# Alohomora: Protecting Files from Ransomware Attacks Using Fine-Grained I/O Whitelisting

Sanggu Lee
*Seoul National University*

Yoona Kim
*Seoul National University*

Dusol Lee
*Seoul National University*

Inhyuk Choi
*Seoul National University*

Jihong Kim
*Seoul National University*

## ABSTRACT

We propose a novel whitelist-based anti-ransomware solution called alohomora. Alohomora is based on our observation that an I/O activity of an application can be an effective abstraction level for managing I/O whitelisting. In alohomora, when a write request is sent to an SSD, its program context value (which is supported by a host CPU register) is passed to the SSD. The SSD checks if the request was pre-approved using the program context value, thus preventing ransomware from modifying files in the SSD. Our experimental results using a prototype alohomora system show that alohomora can achieve a strong security level against sophisticated ransomware attacks without degrading I/O performance.

## 1 INTRODUCTION

Encrypting ransomware secretly encrypts files and demands a ransom for exchanging a secret key that can decrypt the encrypted files. When the encrypted files should be quickly recovered for an attacked system to properly operate, the only feasible solution might be to pay the ransom. For example, a major oil pipeline company, whose billing system was attacked by ransomware, paid about $4.4 million in exchange for a decryption tool in 2021 [1]. The potential high financial payoff from ransomware attacks made ransomware one of the most serious threats in cyber security.

In order to protect files from ransomware attacks, a typical anti-ransomware solution is based on two strategies: pre-detection of ransomware and post-recovery of an attacked system. A wide variety of techniques have been proposed to identify ransomware in advance based on known characteristics of previous ransomware attacks [2–4]. However, since it is not possible to detect all ransomware attacks in advance (e.g., when a ransomware attack is based on zero-day exploits), many solutions focus more on efficient data-recovery schemes. Although there exist multiple proposals for efficiently managing backup files [5–7], most anti-ransomware solutions based on the post-recovery strategy experience a high recovery cost as well as a high back up overhead.

As the third defense strategy for ransomware attacks, we investigate whitelisting-based solutions in this paper. A whitelisting-based solution protects important files by granting access permissions only to pre-approved (i.e., whitelisted) programs. Since explicit permission is needed to access a file, as long as the permission is not compromised by ransomware, the file can be fully protected. For example, the controlled folder access service [8] in Windows allows only pre-approved programs to modify files in a protected folder. A whitelisting-based solution has distinct advantages over other solutions. Unlike the pre-detection approach, it supports zero-day protection for unknown ransomware attacks. Compared to the post-recovery approach, little overhead exists because no backup files are needed.

Although a whitelisting-based approach can be a strong defense for ransomware attacks, the existing application-level whitelisting techniques reveal unnecessarily large attack surface to ransomware. If an adversary can inject malicious code into the address space of a pre-approved program, an application-level whitelisting solution cannot prevent ransomware from accessing important files. For example, an adversary can inject ransomware into the memory space of a whitelisted process using a DLL injection method [9]. Once ransomware becomes a part of the whitelisted process, it becomes impossible to protect from a ransomware attack.

In this paper, we propose alohomora[1], a fine-grained whitelisting technique based on I/O activities. Unlike existing whitelisting-based solutions that check if the *current application* is pre-approved for file accesses, alohomora checks if the *current I/O activity* of an application is pre-approved

[1] Alohomora is the unlocking charm from Harry Potter.



*HotStorage '22, June 27–28, 2022, Virtual Event, USA*



ACM ISBN 978-1-4503-9399-7/22/06…$15.00
https://doi.org/10.1145/3538643.3539749

or not. Intuitively, an I/O activity represents a particular I/O execution semantic context (e.g., an I/O activity for updating a row of a database file) within an application. Since alohomora checks whether the I/O activity is pre-approved or not, an application-level ransomware attack (such as the DLL injection method) is not feasible unless ransomware can modify a whitelist of pre-approved I/O activities.

