memory address that the segment begins along with its allocated size to denote the total length of the tainted area.

De-allocated memory pages Memory pages that have been de-allocated after being allocated by an application handling sensitive data may produce left-overs. These may be readable by attackers, enabling them to retrieve parts or even the entirety of the sensitive information. Even though Linux has a kernel thread responsible for zeroing-out the freed pages, due to internal performance optimizations, there is no guarantee when this will occur. In traditional systems this could pose privacy risks as one may get to read a region of memory that contains sensitive data. In our case, this is not a problem; the reason is that after memory allocation, all sensitive data get always encrypted before being placed in heap. Therefore, sensitive memory disclosure is not possible, as an adversary will read random bits of the ciphertext.

3.3 Protecting memory from illegal access

After ensuring the confidentiality of written-through data, a problem that may arise is in case of DMA attacks, where it is possible not only to read data from

memory, but also to write. As a consequence, an attacker could inject the OS with malicious code. To mitigate this issue, we use the commodity Input-Output Memory Management Unit (IOMMU)[1] [16] to prevent malicious memory access through DMA. The IOMMU is an IO mapping mechanism, which translates device-visible virtual addresses to physical addresses using, OS-provided mappings. Besides, it also provides memory protection, where memory is protected either from malicious devices that attempt to perform DMA attacks or from faulty devices that initiate errant memory transfers. This protection is achieved by enabling the OS to restrict *who can access what memory region.* As a result, a device cannot read or write to memory that has not been explicitly allocated or mapped to it. In our case, IOMMU is properly configured in order to forbid any access to in-memory kernel or application data.

3.4 Key Management

In this section we describe how we protect the AES secret keys that are used for encryption and decryption, against all attacks within our threat-model (described in Section 2.1). Previous works have shown that it is sufficient to prevent sensitive data and algorithmic state from leaking to RAM by implementing the cryptographic operations using on-chip memory only [9, 20].

In our approach, each process is assigned with a different key, that is stored in the Process Control Block (PCB) data structure. The PCB contains all the information needed to manage a particular process, and is placed at the beginning of the kernel stack of the process. Still, since the kernel memory is vulnerable to cold boot attacks, each *process key* is encrypted before it is stored in the PCB. The process keys are encrypted using a master key which is stored, similar to Tresor [20], inside a pair of debug registers[2]. By doing so, we avoid storing any key in main memory. The reason we utilize debug registers is that, by default, they can be accessed only from ring 0 privileged level. As a consequence, they cannot be reached by malicious user-level applications and more importantly, they do not used in procedures like context setjmp/longjmp or signal handling: cases that otherwise one would have to take specific care to prevent them from being spilled into memory. Additionally, we have modified the `ptrace` system call to respond with `EBUSY` error to any application that may request the particular registers, preventing them thus from being accessed from user level[3]. We need to note at this point, that there are studies questioning such use of debug registers to store secrets [5]. According to these, an attacker is able to inject and execute code in ring 0 privilege level by deploying a DMA attack, and consequently, disclose the secrets stored in the debug registers. In our case, with the use of

[1] IOMMU can be considered commodity since both leading x86 vendors (i.e. AMD and Intel) ship their CPUs with this feature supported (see VT-d [1] and AMD-V [2]).

[2] Chosen from the `dr0 - dr3` range group. On 64-bit systems, only 2 are needed to store 2 64-bit words. On 32-bit systems we need 4 for the same amount of data.

[3] Debug registers are used by software debuggers (e.g. GDB) to store breakpoint addresses. However, even without them, debuggers can still operate seamlessly using the rest of the debug register as well as software breakpoints.

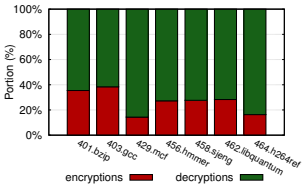


Fig. 4: Portion of cryptographic operations in each `SPEC` benchmark.

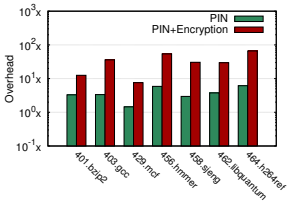


Fig. 5: Runtime overhead of dynamic instrumentation using the `SPEC` suite.

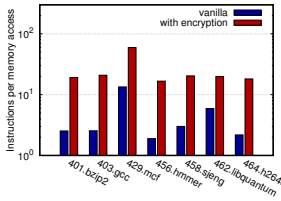


Fig. 6: Number of instructions per memory access with and without memory encryption (vanilla).

