SGXCrypter: IP Protection for Portable Executables using Intel's SGX Technology

Dimitrios Tychalas[1], Nektarios Georgios Tsoutsos[2], and Michail Maniatakos[1]

[1]New York University Abu Dhabi, Abu Dhabi, UAE
[2]NYU Tandon School of Engineering, New York, USA

Abstract— Executable packing schemes are popular for obfuscating the binary code of a target program through compression or encryption, and can be leveraged for protecting proprietary code against analysis and reverse engineering. Although achieving their confidentiality objective, packed executables are prepended with decryption or decompression code that processes the rest of the binary, which is a lucrative target for reverse-engineering attackers. To thwart such attacks, we introduce a novel packing scheme called SGXCrypter, which utilizes Intel's novel Software Guard Extensions to securely unpack and execute Windows binaries. Unlike state-of-the-art crypters, SGXCrypter's code is never flagged as malicious against 35 popular antivirus engines, minimally increasing the loading time of the protected executable by an average of 0.6 seconds per MB.

I. Introduction

The primary objective of reverse engineering is obtaining the original specifications and functionality of a system, after analysing an adequate number of instantiations [1]. In software, the concept of reverse engineering can have desirable effects, such as recovery of design patterns and architectural components or translation to another language. This technique, however, can also impose dangerous consequences, as it is a threat to intellectual property (IP) of software authors. As shown in a recent study, piracy has always been a threat to software manufacturers, resulting in hundreds of billions loss in revenue [2]. In addition, decomposing a program to analyse how it works allows sophisticated attackers to hunt for exploitable bugs in the code, facilitating the creation of various malware and exploits. To prevent this, several anti-reverse engineering strategies have been proposed, such as code obfuscation and self-modifying code [3, 4], as well as anti-debugging protections [5].

A very popular strategy to prevent reverse engineering of software binaries is to use file encoding or encryption, in order to *pack* the executable in incomprehensible form. Typically, executable packers compress or encrypt the target binary, and prepend a small piece of code (called an *unpacking stub*) to handle this decompression/decryption at runtime. Essentially, the entry point of the executable is transferred to the unpacking stub, and the actual program gains control after its unpacking, behaving identically to the original. A packer using encryption as its protection strategy is called more appropriately a *crypter*.

In recent years, packing executables has become a standard antivirus (AV) evasion technique for malware authors as well. Indeed, the principles of obstruction and discouragement that binary encryption imposes on potential software pirates are also applicable to malicious users against AV detection [6]. This creates a constant challenge for cyber-security engineers to develop new reverse engineering methodologies to uncover malware threats protected with packing schemes [7]. At the same time, packer/crypter developers strive to strengthen their schemes and avoid known pitfalls.

From a cyber-security perspective, the most sensitive part of any packing scheme is the unpacking stub, which handles the decryption of the binary using a secret key. This stub is the primary target for the reverse engineering process [8]. The stub code is unencrypted and is vulnerable to static analysis [9], which can reveal the deployed compression and encryption libraries and, more importantly, the secret key. Existing packing techniques detach the secret key from the unpacking stub, mixing it with the encryption code, to effectively hide it [10]. Nevertheless, any method of obfuscating static data will eventually fail during run-time (dynamic) analysis, where a reversing engineer can execute the packed program in a controlled environment (e.g., inside a virtual machine) and analyze its flow and data [11].

In this work, we propose a novel crypter design called SGXCrypter, which leverages the new Software Guard Extensions (SGX) of Intel processors, to protect the IP of Windows binaries, while enabling controlled access to them. To the best of our knowledge, this is the first crypter leveraging the architectural features of commodity processors, which renders the unpacking process untraceable by both static and dynamic analysis. In addition, SGXCrypter is friendly to existing AV engines, as it does not trigger any false positives. In more details, we claim the following contributions:

- Protect the IP of any Windows Portable Executable (PE) against static analysis with a novel packing strategy based on cross-layer security, utilizing hardware primitives to protect software execution.
- Control access to the packed binary through SGX attestation, while preventing any key extraction efforts.
- Full compatibility with popular AV engines, which facilitates seamless deployment of our approach.

