Exzess: Hardware-based RAM Encryption against Physical Memory Disclosure

Alexander Würstlein, Michael Gernoth, Johannes Götzfried, and Tilo Müller

Department of Computer Science
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
{alexander.wuerstlein,michael.gernoth,
johannes.goetzfried,tilo.mueller}@cs.fau.de

Abstract. The main memory of today's computers contains lots of sensitive data, in particular from applications that have been used recently. As data within RAM is stored in cleartext, it is exposed to attackers with physical access to a system. In this paper we introduce Exzess, a hardware-based mitigation against physical memory disclosure attacks such as, for example, cold boot and DMA attacks. Our FPGA-based prototype with accompanying software components demonstrates the viability, security and performance of our novel approach for partial main memory encryption via memory proxies. The memory proxy approach will be compared to other existing mitigation techniques and possible further uses beyond encryption will be discussed, as well. Exzess effectively protects against physical attacks on main memory while being transparent to applications and the operating system after initialization.

Keywords: memory encryption, memory disclosure, physical attacks

1 Introduction

When protecting servers, desktop computers and notebooks against physical access, it is natural to draw on full disk encryption solutions. It is generally overlooked, however, that main memory contains lots of sensitive data, as well, from both the operating system and each process that was recently running. Physical access attacks against RAM range from exploitable Firewire devices that have direct memory access [1,2] to *cold boot attacks* which physically read RAM modules by first cooling them down [7,5]. The property of memory which allows cold boot attacks to work is referred to as the *remanence effect* [13,6]. RAM contents fade away gradually over time instead of being lost immediately after power is cut. Both kinds of attacks enable an attacker to completely recover arbitrary kernel and process memory. The fact that cryptographic keys in main memory are unsafe has been known for two decades now. Nevertheless, almost all vendors of software-based encryption solutions continue to store cryptographic keys inside RAM. Other sensitive data, such as cached passwords, credit card information and confidential emails are always stored in RAM in practice, too.

The problem is hardware-related in nature, but almost all current attempts for a solution are software-based. Those solutions suffer from their limited scope

– often only full disk encryption – or their poor performance. If encrypting RAM could be handled transparently in hardware, the problem of RAM extraction would be turned into the problem of extracting information from a dedicated piece of hardware that can be particularly protected.

Our Contribution

We created Exzess, a hardware-based device that is capable of transparently encrypting and decrypting portions of memory. Exzess is a PCI Express (PCIe) addon card which exploits direct memory access and acts as a transparent memory proxy to any operating system while performing certain functionality such as the encryption and decryption of data. During the development of Exzess, we designed and developed the hardware design for an FPGA-based prototype, wrote drivers and application software for Linux and performed evaluations regarding correctness and performance. In detail our contributions are threefold:

- We designed and implemented an FPGA-based prototype that acts as a transparent memory proxy to an operating system. A certain region of main memory can be accessed indirectly via PCI-Express through Exzess, while Exzess performs a given functionality on the data. We developed Exzess as a prototype to prevent physical memory disclosure attacks such as cold boot attacks by using an AES-128 IP core to encrypt all data that passes through Exzess.
- To be able to use Exzess, we provide device drivers and application software for Linux. Our host tools support the configuration of Exzess and our device driver offers an intuitive interface for using Exzess in end-user applications. Once configured, Exzess creates a number of special files and memory that should be accessed through Exzess. Protected memory can simply be read from and written to by usual system calls, i.e., access is semantically indistinguishable from the access to allocated RAM.
- We evaluated the prototype implementation regarding basic functionality, correctness and performance.

Although our prototype has primarily been developed against physical memory disclosure attacks, the generic design of Exzess as a transparent memory proxy allows for easy adoption to more application scenarios in future – such as error checking, checking the consistency of operating system structures and hardware-based malware detection, to name but a few.

2 Background

In this section, we give background information about the technologies that our work has been built on. Readers familiar with PCI Express (Sect. 2.1) and DMA transfers (Sect. 2.2) may safely skip this section.

