Protecting sensitive information in the volatile memory from disclosure attacks

Stefanos Malliaros, Christoforos Ntantogian, Christos Xenakis
Department of Digital Systems
University of Piraeus
Piraeus, Greece
{stefmal, dadoyan, xenakis}@unipi.gr

Abstract— **The protection of the volatile memory data is an issue of crucial importance, since authentication credentials and cryptographic keys remain in the volatile memory. For this reason, the volatile memory has become a prime target for memory scrapers, which specifically target the volatile memory, in order to steal sensitive information, such as credit card numbers. This paper investigates security measures, to protect sensitive information in the volatile memory from disclosure attacks. Experimental analysis is performed to investigate whether the operating systems (Windows or Linux) perform data zeroization in the volatile memory. Results show that Windows kernel zeroize data after a process termination, while the Linux kernel does not. Next, we examine functions and software techniques in C/C++ programming language that can be used by developers to modify at process runtime the contents of the allocated blocks in the volatile memory. We have identified that only the Windows operating system provide a specific function named SecureZeroMemory that can reliably zeroize data. Finally, driven by the fact that malware scrapers primarily target web browsers, we examine whether it is feasible to extract authentication credentials from the volatile memory allocated by web browsers. The presented results show that in most cases we can successfully recover user authentication credentials from all the web browsers except when the user has closed the tab that used to access the website.**

Keywords: Memory zeroization, Volatile memory, Operating Systems, memory management, Information disclosure.

I. INTRODUCTION

Recent research has shown that authentication credentials and cryptographic keys remain in the volatile memory and can be easily extracted [2]. For this reason, the volatile memory has become a prime target for malicious software. As a matter of fact, a new malware category has emerged named as memory scrapers, which specifically target the volatile memory, in order to steal sensitive information, such as credit card numbers [5]. To achieve this, memory scrapers use regular expression matches, in order to harvest credit card data from the volatile memory, and then the collected data are sent to a malicious server. The first known memory scraper, named StarDust targeted point of sale terminals and compromised nearly 20.000 credit cards in the US [7].

Due to the rise of memory scrapers [1], it is evident that the protection of volatile memory from disclosure attacks is of paramount importance. The majority of the related work in this area focuses on hardware-based solutions that encrypt data in volatile memory. Although these solutions can protect data in volatile memory, they raise deployment and performance issues, since they require modifications at the hardware level. As a result, these solutions are not considered to be generic. Apart from memory encryption solutions, there are also special devices, such as Hardware Security Modules [28] and FPGAs [30] that employ data zeroization to protect the volatile memory from disclosure attacks. The goal of data zeroization is to overwrite sensitive information in the volatile memory with zeroes [4]. On the other hand, it is not widely known whether the generic Operating Systems (OS) such as Windows or Linux provide protection in the volatile memory. Moreover, there are no specific guidelines that the developers should follow to protect sensitive data in the volatile memory of their applications.

In this paper we investigate techniques that can be applied at the software level either from the OS or the applications, in order to zeroize data in the volatile memory. More specifically, first, we investigate whether OS (Windows or Linux) provide built-in safeguards to protect against disclosure attacks from the volatile memory. Experimental results show that Windows kernel zeroize data after a process termination, while the Linux kernel does not. Next, we elaborate on software functions and techniques in C/C++ programming language that can be used by developers to protect the data in the volatile memory of their applications. We have identified that only the Windows OS provide a specific function named *SecureZeroMemory* that can reliably zeroize data. Finally, driven by the fact that malware scrapers primarily target web browsers, we examine whether it is feasible to extract authentication credentials from the volatile memory allocated by web browsers. The presented results show that in most cases we can successfully recover user authentication credentials from all the web browsers except when the user has closed the tab that used to access the website.

The rest of the paper is structured as follows. Section 2 presents the related work, while section 3 presents the protection of the volatile memory data at the software level. Section 4 evaluates the security of web browsers. Finally, the conclusions of this work are presented in section 5.

