Example Security Injections for Hardware Courses

Chenyang Li
Dept. of ECE
Portland State University
Portland, Oregon
chenyang@pdx.edu

Dr. John M Acken
Dept. of ECE
Portland State University
Portland, Oregon
acken@pdx.edu

Dr. Sohum Sohoni
Polytechnic School
Arizona State University
Tempe, Arizona
sohum.sohoni@asu.edu

Abstract - This paper gives examples of security injections in computer engineering courses, including courses on hardware design. More broadly, the paper aims to show how knowledge of hardware and software implementations relate to security exploits is important for students who design computer hardware, and how knowledge of the hardware and architectural features is important for those who focus on computer security. The paper provides examples to illustrate the impact of the knowledge of underlying architectural optimizations and hardware limitations on security features and exploits. Examples of educational tools and methods for integrating security education in context in the computer engineering curriculum are also described.

Keywords

Hardware, encryption, teaching, education

1 INTRODUCTION AND BACKGROUND

Information security is an important issue for many organizations in different disciplines, such as banking, medicine, legal and telecommunications. Computer hardware is a critical part of computer security. However, a traditional computer security class tends to focus on teaching network and software security. So, it is of increasing importance that we incorporate security and hardware in both the undergraduate and graduate curriculums. The goal is to get a basic understanding of security to all electrical and computer engineering students by injecting security examples into all the relevant electrical and computer engineering courses. This paper emphasizes computer hardware concepts related to information security and summarizes our own experiences of including encryption, which is the central part of cryptography, in teaching hardware. The specific cases in this paper demonstrate the injection of security examples relevant to various levels of computer hardware instruction rather than into a lump sum security course.

The broad collection of people (from hardware designers to programmers and information technology personnel) who deal with computer security issues need to know different things about cryptography than do cryptographers or mathematicians. Protecting the implementation from attacks is more important than theoretical knowledge of attacks on the algorithm itself [1]. To meet the current industry demand for qualified computer security professionals, we need innovative courseware that can help students apply information assurance theory into practice. One proposal to improve security education is presented by Chen & Lin [2]. The authors present their hands-on courseware design that combines practice with theory. Specifically, by using well-designed hands-on laboratory exercises, they allow students to experience the technical details of what they have learned from information security lectures. The importance of skepticism and critical thinking in the role of evaluating and procuring cryptosystems should be emphasized, since most people do not understand it, and many claims for security are questionable [3].

Many cryptography methods are difficult to program in lower level languages such as assembly language. To overcome this difficulty, tools and small examples are used. A comprehensive, animated, open-source piece of free software, CrypTool, is a good tool to help us understanding the cryptographic concepts [4]. Another open-source tool is the Progressive Learning Platform or PLP [5,6,7], which is a computer architecture simulation and visualization tool that can be used in multiple computer engineering courses.

In addition to tools, students learning is facilitated by good examples. In the late 1970's, Rivest, Shamir, and Adelman created an asymmetric encryption scheme now called the RSA algorithm [8]. The RSA algorithm provides the context for many of the examples in this paper.

The paper is organized as follows: Section 2 provides some calculations that illustrate the dependency of security algorithms on the underlying hardware implementation. Section 3 covers hardware support for security execution. Section 4 provides recent examples of the interplay between computer architecture optimizations and security issues. Section 5 provides our insights on teaching computer security. Section 6 summarizes the paper.

2 CALCULATIONS ARE HARDWARE IMPLEMENTATION DEPENDENT AS DEMONSTRATED WITH RSA EXAMPLES

2.1 The RSA Algorithm

The security of using encryption to protect messages is dependent upon the security of the key. There are two basic techniques for encrypting information: symmetric encryption and asymmetric encryption. Symmetric key encryption is a type of encryption that makes use of a single key for both the encryption and decryption process, whereas asymmetric key encryption uses different keys for encryption and decryption. In an asymmetric encryption system, the key used to do the encryption is called the public key and the key used to do the decryption is called the private key. Also, encryption can be either a stream cipher or a block cipher. Stream ciphers continuously encrypt the stream of data dependent upon previous data, whereas a block cipher encrypts each fixed size block of data independent of the other blocks. RSA is an asymmetric block cipher.