2.1 PCI Express

Structure of a PCI Express system PCIe forms a packet-switched, hierarchical, tree-shaped network as shown in figure 1. Inner nodes are called *switches*, outer leaves are called *endpoint devices*. Examples of endpoint devices are plug-in cards, such as a network interface card, or soldered-on chips on the mainboard like an integrated graphics processor. The root node, called *root complex*, is formed by the PCIe *root switch* and *root port*.

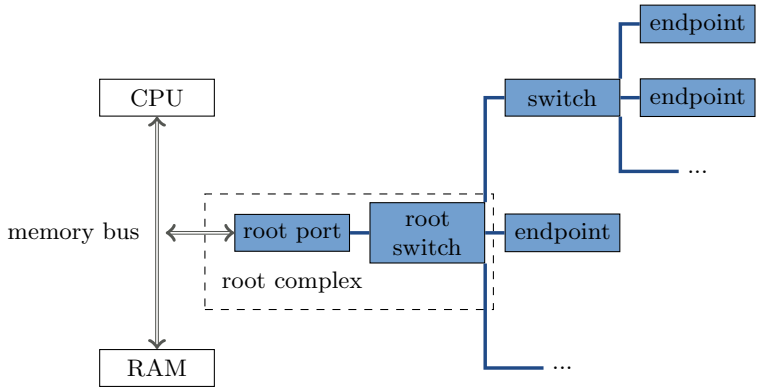

Fig. 1. Architecture of a PCI Express system

The root port has the special role of being the interface between the PCIe bus, the CPU and the main memory of the computer system. Device discovery, configuration and other *housekeeping tasks* are performed through it. Accesses to main memory, referred to as DMA transfers, from PCIe devices are processed by the root port. In the other direction, accesses from the CPU to device memory are also issued via the root port. In both cases, the root port acts as a gateway between the CPU and RAM on one side and the PCIe devices on the other side.

Transaction Layer Protocol An important part of this work consists of implementing the handling of *Transaction Layer Packets* (TLPs). The transaction layer protocol describes the routable topmost layer of the PCIe protocol. This means that a TLP will travel across a number of links from one sending endpoint to one or more receiving endpoints. As the name suggests, a sequence of one or more exchanged TLPs is used to implement logical transactions.

Generally all transactions are initiated by a request TLP from the party subsequently called *requester*. The receiving party, called *completer* performs the request specified and, if necessary, answers the request with a *completion* TLP. *Posted* requests like *memory write* (MWr) require no completion while *non-posted* requests like *memory read* (MRd) have to be answered by a *completion with data* (CplD).

2.2 DMA Transfers and Device Memory

There are two possible memory domains as well as directions for memory access between a PCIe endpoint, software running on the CPU and main memory:

1. Coming from and initiated by the software on the CPU, device memory can be accessed. Device memory is a memory area assigned to a specific endpoint for which that endpoint handles accesses. It is configured using the so-called *base address registers* (BARs) where size, address and access semantics are specified. Device memory accesses are accesses where the completer for a memory transaction is an endpoint and the requester for that transaction is the root port. The term *BAR* is a commonly used metonymy for *device memory*.
2. Coming from and initiated by an endpoint, a computer's main memory can be accessed as well. These accesses happen without interrupting normal program flow of the software running on the CPU. Such accesses, commonly known as *direct memory access* (DMA), are memory transactions where the requester is an endpoint and the completer is the root port. DMA memory areas are made known to the device via device-specific means.

3 Design and Implementation

In this section we will first explain the threat model (Sect. 3.1) with respect to which Exzess was desigend. Afterwards we justify our design choices (Sect. 3.2) before giving an overview of the Exzess architecture (Sect. 3.3). Finally we describe implementation details regarding the hardware design and the device driver (Sect. 3.4).