IOMMU, we forbid such illegal memory injections and as a result, we eliminate the possibility of such attacks.

Bootstrapping. The master key that encrypts the process keys inside PCB is randomly generated at boot time, ensuring this way, forward secrecy. To achieve that, we modified the kernel to use `RDRAND` instruction to perform on-chip hardware random number generation and create the next master key. Apparently, the new master key must be present to every core of the processor. The core responsible for the master key generation (core with ID `0`), is responsible to distribute it across the rest of the cores. Therefore, the newly generated key will be stored not only in the local debug registers, but also in the Memory Type Range Registers (`MTRR`), which are visible to all cores. The rest of the cores will spin on a shared variable till core `0` sets the value to true denoting that the new key has been generated and placed in the `MTRR`. After that, each core can obtain the key, store it in its local debug registers, and finally increment atomically a shared counter. By monitoring this shared counter, core `0` knows how many of the cores have obtain the new key. When they all get informed, it immediately cleans the key from the `MTRR` registers and the boot process continues normally. Obviously, there are cases where data from the memory need to be swapped-out from memory and stored in the disk. Such data, would not be able to get decrypted after boot if it gets stored encrypted with the current master key. In such cases though, we assume that the users have deployed not only memory encryption but also full disk encryption (FDE). This means that the data will get encrypted with the FDE's key, before swapped out to disk.

4 Performance Evaluation

For the performance measurements we used a server that is equipped with two six-core Intel Xeon E5-2620 operating at 2.00GHz, with 15MB L2 cache each. The server contains 8GB RAM and an Intel 82567 1GbE network interface.

4.1 Full Memory Encryption

At first, we measure the overhead imposed for encrypting all data stored in main memory. This way, we determine the cost of the most intensive but secure strategy, where every single byte written to memory is encrypted and respectively

decrypted before it is read.

Dynamic Instrumentation
One way to keep all data residing in memory encrypted is to instrument dynamically every single memory accessing operation at runtime. By leveraging this technique, we instrumented the memory accessing operations and enhanced them with the appropriate AES-NI instructions. To achieve this, we used the execution environment of the Intel's dynamic binary instrumentation tool PIN [4].

We chose this tool due to its high-versatility and support for multiple architectures (x86, x64, ARM, and more). Additionally, PIN enables the developer to inspect and tamper with an application's original instructions, when at the same time, it operates entirely in user space. It just-in-time compiles (JIT) the application's original instructions along with the instrumentation the developer may have added. This results in producing new code which is placed into a code cache awaiting execution. Dynamic instrumentation with PIN guarantees that any memory access will be intercepted either if it belongs to a dynamic library or self-modifying code etc.

Regarding our memory encryption approach, we insert a callback function to PIN[4], thus intercepting each of the original instructions and we instrument the ones that either load or store data from or to the main memory respectively. We then extract the data from the utilized register and we apply encryption or decryption depending on the instrumented instruction. Both encryption and decryption are performed using the AES-NI instructions. The output of these cryptographic operations replaces the original register's value and the program continue its execution to the next instruction.

To evaluate the performance of our approach along with the overhead imposed by the binary instrumentation, we measure the performance of (*i*) a vanilla application (listed as Native), (*ii*) a dynamic instrumentation of the application's store and load instructions using PIN (listed as PIN), and finally (*iii*) our approach: encryption with AES using dynamic instrumentation (listed as PIN+Encryption). To measure the plain instrumentation overhead produced by PIN (case (*ii*)), we perform memory instruction instrumentation with empty function calls, instead of any cryptographic operation.

Benchmarks: In the first experiment, we measure the performance of FME using several representative benchmarks, extracted from the SPEC CPU2006 suite (CINT2006). These benchmarks are comprised of several computational and memory intensive applications aiming to stress both CPU and main memory usage. In Figure 4, we see the portion of cryptographic operations in each benchmark and in Figure 5, the slowdown of a simple dynamic instrumentation of the application's load and store instructions (PIN). This number gives us a baseline for the overhead introduced by the PIN tool. In the same figure, we present the results of the instrumentation with the appropriate AES-NI instructions to encrypt or decrypt every chunk of memory that is stored in or

[4] The confidentiality of Pintool's code falls beyond the scope of this study, in this paper we only care about the sensitive data of an application.

