May-June 2020
ISSN: 0193-4120 Page No. 6227 - 6234

Suspected Software-Code Restoration Using a Dictionary Led System

Menachem Domb and Guy Leshem
Ashkelon Academy College, Ashkelon, Israel

Article Info
Volume 83
Page Number: 6227 - 6234
Publication Issue:
May-June 2020

Article History
Article Received: *19 November 2019*
Revised: *27 January 2020*
Accepted: *24 February 2020*
Publication: *18 May 2020*

Abstract:
The recent increase of devices connected to the Internet and the introduction of new paradigms; Internet of Things [IoT] and Cloud computing], exposes the Internet to sever security risks, especially malicious alteration of the application software code or utilizing malfunction codes to attack the system. Such attack can change the behavior and the outcome of the system. This work proposes a rule-based technique for substituting suspicious-code by secure-code. The technique scans over source code using parsing techniques and identifies key patterns. These patterns are matched against a dictionary which stores mappings from suspicious to secure code. For practical purposes we suggest using the proposed technique in conjunction with a secure execution environment, implemented by Intel Software Guard Extension (SGX). The proposed system may also be helpful at the execution environment by transforming the executable code back to its source code and then apply the proposed system to discover vulnerable code and even correct it. This may also be used to discover code anomalies or security issues and activate the appropriate warning preventing damage to the production environment.

Keywords: *Software vulnerabilities, Rule based system, Software Development Kit (SDK), Intel® Software Guard Extensions (SGX)..*

1. Introduction

The recent advancement of IoT, Cloud computing and the significant increase of devices connected to the Internet, exposes it to severe security risks. One of the common security risks is malicious alteration of critical software components or the utilization of unsecured codes to generate a cyber security attack.

This work introduces a comprehensive system which accepts undetected program codes and outputs safe security- codes. It comprises two components: a system that scans and detects suspicious codes and replaces them with safe codes, and secured software development and execution environment in which the proposed system is executed. The system scans, analyzes, detects and removes suspicious codes and replacing them with equivalent safe codes, while keeping the same functional capabilities it is expected to perform. To ensure a secure development and deployment environment we adopted the Intel Software Guard Extensions (SGX) environment; an isolated and secure environment equipped with a set of instructions providing complete protection from disclosure or modification. The advantage of using the Intel SGX is to execute code snippets in a protected execution area, which can assure that the code is not modified by an external party.

Vulnerable/suspicious code detection is done by scanning the source code and searching for code patterns that comply with one or more predefined rules stored in a ruleset table. The table consists of three columns, the rule id, the rule notation and the proposed safe code replacing the suspected code. Vulnerable/Suspicious events are situations that can be utilized by malicious agents to change the behavior of the program causing wrong results and dangerous impacts. For example, stack overflow or functions that run correctly with input X but are dangerous with input Y. After identifying the

May-June 2020
ISSN: 0193-4120 Page No. 6227 - 6234

suspicious code, the system replaces the relevant code with a safe code, preventing the system from entering a potentially dangerous situation. The safe code is taken from the corresponding rule entry.

To protect the proposed system execution from any external interruptions and intrusions, we selected the Software Guard extensions (SGX) module [1] allowing programs to be executed inside logically separated segments of the CPU called enclaves. An enclave is a general-purpose module used for any kind of program. SGX provides a hardware-based guarantee that the programs and memory inside an enclave cannot be read or modified by any program outside of the enclave. Any type of special access permission can access the memory inside an enclave. It uses special libraries provided by the SGX Software Development Kit (SDK) solely applying C and C++ languages. An adversary is not able to discover what is accessed inside the enclave or what is written back to the RAM when the cache is full. Any data in the enclave that must be written back to the main memory is encrypted and signed so that it cannot be altered by any other program. Fig. 1 depicts the threats an executable application may experience. However, these threats are not effective when the executable application is stored in the SGX environment.

Fig. 1: The processor with the protected area (in the upper left square)

2. Related work

The risk of utilizing malfunction software codes to intentionally change the system's behavior or its outcome is known and appears in many publications. However, the idea of protecting software codes from such situations and proposing active actions to avoid it has increased and various standards have been proposed. The basic idea of defining and enforcing standard coding which blocks common security holes preventing malicious programs from utilizing it is described in [2,3]. A more specific guidelines for the automobile domain appear in [4,5]. Several examples of security-focused coding practices and standards: CERT, OWASP, CWE, MISRA, AUTOSAR, and IEC 61508-based standards. David Svoboda [6] provides secured development training sets intended for software developers. Indeed, this approach prevents some of the security risks but does not provide an automatic way to enforce it. Several commercial security code testing tools are available [7] providing an automatic scan of the code to identify and remediate vulnerabilities. However, the proposed solutions do not cope with the possibility of a malicious altering the executable software code to change its behavior to support its needs.

