# Early mitigation of CPU-optimized ransomware using monitoring encryption instructions

Enomoto, Shuhei
Kuzuno, Hiroki
Yamada, Hiroshi
Shiraishi, Yoshiaki
Morii, Masakatu

(Citation)
International Journal of Information Security,23(5):3393-3413

(Issue Date)
2024-10

(Resource Type)
journal article

(Version)
Version of Record

International Journal of Information Security
https://doi.org/10.1007/s10207-024-00892-2

REGULAR CONTRIBUTION

# Early mitigation of CPU-optimized ransomware using monitoring encryption instructions

**Shuhei Enomoto**[1] · **Hiroki Kuzuno**[1] · **Hiroshi Yamada**[2] · **Yoshiaki Shiraishi**[1] · **Masakatu Morii**[1]



**Abstract**

Ransomware attacks pose a significant threat to information systems. Server hosts, including cloud infrastructure as a service, are prime targets for ransomware developers. To address this, security mechanisms, such as antivirus software, have proven effective. Moreover, research on ransomware detection advocates for behavior-based finding mechanisms while ransomware is in operation. In response to evolving detections, ransomware developers are now adapting an optimized design tailored for CPU architecture (CPU-optimized ransomware). This variant can rapidly encrypt files, potentially evading detection by traditional antivirus methods that rely on fixed time intervals for file scans. In ransomware detection research, numerous files can be encrypted by CPU-optimized ransomware until malicious activity is detected. This study proposes an early mitigation mechanism named CryptoSniffer, which is designed specifically to counter CPU-optimized ransomware attacks on server hosts. CryptoSniffer focuses on the misuse of CPU architecture-specific encryption instructions for swift file encryption by CPU-optimized ransomware. This can be achieved by capturing the ciphertext in user processes and thwarting file encryption by scrutinizing the content intended for writing. To demonstrate the efficacy of CryptoSniffer, the mechanism was implemented in the latest Linux kernel, and its security and performance were systematically evaluated. The experimental results demonstrate that CryptoSniffer successfully prevents real-world CPU-optimized ransomware, and the performance overhead is well-suited for practical applications.

**Keywords** Cloud computing · Operating system · Ransomware prevention · Software security

## 1 Introduction

Ransomware, a form of malicious software (malware), operates against the user's intention by either disabling user operations or pilfering files on targeted computers. Subsequent to these attacks, ransomware demands payment from users in exchange for restoring operations or refraining from exposing stolen files. To incapacitate user operations, ransomware employs file encryption [1–3]. In recent years, attackers planting ransomware have expanded their focus beyond client PCs to include server hosts [4–6] prompting the development of Linux-based ransomware alongside Windows-based variants [1–3, 7–9].

To identify ransomware, antivirus software (antivirus) provides useful detection. Antivirus programs identify malware by matching it with predefined pattern data known as signatures. Furthermore, prior research on ransomware detection [10–22] has introduced effective finding methods, for example, decoy file monitoring [10–12] and file entropy monitoring [13–15]. These monitoring mechanisms, designed to detect file encryption based on the general behavior of ransomware, have proven effective in finding various ransomware families.

In response to existing ransomware detection measures, recent ransomware has undergone advancements in design and implementation. As one of their advancements, ransomware developers now incorporate fast encryption techniques utilizing CPU architecture-specific encryption instructions. For example, LockBit [23, 24] and RansomEXX [25, 26] employ the Advanced Encryption Standard New Instructions (AES-NI) instruction set [27] in Intel x86 for fast encryption.

✉ Shuhei Enomoto
215t801t@gsuite.kobe-u.ac.jp

[1] Kobe University, Kobe, Hyogo 657-8501, Japan

[2] Tokyo University of Agriculture and Technology, Koganei, Tokyo 184-8588, Japan

Published online: 30 July 2024

This study tries to answer the following question: *Is it possible to mitigate the behavior of CPU-optimized ransomware until anti-ransomware mechanisms detect it?* Unlike regular ransomware, CPU-optimized ones perform file encryption so quickly that the systems can be significantly damaged; often, numerous target files can be encrypted before the antivirus scans and detects them. Although current research on behavior-based detection mechanisms for ransomware can identify CPU-optimized ransomware, these mechanisms focus mainly on detecting the target ransomware and do not mitigate damage until malicious activity is alerted.

This study introduces CryptoSniffer, an innovative early mitigation mechanism designed to counteract file encryption by CPU-optimized ransomware. Operating as a software-based solution within the OS kernel, CryptoSniffer is engineered to ensure minimal overhead for benign applications while maintaining high transparency against CPU-optimized ransomware. The primary objective of CryptoSniffer is to enable early mitigation against CPU-optimized ransomware until anti-ransomware mechanisms detect it. Its focus lies in identifying misuses of CPU architecture-specific encryption instructions by ransomware, and it achieves this by monitoring encryption instructions within user processes. Based on the monitoring, CryptoSniffer blocks the file writing from the user processes if suspicious behavior is detected. Notably, CryptoSniffer does not require additional hardware; it can be installed on various server hosts. In summary, the primary contributions of this study are as follows:

1. The proposed mechanism, CryptoSniffer, represents an advanced approach for early mitigation against the latest CPU-optimized ransomware. By monitoring the encryption instructions within user processes, CryptoSniffer prevents the writing of encrypted content to files. Notably, this approach achieves low overhead and is deployable across various server hosts (Sect. 4).
2. The design and implementation details of CryptoSniffer are outlined. Importantly, applying CryptoSniffer necessitates minimal modification to the OS kernel, and it can run simultaneously with existing ransomware detection mechanisms (Sects. 5 and 6).
3. To demonstrate its effectiveness, CryptoSniffer was implemented on Linux 5.7.15 and subjected to security and performance experiments. The security evaluation demonstrated CryptoSniffer's success in preventing file encryption with LockBit-based proof-of-concept (PoC) and RansomEXX. In terms of performance, CryptoSniffer incurred up to 46% overhead on micro-benchmarks and less than 0.2% overhead on real-world server applications (Sect. 7).

# 2 Background

## 2.1 Ransomware attack

### 2.1.1 Target environments

Ransomware is a form of malware designed to extort money by intimidating users on compromised computers. This typically involves encrypting files and demanding payment for their decryption [28]. Frequently, ransomware targets client PCs, as exemplified by instances such as CryptoLocker [1, 2] and WannaCry [3], which focus on targeting Windows hosts on client PCs.

### 2.1.2 Encryption algorithm

To encrypt files, many ransomware variants employ the Advanced Encryption Standard (AES) [29] as a symmetric-key encryption method. AES is a block cipher algorithm that generates ciphertext from a fixed-length key and plaintext. Additionally, several ransomware strains utilize the Rivest Shamir Adleman (RSA) [30] as a public-key encryption method because they need to encrypt the generated AES key.

Figure 1 illustrates an overview of ransomware encryption. Initially, an adversary generates a public- and private-key pair. The public key is embedded in the ransomware, and the private key is located in the Command-and-Control (C&C) server, constructed by an adversary. Subsequently, the ransomware starts execution as the user processes and generates a symmetric key on the infected host. Afterwards, the algorithm encrypts files using the symmetric key and encrypts the symmetric key using the embedded public key.

### 2.1.3 File operations

To locate files for encryption, ransomware scans the OS file system, retrieving file paths. Upon finding files, the ransomware opens them, reads their contents, and then executes the CPU-intensive process of encrypting the file contents. The encrypted contents are subsequently written back to the files.

## 2.2 Existing countermeasures

### 2.2.1 Antivirus

Existing antivirus solutions detect malware, including ransomware, through predefined signatures [31]. These signatures may include hash values and the number of sections in the execution file. Periodic scanning by an antivirus involves

**Table 1** Experimental results for executing ransomware to encrypt 5000 files totaling 300 GiB

| Ransomware family | Encryption method | Execution time | Encryption performance |
|---|---|---|---|
| Conti [38] | Regular | 160 s | 31.25 files per second |
| DarkSide [39] | Regular | 301 s | 16.61 files per second |
| HelloKitty [40] | Regular | 396 s | 12.62 files per second |
| REvil [41] | Regular | 539 s | 9.27 files per second |
| RansomEXX [25] | AES-NI [27] | 4 s | 1250 files per second |

This experiment was evaluated on Ubuntu 20.04.1 LTS with Linux kernel 5.7.15 running on a physical machine equipped with an Intel(R) Core(TM) i7-12700 (4.90 GHz, x86_64) processor and 32 GiB of memory

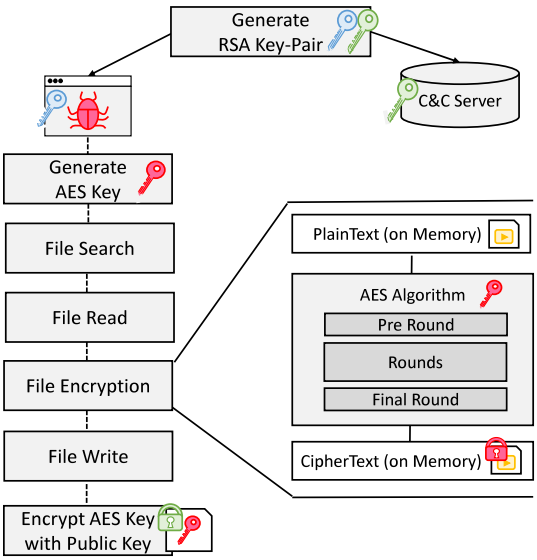


**Fig. 1** Ransomware encryption overview: To encrypt file contents, ransomware often employs symmetric-key encryption. The symmetric key is encrypted by public-key encryption and is additionally written in the encrypted files. These encryption algorithms are calculated by the CPU in user space. File operations, such as read and write, are executed through system calls.

searching for files and user memory spaces that match these signatures and deleting them as malware.

#### 2.2.2 Research on ransomware countermeasures

Research has presented detection mechanisms based on the behaviors of running ransomware. For example, in the past, methods such as decoy file monitoring, which involved concentrating on the file searching behavior of ransomware [10–12], and file entropy monitoring, which involved the assessment of the file I/O entropy [13–15], have been proposed.