In alohomora, we represent an I/O activity using a program context (PrC) that encodes an execution path of a program up to an I/O system call. The PrC value of an execution path is computed by summing program counter (PC) values of all the function calls along the execution path which leads to a write-related system call. When a write system call is invoked, alohomora passes the current PrC value to a storage system. Before a write operation is physically executed within the storage system, the storage system checks if the PrC value belongs to its whitelist of pre-approved I/O activities. Alohomora achieves a near-perfect defense level against ransomware as long as I/O activities of an application are properly selected.

A strong defense capability of alohomora can be attributed to three factors. First, alohomora manages the membership of a whitelist at the I/O activity level, thus minimizing the size of attack surface by ransomware. A whitelist of pre-approved I/O activities works because the PrC value of an I/O activity can be practically *unique* over all the other PrC values (Section 2). Second, alohomora computes a PrC value by hardware (with an ISA extension of a CPU) and maintains the current PrC value in a privileged kernel-mode register that cannot be accessed from a user-mode execution (Section 2). Since the PrC value is read by a kernel when a write system call is invoked, it is not possible for user-level ransomware to modify the PrC value so that it can be tricked to be a member of the whitelist. Third, alohomora maintains its whitelist inside a storage system, thus making it very difficult for ransomware to access the whitelist (Section 3).

In order to evaluate the effectiveness of alohomora, we built a prototype SSD that supports an in-SSD whitelist of pre-approved I/O activities. Alohomora works with the Linux kernel (ver. 5.10) running on top of an extended RISC-V CPU with a PrC register. Our experimental results show that alohomora successfully protect user files from various sophisticated ransomware attacks. Furthermore, our implementation of alohomora shows that there is negligible overhead in supporting alohomora.

The rest of this paper is organized as follows. We explain the key idea of alohomora in Section 2 and describe the design in Section 3. The experimental results are reported in Section 4. Finally, we conclude in Section 5 with a summary.

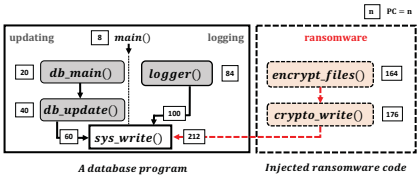


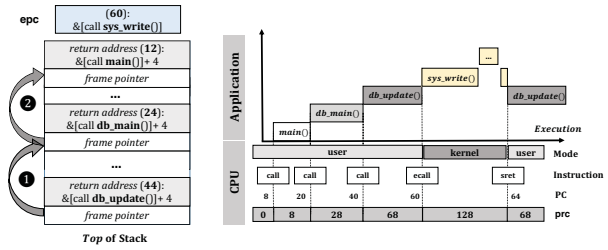

**Figure 1: An example with different I/O activities.**

## 2 I/O ACTIVITY AS A WHITELISTING UNIT

The key differentiating factor of alohomora over existing whitelisting techniques is its whitelisting granularity based on an I/O activity. In alohomora, we grant a write permission for each I/O activity that is represented by its PrC value. Figure 1 illustrates two I/O activities, updating and logging, of a simple database program. The PrC value along the execution path `main()` → `db_main()` → `db_update()` → `sys_write()` is 128 while that along the execution path `main()` → `logger()` → `sys_write()` is 192. If both I/O activities are pre-approved, their PrC values, 128 and 192, are added to a whitelist of pre-approved I/O activities. As shown in Figure 1, when ransomware issues a write system call, which was not pre-approved, the `crypto_write()` function fails because its PrC value does not belong to the whitelist.

Since a PrC value is used to distinguish an I/O activity from other I/O activities, it is critical to guarantee that different I/O activities do not collide with the same PrC value. Intuitively, we can argue that a PrC value is indeed unique over all the other PrC values within a single application. For example, it is not possible for two I/O activities to have the same PrC value because at least one PC value along their execution paths should be different. Otherwise, the execution paths of two I/O activities become identical. I/O activities among different applications can be easily distinguished by incorporating a unique application ID when computing a PrC value. When an application is launched, its hash value, which is computed based on its executable binary, is used as an initial offset of the PrC value.