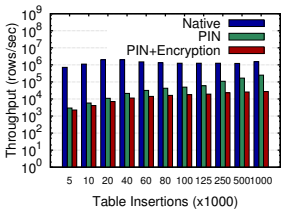


Fig. 7: Achieved throughput when inserting 1M rows into the database.

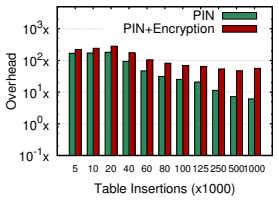


Fig. 8: Slowdown when inserting 1M rows into the database.

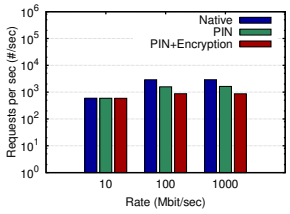
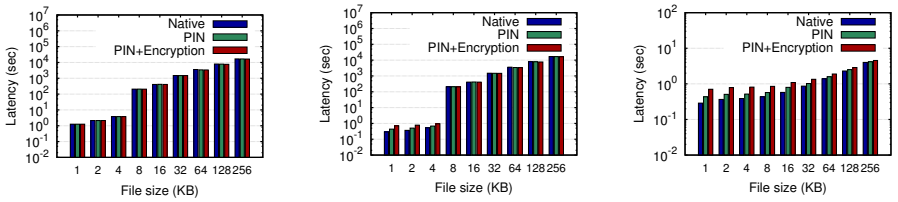


Fig. 9: Req/sec when downloading a file of 1522 bytes using different transfer rates.

loaded from the main memory (PIN+Encryption). As we can see, the run-time overhead of simply instrumenting the application's load and store instructions reaches up to 6 times slowdown for `h264ref` benchmark, while the additional overhead when adding encryption reaches up to 10 times slowdown. The major slowdown in performance arises from the fact that the data are encrypted and decrypted even when residing in the cache. As the caches in `x86` architecture are not addressable, data can reside there in clear-text, without the concern of being leaked. Unfortunately, as it is not possible to check if specific data are cached or not, we cannot benefit from memory locality. In Figure 6, we measure the instructions per memory access with and without our memory encryption approach. As we see, the average encryption cost is an additional 14-18 instructions. This number is not constant; it depends on the benchmark's synthesis of memory accesses and how sequentially the data are being accessed. Due to our pre-fetching mechanism (described in Figures 2 and 3): (i) in case of store, encryption takes place only every 2 words (load from register previous word and encrypt the pair - 28 instr.), when (ii) in case of load, the word can be fetched directly from register (8 instr.) or retrieved after decrypting a block (then the unneeded second word has to be stored in the register - 26 instr.).

Real-world applications: Additionally, we evaluate our approach in a real scenario using two real-world applications. The first, is the SQLite3 relational database management system. We used the C/C++ SQLite interface to implement a simple benchmark that reads a large, 60 MBytes, tab-separated file including 1,000,000 rows of data and updates a table's entries with the respective values. Figure 7 shows the achieved throughput, while Figure 8 shows the slowdown when inserting data into the database as a function of the number of insertions. As expected, the more rows the benchmark updates, the higher the imposed overhead becomes, since the number of memory encryptions increases. In contrast to that, the cost of the instruction instrumentation (PIN) is always proportional to the number of the table insert instructions, resulting to almost linear overhead to the application.

As a second real-world application, we ran the Lighttpd web server both as a vanilla system and with the two versions of dynamic instrumentation In the first experiment, we used a separate machine located on the same local network to repeatedly download a file of 1522 bytes. We synthetically limit the rate of the client's network line to three different network transfer rates: 10, 100 and 1000



(a) Client is over a 10 Mbps network. (b) Client is over a 100 Mbps network. (c) Client is over a 1000 Mbps network.

Fig. 10: Average latency per request when downloading different files from a Lighttpd web server as a function of the requested file's size.

Mbit/sec. As can be seen in Figure 9, when the bandwidth for the client is 10 Mbit/sec, the memory encryption overhead is almost hidden by the the network latency. As a result, the user faces a negligible slowdown of 0.17% for having FME enabled when the cost for the instrumentation is an additional 0.4%. On the other hand, the corresponding overhead for encryption at the higher rate of 1000 Mbit/sec reaches up to 43.7%. Our results indicate that in real-world applications over the Internet the cost for keeping a web server's memory fully encrypted is practically tolerated.

