A Mechanism for Secure Delivery of Bare Machine Computing Software

Robert G. Eyer, Ramesh K. Karne, and Alexander L. Wijesinha
Department of Computer & Information Sciences, Towson University
Towson MD, 21252, USA
{reyer, rkarne, awijesinha}@towson.edu

Abstract

Bare Machine Computing (BMC) software runs directly on computer hardware without any operating system. This approach produces applications that are small, fast, and easily secured. We describe a mechanism for secure delivery of BMC software using paired hardware (USB) devices. The mechanism involves wrapping every application deliverable first with AES encryption, where the key is derived from a passphrase supplied by the software user/owner. This encrypted deliverable is then re-encrypted using RSA where the key is generated by the software developer and used only for deliverables to that specific user. Finally, the doubly encrypted software is placed on a normal USB drive that is paired with a second secure USB device that contains the bare application loader and user specific decryption software. Simulation studies using various application code sizes and RSA/AES key sizes show good performance on an ordinary desktop. Military and other high security environments can benefit from such a delivery scheme since prior attempts at secure software delivery failed to meet the highest FIPS standards.

1 Introduction

Bare Machine Computing (BMC) is a programming paradigm that removes the need for any conventional operating system (OS) by providing application objects that can be loaded and run on bare computer hardware. By removing the need for an OS, BMC developers are able to create applications that are very small, extremely fast, and far more secure than their OS based counterparts. While removal of the OS improves security by reducing the collective size and complexity of executable code, it also requires that BMC developers implement application security without resorting to OS services.

BMC security protocols, like BMC applications, follow a paradigm of self-containment; all functionality is built-in, not added on. Secure delivery of BMC software requires that applications developed at the BMC lab can be delivered directly to trusted users (assumed here to also be the owner of the bare application). Since delivery is typically done through an insecure environment, it is necessary to provide protection mechanisms that prevent a listening or intercepting entity from reading and decrypting the application. A general need exists in the military and other government agencies, as well as high security corporate environments to protect proprietary software in a way that meets the highest levels of FIPS security. At present, BMC software including boot code, loader and application are contained on a single USB device. However, this means that anyone who acquires the USB can boot and run the BMC application. At a minimum, a methodology for secure delivery of BMC software must:

1. Provide authentication to a user that the software is an authentic BMC application.
2. Authenticate the user to the software as a legitimate user of the BMC software.
3. Encrypt the application code and prevent reading of the code until it is decrypted and loaded to a BMC platform.

The issue of BMC software authentication is addressed by the BMC lab acting as a local certificate authority for the BMC domain. We briefly discuss the use of BMC certificates in Section 4. Once the BMC software has been authenticated, the remaining security issues with respect to distribution are addressed as follows.

First, each user authorized to run BMC applications is assigned a user specific RSA key pair. Unlike conventional usage of public and private keys for signatures and encryption, the RSA key pair is generated by (and known only to) the BMC lab. Furthermore, the key pair is unique to the given user. We refer to the individual half-keys in the key pair as the user's and BMC lab's RSA keys (these half-keys are not used in the usual way as public and private keys). In the context of software distribution, no keys are ever shared directly with the user.

To obtain the BMC software and be able to run the application, the user first provides a passphrase to the BMC lab in a physically secure manner or using a secure channel (discussion of this issue is not within the scope of this paper). The passphrase is used by the BMC lab to generate a

unique AES-256 key for the user following the guidelines in [13]. Each user receives a USB device, which contains the BMC boot code and loader, and the user's RSA key encrypted with the user's AES key. When the user enters the passphrase, the BMC loader code automatically generates the user's AES key, which enables the user's RSA key to be decrypted.

All deliverable BMC application code is first encrypted in the BMC lab with the user's passphrase-based AES key. The application code is then re-encrypted using the BMC lab's RSA key. This doubly-encrypted executable is then loaded onto an ordinary USB flash drive. Each new user is initially provided with a USB decryption device that contains their user-specific RSA key (encrypted) used to decrypt multiple bare application deliverables.

These USB decryption devices can be reprogrammed, if it becomes necessary to change the RSA key pair. A boot/decryption device is similar to a secure hardware token. As such, it must be physically secured to prevent unauthorized use of a bare application in the event of passphrase compromise.

The remainder of this paper is organized as follows:

- Section 2 discusses related work dealing with BMC applications, other approaches to secure software delivery, and TLS security concerns.