II. Related Work

In this section, we discuss different approaches towards thwarting reverse engineering attacks, that share similarities with our own protection strategy. First, we elaborate on two alternatives that detach the payload decryption key from the decryption algorithm itself; this is a key concept that is also followed in the architecture of SGXCrypter. Then, we summarize three additional crypter implementations, which we also use in our experiments.

Hyperion is a PE crypter developed by Christian Ammann in 2012 [12]. Its main components are the crypter and the container. The crypter receives an executable, it calculates a checksum for it and encrypts both the binary and the checksum with AES-128. It then creates a new executable file with the encrypted image of the binary and the container in the `.data` section. The new executable contains the unencrypted checksum but not the key used by the crypter, thwarting any static analysis attacks. At runtime, the container essentially brute-forces the key by calculating the checksum of every decryption output and comparing it to the original. Once this procedure produces the actual key, the code is decrypted and the execution jumps to the entry point of the original executable. By eliminating static analysis and rendering run-time debugging a tedious, albeit not impossible, endeavor, Hyperion proves itself as a very secure implementation. It suffers though from significant performance overhead, attributed to the time-consuming method of obtaining the decryption key.

White Box Cryptography is an alternative method to prevent reverse engineering [13]. This methodology focuses on producing cryptographic primitives that aim to be secure in white-box attack context, meaning the attacker has complete access to the underlying encryption scheme. The main idea is to hide the key inside the code by randomizing the decrypting method itself. In an AES implementation, the main algorithm is transformed in a series of look-up tables (LUTs), in which the key is hard-coded, and additional LUTs containing random encodings. These additional LUTs negate each other leaving the original functionality intact, but effectively randomizing the decryption process. Both static and dynamic analysis are successfully hindered with this approach. This method, although secure, results in slow and bulky code, much like Hyperion, shortening its applicability.

Tested Crypters: Aegis is a C/C++ crypter that promises full undetectability, while offering protection against analysis using a sandbox/virtual machine environment, and customization of the decryption stub [14]. Likewise, Yoda's Protector is an open-source crypter, originally developed in 2004, offering polymorphic encryption as well as payload compression [15]. Finally, *CrypterBinder* is a recently developed crypter that enables the users to choose their preferred encryption scheme, and can also be used to bind and encrypt more than one executables [16].

III. Preliminaries

A. Intel Software Guard Extensions (SGX)

In 2015, Intel introduced Skylake, its latest microarchitecture, that introduces a new set of instructions for secure computation called Software Guard Extensions. The centerpiece of this technology is a container called *Enclave*, which is an application in the form of a `.dll`, initialized and managed using the newly added instructions. These instructions always run in user mode, although they resemble privileged instructions such as system calls. This approach protects the Enclave against a compromised Operating System, Hypervisor, or even the System Management Engine, the highest privilege level software that handles power management and system hardware control.

Privacy and protection are granted to the Enclave by isolating its memory space, the Enclave Page Cache (EPC). During boot time, the BIOS binds a portion of the DRAM using the content of specific registers. This memory space is accessed only by Enclaves, denying any other access, be it from system software or through DMA, with the help of the memory controller. Enclave data are encrypted and enter or exit the processor via a hardware cryptographic primitive embedded on the CPU chip [17].

SGX also offers a mechanism for secure exchange of secrets in the form of local and remote attestation [18]. Specifically, when an Enclave is loaded, a cryptographic hash of its contents is computed. This authentication information is then used to convince any remote or local party that they are communicating with an Enclave with a specific measurement, running within an SGX environment. As soon as a secure communication has been established, the two parties can securely exchange sensitive information.

B. Microsoft PE Files

In this work, our focus is the protection and manipulation of Microsoft portable executable (PE) `.exe` files[1], so it is important to introduce the basic organization of this file format. The most important characteristic of PE files is that they are portable across all Windows operating systems versions. The PE format is also used in 32-bit `.dll`s, object files and Windows NT device drivers, and it is generated by the Microsoft linker.

The most important part of the PE format is its header structure. Any 32-bit PE begins with a DOS header, which is a legacy feature of this file format for backwards compatibility, and a DOS stub to contain an MS-DOS compatible application. The header structure is identified by the magic value `4D5A` (or `MZ` in ASCII) and contains a relative offset to the PE header. The PE header is the centerpiece of this format, containing a collection of fields that define what the rest of the file looks like. Relevant information includes the location and size of the code as well as the size of its section table.