Intel introduced a new hardware extension SGX (Software Guard Extensions) [8,9] in their CPUs, starting with the Skylake microarchitecture. SGX is an isolated mechanism, aiming to protect codes and data from modification or disclosure [10]. This protection uses special execution environments, called enclaves, which work on memory areas that are hardware-isolated from the operating system. The memory space used by the enclaves is encrypted to protect the application’s secrets from hardware attackers. Typical use-cases include password input, password managers, and cryptographic operations. It is recommended that cryptographic keys are stored inside enclaves [11]. Apart from protecting software, the hardware supports isolation due to fear of super malware inside enclaves. Rutkowska [12] outlined a scenario where a benign looking enclave fetches encrypted malware from an external server and

May-June 2020
ISSN: 0193-4120 Page No. 6227 - 6234

decrypts and executes it within the enclave. In this scenario, it is impossible to debug, reverse engineer, or in any other way analyze the executed malware. Aumasson and Merino [13] eliminated this fear by arguing that enclaves always run with user space privileges and can neither issue syscalls nor perform any I/O operations. Moreover, SGX is a highly restrictive environment for implementing cache side-channel attacks. Both state-of-the-art malware and side-channel attacks rely on several primitives that are not available in SGX enclaves. Consequently, hitherto no enclave malware has been demonstrated on real hardware.

To adequately address the issue of malware and innocent software that can be exploited or other vulnerabilities, it all executable files need to be transformed to their equivalent source-code in C language using a reverse-engineering tool and then scanned to discover potential threats and correct them with safe codes. Once this is completed, the corrected source code files are compiled back into an executable file replacing the original executable file. The entire process is done in the Intel® SGX environment.

The rest of the paper is organized as follows: in the next section we outline the literature review. Then we proceed with a detailed explanation of the proposed solution. In the dictionary section we describe the dictionary operation by means of a case study. We continue with the details of an experiment we are conducting and preliminary results. We conclude with a summary and recommendations for future work.

3. Proposed System

The solution is based on two main components: a. Rule based vulnerable code detection and cleanup, and b. A strictly secured development and deployment environment [SGX].

Fig. 2 describes in detail the elements and the process-flow of component a. The input to the process is an exe file. In step 1 the exe file is de-compiled to generate the equivalent source code program in the C programing language. The C source code is then loaded to the vulnerable discovery process; an automatic system to detect suspicious operations and convert them into safe actions. The discovery is based on applying a set of predefined rules and each code section that complies with any rule is designated as a suspicious code. The discovered vulnerable code is loaded to the cleanup process for removal of the suspicious code and replaced with a corrected code. The output file is then compiled to generate the exe file, which is deployed to the target computer for production processing.

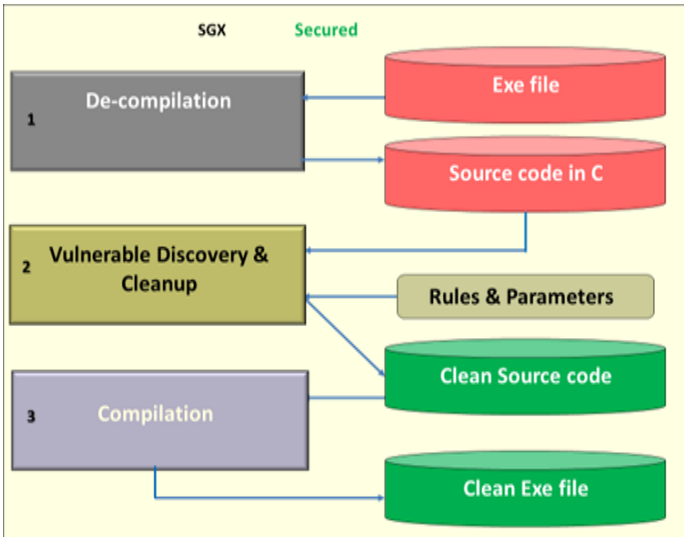

Fig. 2: The proposed system components and process

Fig. 3 details the flow operation of the proposed system. It starts [upper left box] with accepting an executable file, decompiling it to C code, parsing it and checking if it matches any function in the dictionary. If it does, the code is replaced with a safe code. Once the entire code is completed the revised C code is compiled and the new executable file is tested and saved with an indication that it is a safe executable file until the next cycle.