The PrC value of an I/O activity (i.e., an execution path to a write system call) can be extracted in a straightforward fashion if a frame pointer is used for managing a procedure call stack. By backtracking stack frames of a process when `sys_write()` is invoked, we can get all the return addresses along the execution path. Figure 2(a) illustrates how the PrC value is computed for the updating activity of Figure 1. The PC values of `main()`, `db_main()`, and `db_update()`, can be computed from their respective return addresses while the PC value of `sys_write()` can be read from a hardware register (e.g., epc in RISC-V ISA). However, a frame pointer-based SW method is difficult to use in practice because many modern C/C++ compilers do




(a) A frame pointer-based SW method

(b) A hardware-based method

**Figure 2: Two methods for computing PrC values.**

not use a frame pointer for higher performance. For example, when the *-fomit-frame-pointer* option of GCC is used, no frame pointer is used in managing a procedure call stack. Furthermore, since the PrC value is computed using return addresses in the *user* stack memory, if an adversary has a capability to change the content of the user stack memory, using an I/O activity as a whitelisting unit is not a secure approach.

In alohomora, a PrC value is fully computed by hardware within a CPU. We extend a host CPU to support a privileged PrC register, prc, which is updated when a call instruction and a return instruction are executed. Since the PrC value represents an I/O activity of a user application, prc is updated only in the user mode execution. Figure 2(b) illustrates how the prc is managed while the same updating activity is executed. When call (as well as ecall[2]) is executed, prc is incremented by the PC value of the current call instruction (e.g., 8 for call `main()` and 60 for ecall `sys_write()`), and when ret (as well as sret) is executed, prc is decremented by the PC value of its matching call instruction (e.g., 60 for sret `sys_write()`). When `sys_write()` is invoked by a trap instruction (e.g., ecall), a syscall handler simply reads prc in the kernel mode to find the PrC value of the current I/O activity. Since prc is computed by hardware and only accessible in the kernel mode, the hardware-based method provides a strong protection against any user-mode attack attempt.

# 3 DESIGN OF ALOHOMORA

## 3.1 Threat Model

We assume that ransomware can obtain the root privilege but it cannot run in the kernel mode. Therefore, ransomware can manipulate any process within an infected system. For example, it can disable anti-virus daemons and shut down a backup subsystem to evade system-level defenses. However, since ransomware cannot operate in the kernel mode, we exclude any attack that requires to modify kernel-internal operations or data structures. For example, it is not possible to modify a PrC value of an I/O activity that is temporary stored within the kernel-internal memory (i.e., before it is passed to a device driver). Our assumption on the kernel-mode attack is reasonable because such attacks (e.g., kernel mode rootkits [10], and malicious kernel patches [11]) are extremely difficult to succeed in modern OSes. For example, in order to prevent kernel mode rootkits, most modern OSes support signed kernel modules for stronger kernel security[12], thus making it very difficult for ransomware to load a malicious module to a kernel. Furthermore, attacks that maliciously alter the kernel files saved in disk can be defended by checking the integrity of the kernel code when a system reboots [13, 14].

[2]In RISC-V, the ecall instruction is used to invoke an OS syscall handler while the sret instruction is used to return from it.