[1] At the time of writing, an Intel SGX software development kit was only available for the Windows operating system.

Immediately after the PE header, the file structure stores a table of section headers, which describes each subsection of the PE file, followed by the actual machine code, data, resources as well as other portable executables. Typically, the `.text` subsection is used for the code, the `.data` for initialized global and static data, the `.idata` contains information about imported functions, while the `.rsrc` may include various resources such as other executables. Figure 1 presents a hierarchical representation of the PE format [19].

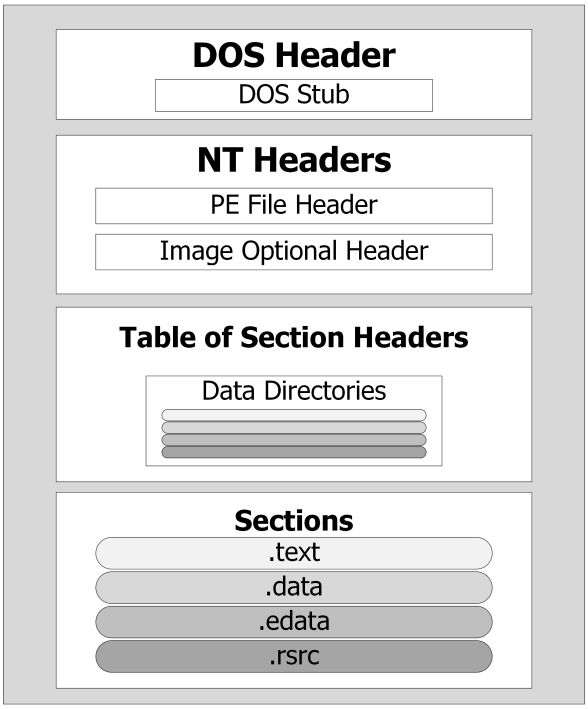

Fig. 1. High level structure of a Portable Executable file.

IV. SGXCrypter Architecture Overview

The basic architecture of the SGXCrypter consists of an encryption engine and a decryption stub. The primary objective of the encryption engine is to protect the IP of a given PE file using a strong cryptographic algorithm. Likewise, the decryption stub reverses the cryptographic protections, and handles memory loading and execution of the original PE. For our SGXCrypter implementation, we extended the baseline crypter from [20] to add support for SGX and Windows API process context initialization, replacing the hard-coded decryption key protection functionality. SGXCrypter is implemented in C/C++ using Visual Studio 2012, which supports development of SGX Enclave applications.

A. Comprehensive Protection Strategy

Payload Encryption: The first step of our packing process is encryption of the PE file payload. This step is executed offline by the original program author and uses our encryption engine. In our approach, the encryption engine is also a Windows executable that receives the PE payload as its command line argument and uses AES-256 to encrypt its contents as raw 16-byte blocks. In case the byte size of the input is not a multiple of 16, our encryption engine also handles all necessary padding. Internally, the encryption engine uses cryptographic libraries selected to minimize the memory footprint to the OS. At the end of this step, the encrypted payload can be combined with our decryption stub.

Decryption Stub: The second step of the packing process combines the encrypted PE payload with our decryption stub. A straightforward option to combine the stub and the payload would be to build a header file containing the decryption stub and use a `constructor` attribute in C, as the latter enables execution of a function outside the standard program flow (before calling the `main` function). This allows on-the-fly decryption and payload execution (oblivious to the various executable file formats), offering cross-platform compatibility of the solution. Nevertheless, Microsoft C/C++ compiler prohibits any control flow altering attributes or `pragmas`, so our decryption stub can leverage the Windows API instead. Given this limitation, our SGXCrypter is designed to include the encrypted payload within the decryption stub, which is then compiled as a standalone application. Specifically, our payload is added as a resource in the `.rsrc` section of the stub, using a typical resource management solution such as *Resource Tuner* [21]. At this point, the stub containing the encrypted payload is ready for distribution.