II. RELATED WORK

In this section, the related work is presented and elaborated which is composed of two parts. In the first part, previous works which show that sensitive information can be discovered in volatile memory are presented. In the second part, the previous works that protect volatile memory using encryption are elaborated.

Regarding previous works that focus on the retrieval of sensitive information in the volatile memory, Darren et al.

tried to recover data remnants from cloud storage applications including Dropbox [8], Skydrive [9], and Google Drive [10]. Similarly, in [11] the authors investigate the volatile memory of cloud services applications, such as Amazon S3, Dropbox, Google Docs and Evernote. In all the aforementioned publications, several artifacts were recovered such as authentication credentials, visited URLs, filenames and hashes. Apart from personal computers, sensitive information was also recovered from the volatile memory of Android devices using two different methods. More specifically, in the first method [12] the authors used the Linux Memory Extractor (LiME) kernel module [13] and a physical Samsung i9000 phone to dump the Android memory, whereas in the second technique [14] the Android emulator was used alongside with Dalvik Debug Monitor Server (DDMS) to acquire the memory data. In both cases, critical and secure applications, such as mobile banking and password managers, were examined and authentication credentials were recovered in plain text from the dumped memory.

Regarding memory encryption, the proposed solutions can be further classified into two categories: software-based and hardware-based. For software-based solutions, in [15], the authors propose a modified secure memory bus controlled by the OS, in which the encryption key is generated each time the system boots up. Peterson, in [16], modified the virtual memory manager of the Linux 2.6.24 kernel and partitioned the volatile memory into a plaintext and an encrypted segment. However, [17] shows that the memory maps, should be maintained in the plaintext segment; thus pointing the addresses to where the encrypted volatile data are stored. The second category of the proposed solutions for memory encryption is based on hardware modifications. In particular, several publications [18-23] [40] for single processor systems propose the addition of an encryption unit to cipher and decipher data from and to the volatile memory. Moreover, for multi-processor systems, [24] proposes a shared bus, containing a crypto engine, to coordinate and secure traffic between processors, while [25] [26] proposed the use of sequence numbers for the coordination between different processors. Lastly, in [27], the authors propose SecBus, a cryptographic coprocessor between the volatile memory and the main processor.

The main limitation of the proposed solutions that perform memory encryption has to do with the fact that hardware-based solutions require extensive changes in the current CPU architecture, while the software-based solutions require modifications at the OS kernel. In contrast to the relevant works, this paper investigates if the latest OS versions (Windows and Linux) provide built-in data zeroization methods as well as whether C/C++ developers can use existing software libraries and methods in order to perform data zeroization in their applications.

III. SOFTWARE LEVEL PROTECTION

A. Operating System level protection

Memory management is the procedure of administering the volatile memory at the system level. This is performed by the kernel of the Operating System (OS) with the support of a part of the central processing unit, named memory management unit. Allocation and deallocation requests are used in order to grant or revoke memory blocks to applications. Allocation is the procedure in which memory blocks are granted to applications, and are then used by them for handling the necessary data for their functionalities. On the other hand, deallocation is the procedure in which the applications free the memory blocks they do not longer need, making them available for other running or starting applications. It is important to note that the OS does not modify the allocated memory blocks, since this action could cause the running applications to crash. Subsequently, during the applications' runtime, only the applications themselves are accountable of modifying their allocated memory blocks.

```
1. void main() {
2.   static int length;
3.   char password[length];
4.   fgets(password, length * sizeof(char), stdin);
5.   sleep(120); } //suspend for 120 seconds
```

Figure 1. First testing application used to discover the total number of instances of the *password* variable in the volatile memory.