A weak point in a system using symmetric encryption is the communication of the key. When Alice sends Bob an encrypted message, the intention is to prevent Eve from eavesdropping on the contents of the message. However, because the same key is used for encryption and decryption, Eve can decrypt the message if she has intercepted the key. Asymmetric encryption uses a different key for encryption than for decryption. Therefore, Bob can publicly reveal the encryption key so that Alice can encrypt messages. However, Bob keeps the decryption key secret without any requirement to reveal it to anyone. Without the decryption key, Eve cannot decipher Alice's secret message. One method of public key encryption invented by Rivest, Shamir, and Adelman is now called the RSA algorithm [8]. The encryption key is two integers (e and n) and the decryption key is two integers (d and n). The integers e and n are made public so that anyone can encrypt a message to Bob, but d is kept secret so that only Bob can decrypt the messages. This is possible because of an interesting mathematical feature of combining the modulus operation with exponentiation. The cypher text (c) is calculated by raising the message (m) to the power (e) mod n. That is $c = m{}^\wedge e$ mod n. The cypher text is decrypted by raising the cypher text (c) to the power d mod n. That is $mxg = c{}^\wedge d$ mod n or $mxg = (m{}^\wedge e$ mod $n){}^\wedge d$ mod n. The result is that the final message (mxg) is equal to the original clear text (m). The common integer (n) is the product or two large prime numbers (p) and (q). Specifically, $n = p{\star}q$. The encryption number (e) and the decryption number (d) are related where $e{\star}d = 1$ (mod $(p\text{-}1){\star}(q\text{-}1)$). The ability to publicly reveal an encryption key to create encrypted messages that can be sent securely over open communication channels is fundamental to information security on the internet. The RSA algorithm is presently the most common public key encryption algorithm used on the internet. A step in information security education is to teach this algorithm to students and have them calculate examples. Another lesson is due to the fact that to crack RSA one needs to factor a big number, therefore the student must learn the effects of the rapid growth in size of a number and hence difficulty in calculating exponentials (i.e. *m^e* and *c^d*). Also, the examples can inject security ideas when used in math and computer science classes as real-life examples of the application of the modulus operator.

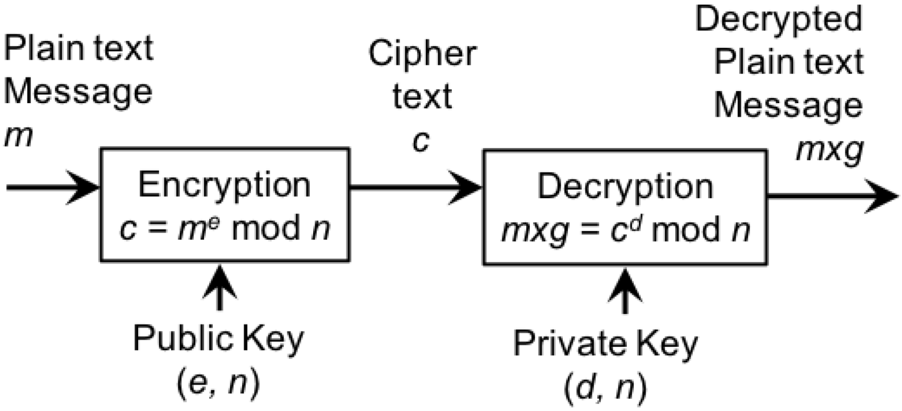

Figure 1: RSA encryption/decryption process

In summary, as shown in figure 1 for RSA the values of *n* and *e* are used in encrypting the message whereas *n* and *d* are used in decrypting the message encrypted. The RSA encryption key is the two numbers *e* and *n*. The RSA decryption key is the two numbers *d* and *n*. *m* is the message we want to send. The original text message (*Plaintext*) is encrypted by using the following equation:

$$c = (m\text{^}e) \bmod n$$

where *c* is the cipher-text which the Alice sends to Bob. On receiving the cipher-text it is decrypted by using the following equation:

$$mxg = (c\wedge d) \bmod n$$

where *mxg* = *m* and is the original text message (*Plaintext*).