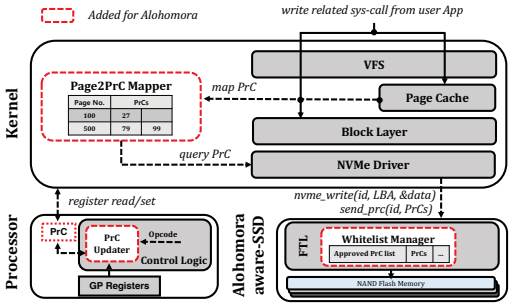



**Figure 3: An overall architecture of alohomora**

## 3.2 Alohomora Overview

Figure 3 depicts an overall organization of alohomora. Since alohomora uses a PrC value to determine if the I/O activity is pre-approved or not, the implementation of alohomora requires several changes from a host CPU to a host OS kernel to an SSD so that PrC values are properly handled. A host CPU is extended to support the prc register (in the PrC Updater module in Figure 3) while an SSD needs to manage a whitelist of pre-approved I/O activities (in the Whitelist Manager in Figure 3).

A host kernel needs to be modified so that a correct PrC value is sent to an SSD. Since an asynchronous write is temporarily stored in a page cache before it is passed to an SSD, the host kernel needs to maintain the PrC value of a written page. The Page2PrC Mapper module in Figure 3 is responsible for maintaining PrC values of pages in the page cache. For example, the page 100 was written by an I/O activity whose PrC value is 27. Since the same page can be written by different I/O activities, the Page2PrC Mapper module maintains a list of PrCs for written page (e.g., a list of two PrC values, 79 and 99, for the page 500). When a page is flushed from the page cache, the mapped PrC list of the page is sent to the SSD. Since the current NVMe write command cannot accommodate multiple PrC values within its current command format, we proposed a new NVMe command for sending a mapped PrC list to the SSD. As shown in Figure 3, the `send_prc()` command is used to transfer a mapped

PrC list of a previous write command. The id field of a normal NVMe write command is used to match a mapped PrC list to a proper previous write request.

Since the prc register is a new addition to an ISA of a host CPU, a host kernel is modified to handle the prc register in a proper fashion. When a CPU is allocated to a different process by a context switch, the current prc register value should be saved as an attribute of a process state so that its value is saved and restored on a context switch. Furthermore, when a new program is started (e.g., by execve()), the prc register should be changed to the initial PrC value of the new program (i.e., its application ID).

### 3.3 Whitelist Management

Alohomora assumes that a PrC value of a pre-approved I/O activity should be known in advance so that the PrC value can be managed inside an SSD. However, statically finding all the possible PrC values of I/O activities of an application in advanced is challenging. In the current version of alohomora, we employ both a static PrC extraction technique as well as a dynamic PrC extraction technique. In order to statically compute a PrC value of an I/O activity, we use a binary-level reverse engineering tool [15] that can generate a call-return graph of an executable binary. Using the call-return graph, all the potential execution paths to write-related system calls are identified and their PrC values are statically computed. Unfortunately, the static PrC extraction technique is not perfect. For example, when a function pointer is used to invoke a function that eventually calls a write system call, it is not possible to find out the PrC value of the corresponding I/O activity.

In order to mitigate the weakness of the static technique, we use an alohomora-aware SSD to identify potentially missed I/O activities during run time. The SSD is configured to notify a host when a PrC value of a write request does not belong to the whitelist of the SSD. Using the feedback information from the SSD, we can find missed I/O activities. Only trusted users can configure an SSD to provide a feedback on the whitelist membership status of a write request.

### 3.4 File Metadata Management

When a file is written by an unauthorized write request, alohomora safely prevents the requested data from being stored to an SSD. However, since a host file system does not know in advance if a file write will be rejected by an alohomora-aware SSD, a special care is needed to guarantee the atomicity of a file data write operation and its metadata update operation. When a file write is issued to an SSD, a host kernel saves an old version of file metadata until the SSD decides if the write activity is a member of a whitelist. When the write activity was not pre-approved, the saved metadata is restored so that the file system consistency is maintained.