- Section 3 provides details of the secure delivery scheme proposed in this paper.

- Section 4 discusses how the proposed delivery scheme provides authentication and protection to deliverables.

- Section 5 details the test simulations and results from proof-of-concept testing of the proposed delivery scheme.

- Section 6 contains the conclusion.

2 Related Work

The application-oriented object architecture forming the basis for the BMC paradigm was first described in [1]. The BMC paradigm serves as the basis for developing completely self-contained bare applications that run on the hardware, and perform memory management, task scheduling, and I/O without the need for any OS or kernel [6][7]. Numerous non-trivial BMC applications have been implemented, including a VoIP softphone [2], web server [3], email server [4], and a SQL database engine [5]. The BMC lab has developed a USB driver allowing BMC applications to securely access USB mass storage devices for application delivery and installation [8][9]. An approach for developing bare applications is described in [11]. It is also possible to convert (port) certain OS-based applications to bare applications [10]. The original implementation of TLS on a bare web server is described in [12]. However, TLS by itself, is insufficient to enable secure delivery of BMC software to users.

There are many studies on software protection. In [22], software integrity protection techniques are classified by considering system, attack and defense views. In [23], an approach to anti-tampering that uses a trusted remote server is described.

The TLS protocol is widely used for the secure transfer of data including software. RSA-PKCS#1 v1.5 still dominates

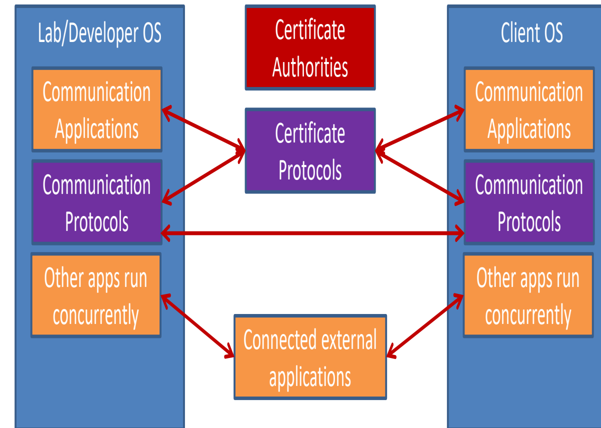

Figure 1a. Attack profile for non-BMC communication applications.

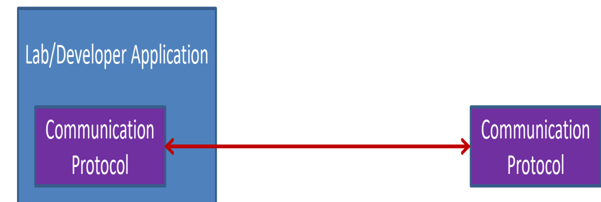

Figure 1b. Attack profile for BMC communication applications.

the SSL/TLS key exchange implementations [15]. Security concerns with TLS include vulnerabilities relating to CAs [16][17][20], and attacks on the session key [18][14]. While TLS 1.3 addresses many issues with TLS 1.2, some attacks are still possible [15]. Elliptic curve cryptography (ECC) is an alternative to RSA having equivalent or better security with smaller key sizes [19].

3 Secure Software Delivery

Secure delivery of trusted software to untrusted users across insecure environments is a major concern for developers. Interception of deliverables impacts both users and BMC developers. Users are concerned for the security of their applications and data, and BMC developers for unauthorized use of their software. For example, users need assurance that purchased software is coming from a legitimate source, and developers need to have their intellectual rights protected. Because of this need for secure software delivery, numerous (often complicated) encryption

schemes have been developed, usually depending on OS security services or peripheral hardware like smart cards to enable protections.

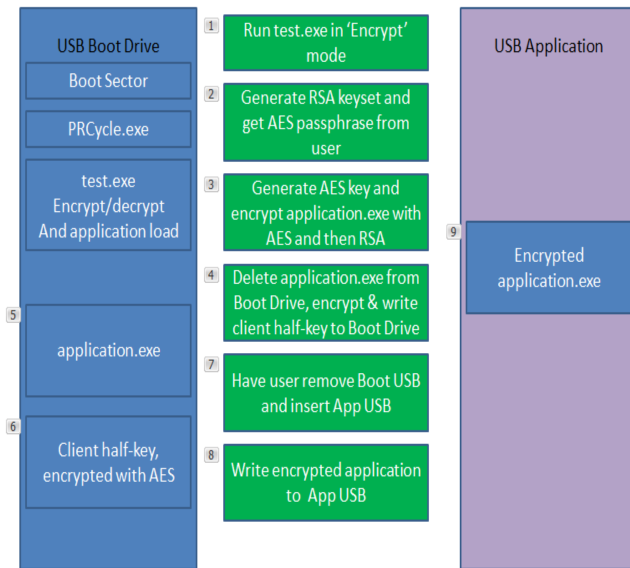

Figure 2. Encrypt mode.