In order to find out whether the OS performs data zeroization, we developed a testing application written in C programming language (see figure 1), that holds a secret value in the variable named as *password*. The aim of the experiments was to investigate how many instances of the *password* variable can be extracted from the volatile memory. More specifically, as shown in figure 1, the testing application defines the *password* variable at line 3, which is an array of type *char* and size *length*. Moreover, the *stdin* (e.g. keyboard input) is used to fill in the array of the *password* variable. For the experiments, three types of memory dumps were considered which are: A) **Process:** This memory dump includes only the memory blocks that are allocated to the executable of figure 1. B) **All-Processes:** This memory dump includes memory blocks allocated to all running user-mode processes in the OS. In this way we can find out whether the password variable of figure 1 can be extracted from other user-mode running processes; C) **System:** This memory dump contains the entire volatile memory including memory allocated not only to user-mode processes but also to the OS kernel, drivers, unallocated blocks. The technical methodology that we followed in order to obtain the memory dumps is as follows. To perform a **Process** dump in Linux, the GNU debugger (i.e. GDB) was used to dump the memory blocks of a process based on its PID. Similarly, the **All-processes** dump was performed using a script that feeds GDB with all the running PIDs. The same methodology was followed in Windows. In particular, we used the Windows Powershell in order to list all the running PIDs, and feed them to ProcDump [41] (i.e., a Windows utility which performs memory dumps of running processes). It is important to note that all the aforementioned memory dumps, were executed using root privileges both in Linux and Windows. To perform **System** dump, we used virtual machines, in order to dump the entire volatile memory of the system in an easy manner.

Moreover, two scenarios were considered. In the first scenario named as "***Running process***" we performed memory dumps (all three types) while the process of the

TABLE I. *NUMBER OF INSTANCES OF THE PASSWORD VARIABLE*

Memory Dump Type	**Operating System**			
	Ubuntu Linux 14.04		***Windows 7/10***	
	Running Process	***After Termination***	***Running Process***	***After Termination***
Process	1	Not Applicable	3	Not Applicable
All-Processes	1	1	3	0
System	9	2	5	0

executable was running. This was achieved during the sleep function (see line 5 of figure 1), where the execution of the process was suspended and we were able to recover the memory dump. In the second scenario named as "***After termination***", we performed the memory dumps immediately after the termination of the executable. Evidently, in this scenario, we performed only **All-processes** and **System** memory dumps, since **Process** dump cannot be performed after the termination of the executable. The experiments were conducted in Windows 7 and 10 and Ubuntu Linux 14.04, fully updated as of 15th of April 2016. In both versions of Windows, the compiler of Microsoft Visual Studio 2015 suite was used, while in Ubuntu Linux we used the latest version of the GCC compiler (i.e., v5.3).

The results of the experiments are summarized in Table 1. We can observe that in the "***Running process***" scenario in all three memory dump types for both Linux and Windows OS we were able to recover the value of the *password* variable. It is interesting to notice that in the **All-processes** memory dump type, the number of the instances of the *password* variable were the same as in the **Process** memory dump type (i.e., 1 time in Linux and 3 times in Windows). This means that apart from the process itself of the testing application (see figure 1), the other processes running in the system did not use the *password* variable. We can also observe that in the **System** memory dump, the number of recovered *password* instances increased (i.e., 9 times in Linux and 5 times in Windows). This result means that i) apart from the process of the testing application itself, the OS kernel stores also the value of the *password* variable and ii) the OS kernels stores in multiple memory regions the value of the *password*.

Regarding the results of the "***After termination***" scenario, we can observe an interesting outcome: for both ***All-processes*** and ***System*** dumps in Linux we were able of recovering the *password* variable (1 and 2 times respectively). On the other hand, in Windows we were not able to recover it. This result means that Windows kernel zeroize the deallocated blocks of a process immediately after its termination. On the other hand, the Linux kernel follows a different approach. That is, instead of zeroizing the deallocated memory blocks of a terminating process, it zeroizes the memory blocks right before their allocation [31]. Thus, in Linux, a malicious software that has access to the entire system memory can extract potentially sensitive information (such as authentication credentials) even from applications that were terminated, in case the related deallocated blocks have not been allocated to a new process. On the contrary, in Windows, a malicious software can extract information only from the memory blocks of running applications.