We conduct follow up experiments modifying the usage scenario in the following way. We use the same machine and the same three different network transfer rates to repeatedly download 9 files of different sizes, ranging from 1 KB to 256 MB. We then measure the average requests per second performed for each file. To make this experiment as realistic as possible, we use the most representative workloads found in production web servers. Such workloads include queries for short snippets of HTML (about 1 KB), e.g. user updates in micro-blogging services like Twitter or Tumblr, or portions of articles found in wikis (2.8 KB on average). Other workloads include photo objects of 25 KB size on average, used in photo-sharing sites that serve thumbnails. In general, as reported in [10], the most common file size is between 2-4 KB and regard HTML files, while 95% of all files are less than 64 KB in size. In Figures 10(a), 10(b) and 10(c) we present our results for the same experiment in the network transfer rates used above: 10 Mbps, 100 Mbps and 1000 Mbps. We immediately notice that in the case of 10 Mbps, the slowdown introduced from the memory encryption is close to zero, regardless the size of the downloaded file. In case of higher rates (i.e. 100 and 1000 Mbps) we observe that bigger files produce higher latency and as a consequence, hide the memory encryption cost. The average performance overhead imposed by encryption as calculated from the results in Figure 10 is 17%.

Static Instrumentation

The alternative of dynamic instrumentation is to statically parse the executable and instrument the load and store instructions. Although this approach requires the instrumentation of all linked shared libraries as well, however it is able to provide significantly better performance. In the following experiment, we measure this performance, and more specifically, compare the execution time of the

	Type	Execution Time (sec)	Overhead	Energy Efficiency (Joules/mbit)	Overhead
Dynamic	PIN	2.064999	-	0.03983	-
	PIN+Encryption	19.73596	9.56x	0.52276	13.12339x
Static	Native	0.406917	-	0.00849	-
	Native+Encryption	1.745001	4.29x	0.03072	3.61776x

Table 1: Encryption cost in the two implementations, in terms of execution time and power consumption.

two different approaches. We use a very simple application which copies an array of 512 MB size, along with two secure versions of it: The first version, encrypts the array contents before storing them on main memory by dynamically instrumenting the store and load instructions using PIN. The second one statically encrypts the array's cells by utilizing in-line AES-NI assembly instructions. In the first two columns of Table 1 we can see the execution time of each approach as well as the imposed latency overhead compared to the unsecured native application and its binary instrumented version respectively. As we can see, the application with the dynamically instrumented encrypt/decrypt operations on the load and store instructions is 9.56 times slower than the plain instrumentation. Additionally, the static memory encryption makes the application 4.29 times slower compared to the insecure version.

Next, we statically instrument the same benchmarks of SPEC suite as previously and we perform main memory encryption measuring again the run-time overhead. In figure 11, we compare the overhead imposed by static and dynamic instrumentation and also the performance improvement of the use of pre-fetching in both cases. As expected static instrumentation performs better (almost 1.7x) than dynamic. In addition, we see that our pre-fetching mechanism, by reducing the number of cryptographic operations in sequential memory accesses, significantly reduces also the performance of our approach (4.9x on average).

Energy efficiency

To accurately measure the energy efficiency of our approach we used 3 Phidgets high-precision current sensors [23] to constantly monitor the 3 ATX power-supply lines (+12.0a, +12.0b +5.0, +3.3 Volts), similar to [15]. The 12.0 Va line powers the processor, the 5.0V line powers the memory, and the 3.3V line powers the rest of the peripherals on the motherboard. For workload, we use the same array copy application from the prior experiment. We measure both versions in our power measurement above and the results are presented in the last two columns of Table 1. We can compare the energy efficiency of the four different approaches: (*i*) unprotected native application, (*ii*) the secure native application that statically encrypts array cells before storing them to memory, (*iii*) the unprotected native application over PIN instrumenting the plain load/-store instructions, and (*iv*) the native application over PIN when instrumenting each load and store instruction with the appropriate AES-NI instructions. From the last column, we observe that the additional overhead is 3.6 times higher in case of the static memory encryption compared to native. When we used pin the cost of encryption/decryption is 13.12 times higher compared to the baseline PIN case.