Defense mechanisms aimed at recovering file contents post-encryption have also been introduced. For example, extracting the encryption key through the interception of encryption API calls [32] or the customized solid-state drive (SSD) for rolling back overwritten data [33–36] allows later recovery of victim files.

### 2.3 Recent ransomware attack

#### 2.3.1 Target environments

In recent years, ransomware has expanded its focus to include server hosts. Specifically, LockBit [23] targets the VMware ESXi, while ransomware families such as IceFire [8] have developed new versions designed to operate on Linux. Furthermore, RansomEXX [25, 26], which is a ransomware strain that emerged in 2020 and was developed by the cybercriminal threat group Gold Dupont [25], operates on the x86 Linux.

#### 2.3.2 CPU-optimized ransomware

Modern ransomware exhibits the capability for accelerated encryption. For example, LockBit, which was designed for the x86 architecture, exploits the AES-NI instruction set [27] to expedite file encryption. Consequently, LockBit has been observed to achieve approximately 8.4 times faster execution than other ransomware families [37]. This study refers to ransomware that exploits CPU-architecture-specific encryption instructions for swift encryption as CPU-optimized ransomware.

AES-NI provides specific instructions for encrypting plaintext. The `aesenc` instruction calculates *SubBytes*, *ShifRows*, *MixColumns*, and *AddRoundKey* in each round of the AES process. Additionally, the `aesenclast` instruction computes *SubBytes*, *ShifRows* and *AddRoundKey* in the final round of AES. The calculation results are stored in XMM registers, which are 128-bit wide registers. CPU-optimized ransomware copies the ciphertext from XMM registers to memory and writes the ciphertext from memory to the file using the write system call.

With faster encryption, the chances of detecting ransomware using existing methods (e.g., antivirus) decrease. This can be attributed to the fact that these mechanisms require sufficient time to detect ransomware. To illustrate this point, an original experiment on the execution time of the ransomware was conducted. Table 1 shows the experimental results of execution time for various ransomware families. The experiment measured the ransomware execu-

tion time to encrypt 5000 files totaling 300 GiB. Conti [38], DarkSide [39], HelloKitty [40], and REvil [41] encrypted files using normal CPU instructions while RansomEXX utilized the AES-NI instruction set. The experimental results indicate that the execution time of RansomEXX is approximately 40 to 134 times faster than that of other ransomware families. These results also indicate that RansomEXX can encrypt more files than other ransomware in a fixed amount of time because CPU-optimized ransomware performs better than other ransomware.

#### 2.3.3 Problems in existing ransomware countermeasures

**Difficulty in early mitigation**

Existing research, including decoy file monitoring [10–12] and entropy monitoring [13–15], can still detect CPU-optimized ransomware even when it employs faster encryption. However, achieving early mitigation from the launch stage of CPU-optimized ransomware is challenging with these mechanisms. For example, decoy file monitoring permits file encryption by CPU-optimized ransomware until the decoy file is accessed. Similarly, entropy monitoring allows file encryption to continue until the entropy differential surpasses the predefined threshold. Furthermore, machine learning-based detection [18] requires about 2 min to identify ransomware, thus CPU-optimized ransomware can encrypt numerous files before detection.

**Difficulty in applying to server environments**

Restoring files after the completion of a CPU-optimized ransomware attack is relatively straightforward when implementing a customized SSD [33–36] on client PCs. However, applying such a mechanism becomes challenging in server computing instances, such as cloud IaaS because cloud IaaS makes it difficult for cloud users to change physical hardware components.

## 3 Threat model

This section explains the threat model that this study assumed.

### 3.1 Victim environments

To deploy ransomware, adversaries often compromise network services on the targeted host by exploiting vulnerabilities in these services [4–6]. Based on this, this study assumed that ransomware is installed through attacks on network services, initiated by exploited user privileges associated with these services.

In victim environments, the adversary typically lacks superuser privileges (e.g., the root user in Linux). Therefore, operations such as installing kernel modules or replacing kernels become challenging. Additionally, this study assumes the safety of kernel, hypervisor, and hardware implementations, thereby excluding kernel exploits and side-channel attacks from their scope.

### 3.2 Ransomware type

An existing survey [28] categorized ransomware into three types: (1) screen lock ransomware, (2) file encryption ransomware, and (3) data exfiltration ransomware. Screen lock ransomware demands money in exchange for unlocking the screen lock while file encryption ransomware requires a trade of money for the recovery of encrypted files. Data exfiltration ransomware threatens to expose stolen data if the money is not paid.

Furthermore, the survey noted that file encryption attacks constituted 90% of all ransomware incidents in 2019. Consequently, the act of demanding ransom through file encryption has recently emerged as a predominant attack method in ransomware. This study specifically considers the adversary's use of file encryption ransomware for ransom demands. Thus, screen locker ransomware and data exfiltration ransomware are beyond the scope of this study. Furthermore, the victim environments are presumed to have an acceleration feature such as AES-NI implemented on the CPU, and the ransomware exploits this feature for file encryption.

## 4 Requirements

This study introduces CryptoSniffer, a novel preventive mechanism designed for CPU-optimized ransomware. As shown in Fig. 2, CryptoSniffer achieves early mitigation of damage via CPU-optimized ransomware until it is detected by anti-ransomware mechanisms. The outlined requirements that CryptoSniffer fulfills are detailed below.

- **Requirement 1:** Prevent CPU-optimized ransomware while minimizing the negative impact on benign applications such as server programs.
- **Requirement 2:** It can be installed and run on various server environments, such as cloud IaaS.

To fulfill requirement 1, CryptoSniffer incorporates a mitigating feature designed to minimize the impact on the program semantics of benign applications.

To satisfy requirement 2, CryptoSniffer is designed as a software component that does not require additional hardware, such as a specific SSD.

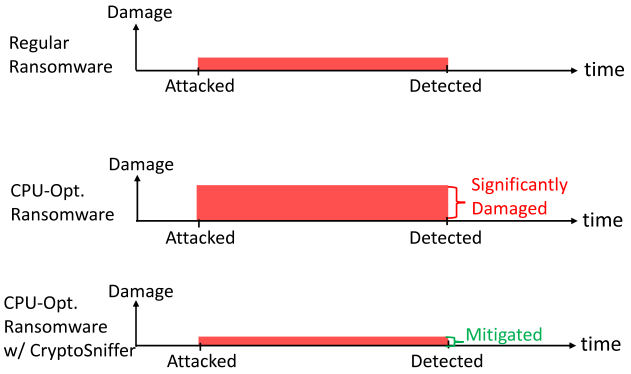


**Fig. 2** Early mitigation by CryptoSniffer: Compared to regular ransomware, CPU-optimized ransomware causes serious damage to the victim host until anti-ransomware mechanisms detect and stop it. CryptoSniffer mitigates damage caused by CPU-optimized ransomware before detection.

### 4.1 Approach

This section explains the core approach of CryptoSniffer.

An adversary gains unauthorized access to the target host by exploiting network services. Subsequently, the adversary installs and initiates CPU-optimized ransomware using the user privilege of the compromised network service. The CPU-optimized ransomware operates as a user process, encrypting files that the user privilege of the network service can access for reading and writing.

CryptoSniffer's focus lies in the misuse of CPU architecture-dependent encryption instructions by CPU-optimized ransomware during file encryption. Leveraging this insight, CryptoSniffer intercepts the ciphertext generated by user processes using encryption instructions, preventing file writing that involves this ciphertext. Initially, CryptoSniffer monitors the issuance of encryption instructions by user processes and captures the generated ciphertext. During file writing from user processes, CryptoSniffer examines the content intended for writing to the file using the captured ciphertext. If the captured ciphertext is present in the content, CryptoSniffer halts the file writing process.

### 4.2 Design challenges

To fulfill the approach indicated in Sect. 4.1, the required design challenges are presented below.

- Transparent monitoring for applications:
  Encryption instructions are executed by the user process in the user space. Existing mechanisms such as software breakpoints and the ptrace system call [42] are valuable for monitoring these instructions. However, because these methods are detectable by existing techniques [43], CPU-optimized ransomware can evade them. For example, Lockbit examines the existence of a debugger that uses ptrace [24]. Thus, a highly transparent monitoring mechanism is essential for user processes.
- Mitigating performance overhead for applications:
  The environment where CryptoSniffer operates hosts benign applications (e.g., web server). Therefore, CryptoSniffer is designed to run with minimal overhead on these benign applications.

## 5 Design

This section outlines the design of CryptoSniffer and provides detailed information on how it safeguards file contents from CPU-optimized ransomware.

### 5.1 Design overview

Figure 3 shows an overview of CryptoSniffer. CryptoSniffer is a software component integrated into the kernel space that monitors the file encryption activities of user processes from the kernel space. Operating in the kernel space ensures that CryptoSniffer is highly transparent to user processes.

The monitoring mechanism of CryptoSniffer encompasses the following phases.

1. **Process Startup Phase:** This stage involves finding encryption instructions within the memory space of user processes. The operation conducted during the Process Startup Phase is referred to as an *instruction search*.
2. **File Encryption Phase:** In this phase, CryptoSniffer captures the ciphertext generated by user processes.
3. **File Writing Phase:** This phase involves identifying ciphertext from memory contents intended for writing to files using the captured ciphertext. The operation conducted during the File Writing Phase is termed a *content search*.

#### 5.1.1 Process startup phase

This phase conducts a search for binary patterns that match encryption instructions within memory pages. It then sets hardware breakpoints into subsequent instructions following the located encryption instructions. Employing hardware breakpoints ensures that user processes remain unaware of CryptoSniffer's monitoring activity within the kernel. The binary patterns utilized in the search are registered by a user with administrator privileges through a dedicated interface. For example, in Linux for x86, a root user registers patterns corresponding to the aesenclast instruction of AES-NI. The search focuses solely on executable pages to minimize overhead, and the subsequent instructions following the found patterns are registered into hardware breakpoints.