The above observation implies that Windows is more secure than Linux to memory disclosure attacks. To overcome this issue, we have identified that there is a Linux kernel patch, named as GRsecurity, which provide several security enhancements for the Linux kernel [32]. One of these enhancements enables the Linux kernel to zeroize the deallocated memory blocks after process termination by compiling the Linux kernel with the *PAX_MEMORY_SANITIZE* option that the GRsecurity provides. To this end, we repeated the experiments (using the testing application of figure 1) in Ubuntu 14.04 compiled with a kernel that has GRSecurity installed and the *PAX_MEMORY_SANITIZE* option enabled. We observed that this time we were not able of recovering instances of the *password* variable after the process termination. Based on the above discussion, we *propose the use of GRsecurity (with the PAX_MEMORY_SANITIZE option enabled),* in order to minimize information disclosure in volatile memory.

Despite the fact that GRsecurity may enable the kernel to perform data zeroization, it is not widely adopted in Linux Distributions. Even those that offer a GRsecurity patched kernel by default, many of them have not enabled the *PAX_MEMORY_SANITIZE* option. In total, we found six Linux distributions [33-34] [42-45] that come with a GRsecuity patched kernel and only three of them have the *PAX_MEMORY_SANITIZE* option enabled.

B. Source code level protection

The previous results show that OS zeroize data only after the termination of the running process which means that during the runtime of a process, sensitive information can be extracted in its allocated memory blocks. In this section, we investigate functions and methods that developers can use in order to zeroize memory blocks during the runtime of their applications. We focus on C/C++ programming language, since it provides low-level memory manipulation. All experiments carried out in this section perform **Process** dump in a "**Running-process**" scenario.

First, we investigate for Windows OS, if there are special functions that can be used in order to zeroize data. More specifically, by including the *windows.h* header file in a C/C++ source code, a developer has the ability of using the macro *SecureZeroMemory*, which calls the function *RtlSecureZeroMemory* that guarantees to zeroize memory blocks, even if it is not subsequently written or accessed by the code [35]. We repeated the experiments

```
1. void main() {
2.   static int length;
3.   char password[length];
4.   fgets(password, length * sizeof(char), stdin);
5.   memset(password, '0', length);
6.   sleep(120); } //suspend for 120 seconds
```

Figure 2. Second testing application used to discover the total number of instances of the *password* variable in the volatile memory.

performed in the previous section (i.e., as mentioned previously only **Process** dump in the "**Running process**" scenario) using the same testing application with the difference that at the end of the code we called the SecureZeroMemory macro. We observed that indeed the macro SecureZeroMemoy replaced the contents of the password variable with zeroes. Thus, in Windows, developers should use the macro SecureZeroMemory to ensure that the memory blocks of their applications are zeroized.

On the other hand, for Linux OS, there is no similar C function that can be used to zeroize data in the volatile memory. To this end, we have used the function *memset* of the C programming language to manually try to zeroize memory blocks allocated to a process. In particular, we have used the testing application of figure 2, which is identical to the code of figure 1, with the difference that figure 2 includes in line 5, the command *memset(password, '0', length).* This command writes in the memory block, which is allocated for the value of the *password* variable, the 0 character as many times as indicated by the value of the *length* variable. This will result in the zeroization of the data of the array *password.* We repeated the experiments of the previous section and we observed that the *memset* function was not operating as we expected, since the value of the password variable was detected in the process dumps. After investigation, we identified that the *memset* function was not being called due to code optimization. The latter is the process in which a compiler tries to improve the generated executable code by making it consume fewer resources, such as CPU and Memory. This is performed by several techniques. One of these methods is to avoid compiling specific code which is not necessary for the execution flow. For this reason, in our experiments, the compiler skipped the calling of the *memset* function, because the new value of the *password* variable (i.e., the zeroized data) is not used after the *memset* function. Note that although the executable of figure 2 was compiled using GCC without optimization flags, the GCC compiler did perform optimization and did not include the *memset* function in the executable.