3.1 Threat Model

For the design of Exzess, we consider our protectable asset to be portions of RAM that are limited in size and that are flagged as such by the application or operating system component to which these memory areas have been allocated. The content of these memory areas is data that is considered more critical than other, unprotected data. Examples of such data are cryptographic keys, user passwords, credit card information, confidential emails and secret documents. By protecting full-disk-encryption keys, the encrypted contents of a hard disk can be considered an indirectly protected asset. The same principle applies to, for example, encrypted data within TLS connections and their respective keys stored within protected RAM on the communication endpoints.

We consider our attacker to be capable of physically accessing a machine running Exzess. In particular, the attacker is able to extract RAM modules from the system and read their contents, even after the system has been recently switched off. The attacker, however, is unable to perform more sophisticated attacks such as chip probing or fault injection as these attacks are by far more

costly and technically difficult. Thus, he is not able to read CPU registers, CPU cache contents, and registers from the Exzess extension board.

We restrict ourselves to a physical attacker, i.e., software based attacks on applications or the operating system are out of scope for this work. Furthermore, we exclude the possibility of physically writing RAM from our attack model as writing RAM by extracting memory modules usually results in a system crash and is therefore not feasible in most scenarios. Observing bus transfers via direct electrical taps or electromagnetic emanations are excluded from our attack model as well.

3.2 Design Rationale

To mitigate physical memory disclosure we decided to encrypt a limited but sufficiently large portion of a computer's RAM. Any read access to the encrypted portion of the RAM is decrypted transparently when loading the respective data into the CPU's caches or registers. No unencrypted copy of this data can be found in physical main memory. Similarly, when writing CPU register or cache contents back to an encrypted portion of the RAM, encryption will happen transparently and without storing plaintext data in RAM. Userspace software and operating system components wishing to encrypt parts of their respective memory address space may allocate an appropriate address range through a special allocation function.

We decided to encrypt a limited portion of overall RAM. Although a cold boot attack exposes the complete contents of a computer's RAM to the attacker, not all data stored in RAM is equally sensitive as explained in section 3.1. By only encrypting important portions of RAM, performance for the unencrypted parts is unaffected, thereby alleviating the severe performance problems of full memory encryption.

One consequence of encrypting selected portions of RAM is that userspace applications have to be modified to make use of our encrypted memory allocations. The modifications, however, only require limited effort: Cryptographic operations employ libraries that are therefore an obvious place to introduce our changes. Furthermore, many applications already identify the relevant parts of their RAM allocations by marking them non-swappable. To make the modifications of userspace applications as simple as possible we provide an easy to use library that offers the Exzess functionality to programmers.

3.3 Exzess Architecture

In figure 2 a schematic overview of a memory access with Exzess is shown. Exzess provides a readable and writeable memory window for the operating system. No additional memory beyond the available main memory is needed for Exzess to work. Instead, any read or write access to the memory window is redirected to a previously configured memory region in RAM via direct memory access. Exzess acts as proxy for those requests and encrypts or decrypts the data written

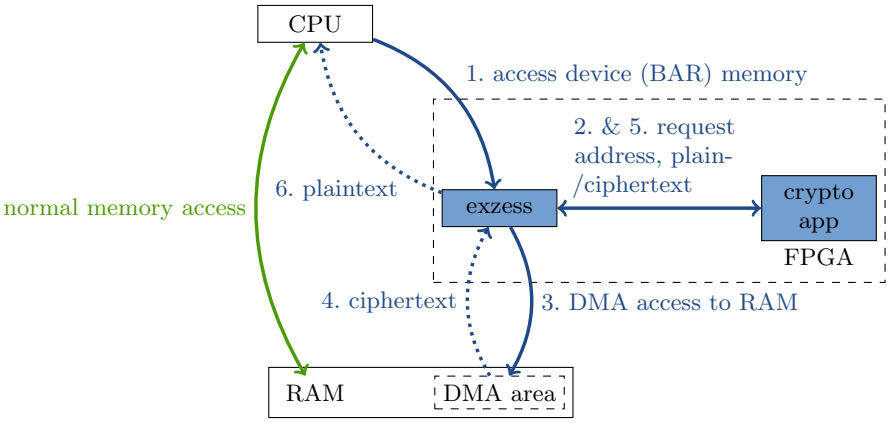

Fig. 2. Schematic overview of Exzess.

to or read from the DMA area. Thus an encrypted memory area has the same semantics as usual memory accesses except for the transparent encryption.