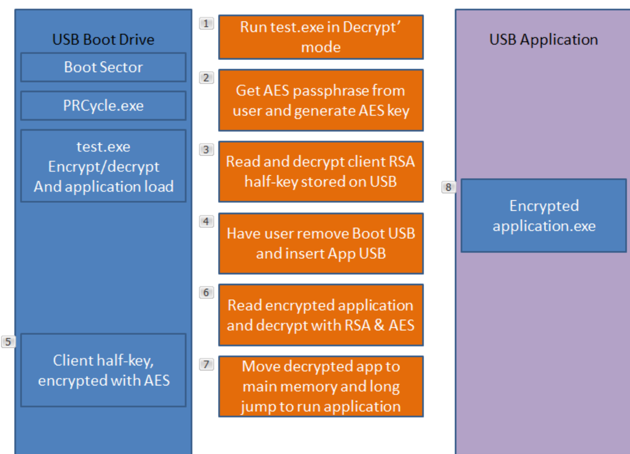

Figure 3. Decrypt mode.

The BMC programming paradigm requires that all software security (including cryptographic capabilities) be built-in to the application object or application loader that is part of the BMC boot code. We define and implement a secure method of software delivery that insures that BMC applications cannot be stolen or reproduced by unauthorized parties. While many of the goals of secure software delivery are the same for BMC applications and OS-based applications, the approaches and expected results are vastly different. Any OS based application must be viewed (and secured) as part of the larger, OS based system. Regardless of how well such application software is secured against piracy, security holes in the underlying OS can open new routes to unauthorized replication. Application developers have no control over OS security, and to a lesser degree OS security can be compromised by running insecure applications. Each component of an OS-based system is, at least, partially dependent on the security of every other component that is running concurrently on the system [21].

For BMC applications, the only code running on the system is the application object, plus the residue of the boot and loader code. Both of these items are within the control of the BMC application programmer, allowing for the development of mechanisms that cannot be weakened by external software components. Figures 1a and 1b compare communication between BMC versus non-BMC (OS-based) applications.

Secure software delivery always involves some form of cryptographic protection to provide authentication, and prevent unauthorized replication. With OS based systems, cryptographic capabilities are usually available as system services. However, since BMC applications load directly to computer hardware, using only the BIOS and BMC loader code, any cryptographic capabilities associated with software deliverables needs to be directly supported by the BMC loader. The secure software delivery mechanism for BMC was developed to take advantage of the secure laboratory environment where BMC applications are presently created. This mechanism needs to be extended in future for scalability.

BMC development computers are not networked beyond the lab itself and a non-networked machine has been designated as a single source for secure application release of BMC deliverables. On this single machine, an encryption database resides that contains all releasable BMC applications, and a table containing BMC user records which include, (for each user), (1) A secret passphrase selected by the user; (2) an AES key (256 bits) generated from the user's secret passphrase; and (3) a 2048-bit RSA key-pair generated on the BMC release workstation. As noted earlier, these RSA keysets are not shared directly with the associated users, but are used as part of the delivery process.

The secure software delivery mechanism involves the following steps:

1. Each application user has a record created in the BMC user database where their AES passphrase, AES key, and RSA keys are stored
2. The BMC Boot Code and Application Loader-Decryptor Code are written to low sectors of the USB Boot Drive along with the user's RSA key which was encrypted with the user AES key
3. The USB Boot Drive is delivered to each user by some (out of band) secure delivery method, and is to be used with any BMC applications delivered separately on standard USB flash drives. The user will confirm receipt of the USB Boot Drive (via an out of band

secure channel or method) before any BMC applications are forwarded

4. As noted earlier, every BMC application delivered to a BMC user will be loaded on a standard USB flash drive after first being encrypted using the AES key generated from the user passphrase and then re-encrypted using the BMC lab's RSA key associated with that particular user. The user's RSA key is also encrypted with the AES key and stored on the flash drive
5. Whenever the user loads a BMC application, the machine is cold-booted with the USB Boot Drive inserted. The boot code then installs the BMC application loader/decryptor which will ask the user to enter their secret passphrase
6. The application loader then decrypts the user's RSA key and asks the user to insert a USB drive containing an encrypted BMC application
7. The encrypted application is read and decrypted by the loader code (using the decrypted RSA key and AES key) and written to main memory
8. The BMC loader code now jumps to the newly decrypted code in main memory and the application is launched.