The above results raise the following question: "is it feasible to avoid optimization caused by the GCC compiler, in order to ensure that the *memset* function will be executed"? To answer this question we tried two different methods. In the first method we used the function *memset_s*. The latter has the same functionality as *memset*. The main difference between those two functions is that the *memset_s* cannot be optimized out by the compilers [38]. However, *memset_s* is included only in the currently last version of the standard of the C programming language (i.e., C11 [39]) in Annex K. Unfortunately, Annex K is not mandatory in C11 specifications and at the time of writing the paper, the latest version of the GCC compiler (i.e., v5.3) has not implemented the Annex K, and thus the developers have no way to use the *memset_s* function.

The second method that we attempted in order to avoid bypassing optimization was to wrote a testing application similar to the one described in [46] (see figure 3), which uses a function pointer of type volatile named *memset_volatile*, as defined at line 1. The declaration of a variable as volatile instructs the compiler not to optimize out functions that access the variable. This is due to the fact the volatile type is used mainly for buffers in communication with hardware devices or other applications. Based on this observation, we defined the function pointer named *memset_volatile* pointing to the function *memset* at line 1. At line 4, a pointer named *password_heap* is defined, which points to a block of memory of size *length*sizeof(char)*. This block of memory is allocated using the *malloc* function, which is used for dynamic memory allocation during the application execution. In line 5, the user enters his password, and in line 6, the memory block allocated at line 4 is freed with the *free* command. It should be noted that the *free* command does not zeroize the data of the memory block it deallocates. Consequently, we used the *memset_volatile* function pointer to indirectly call the *memset* function. We repeated the experiments once again, using all the available optimization flags of the GCC compiler. In all cases we observed that the GCC compiler did not optimize the call to the memset function. Although the experiments showed that the data type volatile in C/C++ programing language prevents the optimization caused by the compilers, it should be noted that GCC compiler can arbitrary perform optimization even in volatile data types as mentioned in [47]. In any case, volatile function pointers can be used to increase the chances that the *memset* function will not be optimized out during compilation.

```
1. void *(*volatile memset_volatile)(void *, int, size_t) = memset;
2. void sensitive_function() {
3.   static int length;
4.   char *password_heap = malloc(length * sizeof(char));
5.   fgets(password_heap, n, stdin);
6.   memset_volatile(password_heap, 0, n * sizeof(char));
7.   free(password)
8.   sleep(120);}
```

Figure 3. Third testing application used to discover the total number of instances of the *password* variable in the volatile memory.

TABLE II. AVERAGE NUMBER OF RECOVERED INSTANCES OF THE AUTHENTICATION CREDENTIALS IN THE WEB BROWSER EXPERIMENTS

Scenario Steps	Operating System					
	Ubuntu Linux 14.04		*Windows 7/10*			
	Mozilla Firefox	*Google Chrome*	*Mozilla Firefox*	*Google Chrome*	*Internet Explorer 11*	*Microsoft Edge*
Login	38U / 8P	25U / 3P	23U / 6P	5U / 2P	21,3U / 6,3P	1,5U / 1P
Logout	12,8U / 3P	18,2U / 1,4P	10U / 0,6P	4,6U / 1.6P	13,3U / 3,6P	1,5U / 1P
Change Webpage	9,5U / 2,2P	6,3U / 0P	7U / 0P	4,6U / 0P	13U / 3,3P	0U / 0P
Close Tab	8,9U / 1,8P	4,8U / 0P	6U / 0P	4,3U / 0P	2U / 2P	0U / 0P

IV. WEB BROWSER EXPERIMENTS

In this section we investigate whether web browsers perform data zeroization. The reason that we have selected web browsers is: i) their rendering engine is written in C++ programming language meaning that they can perform low-level memory management, and ii) malware scrapers primarily target web browsers to extract sensitive information from the volatile memory (such as authentication credentials). First, we present the methodology to obtain memory dumps and next we analyze numerical results.