2.2 M6811 8-bit Processor Teaching Example

The introductory computer principles class at Oklahoma State University used the Motorola 6811 in the labs. The teaching assistant (TA) was instructed to create a lab assignment doing RSA encryption on the 8-bit microprocessor. In the original RSA paper the public key was (*e*, *n*) = 17, 2773), the private key was (*d*, *n*) = (157, 2773), the plain text message was *m* = 920 and the cypher text was *c* = 948. While this was fine for demonstrating the algorithm, the numbers were far too large for an 8-bit processor. The TA was asked to find a small example as small numbers allow hand calculation to check the program in addition to fitting in 8-bit registers. To teach how big the numbers get, consider the tiny example of (*e*, *n*), (*d*, *n*) = (11, 15), (3, 15). For a plain text message of 7, 7^11 = 1,977,326,743. And 1,977,326,743 mod 15 is 13. So, for *m* = 7, *c* = 13. But the intermediate value is large, such as 1,977,326,743 in the previous example. An interesting observation noted by the TA was that for the tiny example of (*e*, *n*), (*d*, *n*) = (11, 15), (3, 15), the majority of the message values slip through the encryption algorithm unchanged [9]. These holes were where the cypher text was equal to the plain text. Larger key values are needed to avoid this problem. Part of the instruction for implementing encryption is to understand that the character strings in a computer are actually numbers. The standard translation uses ASCII codes for the letters. Also, the instruction includes the limitations of the modulo operation – specifically, the part of the key (*n*) needs to be larger than the largest message value (*m*) to be encrypted or else multiple plain text messages will be encrypted to the same value for the cipher text. ASCII codes are 7-bits, so the character values vary from 0 to 127. The TA found the key set of (*e*, *n*), (*d*, *n*) = (23, 143), (47, 143) works for the 8-bit processor. Some programing tricks are required to be sure there is not a calculation overflow, but these are in a later section of this paper. Table 1 shows example cases of keys used for RSA on the 6811. Notice that case 1, the tiny keys as described above, had problems with a majority of plaintext values failing to encrypt. The keys for Case 2 provide values that can be implemented on an 8-bit processor without the problem of a large percentage of plaintext values failing to encrypt. Case 3, which are the keys and values from the original RSA paper, is not readily implemented on an 8-bit processor. The lessons for electrical engineers in an introductory computer principles class are the number limitations for large integer calculations as demonstrated by the real-world example of RSA encryption.

	p	q	(e, n)	(d, n)	m	c
Case 1	3	5	11, 15	3, 15	7	13
Case 2	11	13	23, 143	47, 143	140	17
Case 3	47	59	17, ~~2773~~	157, ~~2773~~	~~920~~	~~948~~

Table 1. RSA encryption examples implemented with 6811 assembly code. Note, the original RSA example which is case 3 exceeded the capability of the 6811. The lined through values identify numbers too large for an 8-bit processor.

2.3 Synergy of Teaching and Research

While creating the small example described in the previous section, the TA found that some key values created cipher text that was equal to the original plain text. The fact that some keys for RSA do fail to encrypt more messages that others is formally described by Blakely and Borosh [10]. Specifically, they identified the limits for these fixed points where *m^e* mod $n = m$, and they proposed a measure of opacity for a cryptosystem for how many fixed points can occur. Further, Chmielowiec presented an estimate of the probability of finding a fixed point and the resulting likelihood of such points from random selection [11]. Behnaz Sadr further characterized the location and likelihood of fixed points [9]. She used the term "holes" for the cases where a message leaked through RSA encryption so that ciphertext was equal to plaintext. Specifically, where $m = c = m\^{}e$ mod n. The other two papers [10, 11] used $x\^{}e = x$ mod n. Sadr found several characteristics of these holes, such as for any set of keys there are at least 6 holes, and that the number of holes is symmetric about $n/2$. Kocakulak and Temel demonstrated Sadr's findings with a Java implementation [12]. The lesson for students is that not all keys are equally good.