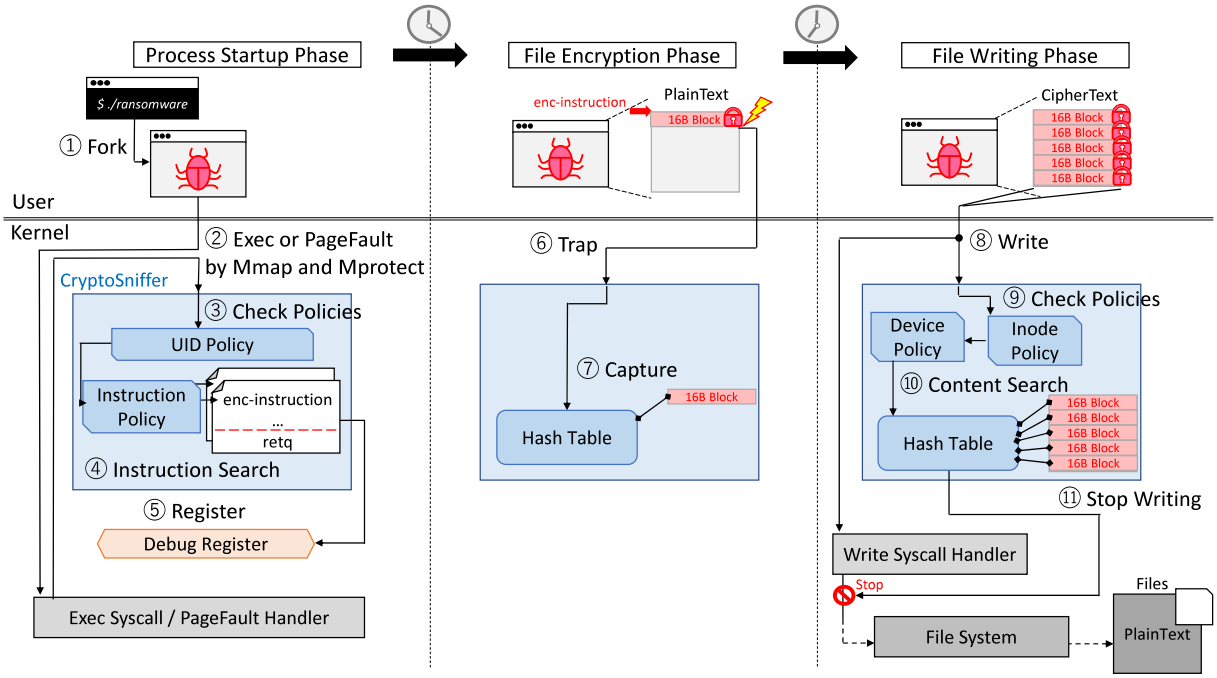


**Fig. 3** CryptoSniffer design overview:To prevent file encryption by CPU-optimized ransomware, the mechanism of CryptoSniffer comprises three phases. First, Process Startup Phase registers the address of encryption instructions to debug registers. Second, File Encryption Phase captures ciphertext. Third, File Writing Phase compares the ciphertext with the file contents that will be written to the file and stops the writing.

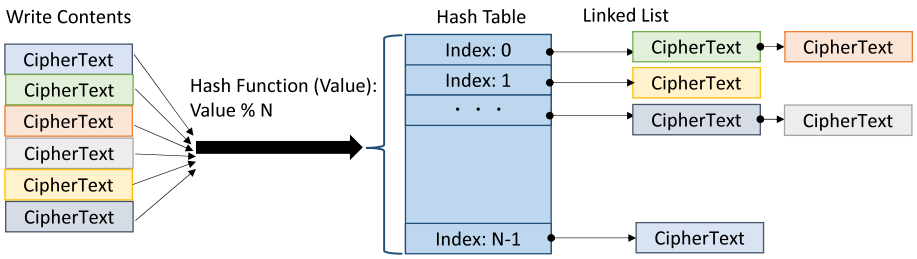


**Fig. 4** Chaining hash table: The captured ciphertext is stored in the chaining hash table on memory. By the chaining hash table, CryptoSniffer can discover the same ciphertext faster in the File Writing Phase.

### 5.1.2 File encryption phase

This phase captures ciphertext when the user processes finish the encryption calculation in the user space. As the next instruction after the encryption instruction is trapped by hardware breakpoints, CryptoSniffer can capture the timing immediately after the completion of the encryption instruction. Subsequently, CryptoSniffer retrieves the ciphertext by reading CPU registers (e.g., xmm0 in x86) and stores the ciphertext in memory within the kernel space. To manage ciphertext in memory, CryptoSniffer utilizes the chaining hash table depicted in Fig. 4. The choice of employing a chaining hash table is to facilitate rapid content searches in the File Writing Phase.

### 5.1.3 File writing phase

This phase scans memory contents intended for writing files by monitoring file-writing-related system calls issued by user processes. The search involves extracting memory contents of specified sizes and verifying their existence in the chaining hash table. The extracted memory contents are used as the index to involve the chaining hash table. As shown in Fig. 4, each index in the chaining hash table links memory contents that are stored in the File Encryption Phase. If the contents are found in the chaining hash table, CryptoSniffer protects the file contents by interrupting the execution of the system call.

### 5.2 Minimizing impact for benign applications

Benign applications, which are not CPU-optimized ransomware, also write encrypted contents to files. Consequently, CryptoSniffer incorporates policies to alleviate the impact of program semantics on these benign applications.

#### 5.2.1 File encryption by trusted users

CryptoSniffer permits file encryption by user processes initiated by trusted users. To enlist these trusted users, CryptoSniffer offers an interface named the UID Policy. The UID Policy is a user-id (UID) list designed to manage users exempted from instruction search during the Process Startup Phase. Before initiating the instruction search, CryptoSniffer obtains the UID of the user processes and cross-references it with the UID Policy. If the same UID is identified in the UID Policy, CryptoSniffer aborts the instruction search.

#### 5.2.2 File encryption to specific files

To enable file encryption for specific files, CryptoSniffer offers an interface called Inode Policy. Inode Policy is an inode list designed to bypass the content search. Files registered in the Inode Policy are excluded from the content search during the File Writing Phase. Consequently, CryptoSniffer writes encrypted contents to these files without interrupting file writing-related system calls.

#### 5.2.3 Network packet encryption

CryptoSniffer facilitates network packet encryption for network devices through an interface called the Device Policy. During the File Writing Phase, CryptoSniffer identifies the device to which content is being written and cross-references it with the Device Policy. If the same device is registered in the Device Policy, CryptoSniffer aborts the content search.

## 6 Implementation

The CryptoSniffer prototype was implemented on the Linux kernel in the x86 architecture. This section explains the implementation details of CryptoSniffer.

### 6.1 Policy details

CryptoSniffer manages four user-defined policies, considering the UID Policy, Instruction Policy, Inode Policy, and Device Policy. The primary role of each policy is explained as follows:

- **UID Policy:** CryptoSniffer excludes the instruction search of the user processes owned by the registered UIDs (e.g., the root user).
- **Instruction Policy:** CryptoSniffer executes the instruction search with the registered instructions (e.g., the aesenclast instruction).
- **Inode Policy:** CryptoSniffer excludes the content search in files writes linked to the registered inodes (e.g., /home/alice/keepass/Database.kdbx).
- **Device Policy:** CryptoSniffer excludes the content search in file writes to registered devices (e.g., network socket).

#### 6.1.1 UID policy

To identify user processes indicated by trusted users, CryptoSniffer utilizes the UID Policy in the Process Startup Phase, as illustrated in Fig. 3. Prior to the instruction search, CryptoSniffer compares the current UID of a running user process with UID patterns extracted from the UID Policy. To manage trusted users, CryptoSniffer utilizes procfs as an interface for registration. Writing the UID to /proc/cryptosniffer/uid_policy notifies CryptoSniffer of the UID. The file permissions under /proc/cryptosniffer are configured to allow read and write access to the root user.

Before initiating the instruction search, CryptoSniffer retrieves the current UID of the running thread and searches for the UID in the UID Policy. If the same UID is identified in the UID Policy, CryptoSniffer bypasses the instruction search.

#### 6.1.2 Instruction policy

For the instruction search execution, CryptoSniffer employs the Instruction Policy in the Process Startup Phase, as depicted in Fig. 3. During the instruction search, CryptoSniffer retrieves instruction patterns from the Instruction Policy and scans executable pages using these patterns. CryptoSniffer offers a designed system call as an interface for registration to the Instruction Policy. In the existing CryptoSniffer prototype, the ability to issue a specific system call is restricted to the root user. Therefore, CPU-optimized ransomware cannot manipulate the policy.

#### 6.1.3 Inode policy

To identify files excluded from the content search, CryptoSniffer employs the Inode Policy in the File Writing Phase, as illustrated in Fig. 3. CryptoSniffer offers a specific system call for registering these files. In this system call, CryptoSniffer identifies an inode from the file path of the system call argument and registers the inode to the Inode Policy.

Within the write system call handler, CryptoSniffer determines an inode from the file descriptor and searches for the

inode in the Inode Policy. Subsequently, if the same inode is identified in the Inode Policy, CryptoSniffer bypasses the content search.

#### 6.1.4 Device policy

To identify devices excluded from the content search, CryptoSniffer utilizes the Device Policy in the File Writing Phase, as shown in Fig. 3. Within the write system call handler, CryptoSniffer verifies the file type and bypasses the content search if the file is identified as a socket.

### 6.2 Process startup phase

The Process Startup Phase indicates the instruction search in CryptoSniffer, as outlined in Sect. 5, and the flowchart is illustrated in Fig. 5. Initially, CryptoSniffer verifies the UID of the user process entering the kernel through system calls. Subsequently, if the UID of the user process is not registered in the UID Policy, CryptoSniffer begins the instruction search for the user process using the Instruction Policy.