A. Methodology

We have selected the most common web browsers to evaluate whether we can recover user login credentials (i.e., username and password) from their allocated memory blocks during their runtime. In particular, Microsoft Edge, Internet Explorer 11, Google Chrome 49.0.2623 and Mozilla Firefox 45.0.1 were investigated running in Windows 7 and Windows 10 OSes. In Ubuntu Linux 14.04, we evaluated Mozilla Firefox 45.0.1 and Google Chrome 49.0.2623. In the experiments we performed process-dumps (sees section 3.1) at the end of four distinct steps, which are: i) after logging to the website using our authentication credentials; ii) after logging out of the website; iii) after browsing to a different webpage in the same tab of the web browser; and iv) after closing the browser tab that we used in the previous steps and browsing to a new page in a new tab. Furthermore, five different websites (i.e. Linkedin, Twitter, Facebook, Pintersest, Instagram) were used that require user login. In the four process-dumps, we searched how many times we can recover the authentication credentials (both in ASCII [36] and Unicode [37] formats).

B. Results

Table 2 shows the average number of the recovered instances of the authentication credentials. In this table, the digit before the notation U indicates the number of occurrences of the username and the digit before the notation P indicates the number of occurrences of the passwords. For instance, the value 5U/2P indicates that we extracted 5 occurrences of the username and 2 occurrences the password. As shown in Table 2, in Firefox running in Windows we recovered 23 and 6 instances of the username and password respectively after the login process has been completed. Similarly, for Linux using the same web browser we recovered 38 and 8 times the username and password respectively. Google Chrome has exact the same behavior, since in Linux we recovered more occurrences of the authentication credentials than in Windows.

A generic observation is that the password variable is found fewer times in comparison to the username in all the performed experiments. This can be attributed to the fact that the username is displayed in the pages of the tested websites, while the password evidently is not shown anywhere in the website. Overall, we can observe that Microsoft Edge has the best performance, since the recovered instances of the authentication credentials are considerably fewer than the other web users. Moreover, Edge is the only browser that deleted all the instances of the authentication credentials after changing the web page without closing the tab. On the other hand, Firefox had the worst performance, since it had the most occurrences of the authentication credentials, both in Linux and Windows.

Finally, it is important to mention that the total occurrences of the authentication credentials decrease when we log out, change webpage and close the tab of the web browser. Subsequently, the authentication credentials are more likely to be disclosed right after the user has entered them into a website.

V. CONCLUSIONS

This paper investigates security measures that can be applied at the OS and the source code level to protect sensitive information in volatile memory from disclosure attacks. Based on our experimental analysis, we observed that Windows delete the data from deallocated memory blocks, while Linux does not. This can be solved using the GRsecurity Linux kernel patch that enables the zeroization of deallocated memory blocks, using the *PAX_MEMORY_SANITIZE* option during the kernel compilation. At the source code level, the Windows developers may use the SecureZeroMemory function for manually modifying volatile memory data without facing any optimization issues. In Linux, we propose the use of *volatile* function pointers to ensure that the call to *memset* will not be optimized out. Lastly, the experiments performed in web browsers show that in most cases we can successfully recover user authentication credentials from

all the web browsers except when the user has closed the tab that used to access the website.

ACKNOWLEDGMENT

This research has been funded by the European Commission in part of the ReCRED project (Horizon H2020 Framework Programme of the European Union under GA number 653417).

REFERENCES

[1] R. J. Rodriguez, "Evolution and characterization of point-of-sale RAM scraping malware" Journal of Computer Virology and Hacking Techniques, pp. 1-14, 2016.

[2] C. Hilgers, H. Macht, T. Muller and M. Spreitzenbarth, "Post-Mortem Memory Analysis of Cold-Booted Android Devices," in IT Security Incident Management & IT Forensics (IMF), 2014 Eighth International Conference on, 2014.

[3] "Intel® 64 and IA-32 Architectures Software Developer Manuals," [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. [Accessed 17 Feb. 2016].

[4] R. Druyer, L. Torres, P. Benoit, P.-V. Bonzo and P. Le-Quere, "A survey on security features in modern FPGAs," Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2015 10th International Symposium on, pp. 1-8, 2015.

[5] N. Huq, "PoS RAM Scraper Malware" Trend Micro, [Online]. Available: http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-pos-ram-scraper-malware.pdf. [Accessed 19 October 2015].