2.4 C Language Implementation Example

Digital computer classes introduce the hardware implementation effects on programming languages. C is a widely used programming language. As a demonstration of the relationship between programming languages and the underlying hardware implementation, we use the implementation of the RSA key generation and encryption/decryption in C.

There are two parts of this implementation. The first part selects two prime numbers, which are used for key generation. The second part uses the generated keys for encryption and decryption. This module encrypts and decrypts the given message/plaintext (integer form) to a cipher-text with the key pair generated in the key generation module.

The examples in the table 2 demonstrate the effect of different key size. Notice that case 3 are the keys and values from the original RSA paper, whereas case 4 uses a similar sized key. Case 5 is an example of overflow during the encryption/decryption process whereas the last one shows the case of overflow in the key generation part: the program will display cipher-text is 0 due to the overflow problem, to be specific in this case, $p \star q$ is not able to fit in a integer data type. The times in the table were generated from running C code on an Intel core i5 laptop with a Mac OS. Although these p and q values seem to be big, this key size is about 150 bits, which is far less than the 2048 key size in real life. The lesson learned here is that even for this small key size a simple implementation cannot handle the intermediate values. The largest integer number for integer data type is 2,147,483,647. Therefore, the lesson to inject is that implementing security features requires the ability to handle very large integers.

One implementation lesson is for the key generation module. One way to implement key generation is using iterative attempts. The Chinese remainder theorem is a theorem of number theory, which states that if one knows the remainders of the Euclidean division of an integer n by several integers, then one can determine uniquely the remainder of the division of n by the product of these integers, under the condition that the divisors are pairwise coprime. Allows an alternative implementation of key generation that directly applies a math concept to replace trial and error.

A second lesson is in the implementation of encryption and decryption module. From modular arithmetic, one can use a trick in ordering calculations to avoid overflow of bits:

$$A^e\ mod\ N =\ A(A(A \ldots ((A^2)mod\ N) \ldots mod\ N)mod\ Nmod\ N)$$

for example, with *e* = 5:

$$A^5\ mod\ N =\ A(A(A(A^2)mod\ N)mod\ N)mod\ N)mode\ N$$

The above property can be used for modular arithmetic calculation for example, (2^3) mod 5 = 3. This calculation can be modified as follows: ((((2 mod 5) *2) mod 5) *2) mod 5 = 3. We inject the example of reordering calculations for encryption to teach the lesson as used in the C code to avoid overflow, since A^e could be a very big number and A^2 is relatively small. The important theme to note here, is how the limitations of the hardware and the awareness of these limitations, dictates the decisions at the algorithmic implementation level.

	p	*q*	*(e, n)*	*(d, n)*	*m*	*c*	encrypt time	decrypt time
Case 3	47	59	17, 2773	157, 2773	140	2114	4.17us	4.33us
Case 4	23	29	13, 667	237, 667	140	487	3.67us	3.83us
Case 5	2204333	2204341	43, ⋆ex5n	⋆ex5d, ⋆ex5n	140	overflow	overflow	overflow
Case 6	⋆ex6p	⋆ex6q	overflow	overflow	140	0	overflow	overflow

⋆ex5n = 4859101609553

⋆ex5d = 1356027125827

⋆ex6p = 2295368686771969123000270782186855260112447232907

⋆⋆ex6q = 3076254225030127069205146053958616692729173275496

Table 2. RSA examples implemented in C

2.5 Python Language Implementation Example

Some programming languages remove many of the hardware implementation restrictions that have been described. Python is a very popular dynamic programming language.

The examples in the table 3 are used to demonstrate the effect of different key sizes. Again case 3 is the original RSA paper case, whereas case 4 uses a similar sized key. Case 5 will have overflow in C implementation but works fine with Python, whereas the last case shows the case of overflow in the key generation part: the program displays "OverflowError". These times were generated from Python code on an Intel core i5 laptop with a Mac OS.