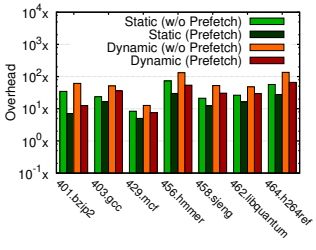
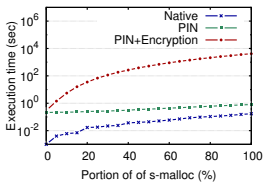
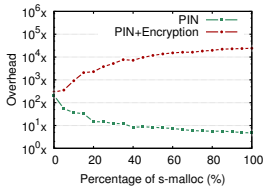


Fig. 11: Overheads of static and dynamic instrumentation with and without pre-fetching for the different benchmarks.



(a) Execution time as a function of the portion of sensitive data.



(b) Overhead over native as a function of the portion of sensitive data.

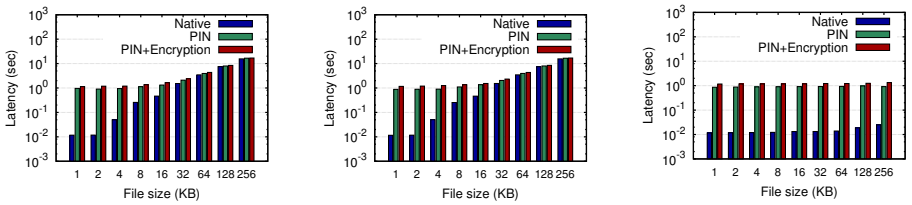
Fig. 12: Storing different portion of an array's data to the heap. Data considered as sensitive gets encrypted before sent to main memory.

4.2 Selective Memory Encryption

As described in Section 3, contrary to Full Memory Encryption one may prefer to follow a more Selective Memory Encryption (SME) strategy to reduce the imposed overhead. To evaluate this strategy, we implemented a `s_malloc` prototype, to explicitly mark some data as sensitive and only encrypt this data before storing them to memory. Additionally we created a custom benchmark which copies different sized chunks of data from a large array to the heap. Figure 12(a) shows the results for execution time as a function of the portion of data considered as sensitive. As expected, the native application using s_malloc without instrumentation increases with the percentage of sensitive data. On the other hand, the cost of instrumentation is also increasing but not as rapidly since it does not depend on the data being stored in memory. Hence, as can be seen in Figure 12(b) the instrumentation overhead over native is actually decreasing as the percentage of data increases. Furthermore, the overhead caused by the memory encryption follows a logarithmic growth with the increasing percentage of data being encrypted. Thus, in case of a chunk of data including 10% of sensitive information, the cost to guarantee its confidentiality is latency 24.90 times higher than the unencrypted case.

In our macro-benchmarks, we used the Lighthttpd web server as a real world application example and the popular Apache HTTP server benchmarking tool of ApacheBench (ab). Web services are a good case of a single physical machine serving multiple users who need to be assured that sessions will be secure during their online transactions. As a result, we can state that the keys used from the web service during the HTTPS protocol are highly sensitive, and in need of protection against unauthorized access.

In our following experiment, we use Lighttpd web server in conjunction with WolfSSL Embedded SSL Library. Inside the latter we integrated s_malloc right at the point that the private key gets stored in memory. This way, while using dynamic instrumentation we are able to selectively encrypt only the particular sensitive information of the key. Figure 13 presents the average latency of the SSL handshake while using SME and considering the server's private key as sensitive. As we can see, this latency has been measured when the client over different



(a) Client is over a 10 Mbps network. (b) Client is over a 100 Mbps network. (c) Client is over a 1000 Mbps network.

Fig. 13: Average latency for performing an SSL handshake during a client's connection to a web server where the latter's private key is considered as sensitive.

network rates (i.e. 10 Mbps, 100 Mbps and 1000 Mbps) downloads different file sizes over the secure channel. It is apparent that since the SSL handshake happens when initializing the connection it is independent from the file size. Still, SSL uses sessions in order to restrict the number of SSL handshakes. As such, for each SSL handshake tens of KBs are typically exchanged over the same session, converging the network latency in both secure and non-secure cases when using a commodity network transfer rate. Consequently, the additional latency caused by SME is concealed by the network latency. The average performance overhead imposed by encryption as calculated from the results in Figure 13 is below 27%.