May-June 2020
ISSN: 0193-4120 Page No. 6227 - 6234

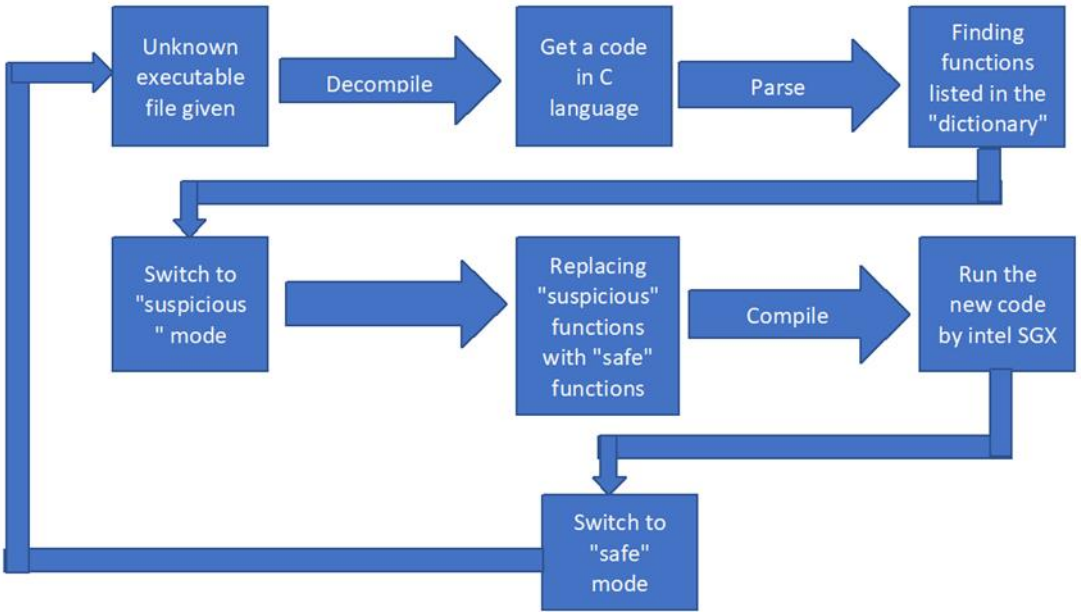

Fig. 3: Flow chart of the proposed system

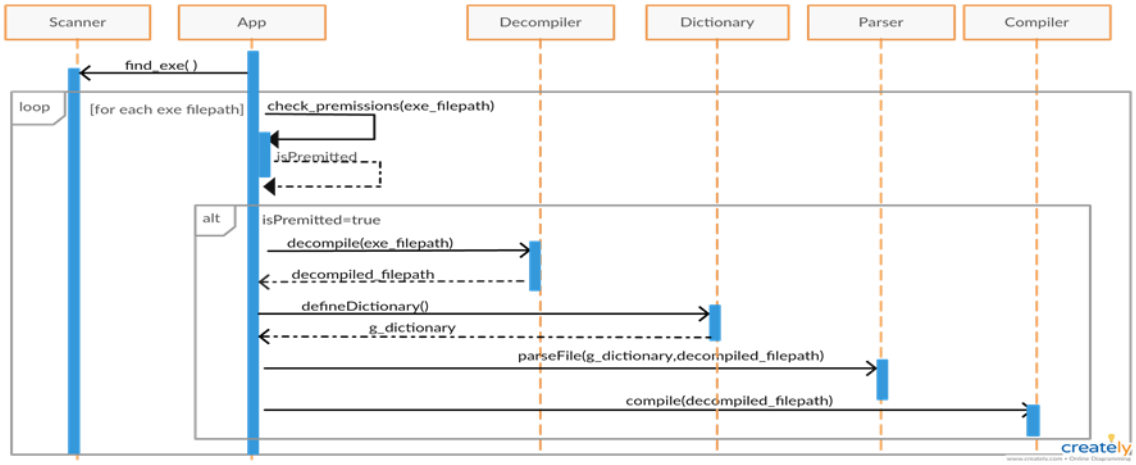

Fig. 4 outlines the sequence of events in a self-explanatory flowchart diagram.

Fig. 4: Sequence diagram of the proposed system

4. Experiment

The experiment was done using the Visual Studio environment with Intel® SGX SDK plugin that enables the enclave functions to run even without the appropriate processor. The system receives an executable file (xxx.exe) and executes all of the stages (as described above) automatically.