Stub Execution: After distribution, our decryption stub can be executed on any SGX-enabled machine. Initially, the stub locates the added resource (i.e., the encrypted payload) using its name and extension (both set up by resource management). After fetching the byte size of the resource and a pointer to its first byte, the encrypted payload is copied to SGX-protected memory. At this point, the decryption process can start within the SGX Enclave. The IP of the payload is protected with the run-time isolation offered by the SGX architecture utilizing decryption libraries mirroring the ones used by our encryption engine. It is also possible to call Enclave functions from the main application, by declaring them in a separate file with the help of Intel's Enclave Definition Language (EDL); the latter is used to describe the function attributes, such as input and output arguments, argument size, etc. An `.edl` file is also included in the main application, so it is possible to build wrapper functions and directly interface with the Enclave.

The Enclave communicates with our decryption stub using the protected Enclave Page Cache (EPC) memory. Specifically, input arguments are written into a buffer and copied to EPC, while output arguments are transferred from the EPC to the application memory space, updating the old argument value without using a return function. In SGXCrypter, payload decryption entails copying 16-byte encrypted strings to EPC, decrypting them within the Enclave and then updating their former value in the main application with the decrypted bytes.

Remote Authentication: Since the decryption key is not present in the Enclave before its creation, it must be provided either by the main application (i.e., the decryption stub) or by a remote service provider. Consistent with our access control requirements, SGXCrypter uses the latter, and our Enclave communicates with an authentication provider via remote attestation. At first, the two parties establish a secure communication channel, and the Enclave establishes itself as a trustworthy application using an RSA signed measurement of its code and data. As soon as the attestation is successful and the enclave is authenticated against the remote database, the remote party securely dispatches the decryption key.

Payload Execution: The last step of this procedure is executing the original payload. Since its decrypted binary code is not a part of the stub main function, it must be executed as an independent process. Indeed, this is possible using the Windows API, which allows to first build a DOS header structure for our decrypted payload, using a pointer to its first byte. As soon as the DOS header is created, our payload is recognized as a valid executable. Likewise, the PE header and the section headers are created at the correct offsets, followed by the actual sections of the executable. This allows initialization of all process related information (e.g., context data, security attributes, among others), and a process for our payload can be created. Before execution of the payload begins using the `ResumeThread` function, the memory protection attributes are updated in the PE header to "read/write/execute", and the thread context is set as well. A block diagram describing the lifecycle of a payload is illustrated in Figure 2.

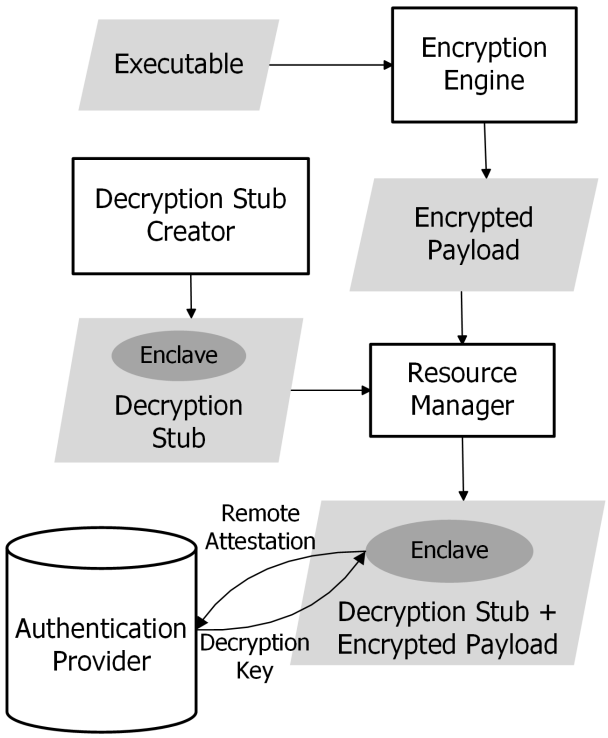

Fig. 2. SGXCrypter block diagram

B. Security Assessment of SGXCrypter

In this work, our primary objective is to safeguard the IP of an executable against static analysis techniques; for that matter, we developed a crypter that can be applied to any precompiled `.exe` payload, which is securely decrypted within an SGX Enclave. In theory, any program can be compiled and deployed within an SGX Enclave (as if it were a native SGX application), to be protected against runtime attacks. In practice, however, the SGX Software Development Kit does not allow Windows API function calls, such as the ones used in the decryption stub, from inside an Enclave [22]. Moreover, the maximum protected memory (EPC) is limited to 128MB [22], so the performance of any native SGX application could be impacted due to expensive EPC page swapping. Furthermore, redesigning a regular application to its SGX counterpart would impose an extra design cost in addition to requiring access to the original source code (i.e., existing PE files need recompilation).