5 Limitations

A major limitation of memory encryption approaches arises in cases where shared memory is deployed across different processes. To communicate correctly, processes have to maintain the same secret key, or use a different secret key which will use separately for encrypting and decrypting the contents of the shared memory. To deal with such cases, the OS kernel should be responsible for creating different secret keys for each memory segment that is instantiated, and attached it to each participant process. Similar inconveniences also arise for devices that allow data transfers via DMA. The exchanged data have to be unencrypted, since the connected devices are not aware of the encryption scheme. As it is easy to overcome these scenarios in hardware-based implementations (e.g. by performing the corresponding cipher operations at the I/O bus), it is not straightforward to provide a solution in software-only approaches. In some cases, where the device already provides a programmable interface (e.g. Endage DAG network cards, general-purpose graphics cards, etc.), it would be possible to implement the encryption and decryption operations on the device and pre-share the secret key with them.

6 Related Work

There are various approaches proposed, implemented either in software or hardware, aiming to defend against cold-boot attacks in both academia and industry.
Software-based mechanisms. Halderman et al. described cold boot attacks [12], and also discussed some forms of mitigation. Mitigations included deleting sensi-

tive data and keys from memory when an encrypted drive is unmounted, obfuscation techniques, and hardware modifications such as intrusion-detection sensors or encased RAM. However, the authors, eventually admit that these solutions do not constitute complete countermeasures, applicable to general-purpose hardware. In [30], the authors assume a powerful attacker with physical access to the machine and able to launch DMA attacks, bus snooping attacks and cold boot attacks in order to disclosure sensitive data residing in the main memory. Their approach focuses on encrypting sensitive data and code residing in the main memory and decrypting and locking them when moved in the cache. Contrary to our approach, their work is tightly woven with the ARM System-on-Chip (SoC) specific features, cache locking and TrustZone. Towards the same direction, Sentry [9], uses ARM-specific mechanisms in smarphones and tablets to keep sensitive application code and data on the SoC rather than on DRAM. They observe that sensitive state data only need to be encrypted when the device is screen-locked. Consequently, Sentry decrypts and encrypts the memory pages of sensitive applications as they are paged in and out, thus avoiding leakage of sensitive information to DRAM when the device is screen-locked. AESSE [19] was designed to provide Full Disk Encryption (FDE) and protect the required keys by storing the encryption key in the Streaming SIMD Extension (SSE) registers of the CPU, while access to these registers is disabled for user-level code. The authors however, admit that many common applications (like multimedia applications e.g. OpenGL) really need SSE registers and therefore there is a significant collision with AESSE. TRESOR [20] is a kernel module and successor of AESSE. Instead of the SSE registers it utilizes the debug registers to store the encryption key. In addition, similar to our approach, it leverages AES-NI instruction set to eliminate cold boot attacks achieving this way far better performance than the AESSE. PrivateCore's commercial product, namely vCage [25], relies on a trusted hypervisor to implement FME for commodity hardware by executing guest VMs entirely in-cache and encrypting their data before they get evicted to main memory. Although it is more cloud-oriented, vCage shares with our approach similar resistance to the same type of physical attacks.

Hardware-based mechanisms. Trustwave's BitArmor [17], is a commercial solution that claims to be resistant against cold boot attacks. BitArmor tries to shield the system as soon as abnormal environment conditions are detected. More specifically, it uses temperature sensors and in case a sudden temperature drop is detected it initializes a memory wiping process. As demonstrated in a recent study [11], this approach raises the bar, but it cannot prevent the attack. Finally, Intel provides processors with Software Guard Extensions (SGX) [14]. These extensions aim to enable applications to encrypt specific data by placing them inside secure memory regions, called enclaves. The data that reside in enclaves are protected even in the presence of privileged malware. However, SGX does not allow dynamic creation of enclave pages at runtime, it can currently be used only to encrypt static data, typically secret or private keys [3]. As such, SGX is a complementary technology to our approach, that can be used to provide us with a protected area of storing the secret keys that are used to encrypt the full

application's data, that are stored in either statically or dynamically allocated memory areas.

7 Conclusions

In this paper we design the first to our knowledge software-only main memory encryption of a running process and we set out to explore the imposed overhead when following different strategies (full Vs. selective memory encryption - dynamic instrumentation Vs. static patching). Contrary to hardware-based approaches, our work can be directly applied to commodity systems without any architectural support. Our solution leverages AES-NI instructions when our performance analysis uses both benchmarks and real world applications. Results of our work show that the average overhead of the encryption cost in real-world applications was 17% and 27% for HTTP and HTTPS respectively.