#### 6.2.1 Check UID policy

CryptoSniffer retrieves UID from the currently running user process using the current_uid function and cross-references UID with the UID Policy. If the UID is identified in the UID Policy, CryptoSniffer concludes that the user process is executed by a trusted user and skips the instruction search. Conversely, if the UID is not found in the UID Policy, CryptoSniffer initiates the instruction search.

#### 6.2.2 Trigger timing for starting instruction search

CryptoSniffer begins the instruction search when creating new executable pages for the user process. The initiation timing of the instruction search can be categorized into three types.

1. **Exec System Call Trigger:** The instruction search is triggered by the exec system call, which creates new physical pages mapped to executable virtual address areas.
2. **Mmap System Call Trigger:** The instruction search is indicated by the mmap system call, which employs demand paging to create new physical pages mapped to executable virtual address areas.
3. **Mprotect System Call Trigger:** The instruction search is activated by the mprotect system call, altering the permission of existing physical pages mapped to non-executable virtual address areas to executable pages.

**Exec system call trigger**

The exec system call trigger indicates the instruction search when launching user processes. In the exec system call handler, CryptoSniffer identifies executable virtual address areas and provides these address areas to a component responsible for the instruction search. To accomplish this, CryptoSniffer retrieves the virtual address areas of the running thread and searches for executable virtual address areas. Subsequently, CryptoSniffer passes the address range to the function for starting the instruction search.

**Mmap system call trigger**

The mmap system call trigger indicates the instruction search after launching user processes. Since CPU-optimized ransomware may map executable pages using the mmap system call, CryptoSniffer performs an instruction search similar to the case of an exec system call. In the Linux kernel, physical pages are not created in the mmap system call because Linux utilizes demand paging, which generates physical pages during a page fault. Consequently, CryptoSniffer executes the instruction search after a page fault occurs.

Initially, when the mmap system call is invoked, CryptoSniffer checks the flags argument of the mmap system call within the system call handler. If an executable bit is present in the flags, CryptoSniffer records the executable virtual address and size. Subsequently, when the page fault handler is invoked, CryptoSniffer verifies whether the faulted virtual address has already been recorded by CryptoSniffer. If the virtual address is recorded, CryptoSniffer concludes that the virtual address is mapped to executable physical pages. CryptoSniffer then calls the function for starting the instruction search.

CPU-optimized ransomware might alter page permissions from non-executable to executable after starting the user process. To capture the encryption instructions on the pages changed to executable, CryptoSniffer also performs an instruction search in the mprotect system call, similar to the mmap case.

#### 6.2.3 Instruction search

**Find instructions**

After user processes enter the kernel using system calls such as exec, mmap, and mprotect, CryptoSniffer invokes the function for the instruction search. This function performs two main tasks: (1) identifying virtual addresses of encryption instructions from executable pages and (2) registering the identified virtual addresses to hardware debug registers.

In the CryptoSniffer prototype for x86, the function specifically searches for the aesenclast instruction, which calculates the final round of AES involving *SubBytes*, *ShifRows*, and *AddRoundKey*. Because the aesenclast instruction is a variable-length instruction, the function determines the length of the instruction by parsing the found instruction.

Subsequently, the function provides the hardware debug registers with the virtual addresses of instructions, considering the length of the identified instructions. By controlling these debug registers, CryptoSniffer successfully traps events after the execution of the encryption instructions is completed.

**Debug registers**

To write virtual addresses to debug registers, CryptoSniffer utilizes the API function provided by the Linux kernel for debug register control. This registration triggers a hardware debug interruption when a thread attempts to execute instructions at the registered virtual address. During the hardware debug interruption, CryptoSniffer identifies whether the interrupted thread is a target of CryptoSniffer monitoring by checking the added entry for marking in the process control block structure (e.g., task_struct in Linux).

Because of the limited number of hardware debug registers, the count of discovered encryption instructions may surpass the available registers. However, even where the number of found encryption instructions exceeds the number of hardware debug registers, CryptoSniffer can still trap these instructions. Initially, CryptoSniffer revokes execute permission on pages containing encryption instructions, leading to a page fault caused by non-executable pages. In the page fault handler, CryptoSniffer restores execution permission on the page and registers the virtual address of the encryption instruction on the faulted page to the hardware debug register. During this process, CryptoSniffer also revokes execution permission on the page pointed to by the evicted virtual address from the hardware debug register. This swapping mechanism enables CryptoSniffer to trap encryption instructions, even when the number of debug registers is exceeded.

### 6.3 File encryption phase

Figure 6 illustrates the flowchart for the File Encryption Phase. During this phase, CryptoSniffer traps hardware debug interruptions triggered by debug registers and captures the ciphertext generated by CPU-optimized ransomware.

#### 6.3.1 Trap hardware debug interruption

When a hardware debug interruption occurs, the function for the debug interruption handler in the Linux kernel is invoked. Within the function, CryptoSniffer checks whether the marking flag in the process control block structure is activated. If it is activated, CryptoSniffer concludes that the thread has completed the execution of the encryption instruction. Conversely, if the marking flag is not enabled, CryptoSniffer deems that the interruption falls outside the scope of CryptoSniffer monitoring and dispatches a SIGTRAP signal to the thread.

#### 6.3.2 Capture ciphertext

CryptoSniffer captures the generated ciphertext if the trapped thread has been marked by CryptoSniffer. The x86-based CPU-optimized ransomware, utilizing the AES-NI instruction set, stores calculation results in 128-bit XMM registers. Consequently, CryptoSniffer retrieves the ciphertext of the trapped thread by reading XMM registers. The extracted ciphertext is subsequently inserted into a chaining hash table managed by CryptoSniffer and referenced during the content search of the write system call.

### 6.4 File writing phase

Figure 7 shows the flowchart for the File Writing Phase. CryptoSniffer evaluates the policies associated with the thread, considering Inode Policy and Device Policy. If the thread falls outside the defined policies, CryptoSniffer scans the contents that are about to be written to the file through content search.

#### 6.4.1 Check inode/device policy

In the write system call, CryptoSniffer examines the threads marked by CryptoSniffer against Inode Policy and Device Policy. Initially, CryptoSniffer determines the inode associated with the file using the file descriptor specified in the system call argument. During this process, CryptoSniffer verifies whether the inode is enlisted in the Inode Policy. Subsequently, CryptoSniffer determines the device linked to the inode, and examines whether this device corresponds to the Device Policy.

#### 6.4.2 Content search

Following the evaluation of the Inode Policy and Device Policy, CryptoSniffer proceeds to scrutinize the memory contents slated for writing to the file. During the content search, CryptoSniffer segments the contents into 128-bit portions and explores the chaining hash table by employing the hashed 128-bit contents as an index. If the matched contents are identified in the chaining hash table, CryptoSniffer obstructs the file writing process by interrupting the system call.

## 7 Evaluation

To assess the effectiveness of CryptoSniffer, this section describes the experiments conducted in both the security and performance domains.

– Security Capability Experiment:
  The security experiment validates the prevention of

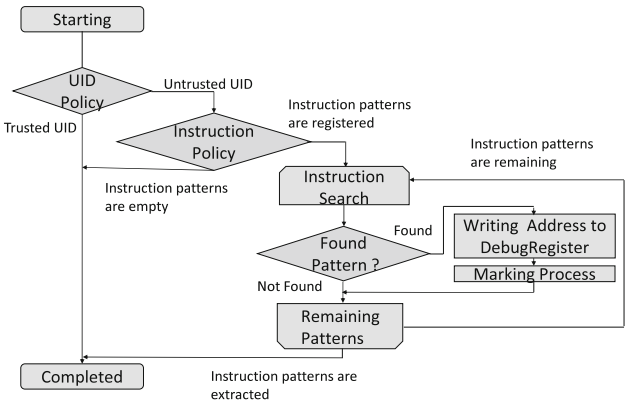


**Fig. 5** Flowchart of the process startup phase: This phase checks UID and instruction policies and starts the instruction search. This phase is not only called at the beginning of the user processes in the exec system call. For example, this phase is also called after creating executable pages using the mmap and mprotect system call.

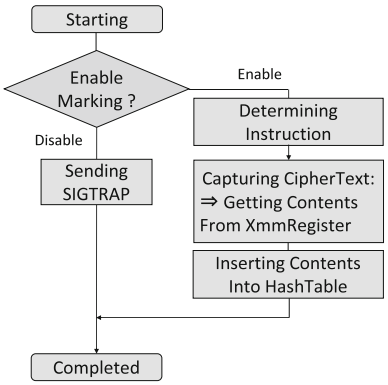


**Fig. 6** Flowchart of file encryption phase: Received hardware debug interruption, this phase extracts encrypted contents from XMM registers and stores the contents in the chaining hash table.

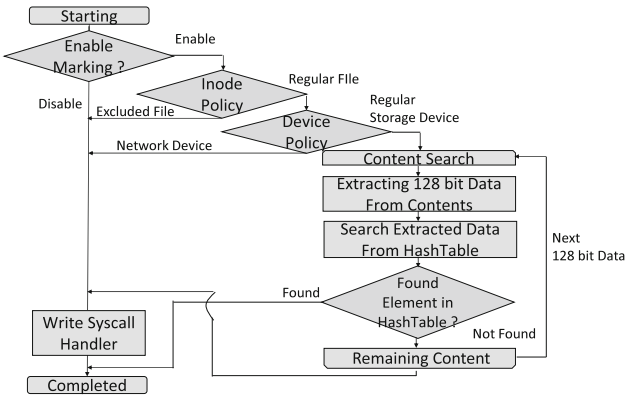


**Fig. 7** Flowchart of file writing phase: This phase checks inode and device policies and starts the content search. If encrypted contents are found in contents that will be written to the file, writing to the file is stopped. After that, the write system call is returned to the user processes.

encryption using the following programs. First, a PoC program that encrypts file contents with AES-NI was executed. Second, a real-world CPU-optimized ransomware samples were also executed.