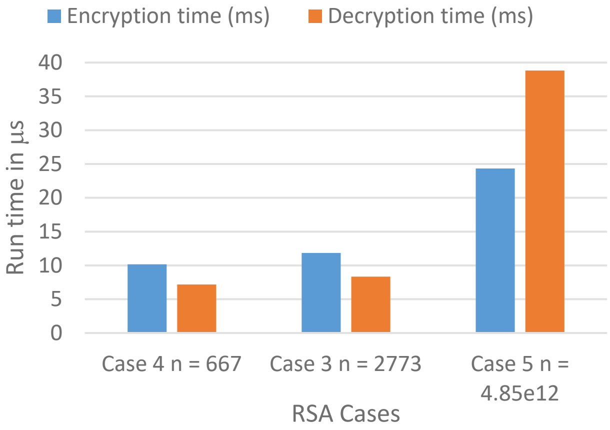

Figure 2: Encryption and Decryption time for message=140

Alese, et al. [13] have conducted a similar experiment, but they have compared the key size to the encryption, decryption and the key generation time as shown in figure 3. The lesson here is that security due to increasing key size has an impact on computation time.

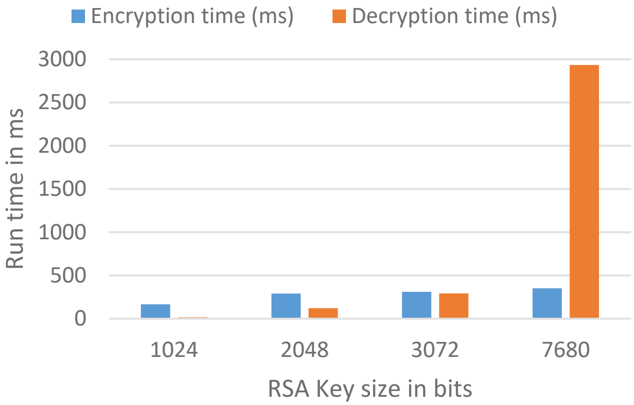

Figure 3: Key size vs Encryption and Decryption time [13]

In an RSA operation, the computation is performed by a series of modular multiplications. In practical applications, a small public exponent can be considered as a public key. Many users can use the same public exponent, each with a different modulus. This makes encryption faster than decryption and verification faster than signing. Similarly, this principle can be used for larger numbers, and computation can be simplified and overflow of bits can be prevented.

This section shows how RSA encryption examples are used in general computer hardware instructions to demonstrate implementation principles and limitations. The next section shows an example of a specific hardware implementation directed at security.

	p	*q*	*(e, n)*	*(d, n)*	*m*	*c*	encrypt time	decrypt time
Case 3	47	59	17, 2773	157,2773	140	0x617	11.83us	8.33us
Case 4	23	29	13, 667	237, 667	140	0x1e7	10.17us	7.17us
Case 5	2204333	2204341	43, ⋆ex5n	⋆ex5d, ⋆ex5n	140	⋆ex5c	24.33us	38.83us
Case 6	ex6p	Ex6q	overflow	overflow	140	overflow	overflow	overflow
⋆ex5c = 0x313f09ce84d								

Table 3. RSA implementation in Python

3 EXAMPLE HARDWARE SUPPORT FOR SECURITY

Hardware features make security implementation easier, for example, Intel's Software Guard Extensions (SGX) as one of the ideas. SGX is a set of extensions to the Intel architecture that aims to provide confidentiality and integrity even in the presence of privileged malware [14, 15].

These isolated execution environments, called enclaves, are designed to run software and handle secrets in a trustworthy manner, even on a host where all the system software (including OS, hypervisor, etc.) and system memory are untrusted. When enclave code and data are cache-resident, they are guarded by CPU access controls; when flushed to DRAM or disk, they are transparently encrypted and integrity protected by an on-chip memory encryption engine [16]. There are three main functionalities that enclaves achieve: Isolation–code and data inside the enclave protected memory cannot be read/modified by any process external to the enclave. Sealing– the process of encrypting it so that it can be written to untrusted memory or storage without revealing its contents [17] And Attestation–a special signing key and instructions are used to provide an unforgeable report attesting to code, static data, and (hardware-specific) metadata of an enclave, as well as outputs of computations performed inside the enclave [18]. There are two forms of attestation: local and remote. Local attestation is between two enclaves on the same platform Remote attestation generates a report that can be verified by any remote party. The lesson for a hardware architecture class is that this an example of how the security of a system can be enhanced by the support of specific hardware features. This section describes just one example of a specific hardware implementation to improve security. The next section describes how a student can learn about security related to computer hardware in general.