Encryption through a Memory Proxy Exzess presents a PCI device memory area to the operating system which is a proxied cleartext view of a given memory window. The memory window can be configured in size and location with the help of our device driver. On each request encryption or decryption is performed on-the-fly. The encrypted data is stored in a DMA memory area which is a portion of regular main memory according to the previously configured memory window.

3.4 Exzess Implementation

In this section we will present the implementation of Exzess. We first will explain details of our hardware design and afterwards describe how the device driver is able to interact with our memory proxy.

FPGA Hardware with PCI Express Interface Our prototype is implemented on an Enterpoint Raggedstone 2 board [4]. The Xilinx Spartan6 FPGA on this board includes a PCIe interface that is able to handle the physical layer and configuration portions of the PCIe protocol. The VHDL code written for our prototype handles the creation and sequencing of PCIe TLPs as well as encryption. To be able to encrypt data passing through Exzess, an open source AES core [12] has been utilized.

As shown in figure 3, the hardware is structured in a four-layered architecture: Physical PCIe interface and configuration tasks are performed by the hard IP core included on the FPGA. Above this lowermost layer, three layers were

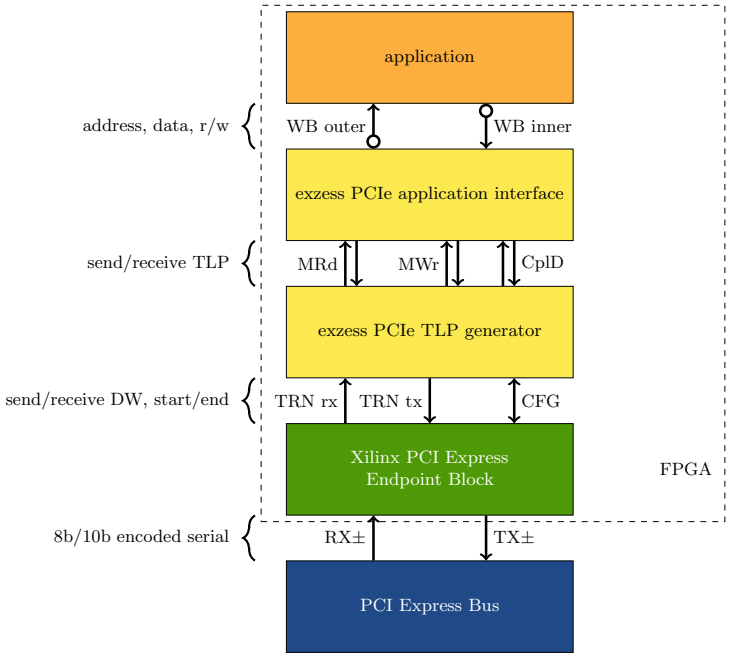

Fig. 3. Block diagram of the software stack of Exzess.

implemented in VHDL: (1) a basic, reusable generator and decoder for PCIe TLPs, (2) an abstraction layer providing memory access primitives over the on-chip Whishbone bus as an abstraction to sending and receiving TLPs, and (3) the actual application responsible for encrypting and decrypting memory accesses.

PCI Express Communication The memory proxy is implemented by a distinctive sequence of TLPs. After a request, e.g., a read request (MRd) addressing the BAR device memory, has arrived at the FPGA, a corresponding MRd for the encrypted data in the DMA area will be generated by the FPGA. After this second request has been answered by a CplD containing the encrypted data, decryption will take place within the FPGA. The decrypted data is then used to answer the initial MRd with a completion containing the decrypted data. As long as no timeout violations for the initial request occur, i.e., the final completion arrives in time, PCIe allows multiple larger or smaller requests between the initial and final TLP to be sent.