An important benefit of our SGX-based approach is protection against static and dynamic analysis of the developed decryption stub. Indeed, before the decryption stub is engaged, the confidentiality of our PE payload is protected using strong AES-256 encryption; this enables protection of the payload during storage. Since breaking the symmetric encryption is assumed to be an intractable problem, all attack efforts would be directed to the decryption stub and the extraction of the decryption key.

Static Analysis Protection: Decryption key extraction via static analysis of an executable is usually thwarted using heuristic methods, such as hiding the key within the code section, fragmenting it in memory and reassembling it during execution, by exploiting more sophisticated methods such as steganography [23]. Given file access and enough time, however, an attacker can extract any key stored within a PE file, regardless of the obscurity & sophistication level hiding it or size of the PE file [24]. SGXCrypter, unlike state-of-the-art crypters, detaches the key from the executable, requiring it to be loaded by a remote entity, effectively protecting against static analysis[2].

Connection between the application and the service provider is secured in two steps. The first is remote attestation, where the Enclave authenticates itself to the remote entity. With SGX technology, this is achieved by deploying an anonymous signature scheme, called Enhanced Privacy ID (EPID) [25]. When the remote service provider acknowledges that it is communicating with an application running in a safe environment, secure communication is established to ensure privacy (second step). Specifically, the secure channel is protected with AES-128 encryption of the message sent, and the SIGMA secure session establishment protocol [26], which is an enhanced version of Diffie-Hellman key exchange.

[2]While all dynamic data are protected (including the keys), the SGX technology does not prevent application disassembly [22, p. 5].

Dynamic Analysis Protection: A major security guarantee offered by SGXCrypter is defending against run-time analysis during decryption. Without the latter, the aforementioned protection against static attacks will fail, if an adversary could access the running code and extract the key during decryption. Thus, our approach employs cross-layer security, utilizing newly added and modified hardware primitives residing in SGX CPUs to achieve truly isolated execution. Leveraging SGX specific registers that map every memory range loaded with Enclave data, the memory controller rejects all memory accesses to SGX data, except for those performed by the SGX instructions. Filtered accesses can be direct, from a peripheral like a PCIe device, or indirect by the OS or Hypervisor via virtual addressing.

Specifically, the Memory Management Unit, which handles address translation, holds EPT tables mapping virtual to physical addresses and ensures that the specific virtual addresses of EPC will be reserved during execution. In addition, every block of data that enters or exits the CPU is encrypted with the help of a hardware encryption primitive, dubbed Memory Encryption Engine (MEE) [27]. During execution, all data traffic between the CPU and main memory, is secured by the MEE with AES encryption and Message Authentication Codes, to ensure both confidentiality and integrity. Encryption keys are kept inside the MEE, inaccessible by software, and refreshed during boot or reset. Furthermore, debugging is disabled for all Enclave applications compiled in *release mode*, by clearing the relevant flag in the memory page holding the Enclave attributes (such as base address and length). Since external modification of the Enclave memory space is not possible, debugging as a means of run-time analysis is ineffective.

V. Experimental Evaluation

SGXCrypter experiments were conducted on a Dell Optiplex 5040 SFF desktop, with an i7-6700 CPU, 8 GB DRAM and a Windows 10 environment. Our benchmarks measured two metrics: *performance* and offered *AV evasion*. Performance is measured in terms of execution time, with the help of a custom batch script. Furthermore, we also investigate AV evasion, considering that detecting known AV samples in spite of crypter processing can be attributed to either weak encryption or insufficient protection. We tested three PE files: two innocuous programs (a simple AES-128 engine we developed and `DebugView` [28]), used as controls to verify compatibility, as well as a `Nivdort` malware sample taken from a vulnerability research database [29], to test for maximum AV detection. Our evasion tests were performed on July 7, 2016 using the AV engines from [30], and the results are presented in Table I.