Below are the data items we use to demonstrate the system functionality.

The suspicious actions identified in this research are:

1) The strcpy operation – which can cause buffer overflow in some cases.
2) Division by zero – division where the divisor (denominator) is zero.
3) Recursion - which can result in stack overflow.

For the *strcpy* operation and *division by zero*, safe operation was identified and replaced. For the *recursion* operation, detection was performed, and a safe operation was investigated that would prevent the recursion from causing stack overflow. After investigating this case and writing a safe operation for the recursion operation, the recursion operation was replaced by a safe operation. Intel® SGX technology protects the code and the data from exposure or modification by placing parts of the code in special areas of the processor.

May-June 2020
ISSN: 0193-4120 Page No. 6227 - 6234

Below are the process stages, from loading the executable file to replacing it with a safe code:

Stage 1: **Searching** – The system scans looking for executable files in the computer system.
The scan is performed on two levels:
a. Scan the computer recursively looking for executable files.
b. Checking access permissions for each executable file found.
The function accepts the following parameters:
a. The folder name to scan
b. A filter on a file name
c. Pointer to the file list that is returned

Fig. 5 describes the file scanning process.

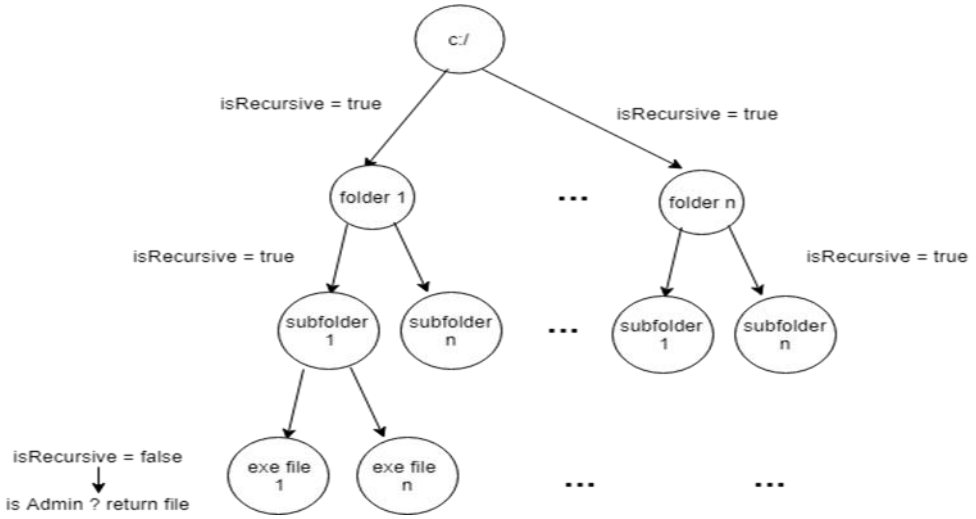

Fig. 5: Scanning methods that search for exe files

Stage 2: **De-compilation** – we used the *Retargetable Decompiler* system to transform the executable file to a code in C programming language. The function returns a path to the C file received after the de-compilation.

Stage 3: **Identify suspicious situations** – The C code file is parsed, and its function names are searched against the dictionary; a table containing unsafe functions, such as the strcpy function which may cause buffer overflow. Table 1 lists examples of suspected strings and the associated action to perform.

Table 1: The string/characters found and the action that needs to be taken to prevent dangerous situations

String	Action	Explain
#	*break*	include - Go to the next line
// , /* */	*isComment = true* *break*	go to the next line
main	*isMain = true*	for adding rows to enable Enclave
{ && isMain	*q.push('{')* *if (q.size() == 1)*	adding the rows required to create the enclave link to the rows below

May-June 2020
ISSN: 0193-4120 Page No. 6227 - 6234

} && isMain	*q.pop()* *if (q.size() == 0)*	the end of the **main**, adding the lines needed to destroy the enclave link to the lines below
dictionary.count(str)>0	*isSuspiciousFuncFound = true* *replaceLineInTempFile*	sending a function to replace the suspicious operation to ensure a safe operation.