Exzess Encryption Applications To be able to evaluate Exzess, especially regarding performance, two different applications have been implemented: The first one performs a simple "XOR encryption" with a configurable constant on all

data passing through Exzess while the second actual application performs AES encryption on the data and thus effectively prevents physical memory disclosure.

For this prototype AES has been used in CTR mode of operation with the concatenation of a nonce and the currently requested address as start value. The key and the nonce can be set at configuration time, i.e., boot time, and are not readable by software afterwards. In real world scenarios, both the nonce and the key should be randomly generated on each boot.

Device Driver The Linux device driver for Exzess is responsible for initializing the FPGA as well as for allocating and configuring the DMA and device memory areas, e.g., the current memory window. After configuration, Linux automatically exposes all device memory areas as special root-accessible files in the `/sys` filesystem (sysfs). The allocated DMA area is exposed by the driver via a character device `/dev/exzess` for debugging and verification purposes. Since it only enables access to the encrypted data, this device file would be unnecessary in a production environment.

Both the device memory areas and the DMA area can be accessed through the usual POSIX `read`/`write` and `mmap` system calls using the aforementioned files. For convenience, a library wrapping those calls with appropriate arguments into `secure_malloc` and `secure_free` routines is provided.

4 Evaluation

In this section we evaluate Exzess regarding basic functionality and correctness (Sect. 4.1) followed by performance (Sect. 4.2).

4.1 Correctness

Both applications, "XOR encryption" and AES encryption, were used to verify that the application interface, the device driver and the hardware are working together as expected: By inspecting both the contents of the encrypted DMA area and the plaintext device memory window using XOR encryption a first hint is given that the data is processed correctly by our memory proxy. To verify the correctness of our actual application, the AES encryption, OpenSSL was used. On the one hand, data has been encrypted by Exzess and decrypted by OpenSSL and on the other hand, it has been encrypted by OpenSSL and decrypted by Exzess. For both scenarios, a successful comparison of a large number of plain- and ciphertext pairs leads to the conclusion that the Exzess works as intended.

Furthermore, the device memory area was mapped into a Linux userspace application and known values were written to this area. By monitoring received and generated PCIe TLPs on the FPGA and viewing the contents of the DMA area, the correct functionality of the FPGA could be verified.

To examine the leakage of sensitive data, known patterns were written to the protected device memory area. Afterwards full physical memory dumps have been obtained via the `/dev/mem` special device. We searched for the previously written

known patterns within these images and could not find a single pattern within one of the dumps. Note that obtaining a physical memory image via `/dev/mem` is more strict then performing an actual cold boot attack, because the images do not contain bit errors and therefore no patterns can be missed.

4.2 Performance

To measure the performance of the AES and XOR application, a benchmarking application has been implemented which reads and writes device memory areas 100 times and calculates average measurements and error estimates. These measurements are shown in figure 4 and table 1.