Figures 2 and 3 show the contents of the USBs (on left and right) and the sequence of operations (in the middle) during encrypt and decrypt respectively.

4 Discussion

Authentication issues are partially addressed by using the proprietary USB devices, which may be viewed as a form of "two-factor" protection. As discussed previously, protection against counterfeit boot code and application code requires that these be signed by the BMC lab acting as a certificate authority (this signature must be verified in the usual manner). An encrypted BMC application can only be decrypted if the decryption agent possesses the user's RSA key and the user's AES key. Since the AES key is not stored on either device, only the legitimate user with knowledge of the passphrase can decrypt the application. Conversely, if the application is successfully decrypted, the boot code has the lab's RSA key. Thus, we have authenticated the application to the user and the user to the application.

Protection against software interception is provided by the double encryption scheme. This will make it harder for an attacker to decrypt the software by brute-force. No key information resides on the application drive along with the encrypted executable. Additionally, on the user's bare machine, the decrypted code is never exposed except during BMC run-time. The encrypted executable is read, decrypted, and copied directly to main memory, without any external write to persistent storage.

The double encryption scheme for deliverables was selected for 3 reasons:

1. While all encryption keys are known at some point to the BMC lab, the AES component belongs primarily to individual users and is changeable at their discretion with simple notice to the BMC lab.
2. A single layer of encryption represents a single point of failure if the relevant key is compromised. Any RSA key could be attacked directly using known attack patterns. Given the security issues noted in Section 2, this is a valid concern.
3. Two-layer encryption provides developers with expanded choices on the RSA key size used to encrypt executables. Since any brute-force attempt to decrypt an executable would need to find both the AES and RSA keys, successful decryption must try each AES key with every possible RSA key. The search space is the product of all valid AES keys and all valid RSA keys. As such, a developer could use a combination of AES-256/RSA-1024 and still exceed the cipher security of a 2048 bit RSA key. An RSA key size of 4,096 is considered roughly equivalent to a symmetric key size of 148 bits. Combining AES-256 with a 1.024 bit RSA key far exceeds the strength of even the highest available RSA key sizes.

The USB boot and application devices are currently written to standard 2 gigabyte USB flash drives. Future research could investigate adding support for USB hardware based decryption and providing additional security in the form of masked storage (and a security housing that grounds/deletes ROM memory when breached). The cost of this hardware development would be significant and the tradeoffs would need to be considered. Future versions of the secure delivery software could include ECC to take advantage of the smaller key sizes and associated improvements to execution time and memory usage.

We now discuss the issue of transferring BMC software from the BMC lab to a trusted BMC user/owner of a BMC application (referred to from now on as the BMC user or simply the user). The requirement is that the user has 1) received a USB drive containing BMC boot code/loader and the user's RSA half key encrypted with the user's AES key; and 2) the BMC application code encrypted first with the user's AES key and then with the BMC lab's RSA half key for this user.

As noted earlier, we assume that:

a) The user's passphrase, AES key (derived from the passphrase), and RSA key pair (BMC lab and user half keys for this user) are known to the BMC lab. This information is stored on an isolated physically secured non-networked machine in the BMC lab that can only be accessed with the necessary administrative privileges. If this machine (or the information it stores) were to be compromised, the secure

delivery mechanism for BMC software as proposed in this paper fails.

b) The USB containing the boot code/loader and user's encrypted RSA half key were delivered to the user by some secure means. This USB must be physically secured to prevent tampering or unauthorized use.

Next, we consider secure delivery of the (doubly encrypted) BMC application code. If an ordinary USB containing this code cannot be physically delivered by some secure means to the user, one option would be to transfer the encrypted code over a secure Internet connection between the BMC lab and the user's network. The user could then create the USB containing the application code.