### 3.5 Code Relocation Management

Alohomora assumes that a PrC value of a write activity in an application does not change over different executions. However, this assumption may not hold when position-independent codes (PIC) [16] are loaded in memory. For example, when a shared library is dynamically loaded, its executable can be loaded to any page-aligned virtual address, thus making a PrC value variable depending on loaded memory locations of a shared library. To address this issue, alohomora slightly modifies the load procedure of an executable code. In our current version of alohomora, we require that an executable code is aligned to the 24-bit boundary, thus guaranteeing that the lowest 24 bits of PC does not change over different executions. When a PrC value is computed by a host CPU, we use only the lowest 24 bits of PC, ignoring the upper bits (although the prc register maintains a 64-bit value). Meeting the 24-bit alignment requirement is not a problem because modern OSes support large virtual address spaces (e.g., 256 TB with 48 bits).

## 4 EXPERIMENTAL RESULTS

### 4.1 Experimental Setup

In order to understand the effect of alohomora and its performance implications, we have built a prototype alohomora system. Figure 4 shows an overview of our prototype implementation. The host side of our prototype runs Ubuntu 20.04 with a modified Linux kernel (ver. 5.10) on an open-source RISC-V development platform [17]. The enhanced RISC-V CPU with prc register was synthesized on an FPGA-based VC707 board [18] with four RISC-V cores and 4-GB DRAM. Each RISC-V core supports RV64GC ISA and runs at 100-MHz clock speed. An alohomora-aware SSD was implemented on the Cosmos+ OpenSSD board [19] with an ARM Cortex A9 processor and 1-GB DRAM that has 512-GB storage capacity. We extended the OpenSSD firmware, greedy-FTL [20], to implement the I/O whitelisting module. Although several layers of an I/O stack need to be modified to support alohomora, most changes are rather straightforward. As shown in Table 1, the implementation complexity is not high. For example, 38 lines of Chisel code were added to support the prc register while 230 lines of code were modified in Linux kernel.

### 4.2 Ransomware Defense Capability

In order to demonstrate that alohomora can protect files from various ransomware attacks, we used 37 public ransomware programs that included a total of 162 malicious write activities. Although the attacked files were encrypted in the

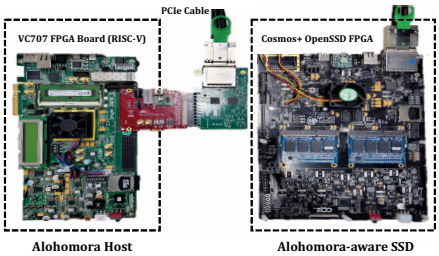



**Figure 4: A prototype alohomora system setup.**

host memory, the file contents in the SSD were not compromised. When the SSD was remounted (or a page cache/buffer was flushed), all the files were fully recovered. Furthermore, when an unauthorized write request was detected by the SSD, the host kernel was notified by a special I/O failure flag so that a ransomware attack can be detected early.

In order to show the key differentiating aspect of alohomora over the existing application-level whitelisting techniques, we simulated the worst-case user-level attack scenarios for alohomora where a pre-approved program is intentionally compromised by injecting ransomware into the address space of the pre-approved program. To inject a ransomware program into the pre-approved program, we manually modified the LD_PRELOAD environment variable so that the ransomware program can be placed within the pre-approved application.

Table 2 summarizes six combinations of a pre-approved program and a injected ransomware program with their respective numbers of PrC values. For ransomware programs, we used five open-source ransomware programs, GonnaCry [21], RAASNet [22], Ransom0 [23], Hidden-tear [24], and FSociety [25] as well as one custom-built program, CustomRS. Unlike open-source ransomware, CustomRS employed a large number of write activities for simulating more intensive attack scenarios. For pre-approved applications, we used MariaDB (a database program [26]), RocksDB (a key-value database [27]), GCC (a GNU C compiler [28]), and Bacula (a backup application [29]). As expected, alohomora successfully defended the pre-approved applications from six ransomware programs even when a ransomware program was injected as a part of a whitelisted program.