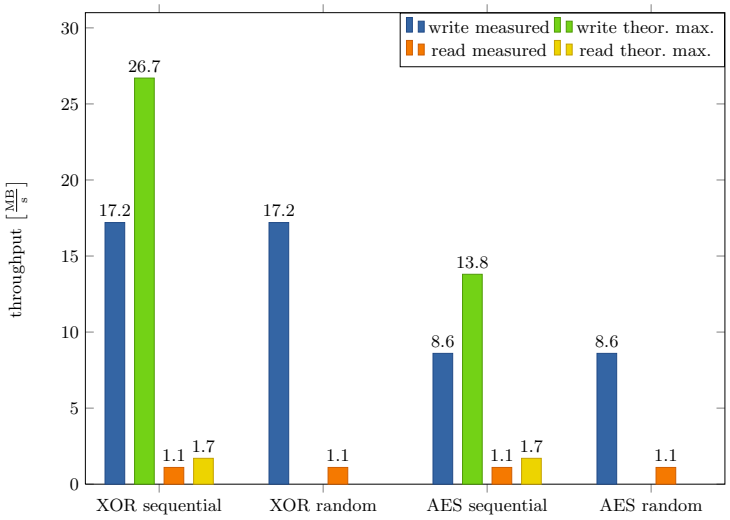

Fig. 4. Benchmark results for both applications, XOR and AES in read and write mode. The measured values are averaged over 100 iterations.

As expected, the measured values are lower than the calculated theoretical maximum values which are derived from transaction runtimes. The performance for read transactions, when comparing the XOR and AES applications, is equal within the precision of measurement. Performance values for write transactions are lower in the case of AES and the XOR application has roughly twice the write throughput of the AES application. This is also consistent with the predicted behaviour.

Random accesses have essentially the same performance as sequential accesses, at least for writes. In the case of reads, the performance is approximately 1 % lower for random reads than for sequential reads, both in the XOR and AES application. The difference is small but much larger than 5σ and therefore significant. While

application	access pattern	access direction	theoretical maximum $\left[\frac{\text{kB}}{\text{s}}\right]$	measured $\left[\frac{\text{kB}}{\text{s}}\right]$
XOR	sequential	write	26667	17223 ± 7
XOR	sequential	read	1739	1085 ± 1
XOR	random	write		17221 ± 6
XOR	random	read		1078 ± 1
AES	sequential	write	13793	8621 ± 3
AES	sequential	read	1739	1086 ± 1
AES	random	write		8620 ± 3
AES	random	read		1077 ± 1

Table 1. Benchmark results for both applications in read and write mode.

we have no definitive explanation, readahead behaviour is probably the most likely cause for this difference. Of course there are possibilities to improve performance and some of them will be discussed in section 6.2.

5 Related Work

Theoretical approaches for *full memory encryption* [3] are described but practical implementations to encrypt, for example, swap space [11,10] are also available. Furthermore, a solution for use in embedded hardware [8] exists. All these solutions, however, are implemented in software and suffer from serious performance drawbacks while Exzess is a fast hardware solution that protects sensitive data on the users choice. For embedded hardware there exists work on hardware-based full memory encryption [9] as well, yet for the most critical and vulnerable species of personal computers no solution seems to be available.

6 Discussion

In this section, we will list current limitations (Sect. 6.1) of Exzess and give an outlook over future research directions (Sect. 6.2).

6.1 Limitations

The BAR memory area is generally limited in size by the hardware since all BAR memory areas of all PCI and PCIe devices in a system need to fit into the PCI address space which is a hardware-dependent fixed subset of the 32 bit address space, usually a few hundred megabytes in size. A switch to 64 bit addresses on the PCIe interface could diminish this limitation. This would also allow for DMA areas larger than what is addressable by 32 bit addresses.

The current interface to access BAR memory areas from a userspace program is not very convenient as root access privileges are necessary and the whole memory area can be accessed. To handle accesses for a *normal* user and to ensure the usual page-granular memory mapping and protection, an operating system

interface to map these pages from the BAR memory area into process address spaces is necessary.

6.2 Future Work

An increase in performance to levels similar to normal RAM accesses could be achieved with some enhancements. Currently, the configuration of the BAR and DMA area forbids caching. This means that reads of previously accessed data are as slow as if accessed for the first time. With caching enabled, each access after the first one would access the cache, speeding up operations significantly. Additionally, read accesses are usually performed in cache-line-sized portions and following data would then be subject to read-ahead, possibly gaining even more speed.

PCIe TLPs not only contain the memory address subject to a read or write request but also the bus address of the requester to return data or error messages to. By only allowing requests from known-good bus addresses, i.e. the root complex where requests from the CPU originate, and rejecting all requests from other, possibly malicious PCIe devices, attacks from those devices could be mitigated as well.