Acknowledgements

The research leading to these results has received funding from the European Unions Horizon 2020 Research and Innovation Programme, under Grant Agreement no 700378 and project H2020 ICT-32-2014 "SHARCS" under Grant Agreement No. 644571.

References

1. D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel virtualization technology for directed i/o. *Intel technology journal*, 10(3).
2. Advanced Micro Devices Inc. AMD I/O Virtualization Technology (IOMMU). `http://support.amd.com/TechDocs/48882_IOMMU.pdf`.
3. A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. *TCS*, 33(3), Aug. 2015.
4. S. Berkowits. Pin - a dynamic binary instrumentation tool. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`, 2012.
5. E.-O. Blass and W. Robertson. Tresor-hunt: Attacking cpu-bound encryption. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12.
6. A. Boileau. Hit by a bus: Physical access attacks with firewire. *Presentation, Ruxcon*.
7. D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, HPCA'10.
8. E. M. Chan, J. C. Carlyle, F. M. David, R. Farivar, and R. H. Campbell. Bootjacker: compromising computers using forced restarts. In *Proceedings of the 15th ACM conference on Computer and Communications Security*, CCS '08.
9. P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman. Protecting data on smartphones and tablets from memory attacks. In *Proceedings of the Twentieth International Conference on ASPLOS '15*.
10. J. A. Dilley. *Web server workload characterization*. Hewlett-Packard Laboratories, Technical Publications Department.
11. M. Gruhn and T. Müller. On the practicability of cold boot attacks. In *Proceedings of the 2013 International Conference on ARES '13*.

12. J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM*, 52(5), May 2009.
13. M. Henson and S. Taylor. Memory encryption: A survey of existing techniques. *ACM Comput. Surv.*, 46(4), Mar.
14. Intel Corporation. Software guard extensions programming reference. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`.
15. L. Koromilas, G. Vasiliadis, I. Manousakis, and S. Ioannidis. Efficient software packet processing on heterogeneous and asymmetric hardware architectures. In *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '14.
16. A. Markuze, A. Morrison, and D. Tsafrir. True iommu protection from dma attacks: When copy is faster than zero copy. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16.
17. P. McGregor, T. Hollebeek, A. Volynkin, and M. White. Braving the cold: New methods for preventing cold boot attacks on encryption keys. In Black Hat Security Conference 2008.
18. R. Morris and K. Thompson. Password security: A case history. *Commun. ACM*.
19. T. Müller, A. Dewald, and F. C. Freiling. Aesse: A cold-boot resistant implementation of aes. In *Proceedings of the Third European Workshop on System Security*, EUROSEC '10.
20. T. Müller, F. C. Freiling, and A. Dewald. Tresor runs encryption securely outside ram. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11. USENIX Association, 2011.
21. V. Nagarajan, R. Gupta, and A. Krishnaswamy. Compiler-assisted memory encryption for embedded processors. In *Proceedings of the 2nd International Conference on HiPEAC'07*. Springer-Verlag.
22. G. Ou. Cryogenically frozen ram bypasses all disk encryption methods. http://www.zdnet.com/article/cryogenically-frozen-ram-bypasses-all-disk-encryption-methods/.
23. Phidgets, Inc. 1122_0 − 30 Amp Current Sensor AC/DC. `http://www.phidgets.com/products.php?category=8&product_id=1122_0`.
24. D. R. Piegdon. Hacking in physically addressable memory: a proof of concept. `http://eh2008.koeln.ccc.de/fahrplan/attachments/1067_SEAT1394-svn-r432-paper.pdf`.
25. PrivateCore. Trustworthy computing for OpenStack with vCage. `http://privatecore.com/vcage/`.
26. P. Simmons. Security through amnesia: A software-based solution to the cold boot attack on disk encryption. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11.
27. S. Skorobogatov. Low temperature data remanence in static ram. 2002.
28. P. Stewin and I. Bystrov. Understanding dma malware. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12.
29. J. Wetzels. Hidden in snow, revealed in thaw: Cold boot attacks revisited. *CoRR*, abs/1408.0725.
30. N. Zhang, K. Sun, W. Lou, and Y. T. Hou. Case: Cache-assisted secure execution on arm processors. In *Security and Privacy (SP), 2016 IEEE Symposium on*, S&P '16.