The results indicate that our approach offers the highest AV compatibility, as all PE samples encrypted by SGXCrypter did not trigger any detection alarms, even when the packed executable was a registered malware. In comparison, the second best AV evasion performance in this benchmark came from `Yoda's Protector`, which generated nearly undetected executables for our control PEs, but the encrypted `Nivdort` sample was detected by one fourth of the tested AVs. Likewise, `CrypterBinder` demonstrated equivalent results to `Yoda's Protector`, but `Hyperion` and `Aegis` offered sub par protections. It is worth noting that SGXCrypter was also able to evade detection against all 35 AV engines in [30] after packing *eicar*, the universal AV testing sample[3].

TABLE I
Antivirus Detection Rate (From [30] on July 7, 2016)

Crypter	Benchmark		
	AES	**DbgView**	**Nivdort**
SGXCrypter	0/35	0/35	0/35
Yoda's	3/35	2/35	8/35
CrypterBinder	9/35	2/35	11/35
Aegis	15/35	10/35	15/35
Hyperion	21/35	21/35	20/35

TABLE II
Loading Overhead of Protected PEs Against Originals

Benchmark	SGXCrypter	Original
HelloWorld (7KB)	265ms	9ms
Hamming (7KB)	271ms	11ms
AES-128 (17KB)	297ms	12ms
SFX rar (62KB)	374ms	33ms
SFX rar (474KB)	502ms	53ms
GNU gcc (797KB)	861ms	67ms
SFX rar (4.8MB)	3686ms	118ms

The next set of experiments, summarized in Table II, measure the extra delay SGXCrypter imposes during application loading. The applications used for this benchmark are: (a) `HelloWorld`, (b) a fast hamming distance program, (c) an AES-128 encryption program, (d) three self-extracting (SFX) RAR archives (of different sizes) using [31], and (e) GNU `gcc` 4.9.3 from MinGW [32]. Our results show that SGXCrypter increases the total loading time depending on the size of the PE payload. With respect to scalability, for small executables ($<$ 1MB) the SGXCrypter overhead is barely noticeable to the user, with an average penalty of 0.5s. For larger payloads, the SGX decryption cost yields an overhead of about 600ms for each MB of the payload.

[3]This sample is not presented in the benchmarks, as all the other crypters were unable to process it, due to lack of DOS header; SGXCrypter does not have this limitation.

VI. Conclusion and Future Work

In this work we present SGXCrypter, a novel encryption based packing solution, which offers strong protection against reverse engineering techniques such as static analysis. SGX architectural features deter runtime analysis techniques, such as debugging, while our security architecture of decrypting within an Enclave and remotely receiving the decryption key ensures protection in storage. SGXCrypter outperforms its competition in terms of AV evasion while imposing minimal performance penalties.

At the time this work was developed, a Linux development environment for SGX was not available; arguably, a GNU C based implementation would simplify the unpacking process and render SGXCrypter a universal approach, regardless of file type. A native Linux implementation of our crypter will be explored in future work. Finally, as soon as Intel provides support for the Windows API from within Enclaves, our solution could be extended to not only decrypt PE files, but also execute them directly within the SGX environment, evolving into a versatile crypter solution.

Acknowledgments

This work was partially supported by the NYU Abu Dhabi Global Ph.D. Student Fellowship program.

Resources

The source code of SGXCrypter is available at https://github.com/momalab/SGXCrypter.

References

[1] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy," *IEEE software*, vol. 7, no. 1, pp. 13–17, 1990.

[2] B. Kerr, "Software Piracy Hampering U.S. Manufacturing." www.manufacturing.net/news/, 2014. [May 10, 2016].

[3] W. Gregory, *General Method of Program Code Obfuscation*. PhD thesis, Institute of Engineering Cybernetics, Wroclaw University of Technology, 2002.

[4] N. Mavrogiannopoulos, N. Kisserli, and B. Preneel, "A taxonomy of self-modifying code for obfuscation," *Computers & Security*, vol. 30, no. 8, pp. 679–691, 2011.