– Performance Measurements:
Performance measurements gauge the impact of CryptoSniffer on application performance through both micro-benchmark and real-world server applications.
– Extensive Experiment for Security Evaluation:
To evaluate the security capability of CryptoSniffer, real-world CPU-optimized ransomware samples were collected and executed. This experiment derived the number of CPU-optimized ransomware from wild Linux ransomware families.

CryptoSniffer was evaluated using a Linux kernel 5.7.15 in an environment running on a physical machine equipped with an Intel(R) Core(TM) i7-12700 (4.90 GHz, x86_64) processor and 32 GiB of memory.

The Linux distribution used was Ubuntu 20.04.1 LTS on the Linux kernel 5.7.15. CryptoSniffer was implemented by extending 1,366 lines to the Linux kernel.

## 7.1 Security capability experiments

The experiments involved running ransomware samples in environments with CryptoSniffer and a vanilla kernel (without CryptoSniffer). Subsequently, the experiments verify the success or failure of the encryption attack.

– A PoC Program for File Encryption: This experiment executes the PoC program ported LockBit 2.0 from Windows to Linux.
– A Real-World CPU-Optimized Ransomware: This experiment executed RansomEXX, RansomEXX2 [44], and Erebus [45], which are real-world CPU-optimized ransomware packages on Linux.

These experiments involve comparing the SHA256 hash values of victim files before and after executing the ransomware samples. Successful prevention of encryption is indicated if the hash values remain identical. If a different hash value is detected, the experiments further examine the binary contents using tools such as binary viewers [46] because the ransomware sample could have written metadata (e.g., encrypted AES key) to the file. Successful prevention is affirmed if the original binary contents remain unchanged.

To account for real-world malware often packing its original code, each experiment includes two versions of the executable files: the vanilla version (unpacked) and the packed version (using the UPX packer [47]).

#### 7.1.1 PoC program

The PoC program conducts a file search for target extensions (e.g., .pdf) within file systems and subsequently encrypts the discovered files using the AES-NI instruction set. It employs partial encryption, focusing on encrypting only the initial 4 KiB of the file offset, and utilizes multi-threading for searching and encrypting. The encryption algorithm is AES-CBC, with a static 128-bit key applied uniformly across all files. Following encryption, the program overwrites the original file contents and renames the file extensions to .enc.

The results, as presented in Table 2, demonstrate that for both the vanilla and packed binaries, the kernel with the CryptoSniffer installed maintains identical hash values before and after encryption. This signifies successful prevention of encrypted content writing.

#### 7.1.2 RansomEXX

RansomEXX examines the instruction set supported by the CPU architecture of the targeted machine using the cpuid instruction. If the CPU architecture supports the AES-NI instruction set, RansomEXX leverages AES-NI for encryption. The encryption algorithm employed is AES-ECB, and a unique 256-bit key is generated for each file. The encrypted file contents are overwritten to the victim and the files are renamed extensions to .31gs1 after the writing is complete.

The results are presented in Table 2. While the hash value of the file differed between the environments with CryptoSniffer and the vanilla kernel lacking CryptoSniffer, CryptoSniffer preserved the original binary contents of the file, whereas the vanilla kernel permitted file encryption. Consequently, CryptoSniffer effectively thwarted the encryption attempts by RansomEXX in both the vanilla and packed executable files scenarios.

#### 7.1.3 RansomEXX2

RansomEXX2 is a new version of RansomEXX and is implemented in Rust, while the classical RansomEXX is implemented in C++. The encryption algorithm is the same as that of RansomEXX and the files are compromised by overwriting the file contents and renaming the extensions to .cs1c4t.

The results are shown in Table 2. In both the packed and unpacked versions, the hash value of the victim file was the same before and after the execution of RansomEXX2. Therefore, CryptoSniffer successfully prevented file contents from being compromised by RansomEXX2.

**Table 2** Prevention results of CryptoSniffer for CPU optimized ransomware

| Program | CryptoSniffer |
|---|---|
| LockBit 2.0 Based PoC w/o UPX | ✓ |
| LockBit 2.0 Based PoC w/ UPX | ✓ |
| RansomEXX w/o UPX | ✓ |
| RansomEXX w/ UPX | ✓ |
| RansomEXX2 w/o UPX | ✓ |
| RansomEXX2 w/ UPX | ✓ |
| Erebus w/o UPX | ✓ |
| Erebus w/ UPX | ✓ |

(✓ Success; − Failure)

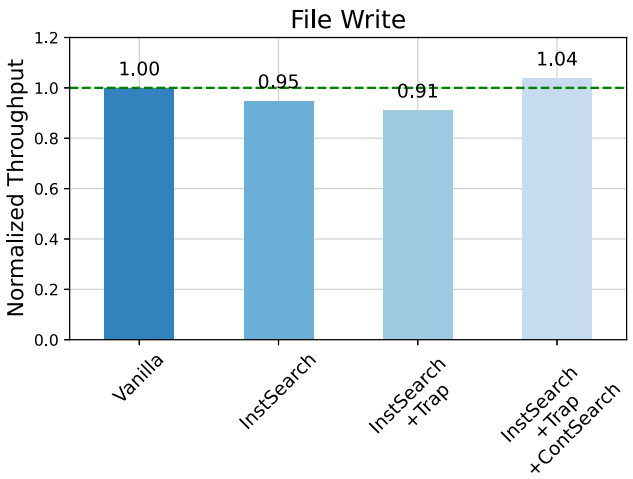


**Fig. 8** Performance overhead in file Write benchmark

#### 7.1.4 Erebus

Erebus overwrites file contents with encrypted file contents using the AES algorithm with AES-NI. In addition, Erebus encrypts the encryption key using the RSA algorithm and writes the key to each victim file. The victim file extensions are renamed to .ecrypt once the writing is complete.

The results are presented in Table 2. In both the packed and unpacked versions, the hash value of the victim file differs before and after the execution of Erebus. This is because Erebus writes the encryption key at the end of the file. The original binary contents of the file were protected in CryptoSniffer. In addition, the experiment confirmed that Erebus initiated malicious activity after suspending its execution for 5 min. However, CryptoSniffer successfully prevented the file contents from being compromised by Erebus.

### 7.2 Micro-benchmark measurements

This experiment evaluates the performance of each component of CryptoSniffer using original benchmark programs.

**Table 3** Analysis results for the number of downloadable and runnable Linux-based CPU-optimized ransomware

| Ransomware families | Downloadable samples | Runnable samples | AES-NI samples |
|---|---|---|---|
| 37 | 22 | 12 | 3 |

**Table 4** Analysis results for the methods of encrypting and compromising files in runnable Linux-based ransomware

| Sample name | Encryption method | File remove/overwrite |
|---|---|---|
| Conti [58] | Regular | Overwrite |
| DarkSide [59] | Regular | Overwrite |
| Erebus [60] | AES-NI | Overwrite |
| HelloKitty [61] | Regular | Overwrite |
| Hive [62] | Regular | Overwrite |
| Kuiper [63] | Regular | Overwrite |
| LockBit [64] | Regular | Overwrite |
| Monti [65] | Regular | Overwrite |
| RansomEXX [66] | AES-NI | Overwrite |
| RansomEXX2 [67] | AES-NI | Overwrite |
| RedAlert [68] | Regular | Overwrite |
| REvil [69] | Regular | Overwrite |

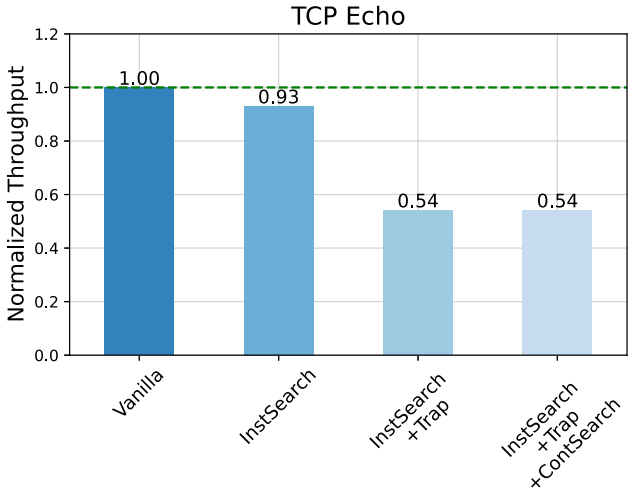


**Fig. 9** Performance overhead in TCP Echo benchmark

The benchmark programs include the following: (1) File Write: Measures the performance of writing encrypted content using the AES-NI to a file. (2) TCP Echo: Measures the performance of sending encrypted payloads using AES-NI to the server through the TCP protocol.

The experimental patterns comprised the following steps. (1) Vanilla: The scenario without the CryptoSniffer was installed. (2) InstSearch: The component responsible for the instruction search. (3) InstSearch+Trap: Components of InstSearch and trapping by debug registers. (4) InstSearch+Trap+ContSearch: Components for InstSearch+Trap and content search. InstSearch+Trap +ContSearch is the full configuration of CryptoSniffer and includes all the components.

#### 7.2.1 File write

The program of File Write allocates the 1024 bytes buffer, the process involves encrypting the contents of the buffer with AES-NI, and then writing the encrypted contents to a file on local storage.

The results, depicted in Fig. 8, reveal that compared to those of Vanilla, InstSearch and InstSearch+Trap incurred a performance overhead ranging from 5 to 9%. On the other hand, InstSearch+Trap+ContSearch demonstrated a 4% improvement in performance. This is attributed to ContSearch skipping the write to a file on storage.

#### 7.2.2 TCP echo

The TCP Echo benchmark comprises a TCP client and server programs. The TCP client allocates the 16-byte buffer, encrypts its contents with AES-NI, and transmits the encrypted data to the TCP server. This transmission process is repeated 1024 times, resulting in the sending of 16,384 bytes of content to the server. Subsequently, the TCP client calculates throughput based on the execution time.

The results, depicted in Fig. 9, indicated that compared to Vanilla, InstSearch incurred a 7% performance overhead, while InstSearch+Trap incurred a 46% performance overhead. Notably, because writing to the network socket is beyond the scope of the content search in this experiment, InstSearch+Trap and InstSearch+Trap +ContSearch recorded identical performances.