4 RECENT LESSONS DEMONSTRATING HARDWARE IMPLEMENTATION EFFECTS ON SECURITY

Computer Architecture can improve hardware performance by implementing specific hardware for instruction execution improvement and cache access speed up.

These performance-enhancing features provide opportunities for security problems. One example is out of order execution, which allows an instruction to execute when it is ready rather than waiting for its turn in the sequence of instructions. Out of order execution is utilized by the recent attack know as meltdown [19, 20]. The details of the attack and its effects are too large for an explanation in this paper. However, the two lessons to be learned for hardware designers illustrate the main theme of this paper. The first lesson is that there are tradeoffs with security and performance. This has been demonstrated by the patch releases for this bug, which simply disable the out of order execution thereby degrading performance while improving security. The second lesson from this example is that security side effects need to be considered at design time. As described in the next section, the Progressive Learning Platform was used to teach students about stack overflow as a general concept and as applied to security.

The second recent example comes from the performance enhancement hardware to support speculative execution. When a conditional branch is reached, the performance can be enhanced by executing both paths of the branch in parallel, and then saving only the path that should have been executed after the condition decision is resolved. The Spectre [20, 21] attack takes advantage of this by accessing the cache illegally down the conditional path that will be cancelled. Once again, Spectre is far too complex to describe in this short paper, but there is still a hardware lesson to be learned. The specific lesson for this attack is that the hardware implementation can have side effects outside the security control of the operating system.

5 TEACHING COMPUTER SECURITY

Whether it is block-chains, RSA, or other security fundamentals, they all rely on certain functionality and limitations of the underlying hardware. For example, if “one-way functions” could have all the possibilities for their reverse calculations computed within milliseconds, they would not remain one-way functions. While this example is far-fetched with today’s technology, there are many examples within the last two decades, where the growth in computational performance has resulted in changes to security protocols. It is thus extremely important for students in computer science and computer engineering to understand security issues in the context of various courses that they take, including courses related to hardware. A strong and accurate mental model of computing is not only essential for becoming a good programmer, but is crucial to understanding various exploits, as they often occur at the boundary of hardware and software. The idea of injecting security concepts into various hardware courses rather than just tacking on a "security hardware course" requires a cultural shift in electrical and computer engineering departments to take a holistic view of the curriculum as opposed to isolated approach of separate course topics.

To aid the adoption of security injections in various courses, educational tools can be leveraged effectively. The Progressive Learning Platform (PLP) is a simulation and visualization tool that was designed for use in multiple computer science and computer engineering courses. It allows students to visually inspect the state of the machine (registers, memory locations, even buses and elements of the datapath and control) for their programs at their own pace. This very different from canned animations over which students have little control and no ownership. PLP simulates not only CPU internals, but the I/O available on some standard development boards like the Nexys 3 from Digilent. Additionally, the CPU itself is a Verilog description that can be synthesized on the board, so that PLP can be tested with real-time input and output, not just in simulation. Students can thus experience user-input related timing issues that are often undetectable in simulations.

6 SUMMARY

Underlying hardware implementation details affect information security solutions. Computer hardware design courses that included RSA encryption examples showed the students the effects and limitations of hardware details upon software programs. Additionally, digital computer simulation and instruction tools are useful to teach the students how specific computer implementation limitations (such as stack overflow) relate to information security problems. Basic hardware design features (such as Intel's SGX) provide tools for security aware software implementations. The key thesis is that all hardware designers should be introduced to information security implementation implications in much the same way they are introduced to power, testing, and reliability concepts, that is by injecting security examples into many different courses.