Stage 4: Replacing the 'suspicious' functions with 'safe' functions –

The "safe" functions perform the same actions as the "suspicious" functions. In this experiment we focused on 3 examples. For example, the *strcpy* operation is replaced by the *enclaveStrcpy* operation. Fig. 6 depicts the process of identifying a suspected function and handling it based on its recursion functionality. In the experiment of replacing a suspicious operation with a safe operation, we use the *replaceLineInTempFile* function, as follows:

1)**Replace strcpy operation** - The call to the *strcpy* function is replaced by the *enclaveStrcpy* function. The function expects that both parameters will be sent to the original *strcpy* function. Hence, the parameters sent to the *strcpy* function are used for the new function. This is done with the *getFuncParams* function which returns the parameters.

2) **Replace Divided by Zero operation** – When a division action is found in the file, it is converted to the *enclaveDevideByZero* function. Because a value cannot be returned from the *enclave* by the "return" command, an out-of-size * byte size pointer is sent to this function as well as the parameters sent to the original function. After the division has been copied, the *memcpy* is copied into the pointer.

3) **Replacing the recursion operation** – The recursion call is switched to *enclaveRecursive*.
At this stage the updated C code file contains safe actions only.

Stage 5: **Compilation** – The updated file is compiled.
Once all changes are done, the executable file is generated by compiling the C code file accepted in Stage 4. Since the above files may contain secured functions that run only in the enclave section of the processor, the output executable is stored in the same project that would contain the settings allowing enclave to run.
Stage 6: Replacing – The original file is replaced with the new executable file. To place the new executable file in the original file location the original executable file needs to be "cancelled" so that it is discontinued (unreadable), and then the new file is renamed so that its name and location are the same as the original.

May-June 2020
ISSN: 0193-4120 Page No. 6227 - 6234

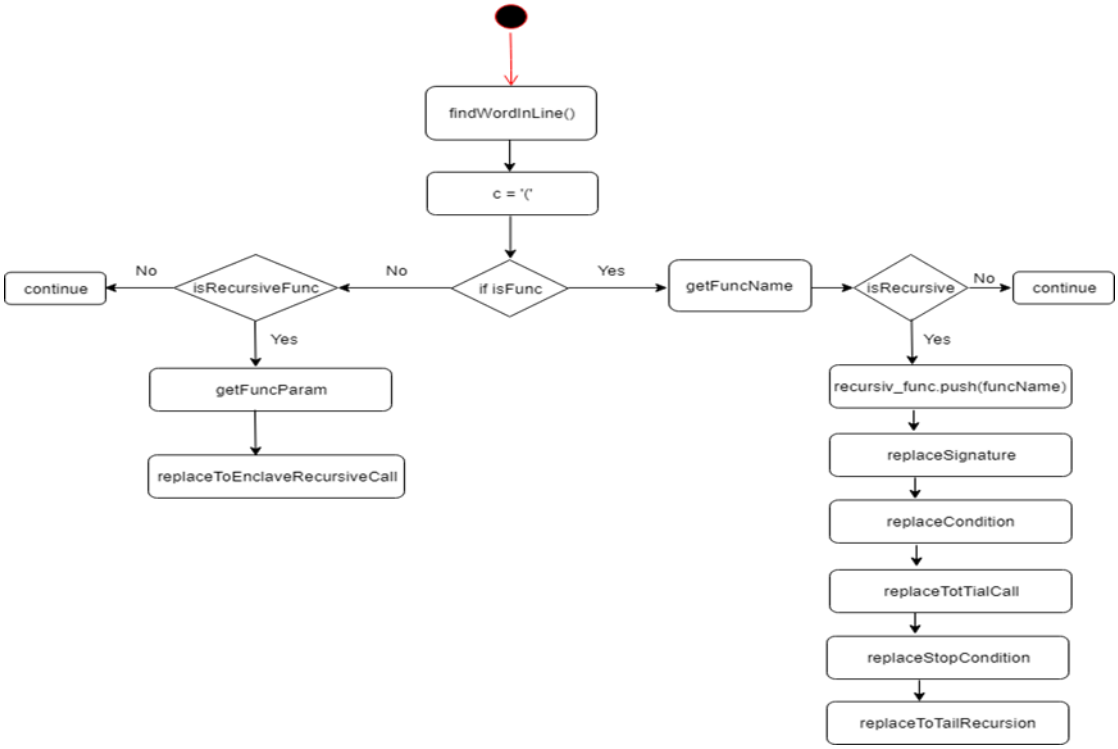

Fig. 6: Activity diagram of investigation of the recursion operation

The Dictionary (containing suspicious functions)

The dictionary contains all the functions known to be vulnerable, such as: all types of overflow (e.g., heap, Buffer, Stack, Integer), Format string, and C language functions, such as: strcpy(), sprintf(), vsprintf(), strcat(), scanf(), bcopy(), gets()), recursive. Each vulnerable function must have a safe function that will replace the vulnerable one.