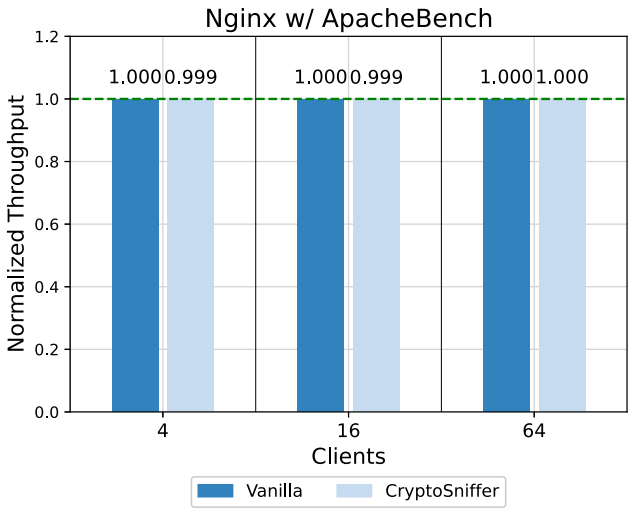


**Fig. 10** Performance overhead in Nginx

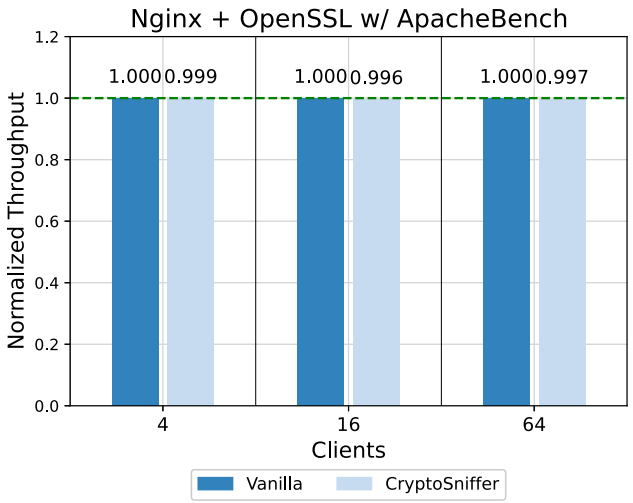


**Fig. 11** Performance overhead in Nginx + OpenSSL

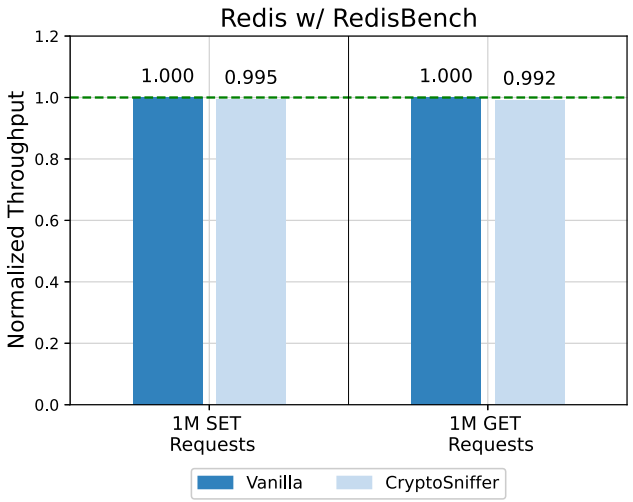


**Fig. 12** Performance overhead in Redis

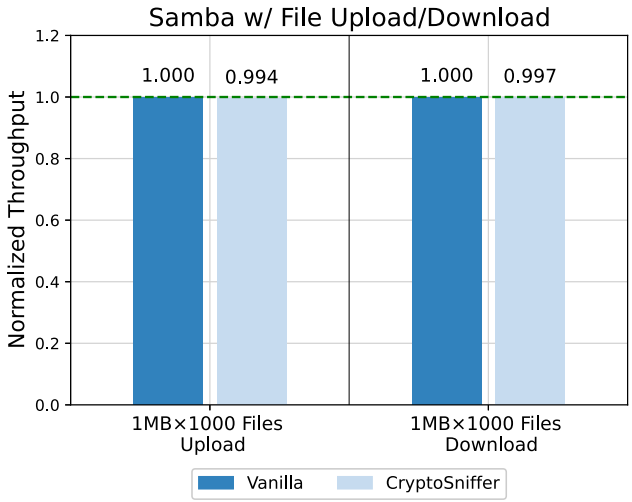


**Fig. 13** Performance overhead in Samba

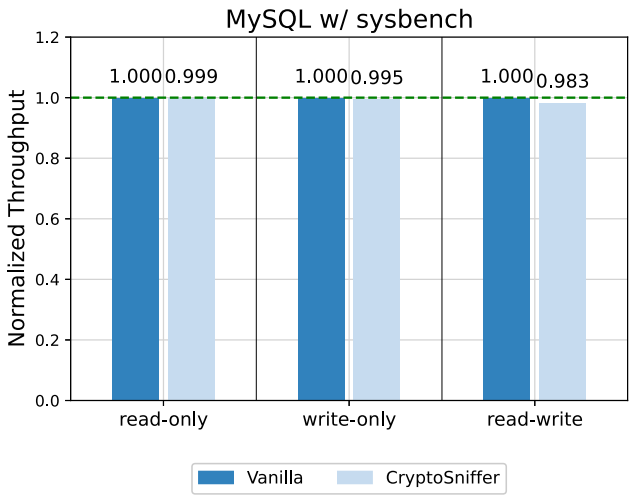


**Fig. 14** Performance overhead in MySQL

### 7.3 Real-world applications measurements

This experiment utilizes real-world server applications to showcase the performance overhead in the full configuration of CryptoSniffer. The following four server applications were executed on a physical machine: (1) on the vanilla kernel without the CryptoSniffer installed and (2) on the CryptoSniffer.

- Web Server: This experiment measured the throughput of Nginx 1.22.1 [48] via HTTP protocol with ApacheBench 2.3 [49]. In addition, this experiment also measured the throughput via HTTP over TLS protocol using OpenSSL 1.1.1f [50].
- In-memory KVS: This experiment measured the throughput of Redis 7.0.11 [51] with RedisBench 7.0.11 [52]. RedisBench issued SET and GET commands in the

benchmark and each command was executed one million times.
- File Server: This experiment measured the throughput of Samba 4.15.13 [53] by uploading and downloading files. The size of each file was 1 MiB, and the number of files for the operations was 1000.
- Database Server: This experiment measured the throughput of MySQL 8.0.33 [54] with sysbench 1.0.18 [55]. The workloads of sysbench were used for read-only, write-only, and read-write.

The results are depicted in Figs. 10, 11, 12, 13, and 14. Across all the applications, the performance overhead imposed by CryptoSniffer was negligible. The maximum overhead for each application was as follows: Nginx (0.001%), Nginx + OpenSSL (0.001%), Redis (0.008%), Samba (0.006%), and MySQL (0.017%).

### 7.4 Extensive experiment for security evaluation

This experiment analyzed the number of real-world CPU-optimized ransomware in wild Linux-based ransomware families. First, the names of existing Linux-based ransomware families were retrieved from Malpedia [56], which is an online database service for recording existing malware families. Based on the names, samples were found from MalwareBazaar [57] which is a repository for sharing real-world malware binaries. Subsequently, the experiment conducted static and dynamic analyses of the downloaded samples using existing tools for reverse engineering and identified CPU-optimized ransomware from the samples.

The results are presented in Tables 3 and 4. The experiment identified 12 runnable ransomware samples from 37 real-world Linux ransomware families. Furthermore, three samples of CPU-optimized ransomware were found in the 12 runnable ransomware samples. The identified samples were RansomwareEXX, RansomwareEXX2, and Erebus.

## 8 Discussion

### 8.1 Evaluation consideration

#### 8.1.1 Security capability experiment

As RansomEXX writes metadata to victim files, the hash value of the victim file did not match in CryptoSniffer. This metadata, written to the victim file, consists of an encrypted 256-bit AES key. RansomEXX encrypts the AES key using a 4096-bit RSA public key embedded in its execution binary and decrypts the AES key using an RSA private key received from the attacker's server. Unfortunately, the current CryptoSniffer prototype cannot prevent the writing of metadata from CPU-optimized ransomware. However, the experimental results indicate that CryptoSniffer can protect original file contents from RansomEXX. Therefore, the experimental results demonstrate that CryptoSniffer has achieved good security. The user can restore files that have the same hash value as before the attack by extracting the original file contents from the victim file.

#### 8.1.2 Performance measurements

In both micro-benchmarks and real-world applications, the user processes and the CryptoSniffer kernel remained stable.

With respect to the micro-benchmarks, the performance overhead of InstSearch ranged from 3 to 7%. On the other hand, the performance overhead of InstSearch+Trap ranged from 9 to 46%. In the Trap scenario, the performance overhead depends on the number of hardware debug interruptions that occur. Therefore, File Write, which incurs 64 interruptions, results in a 9% overhead, while TCP Echo, which incurs 1024 interruptions, results in a 46% overhead.

In real-world applications, all the programs excluding Nginx with OpenSSL that were used in the experiments issue no AES-NI instructions, and benchmarks start after the user processes open the network services. Although Nginx with OpenSSL issues AES-NI instructions, CryptoSniffer allows the writing of encrypted content to the network socket because the content search is canceled by the Device Policy.

Thus, CryptoSniffer incurs only negligible overheads ranging from 0.001 to 0.017%. These minor overheads are attributed to the instruction search of the mmap and mprotect which contain a PROT_EXEC flag in running benchmarks.

### 8.2 Portability consideration

#### 8.2.1 Porting to Other CPU architectures

In modern CPU architectures, the acceleration feature for fast encryption is implemented. For instance, ARMv8-A has hardware AES acceleration, such as the AESE instruction [70]. In recent years, cloud IaaS services have started to provide ARM-based computing instances [71, 72]. Consequently, attackers may design and implement new CPU-optimized ransomware for other CPU architectures.