[5] M. N. Gagnon, S. Taylor, and A. K. Ghosh, "Software protection through anti-debugging," *IEEE Security & Privacy*, vol. 5, no. 3, pp. 82–84, 2007.

[6] T. Brosch and M. Morgenstern, "Runtime packers: The hidden problem," *Black Hat USA*, 2006. Available: www.av-test.org/ [April 25, 2016].

[7] K. Babar and F. Khalid, "Generic unpacking techniques," in *International Conference on Computer, Control and Communication*, pp. 1–6, IEEE, 2009.

[8] W. Yan, Z. Zhang, and N. Ansari, "Revealing packed malware," *IEEE Security & Privacy*, vol. 6, no. 5, pp. 65–69, 2008.

[9] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security & Privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[10] D. Glynos, "Packing heat!," 2012. Available: census-labs.com/ [May 5, 2016].

[11] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Computer and Communications Security (CCS)*, pp. 51–62, ACM, 2008.

[12] C. Ammann, "Hyperion: Implementation of a PE-Crypter," 2012. Available: nullsecurity.net/papers [May 20, 2016].

[13] S. Chow *et al.*, "A white-box DES implementation for DRM applications," in *ACM Workshop on Digital Rights Management*, pp. 1–15, Springer, 2002.

[14] Aegis Crypter, "The official website of Aegis Crypter," 2012. Available: www.aegiscrypter.com/ [June 30, 2016].

[15] "Yoda's Protector," 2004. Available: yodap.sourceforge.net/ [June 30, 2016].

[16] "CrypterBinder," 2016. Available: sourceforge.net/projects/crypterbinder [July 2, 2016].

[17] F. McKeen *et al.*, "Innovative instructions and software model for isolated execution.," in *Hardware and Architectural Support for Security and Privacy (HASP)*, p. 10, ACM, 2013.

[18] I. Anati *et al.*, "Innovative technology for CPU based attestation and sealing," in *Hardware and Architectural Support for Security and Privacy (HASP)*, vol. 13, ACM, 2013.

[19] M. Pietrek, "Peering inside the PE: a tour of the Win32 portable executable file format," *Microsoft Systems Journal-US Edition*, pp. 15–38, 1994.

[20] DrIdle, "Baseline crypter source code," 2014. Available: https://github.com/iGh0st/Crypters [June 13, 2016].

[21] H. Software, "Resource Tuner 2.05 - Visual Resource Editor." Available: www.restuner.com [June 10, 2016].

[22] Intel, "Software Guard Extensions SDK for Windows OS (Developer Reference)," 2016. Available: https://software.intel.com/sites/default/files/managed/b4/cf/Intel-SGX-SDK-Developer-Reference-for-Windows-OS.pdf [July 7, 2016].

[23] R. Ibrahim and T. S. Kuan, "Steganography algorithm to hide secret message inside an image," *arXiv preprint arXiv:1112.2809*, 2011.

[24] A. Shamir and N. Van Someren, "Playing 'hide and seek' with stored keys," in *International Conference on Financial Cryptography*, pp. 118–124, Springer, 1999.

[25] E. Brickell and J. Li, "Enhanced Privacy ID: A direct anonymous attestation scheme with enhanced revocation capabilities," in *Privacy in electronic society*, pp. 21–30, ACM, 2007.

[26] H. Krawczyk, "SIGMA: The 'SIGn-and-MAc' approach to authenticated Diffie-Hellman and its use in the IKE protocols," in *Annual International Cryptology Conference*, pp. 400–425, Springer, 2003.

[27] Intel, "Intel Software Guard Extensions (Programming Reference)," 2014. Available: https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf [April 25, 2016].

[28] M. Russinovich, "DebugView v4.81," 2012. Available: technet.microsoft.com/en-us/sysinternals/debugview [June 23, 2016].

[29] "theZoo aka Malware DB," 2014. Available: ytisf.github.io/theZoo/ [July 6, 2016].

[30] NoDistribute, "NoDistribute Antivirus Scanner," 2013. Available: nodistribute.com/ [July 7, 2016].

[31] "WinRAR archiver," 2016. Available: rarlab.com [July 2, 2016].

[32] "MinGW: Minimalist GNU for Windows," 2016. Available: mingw.org [July 2, 2016].