REFERENCES

[1] William Hugh Murray: What the Graduate Needs to Know about Cryptography. 12th Colloquium for Information Systems Security Education University of Texas, Dallas, TX June 2 - 4, 2008

[2] Li-Chiou Chen and Chienting Lin: Combining Theory with Practice in Information Security Education. 11th Colloquium for Information Systems Security Education, June 4-7 Boston University (2007)

[3] Barry S. Fagin, Leemon C. Baird, Jeffrey W. Humphries and Dino L. Schweitzer: Teaching Information Security With Skepticism and Critical Thinking. 11th Colloquium for Information Systems Security Education Boston University

[4] Rong Yang, Layne Wallace, Ian Burchett: Teaching Cryptology At All Levels Using CrypTool. 15th Colloquium for Information Systems Security Education Fairborn, Ohio June 13-15, 2011.

[5] D. Fritz, W. Mulia, S. Sohoni, "The Progressive Learning Platform"; Workshop on Computer Architecture Education in conjunction with IEEE HPCA-17, San Antonio, TX, February 2011.

[6] D. Fritz, W. Mulia, S. Sohoni, B. Gordon, K. Kearney, M. Mwavita, "The Progressive Learning Platform for Computer Engineering", Proc. 2011 American Society for Engineering Education Annual Conference and Expo (ECE Division), pp 22.1491.1 - 22.1491.14, Vancouver, Canada, June 2011. https://peer.asee.org/18550.

[7] S. Sohoni, D. Fritz, W. Mulia, "Transforming a Microprocessors Course through the Progressive Learning Platform", Proc. the American Society for Engineering Education Midwest Section, Russelville, AR, September 2011.

[8] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," Comm. ACM 21(2) pp 120-126, 1978.

[9] B. Sadr, "Finding cases of ciphertext equal to plaintext in the RSA algorithm, MS Thesis, Oklahoma State University, Stillwater OK, July 2011.

[10] R. R. Blakely and I. Borosh, "Rivest-Shamir-Adleman Public Key Cryptosystems Do Not Always Conceal Messages," Comp. & Maths, Vol5. Pp169-178.

[11] Andrzej Chmielowiec, "Fixed points of the RSA encryption algorithm," In Theoretical Computer Science, Volume 411, Issue 1, 2010, Pages 288-292,

[12] Mustafa Kocakulak and Turgay Temel, "Implementation of Special Cases for RSA Algorithm Where Plaintext is Equal to ciphertext in Java," ICENS, Sarajevo, May 2016.

[13] Alese, B. K., Philemon E. D., Falaki, S. O. "Comparative Analysis of Public-Key Encryption Schemes" International Journal of Engineering and Technology Volume 2 No. 9, September, 2012

[14] Costan & Devadas, Intel SGX Explained, eprint 2016/086

[15] Andrew Baumann, Hardware is the new Software, Proceedings of the 16th Workshop on Hot Topics in Operating Systems, p.132-137, May 07-10, 2017, Whistler, BC, Canada

[16] P. L. Aublin, F. Kelbert, D. O'Keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers, and P. Pietzuch. TaLoS: Secure and Transparent TLS Termination inside SGX Enclaves. Technical Report 2017/5, Imperial College London, March 2017.

[17] Intel® Software Guard Extensions Tutorial Series: Part 1, Intel® SGX Foundation: https://software.intel.com/en-us/articles/intel-software-guard-extensions-tutorial-part-1-foundation.

[18] Ben A. Fisch, Dhinakaran Vinayagamurthy, D.B.S.G.: Iron: Functional Encryption using Intel SGX. Cryptology ePrint Archive, Report 2016/1071 (2016).

[19] Moritz Lipp, Michael Schwartz, Danidal Gruss, Thomas Prescher, Wiener Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval, Yarom, Mike Hamburg. "Meltdown" Graz University of Technology, Cberus Technology GmBH, University of Pennsylvania, University of Maryland, University of Adelaide, and Rambut Cryptography research division. Cornell University Report: arXiv:1801.01207 [cs.CR] 3 Jan 2018

[20] Jann Horn, "Reading privileged memory with a side channel", Project Zero at Google. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

[21] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Mortiz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, Yuval Yarom, "Specter Attacks: Exploiting Speculative Execution" Universtiy of Pennsylvania, University of Maryland, Graz University of Technology, Rambus, University of Adelaide. Cornell University Report: arXiv:1801.01203v1 [cs.CR] 3 Jan 2018