The current prototyped CryptoSniffer for x86 traps encryption instructions using hardware debug registers. If hardware debug registers are supported, CryptoSniffer can implement the same mechanism in other CPU architectures. For example, the ARM Cortex-A7 has six hardware debug registers [73], and the Linux kernel for the ARM supports the control of the debug registers [74].

#### 8.2.2 Porting to other operating systems

With other operating systems that support hardware debug registers, the current prototyped CryptoSniffer for Linux can be adapted. For instance, FreeBSD supports debug registers for running threads within the structure of struct pcb [75]. Moreover, setting the debug registers can be accomplished using functions such as load_dr0 to load_dr7 [76].

On the other hand, Windows can set and obtain debug registers by utilizing kernel mode driver API functions, such as PsSetCreateProcessNotifyRoutine [77] and PsSetContextThread [78].

### 8.3 Implementation limitation

#### 8.3.1 Circumventing CryptoSniffer

This section discusses potential circumvention methods for CryptoSniffer. Two possible approaches are considered for the current CryptoSniffer prototype:

First, ransomware might inspect file contents after CryptoSniffer stops writing encrypted data. By checking the file contents, ransomware could detect that the encryption process could fail, and the ransomware could attempt to bypass CryptoSniffer. For example, ransomware might switch from CPU-optimized encryption to regular encryption without acceleration features. While the current CryptoSniffer prototype might struggle to prevent such circumvention, the delayed encryption process in the absence of CPU optimization could increase the detection chances for existing solutions such as antivirus and incident response.

Second, if an attacker compromises users registered in UID Policy, they could bypass the instruction search in the Startup Phase. This involves leaking the accounts of registered users because of weak security settings, gaining access to the server host with leaked credentials, and executing ransomware as a trusted user for CryptoSniffer. However, the intrusion channels defined in Sect. 3 for ransomware typically involve exploiting network services (e.g., vulnerable web servers [79]). To run these network services, specific users are created per application (e.g., www-data user in Nginx), and as such. These users are generally distinct from the registered users in UID Policy (e.g., the root user).

### 8.4 False-positive to benign applications

Certain cryptographic libraries have the potential to implement encryption algorithms using encryption instructions. For example, the OpenSSL library implements Envelope API [80], which provides fast encryption using AES-NI. As other examples, GnuTLS [81] and Crypto++ [82] also implement AES-NI in their libraries [83, 84].

If benign applications employ these libraries and the administrator fails to register with CryptoSniffer's policies, CryptoSniffer aborts the file encryption of benign applications by raising a false-positive. In the future, the plan is to suspend the applications that issue encryption instructions at the File Writing Phase and to require authentication from the administrator.

### 8.5 File remove-based ransomware

The current CryptoSniffer prototype cannot protect file contents from ransomware that removes original files and creates encrypted files. Thus, the challenge against file remove-based ransomware remains.

However, the analysis presented in Sect. 7.4 shows that modern Linux-based ransomware tends to overwrite rather than remove file contents. Furthermore, existing survey work [22] explains that modern ransomware prefers file encryption implementation that overwrites part of the file. This is because partial file encryption is effective for evading ransomware detection based on monitoring file system activities.

### 8.6 Evading CryptoSniffer

The security capability experiment described in Sect. 7.1 indicates that Erebus attempts to behave in a similar manner to benign applications by delaying the initiating encryption. However, CryptoSniffer successfully protected file contents from Erebus. CPU-optimized ransomware needs to map executable pages including encryption instructions before initiating encryption and issue the instructions before compromising victim files. As CryptoSniffer finds the instructions at the mapping and traps the instructions at the issuing, it overcomes evasion, such as initially refraining from encryption and later attempting to encrypt.

However, if the adversary injects code of ransomware into the applications executed by the users that are registered in the UID Policy, CryptoSniffer cannot protect file contents from the attack. However, as the operation to inject code into running user processes requires privileged user permission in modern OSes [42], the adversary suffers from the achievement of successful code injection.

### 8.7 Resistance to obfuscation techniques

The security capability experiment in Sect. 7.1 showed that CryptoSniffer can prevent file encryption from CPU-optimized ransomware that obfuscates executable binaries with UPX packer. Existing survey work [85] explains that UPX packer accounts for the majority of obfuscation techniques in wild Linux-based malware. Therefore, the experiment was sufficiently realistic to evaluate the security capability of CryptoSniffer.

Furthermore, it is considered that CryptoSniffer is fully capable of protecting file contents from Linux-based CPU-optimized ransomware that employs other obfuscation techniques. Malware is required to decode the obfuscated binaries until the original code is placed in the executable pages. As CryptoSniffer initiates the scanning of executable pages after the original code is placed, it is resistant to obfuscation techniques.

## 9 Related work

This section provides an overview of related works in the field of ransomware countermeasures and highlights the distinctions from CryptoSniffer. The existing survey work [28] categorizes existing ransomware countermeasures into three groups. First, ransomware detection involves mechanisms that detect ransomware based on its behavior during execution. Second, ransomware defense focuses on recovery by restoring victim files after the ransomware completes its attack. Third, ransomware prevention aims to block file encryption before it occurs.

### 9.1 Ransomware detection

Ransomware detection focuses on the behaviors exhibited by ransomware running on victim computers and identifies ransomware based on these malicious behaviors. Examples of mechanisms within this category include decoy file monitoring, file entropy monitoring, and file system activity monitoring.

#### 9.1.1 Decoy file monitoring

Decoy file monitoring [10–12] operates on the key insight that ransomware typically accesses various files on the file system for encryption. Utilizing this insight, decoy file monitoring creates specific files for monitoring and identifies user processes accessing these files as potential ransomware.

#### 9.1.2 File entropy monitoring

UNVEIL [13] is a ransomware detection mechanism that monitors the entropy of file contents. It capitalizes on the observation that file encryption introduces entropy bias compared to the state before encryption. To identify this entropy bias, UNVEIL observes file I/O requests and computes the entropy of file contents. If an entropy bias is detected in a file, UNVEIL halts the user processes responsible for writing to that file.

#### 9.1.3 File system activity monitoring

ShieldFS [16] is a ransomware detection mechanism that monitors file system behaviors. From a file system perspective, many ransomware implementations exhibit certain characteristics. First, they often search for files with specific extensions (e.g., .pdf, .jpeg, etc.) before initiating file encryption. Additionally, many ransomware variants rename files with specific extensions (e.g., .lockbit, .31gs1, etc.) after completing the encryption process. Leveraging these behaviors, ShieldFS observes file system activities and identifies user processes that exhibit similarities to known ransomware patterns.

#### 9.1.4 Combination mechanisms

Combination mechanisms [19–21] in ransomware detection integrate multiple detection methods. Each mechanism, such as decoy file monitoring, file entropy monitoring, and file system activity monitoring, contributes to an ad hoc detection method. Therefore, combining multiple detection approaches enhances the ability to detect a broader range of ransomware variants.

### 9.2 Ransomware defense

Ransomware defense permits file encryption by means of ransomware but focuses on restoring the contents of the files after the ransomware attack is completed. Mechanisms within this category may involve monitoring encryption API calls and incorporating data-recoverable SSD.

#### 9.2.1 Encryption API monitoring

PayBreak [32] facilitates the recovery of victim files by tracking encryption keys during file encryption. This is achieved by hooking API calls in existing cryptographic libraries, such as CryptoAPI in Windows. The encryption key is stored in the key vault and utilized for recovery once a ransomware attack is completed.

#### 9.2.2 Ransomware-tolerant SSD

FlashGuard [33] is a hardware-based file recovery mechanism leveraging SSD characteristics. When the OS requests changes to data contents on the SSD, the new data contents are written to different data blocks than the original ones. Utilizing this characteristic, FlashGuard retains the old data contents of the original data blocks, enabling the recovery of victim files.

## 9.3 Ransomware prevention

Ransomware prevention enables the execution of ransomware but hinders file encryption. This mechanism may involve implementing file-level access control.

### 9.3.1 File-level access control

AntiBotics [86] is a mechanism that regulates user access to specific directories. When users attempt to access controlled directories, user-interactive validation involving biometrics and CAPTCHA is triggered. Access is granted upon successful validation by the logged-in OS user.

## 9.4 Comparison

This section outlines the distinctions between existing works and CryptoSniffer. Figure 15 illustrates the positions of both existing works and CryptoSniffer in terms of the implementation layer and countermeasure types. Additionally, Table 5 provides a comparison in (1) Early Mitigation, (2) File Recovery, (3) Ransomware Types, and (4) Computing Environment Types.

Existing detection mechanisms [10–18], such as decoy file monitoring and file entropy monitoring, can also detect and halt CPU-optimized ransomware. However, existing detection mechanisms focus on the behaviors of running ransomware, requiring them to permit file encryption for detection. Consequently, these mechanisms face challenges in achieving early mitigation against CPU-optimized ransomware. In constant, CryptoSniffer can accomplish early mitigation from the first instance of file encryption, as it monitors the issuance of encryption instructions.

Among the existing defense mechanisms [32–36], encryption API monitoring relies on hooking API calls related to file encryption. Consequently, CPU-optimized ransomware can potentially evade API monitoring by utilizing encryption instructions without making API calls. Alternatively, ransomware-tolerant SSD can contribute to defending against CPU-optimized ransomware. However, as it is a hardware-based defense mechanism, implementing this mechanism in server computing environments, such as cloud IaaS, can be challenging. CryptoSniffer, which requires only hardware debug registers as a hardware feature, is easily installable in server computing environments.

File-level access control [86], an existing prevention mechanism, necessitates real-time user-interactive validation. This makes it challenging to apply in server computing environments, where there is typically no logged-in user, unlike client PCs. CryptoSniffer does not require user-interactive validation and completes prevention automatically.