## 4.3 Overhead Evaluation

To understand the impact of alohomora on I/O performance, we compared the I/O throughput of the alohomora SSD over that of a baseline SSD with no alohomora support. We measured IOPS values from the baseline SSD (baseline) and three alohomora SSDs (aloho-1K, aloho-10K, and aloho-100K). Three alohomora SSDs are different in their whitelist sizes (i.e., from 1-K entries to 100-K entries). All the measurements were normalized to ones from the baseline SSD. We used four applications in Table 2 as benchmark programs. Since only write requests affect the performance of the alohomora SSD,

| Implementation Layer | Lines of Codes (P/L) |
|---|---|
| RISC-V Extensions | 38 (Chisel) |
| Linux Kernel Extensions | 211 (C), 19 (Assembly) |
| FTL Extensions | 127 (C) |

**Table 1: A summary of implementation complexity.**

| Ransomware | | Application | |
|---|---|---|---|
| Name | # of PrCs | Name | # of PrCs |
| GonnaCry | 7 | MariaDB | 152 |
| RAASNet | 5 | RocksDB | 51 |
| Ransom0 | 3 | GCC | 131 |
| Hidden-tear | 4 | Bacula | 35 |
| FSociety | 4 | MariaDB | 152 |
| CustomRS | 60 | RocksDB | 51 |

**Table 2: A summary of synthetic test cases.**

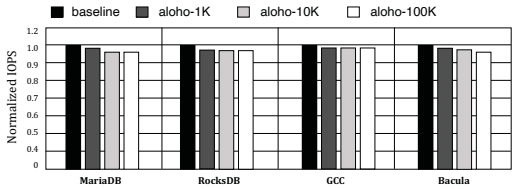



**Figure 5: A comparison of normalized IOPS values.**

we ran benchmark programs under write-intensive workloads. For MariaDB and RocksDB, we used Sysbench [30] and db_bench [31], respectively, with a read/write ratio of 2:8. For GCC, Linux source files were compiled. For Bacula, a large directory and its sub-directories were backed up.

Figure 5 shows normalized IOPS values of three alohomora SSDs over the baseline SSD. All three alohomora SSDs exhibit negligible IOPS degradations. For aloho-1K, only 1.9% of average IOPS drop occurs. Even for aloho-100K, which has a unrealistically large whitelist, there is an average IOPS penalty of 3.7% over baseline. A small performance overhead of the alohomora SSD can be attributed to two reasons. First, a whitelist search overhead is negligible over a typical flash page program time. For example, for aloho-1K, the average whitelist search time was 1.2 $\mu$s while a typical flash page program time is about 400 $\mu$s [32]. Second, since a whitelist can be constructed in advance from pre-approved applications, its data structure can be easily optimized. For example, in our current implementation, a whitelist is organized as a binary search tree, thus making a whitelist search efficient.

# 5 CONCLUSIONS

We have presented a new whitelisting-based anti-ransomware solution, alohomora, that provides a strong protection capability based on a cross-layer implementation approach. Unlike existing application-level whitelisting techniques, alohomora employs an I/O activity as a whitelisting unit, thus successfully protecting files from existing attack schemes for application-level whitelisting solutions. By checking if an I/O activity was pre-approved or not inside an alohomora-aware SSD, our solution provides near-perfect protection support for files stored in the SSD. Our experimental results based on a prototype alohomora system implementation show that alohomora successfully blocked sophisticated ransomware attacks with little SSD performance degradation.

## ACKNOWLEDGMENTS

This work was supported by SNU-SK Hynix Solution Research Center (S3RC) and by the National Research Foundation of Korea (NRF) Grant funded by the Ministry of Science and ICT (MSIT) under Grant NRF-2021R1H1A2093240. The ICT at Seoul National University provided research facilities for this study. (Corresponding author: Jihong Kim.)

## REFERENCES

[1] W. Turton and K. Mehrotra. Hackers breached colonial pipeline using compromised password. https://www.bloomberg.com/news/articles/2021-06-04/hackers-breached-colonial-pipeline-using-compromised-password, 2021.