The modular design of Exzess as a transparent memory proxy allows the development of a lot of different applications far beyond encrypting data to prevent physical memory disclosure and guarantee confidentiality. Possible applications include data integrity and authenticity by using, for example, an AEAD cipher mode. This is possible because both memory areas and the respective requests do not have to be equal in size. Furthermore, completely different application scenarios such as error checking, checking the consistency of operating system structures or hardware-based malware detection could be adopted.

In combination with those integrity checks, the Exzess threat model could be extended to include certain software-based attacks. When moving beyond the current FPGA prototype, e.g., towards an ASIC, certain limitations in the attacker model pertaining to the relative vulnerability of the FPGA could be replaced by those of a sufficiently hardened ASIC.

7 Conclusion

In this paper we presented Exzess, a hardware-based solution against physical memory disclosure attacks such as cold boot attacks. The design and implementation of Exzess as a transparent memory proxy with on-the-fly AES encryption for all data passing through allows applications and operating systems to transparently access sensitive data within main memory without leaving the data at risk. We have shown that Exzess successfully mitigates physical attacks on main memory while maintaining the overall performance of a computer system where other approaches either fall short or are very limited regarding the data they are able to protect.

Acknowledgment This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

References

1. Becher, M., Dornseif, M., Klein, C.N.: FireWire - All Your Memory Are Belong To Us. In: Proceedings of the Annual CanSecWest Applied Security Conference. Laboratory for Dependable Distributed Systems, RWTH Aachen University, Vancouver, British Columbia, Canada (2005)
2. Carrier, B.D., Grand, J.: A Hardware-Based Memory Acquisition Procedure for Digital Investigations. Digital Investigation 1(1), 50–60 (Feb 2004)
3. Duc, G., Keryell, R.: Cryptopage: An efficient secure architecture with memory encryption, integrity and information leakage protection. In: 22nd Annual Computer Security Applications Conference ACSAC 2006), 11-15 December 2006, Miami Beach, Florida, USA. pp. 483–492 (2006)
4. Enterpoint Ltd.: Raggedstone 2 - Xilinx Spartan 6 FPGA Development Board, Manufacturer Website. http://www.enterpoint.co.uk/products/spartan-6-development-boards/raggedstone-2/
5. Gruhn, M., Müller, T.: On the practicability of cold boot attacks. In: 2013 International Conference on Availability, Reliability and Security, ARES 2013, Regensburg, Germany, September 2-6, 2013. pp. 390–397 (2013)
6. Gutmann, P.: Data remanence in semiconductor devices. In: 10th USENIX Security Symposium, August 13-17, 2001, Washington, D.C., USA (2001)
7. Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest We Remember: Cold Boot Attacks on Encryptions Keys. In: Proceedings of the 17th USENIX Security Symposium. Princeton University, USENIX Association, San Jose, CA (Aug 2008)
8. Henson, M., Taylor, S.: Beyond full disk encryption: Protection on security-enhanced commodity processors. In: Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings. pp. 307–321 (2013)
9. Kurdziel, M., Lukowiak, M., Sanfilippo, M.: Minimizing performance overhead in memory encryption. Journal of Cryptographic Engineering 3(2), 129–138 (2013), `http://dx.doi.org/10.1007/s13389-013-0047-5`
10. Peterson, P.: Cryptkeeper: Improving security with encrypted RAM. In: Technologies for Homeland Security (HST), 2010 IEEE. pp. 120–126 (Nov 2010)
11. Provos, N.: Encrypting virtual memory. In: 9th USENIX Security Symposium, Denver, Colorado, USA, August 14-17, 2000 (2000)
12. Satyanarayana, H.: AES128 Crypto Core in VHDL, licensed under LGPL. http://opencores.org/project,aes_crypto_core (2004)
13. Skorobogatov, S.P.: Data remanence in flash memory devices. In: Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings. pp. 339–353 (2005)