The secure Internet connection in this case could use TLS with mutual authentication. However, this requires that the secured USB containing the boot code/loader also include a BMC TLS client application. When the user boots and enters its correct passphrase, the TLS bare client will be launched.

During the TLS handshake, the BMC server in the lab will send its X.509 (RSA/SHA256) certificate, which would be verified by the BMC client. As usual, the BMC client will send the 48-byte random premaster secret encrypted with the BMC server's public RSA key (this key will be included in the secured USB containing the boot code/loader). Only the BMC server knows the corresponding private RSA key needed to decrypt this premaster secret. In lieu of sending its certificate, the BMC client will authenticate itself to the server by encrypting a nonce generated by the BMC server with the correct AES key (generated in turn when the user enters the correct passphrase). After the TLS handshake is successfully completed, the server can transfer the doubly encrypted application code (as TLS application data) over the TLS connection.

An alternative option, which is less secure, is to use an ordinary OS-based browser instead of the BMC client. This means that the browser must be able to verify the BMC server's certificate, which should be signed by a CA. This requires a trusted OS-based browser and a trusted CA. As discussed earlier, OS-based browsers and CA certificates can be controlled by attackers.

5 Software Simulations

A major factor affecting performance of the software is the overhead due to RSA operations. A number of RSA software implementations were tested, and the fastest were those that used the Chinese Remainder Theorem [24] to optimize RSA encryption. The algorithms typically used to generate RSA keysets always chose a small exponent for the public key to reduce encryption time. The multiplicative inverse exponent that is part of the private key is far more complex and needs to be more highly optimized.

Encryption elements of the BMC load and delivery protocol have been tested across a range of possible RSA and AES key sizes. Our implementation can generate and use 16 different RSA key sizes from 512 to 4,096 bits and all possible AES key sizes (128, 192, and 256 bits). We selected 1,024 bit RSA encryption combined with AES-256 to provide decrypt operations that were reasonably fast, but which still provided overall security exceeding the highest stand-alone RSA encryption.

Tests were run using a range of application sizes and, on average, decryption with this combination of RSA and AES required 360 msec/kB of original application size on a Dell OptiPlex 960. RSA encryption in our case involves a scheme where all keys are held as private, so the normal encryption/decryption pattern can be reversed without compromising security. In this protocol the application is first encrypted using the more complex key (most 1s set) and later decrypted using the shorter key. Initial encryption will always be performed in the BMC lab and does not need to be fast. The decryption process, on the other hand, needs to be fast since it is part of a user's load process.

These tests were designed to prove that our approach using RSA/AES encryption provides a high level of security with reasonable encryption/decryption times for all current BMC applications. This enables us to avoid techniques that compromise security such as the intentional selection of Fermat Primes to improve encryption speed. While the decryption time of 360 msec/kB may seem slow, it is noteworthy that these test decryptions included IO read/write time, and full data verification routines. Additionally, our proof-of-concept testing did not use multi-processor or any built-in processor hardware acceleration. Finally, as BMC applications are very small and this scheme only requires decryption once at the start of an application or server process, timing is not of paramount importance. A companion project has been proposed to utilize on-USB hardware acceleration that would improve decryption speeds by 100 times, allowing for large executables.

6 Conclusion

We described a mechanism for secure delivery of BMC software, which requires a pair of USBs containing the boot code and application code respectively. Each user has a proprietary RSA key pair, where only the BMC developer has access to either side of the key pair. This approach enables the encryption strength of RSA to be fully realized, and when coupled with the AES encryption layer, to provide a high level of security for software distribution. Our software simulations of the software delivery mechanism suggest that implementations will provide strong encryption at speeds acceptable for user software loading while achieving secure delivery across a hostile environment.

References

[1] R. K. Karne, Object-oriented Computer Architectures for New Generation of Applications, Computer Architecture News, December 1995, Vol. 23, No. 5.

[2] G. H. Khaksari, A. L. Wijesinha, R. K. Karne, L. He, and S. Girumala, A Peer-to-Peer Bare PC VoIP Application, IEEE Consumer and Communications and Networking Conference (CCNC), Las Vegas, Nevada, January 2007.

[3] He, Karne, and Wijesinha, Design and performance of a bare PC web server, International Journal of Computers and their Applications, June 2008.

[4] G. Ford, R. Karne, A. Wijesinha, and P. Appiah-Kubi, The Design and Implementation of a Bare PC Email Server, 33rd IEEE International Computer Software and Applications Conference (COMPSAC), Seattle, Washington, July 2009, pp. 480-485.