[6] CaseyCammilleri, "PowerShell Memory Scraping for Credit Cards," #_shellntel, 18 September 2015. [Online]. Available: http://www.shellntel.com/blog/2015/9/16/powershell-cc-memory-scraper. [Accessed 22 September 2015].

[7] D. Goodin, "Credit card fraud comes of age with advances in point-of-sale botnets," asktechica, 5 Dec 2013. [Online]. Available: http://arstechnica.com/security/2013/12/credit-card-fraud-comes-of-age-with-first-known-point-of-sale-botnet/. [Accessed 19 Oct 2015].

[8] Q. Darren and R. C. Kim-Kwang, "Dropbox analysis: Data remnants on user machines," Digital Inverstigation, vol. 10, no. 1, pp. 3-18, June 2013.

[9] Q. Darren and R. C. Kim-Kwang, "Digital droplets: Microsoft SkyDrive forensic data remnants," Future Generation Computer Systems, vol. 29, no. 6, pp. 1378-1394, August 2013.

[10] Q. Darren and R. C. Kim-Kwang, "Google Drive: Forensic analysis of data remnants," Journal of Network and Computer Applications, vol. 40, pp. 179-193, April 2014.

[11] C. Hyunji, P. Jungheum, L. Sangjin and K. Cheulhoon, "Digital forensic investigation of cloud storage services," Digital Investigation, vol. 9, no. 2, pp. 81-95, November 2012.

[12] D. Apostolopoulos, G. Marinakis, C. Ntantogian and C. Xenakis, "Discovering Authentication Credentials in Volatile memory of Android Mobile Devices," in 12th IFIP conference on e-business, e-services, e-society (I3E 2013), Athens, Greece, 2013.

[13] J. Sylve, A. Case, L. Marziale and G. G. Richard, "Acquisition and analysis of volatile memory from android devices," Digital Investigation, vol. 8, pp. 175-184, 2012.

[14] C. Ntantogian, D. Apostolopoulos, G. Marinakis and C. Xenakis, "Evaluating the privacy of Android mobile applications udner forensic analysis," Computers & Security, vol. 42, pp. 66-76, May 2014.

[15] X. Chen, R. Dick and A. Choudhary, "Operating System Controlled Processor-Memory Bus Encryption," Design, Automation and Test in Europe, 2008. DATE '08, pp. 1154-1159, 2008.

[16] P. Peterson, "Technologies for Homeland Security (HST), 2010 IEEE International Conference on," pp. 120-126, 2010.

[17] V. Nagarajan, R. Gupta and A. Krishnaswamy, "Compiler-assisted memory encryption for embedded processors," in HiPEAC'07 Proceedings of the 2nd international conference on High performance embedded architectures and compilers, 2007.

[18] Y. Chenyu, B. Rogers, D. Englender, D. Solihin and M. Prvulovic, "Improving Cost, Performance, and Security of Memory Encryption and Authentication," in Computer Architecture, 2006. ISCA '06. 33rd International Symposium on, 22006.

[19] H. Daeyoung, B. Luis, S.-S. Lim and N. Dutt, "DynaPoMP: dynamic policy-driven memory protection for SPM-based embedded systems," in Proceedings of WESS '11 Proceedings of the Workshop on Embedded Systems Security, 2011.

[20] B. Rogers, Y. Solihin and M. Prvulovic, "Memory predecryption: Hiding the latency overhead of memory encryption," ACM SIGARCH Computer Architecture News, vol. 33, pp. 27-33, 2005.

[21] G. Duc and R. Keryell, "CryptoPage: An Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection," in Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual, 2006.

[22] D. Lie, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell and M. Horowitz, "Architectural support for copy and tamper resistant software," in ASPLOS IX Proceedings of the ninth international conference on Architectural support for programming languages and operating systems, 2000.

[23] G. Suh, C. O'Donnell and S. Devadas, "Aegis: A Single-Chip Secure Processor," Design & Test of Computers, IEEE, vol. 24, no. 6, pp. 570-580, 2007.

[24] S. Chhabra and Y. Solihin, "i-NVMM: A secure non-volatile main memory system with incremental encryption," in Proceedings of the International Symposium on Computer Architecture (ISCA), 2011.