Example: ***Stack Overflow***, a software vulnerability that causes the program to crash due to stack overflow. The common cause of stack overflow is infinite or excessively deep recursion, because the recursion process in some cases requires vast memory allocation while running. In some cases, the stack may grow significantly causing the program to run slower or even crash. It may be replaced by "*tail recursion*" where it is performed by tail reading optimization ***TCO***. This is a process where a smart compiler may call a function without requesting additional stack space.

The replacement is performed as follows:

1) The file is scanned for a ***recursive*** function.
2) If a recursive function was found, it was "converted" into a ***tail recursion*** by replacing its ***signature***, ***stopping conditions***, and ***reading lines***.

When a *recursive* function is found in the executable file (after de-compilation) the function is replaced by the ***enclaveRecursive*** function. The function is defined as ECALL, enclave functions that the user can access.

Experiment Summary

In order to create an automatic system that adequately addresses these problems, all the executable files on the computer need to be scanned to ensure there is no threat. Since there are executables with access privileges that do not allow the file to be read, the system will fail in these cases. Thus access permissions of the file need to be checked before the system performs its actions. The file scan should contain all possible end cases for a suspicious action to be replaced by an appropriate 'safe' action. After replacing the 'suspicious' operation with the 'safe' operation, the file needs to be converted back to the executable file to replace the original executable file. This assumes that the

May-June 2020
ISSN: 0193-4120 Page No. 6227 - 6234

compilation will run with the special settings of Intel SGX projects.

5. Conclusions and future work

In this paper we demonstrate the feasibility of a system that protects software systems by making them immune to code alteration attacks by detecting and replacing existing vulnerable codes with equivalent safe codes using SGX and the relevant enclave functions.

Future work will focus on expanding the dictionary and writing additional functions that address software vulnerabilities.

REFERENCES

1. Intel Corp., "Intel® 64 and IA-32 Architectures Software Developer's Manual," April 2016. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/docum ents/manuals/64-ia-32-architectures-software-developersmanual.pdf. [Accessed 7 May 2016].
2. White paper, Addressing Security vulnerabilities in embedded applications using best practice software development processes and standards, an introduction to applying CWE coding guidelines and achieving CERT security compliance using static analysis tools
3. PRQA, Programming Research, www.programmingresearch.com
4. Richard Bellairs, CWE List & CERT Secure Coding Standards-An Overview, Security & compliance static analysis, , Oct 8.2018
5. Priyasloka Arya, Software Vulnerability Analysis & Secure Coding in Vehicle Systems, Auto tech review, Technology, 17 July 2017
6. R. Kurachi et al., "Improving secure coding rules for automotive software by using a vulnerability database," 2018 IEEE International Conference on Vehicular Electronics and Safety (ICVES), Madrid, 2018, pp. 1-8
7. David Svoboda, Using the SEI CERT Coding Standards to Improve Security of the Internet of Things, Carnegie Mellon University, Software Engineering Institute blog, Feb 2019
8. Kiuwan web site, Secure Coding Testing Tool, Identify Security Risks Faster, https: //www.kiuwan.com/code-security-sast
9. F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd and C. Rozas, "SGX Instructions to Support Dynamic Memory Allocation Inside an Enclave," in HASP, Seoul, South Korea, 2016.
10. Intel Corporation (2016a) Intel Software Guard Extensions (Intel® SGX). https:// software.intel.com/en-us/sgx. Accessed 7 Nov 2016
11. Costan V, Devadas S (2016) Intel sgx explained. Technical report. Cryptology ePrint Archive, Report 2016/086.
12. Intel Corporation (2016b) Hardening Password Managers with Intel Software Guard Extensions: White Paper. https://pdfs.semanticscholar.org/ec40/ 833215b3d415c9525940690d0a94d2a178ca.pdf
13. Rutkowska J (2013) Thoughts on Intel's upcoming Software Guard Extensions (Part 2). http://theinvisiblethings.blogspot.co.at/2013/09/thoughts-onintels-upcoming-software.html. Accessed 20 Oct 2016
14. Aumasson J-P, Merino L (2016) SGX Secure Enclaves in Practice: Security and Crypto Review. In: Black Hat 2016 Briefings. https://www.blackhat.com/ docs/us-16/materials/us-16-Aumasson-SGX-Secure-Enclaves-In-PracticeSecurity-And-Crypto-Review-wp.pdf