[5] U. Okafor, R. K. Karne, A. L. Wijesinha and B. Rawal, Transforming SQLITE to Run on a Bare PC, 7th International Conference on Software Paradigm Trends (ICSoft), pages 311-314, Rome, Italy, July 2012.

[6] H. Chang, R. Karne and A. Wijesinha, Insight Into the x86-64 Bare PC Application Boot/Load/Run Methodology, 22nd International Conference on Software Engineering and Data Engineering (SEDE), Los Angeles, CA, 2013.

[7] U. Okafor, R. Karne, A. Wijesinha, and P. Appiah-Kubi, Eliminating the Operating System via the Bare Machine Computing Paradigm, 5th International Conference on Future Computational Technologies and Applications (Future Computing), Valencia, Spain, 2013.

[8] R. K. Karne, A. L. Wijesinha, and S. Liang, A Bare PC Mass Storage USB Driver, International Journal of Computers and their Applications, March 2013.

[9] S. Liang, R. K. Karne, and A. L. Wijesinha., A Lean USB File System For Bare Machine Applications, 21st International Conference on Software Engineering and Data Engineering (SEDE), June 2012, pp. 191-196.

[10] U. Okafor, R. Karne, A. Wijesinha, and P. Appiah-Kubi, A Methodology to Transform an OS-based Application to a Bare Machine Application, 12th IEEE International Conference on Ubiquitous Computing and Communications (IUCC), Melbourne, Australia, 2013.

[11] G. H. Khaksari, R. K. Karne and A. L. Wijesinha, A Bare Machine Application Development Methodology, International Journal of Computers and their Applications, Vol. 19, No.1, March 2012, pp. 10-25.

[12] A. Emdadi, R. K. Karne, and A. L. Wijesinha. Implementing the TLS Protocol on a Bare PC, 2nd International Conference on Computer Research and Development (ICCRD), Kuala Lumpur, Malaysia, May 2010.

[13] K. Moriarty, B. Kaliski, and A. Rusch, PKCS #5: Password-Based Cryptography Specification Version 2.1, RFC 8018, 2017.

[14] K. Böttinger, D. Schuster, and C. Eckert, Detecting Fingerprinted Data in TLS Traffic, 10th ACM Symposium on Information, Computer and Communications Security (ASIA CCS), April 2015.

[15] T. Jager, J. Schwenk, and J. Somorovsky, On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption, 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS), October 2015.

[16] A. Bates, J. Pletcher, T. Nichols, B. Hollembaek, D. Tian, K. R. B. Butler, and A. Alkhelaifi, Securing SSL Certificate Verification through Dynamic Linking, ACM SIGSAC Conference on Computer and Communications Security (CCS), November 2014.

[17] A. Bates, J. Pletcher, T. Nichols, B. Hollembaek, and K. R. B. Butler, Forced Perspectives: Evaluating an SSL Trust Enhancement at Scale, Conference on Internet Measurement (IMC), November 2014.

[18] F. Giesen, F. Kohlar, and D. Stebila, On the security of TLS renegotiation, ACM SIGSAC conference on Computer & Communications Security (CCS), November 2013.

[19] Federal Office for Information Security, Technical Guideline TR-03111 Elliptic Curve Cryptography Version 2.0, Germany, 2012.

[20] N. Vratonjic, J. Freudiger, V. Bindschaedler, and J.P.Hubaux, The Inconvenient Truth About Web Certificates, In B. Schneier, editor, Economics of Information Security and Privacy III, pages 79-117. Springer New York, 2013.

[21] R. Eyer, R. Karne, and A. Wijesinha, Isolation as a Threat Reduction Strategy for Super-Systems, 22nd International Conference on Computers and Their Applications in Industry and Engineering (CAINE), 2009.

[22] M. Ahmadvand, A. Pretschner, and F. Kelbert, A Taxonomy of Software Integrity Protection Techniques, Advances in Computers, ISSN 0065-2458, Feb 2018.

[23] A. Viticchie, C. Basile, M. Ceccato, B. Abrath, and B. Coppens, Reactive Attestation: Automatic Detection and Reaction to Software Tampering Attacks, (SPRO), ACM Workshop on Software Protection, 2016.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, An Introduction to Algorithms, 3rd ed MIT, 2009.