[25] Z. Youtao, G. Lan, Y. Jun, Z. Xiangyu and R. Gupta, "SENSS: security enhancement to symmetric shared memory multiprocessors," in High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on, 2005.

[26] S. Weigond, Lee H, M. Ghosh and L. Chenghuai, "Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems," in Parallel Architecture and Compilation Techniques, 2004. PACT 2004. Proceedings. 13th International Conference on, 2004.

[27] L. Su, S. Courcambeck, P. Guillemin, C. Schwarz and R. Pacalet, "SecBus: Operating System controlled hierarchical page-based memory bus protection," in Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE '09., 2009.

[28] F. I. P. S. Publiation, "FIPS PUB 140-2: Security Requirements for Cryptographic Modules," 25 May 2001. [Online]. Available: http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf. [Accessed 21 Mar. 2013].

[29] X. networks, "Cryptography: What is ANSi X9.17?," [Online]. Available: http://x5.net/faqs/crypto/q159.html. [Accessed 28 Feb 2016].

[30] J. J. M. Stephen M. Trimberger, "FPGA Security: Motivations, Features, and Applications," Proceedings of the IEEE, vol. 102, no. 8, pp. 1248-1265, 2014.

[31] "The Linux Kernel Archives," [Online]. Available: https://www.kernel.org/doc/gorman/html/understand/understand009.html. [Accessed 26 Mar 2016].

[32] "GRsecurity," [Online]. Available: http://grsecurity.net/. [Accessed 9 June 2015].

[33] "Atomicorp Linux Distribution," [Online]. Available: https://atomicorp.com/. [Accessed 2016 Mar 27].

[34] "IPFire Linux Distribution," [Online]. Available: http://www.ipfire.org/. [Accessed 2016 Mar 27].

[35] M. MSDN, "RtlSecureZeroMemory routine," [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/hardware/ff562768%28v=vs.85%29.aspx. [Accessed 13 Mar 2016].

[36] ASCII Table and Description, [Online] Available at: http://www.asciitable.com

[37] What is Unicode? [Online] Available at: http://unicode.org/standard/WhatIsUnicode.html

[38] J. Damato, “MSC06-C. Beware of compiler optimizations”, Software Engineering Institute – Carnegie Mellon University, 30 September 2015, [Online] Available: https://www.securecoding.cert.org/confluence/display/c/MSC06-C.+Beware+of+compiler+optimizations

[39] T. Plum “C11: The New C Standard”, [Online] Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3631.pdf [Accessed 2016 Mar 27]

[40] L. Guan, J. Lin, B. Luo, J. Jin and J. Wang, "Protecting Private Keys against Memory Disclosure Attacks Using Hardware transactional memory" in Proceedings of the IEEE Symposium on Security and Privacy, pp. 3-19, 2015.

[41] M. Russinovich, “Windows Sysinternals, ProcDump v8.0”, [Online], Available: https://technet.microsoft.com/en-us/sysinternals/dd996900.aspx

[42] "Alpine Linux Distribution," [Online]. Available: http://www.alpinelinux.org/. [Accessed 2016 Jun 24].

[43] "Pentoo Linux Distribution," [Online]. Available: http://www.pentoo.ch/. [Accessed 2016 Jun 24].

[44] "Hardened Linux Linux Distribution," [Online]. Available: http://hardenedlinux.sourceforge.net/. [Accessed 2016 Jun 24].

[45] "Subgraph OS Linux Distribution," [Online]. Available: https://subgraph.com/sgos/. [Accessed 2016 Jun 24].

[46] S. Guelton, “A glance at compiler internals: Keep my memset”, [Online], Available: http://blog.quarkslab.com/a-glance-at-compiler-internals-keep-my-memset.html [Accessed 2016 Mar 22]

[47] ARMKEIL Microntroller Tools, “Compiler optimization and the volatile keyword”, Available: http://www.keil.com/support/man/docs/armcc/armcc_chr1359124222941.htmssss