[2] G. Cusack, O. Michel, and E. Keller. Machine learning-based detection of ransomware using SDN. In *Proceedings of ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, 2018.

[3] C. Chew and V. Kumar. Behaviour based ransomware detection. In *Proceedings of International Conference on Computers and Their Applications*, 2019.

[4] A. Elhadi, M. Maarof, and B. Barry. Improving the detection of malware behaviour using simplified data dependent API call graph. *International Journal of Security and Its Applications*, 7(5):29–42, 2013.

[5] H. Jian, J. Xu, X. Xing, P. Liu, and MK. Qureshi. FlashGuard: leveraging intrinsic flash properties to defend against encryption ransomware. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[6] J. Park, Y. Jung, J. Won, M. Kang, S. Lee, and J. Kim. RansomBlocker: a low-overhead ransomware-proof SSD. In *Proceddings of Design Automation Conference*, 2019.

[7] S. Baek, Y. Jung, A. Mohaisen, S. Lee, and D. Nyang. SSD-insider: internal defense of solid-state drive against ransomware with perfect data recovery. In *Proceedings of International Conference on Distributed Computing Systems*, 2018.

[8] Microsoft documentation. https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/enable-controlled-folders?view=o365-worldwide, 2022.

[9] L. Abrams. Windows 10 ransomware protection bypassed using DLL injection. https://www.bleepingcomputer.com/news/security/windows-10-ransomware-protection-bypassed-using-dll-injection, 2018.

[10] A. Lopez. Malware in Linux: kernel-mode-rootkits. https://www.incibe-cert.es/en/blog/kernel-rootkits-en, 2015.

[11] Rootkits: kernel mode. https://resources.infosecinstitute.com/topic/rootkits-user-mode-kernel-mode-part-2/, 2015.

[12] The Linux kernel documentation. https://www.kernel.org/doc/html/v4.15/admin-guide/module-signing.html, 2022.

[13] Windows 7 code integrity security policy. https://csrc.nist.gov/csrc/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp1327.pdf, 2022.

[14] Microsoft Windows 8.1 kernel patch protection analysis. https://www.ptsecurity.com/upload/corporate/ru-ru/analytics/Windows_81_Kernel_Patch_Protection_Analysis.pdf, 2014.

[15] Radare2. https://github.com/radareorg/radare2.

[16] Linker and libraries guide. https://docs.oracle.com/cd/E26505_01/html/E26506/glmqp.html.

[17] Freedom. https://github.com/sifive/freedom.

[18] Xilinx virtex-7 FPGA vc707 evaluation kit. https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html.

[19] OpenSSD. http://openssd.io/.

[20] OpenSSD greedy-FTL (run-gftl3). https://github.com/CRZ-Technology/OpenSSD-OpenChannelSSD/tree/main/CosmosPlus/OpenSSD/Toshiba_NAND/cosm-plus-ns_19.1_20211216/cosm-plus-sys/cosm-plus-sys.sdk/run-gftl3.

[21] GonnaCry. https://github.com/tarcisio-marinho/GonnaCry.

[22] RAASNet. https://github.com/leonv024/RAASNet.

[23] Ransom0. https://github.com/HugoLB0/Ransom0.

[24] Hidden-tear. https://github.com/starf1ame/Hidden-tear.

[25] FScoiety. https://github.com/graniet/fsociety-ransomware-MrRobot.

[26] MariaDB. https://mariadb.com.

[27] RocksDB. http://rocksdb.org.

[28] GCC, the GNU compiler collection. https://gcc.gnu.org.

[29] Bacula. https://www.bacula.org.

[30] Sysbench. https://github.com/akopytov/sysbench.

[31] RocksDB document. https://www.bookstack.cn/read/rocksdb-en/961dd924ca767aa1.md.

[32] Micron technical note. https://www.micron.com/-/media/client/global/documents/products/technical-note/nand-flash/tn2914.pdf.