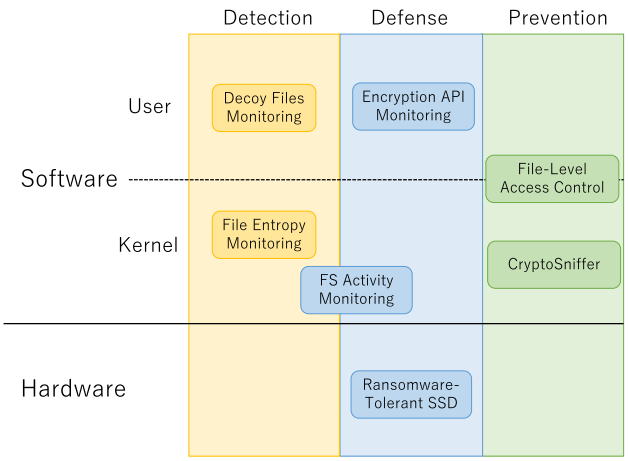


**Fig. 15** Positioning of existing works in the countermeasure types and implementations: CryptoSniffer is a new prevention mechanism implemented in the kernel. Since CryptoSniffer requires only hooking some kernel functions and debug registers, implementation conflicts with other works are unlikely to occur. Thus, CryptoSniffer can also be used simultaneously with other works.

While CryptoSniffer is not specifically designed for client PCs, it can be installed and run on them. In client PCs, where the logged-in user often has administrator privileges and performs actions that may trigger ransomware (e.g., web browsing), CryptoSniffer can allow ransomware execution if the login user is registered to the UID Policy. Consequently, on client PCs with CryptoSniffer installed, it is advisable to separate login users and administrative users, registering only the administrative user to the UID Policy.

### 9.4.1 Comparison with ransomware-tolerant SSD

This section compares the traditional SSD-level backup and recovery systems [33–36] and CryptoSniffer in the following situations:

- **Cloud User Level:** Each user of the cloud IaaS takes measures against CPU-optimized ransomware to protect their file contents on the VM.
- **Cloud Provider Level:** The cloud provider is motivated to protect the file contents of all users on the VMs against CPU-optimized ransomware.
- **Local Level:** Users and administrators on the local machines (e.g., PCs and, on-premise servers) consider measures against CPU-optimized ransomware to protect their storage.

On the cloud user level, CryptoSniffer has an advantage over SSD-level systems. This is because SSD-level methods require a hardware modification, even if the cloud provider does not permit users to change the hardware, while CryptoSniffer requires no hardware modification.

**Table 5** Comparison of ransomware countermeasures features

| Feature | Decoy file monitoring [10–12] | File entropy monitoring [13–15] | FS activity monitoring [16–18] | API monitoring [32] | Ransomware-tolerant SSD [33–36] | File-level ACL [86] | Crypto-sniffer |
|---|---|---|---|---|---|---|---|
| Countermeasure category | Detection | Detection | Detection | Defense | Defense | Prevention | Prevention |
| Early mitigation | | | | | | ✓ | ✓ |
| File recovery | | | ✓ | ✓ | ✓ | | |
| CPU-Opt. ransomware | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ |
| Regular ransomware | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| Client PCs | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | △ |
| Servers | ✓ | ✓ | | ✓ | | | ✓ |

(✓ is supported; △ is partially supported)

On the cloud provider level, combining CryptoSniffer and SSD-level methods is effective. Unlike cloud users, cloud providers can easily introduce hardware such as ransomware-tolerant storage into their data centers. However, replacing existing storage with other storage in the data center may require a huge amount of time because data migration on the petabyte and/or exabyte scale is required [87]. Consequently, employing CryptoSniffer until the cloud provider completes the data migration is an effective solution.

On the local level, SSD-level methods are more suitable than CryptoSniffer. This is because introducing storage and migrating data on local machines, are easier than in the cloud provider's data centers. However, employing CryptoSniffer until the data migration on local machines is completed, is still effective.

## 10 Conclusion

To counter ransomware attacks, antivirus measures and ransomware detection research are practical. However, early mitigation against running CPU-optimized ransomware on servers proves challenging for these mechanisms. First, antivirus programs may struggle to swiftly detect CPU-optimized ransomware because of its rapid file encryption. Second, existing detection mechanisms, such as decoy file monitoring, are less effective at early mitigation because they permit ransomware to execute until malicious behaviors are detected.

This study introduces CryptoSniffer, a novel early mitigation mechanism designed to prevent file encryption by CPU-optimized ransomware. CryptoSniffer, an answer to the research question described in Sect. 1, mitigates the behavior of CPU-optimized ransomware until it is detected by anti-ransomware mechanisms. To achieve high transparency and low overhead in user processes, CryptoSniffer acts as a software component within the OS kernel, monitoring encryption instructions issued by user processes. It captures the ciphertext generated by these processes and halts file encryption by comparing the captured ciphertext with the memory contents intended for writing to files. To validate the security and performance of CryptoSniffer, experiments were also conducted. The results demonstrated the efficacy of CryptoSniffer in preventing file encryption via real-world CPU-optimized ransomware, with reasonable performance overheads observed in real-world applications.

**Acknowledgements** This work was partially supported by the contract of "Research and development on new generation cryptography for secure wireless communication services" among "Research and Development for Expansion of Radio Wave Resources (JPJ000254)" supported by the Ministry of Internal Affairs and Communications, Japan.

**Author Contributions** First Author helped in the conceptualisation, methodology, literature review and writing of the original draft, Second Author helped in the conceptualisation, review, and supervision. All authors read and approved the final manuscript.

**Funding** Open Access funding provided by Kobe University.

**Data Availability** All data are either included in the paper. Also, will be available to those requests for researchers.

## Declarations

**Conflict of interest** The authors declare that they have no Conflict of interest as defined by Springer or other interests that might be perceived to influence the results and/or discussion reported in this paper.

## References


1. TREND MICRO, Ransomware Raises the Stakes with CryptoLocker. https://www.trendmicro.com/vinfo/de/threat-encyclopedia/web-attack/3132/ransomware-raises-the-stakes-with-cryptolocker (2013). Accessed 23 Jan 2024
2. Kharraz, A., Robertson, W., Balzarotti, D., Bilge, L., Kirda, E.: Cutting the Gordian knot: a look under the hood of ransomware attacks. In: Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '15), Milan, Italy, pp. 3–24. Springer (2015)
3. TREND MICRO, WannaCry/Wcry Ransomware: how to defend against it. https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/wannacry-wcry-ransomware-how-to-defend-against-it (2017). Accessed 23 Jan 2024
4. csoonline.com, Apache Struts 2 exploit used to install ransomware on servers. https://www.csoonline.com/article/561031/apache-struts-2-exploit-used-to-install-ransomware-on-servers.html (2017). Accessed 23 Jan 2024
5. Sophos, DearCry ransomware attacks exploit Exchange server vulnerabilities. https://news.sophos.com/en-us/2021/03/15/dearcry-ransomware-attacks-exploit-exchange-server-vulnerabilities/ (2021). Accessed 23 Jan 2024
6. The BlackBerry Research & Intelligence Team, ESXiArgs Ransomware: Knocking Out Unpatched VMware ESXi Linux Servers Worldwide. https://blogs.blackberry.com/en/2023/02/esxiargs-ransomware-knocking-out-unpatched-vmware-esxi-linux-servers-worldwide (2023). Accessed 23 Jan 2024
7. Microsoft Security Intelligence, Threat description search results: Win32/Reveton. https://www.microsoft.com/en-us/wdsi/threats/threat-search?query=Trojan:Win32/Reveton.A (2018). Accessed 23 Jan 2024
8. bleepingcomputer, Icefire ransomware now encrypts both linux and windows systems. https://www.bleepingcomputer.com/news/security/icefire-ransomware-now-encrypts-both-linux-and-windows-systems/ (2023). Accessed 23 Jan 2024
9. bleepingcomputer, Linux version of Royal Ransomware targets VMware ESXi servers. https://www.bleepingcomputer.com/news/security/linux-version-of-royal-ransomware-targets-vmware-esxi-servers/ (2023). Accessed 23 Jan 2024
10. Moussaileb, R., Bouget, B., Palisse, A., Le Bouder, H., Cuppens, N., Lanet, J.L.: Ransomware's early mitigation mechanisms. In: Proceedings of the 13th International Conference on Availability, Reliability and Security (ARES '18), Hamburg, Germany, pp. 1–10. ACM (2018)
11. Lee, J., Lee, J., Hong, J.: How to make efficient decoy files for ransomware detection? Proceedings of the International Conference on Research in Adaptive and Convergent Systems (RACS '17), Krakow, Poland, pp. 208–212. ACM (2017)
12. Gómez-Hernández, J.A., Álvarez-González, L., García-Teodoro, P.: R-Locker: thwarting ransomware action through a honeyfile-based approach. Comput. Secur. **73**, 389–398 (2018)
13. Kharaz, A., Arshad, S., Mulliner, C., Robertson, W., Kirda, E.: UNVEIL: a large-scale, automated approach to detecting ransomware. In: Proceedings of the 25th USENIX Security Symposium (USENIX Security '16), Austin, TX, pp. 757–772. USENIX Association (2016)
14. McIntosh, T., Jang-Jaccard, J., Watters, P., Susnjak, T.: The inadequacy of entropy-based ransomware detection. In: Proceedings of the 25th International Conference on Neural Information Processing (ICONIP '18), Siem Reap, Cambodia, pp. 181–189. Springer (2019)
15. Lee, K., Lee, J., Lee, S.Y., Yim, K.: Effective ransomware detection using entropy estimation of files for cloud services. Sensors **23**(6), 3023 (2023)
16. Continella, A., Guagnelli, A., Zingaro, G., De Pasquale, G., Barenghi, A., Zanero, S., Maggi, F.: ShieldFS: A self-healing, ransomware-aware filesystem. In: Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC '16), New York, USA, pp. 336–347. ACM (2016)
17. Palisse, A., Durand, A., Le Bouder, H., Le Guernic, C., Lanet, J.L.: Data aware defense (DaD): towards a generic and practical ransomware countermeasure. In: Proceedings of the 22nd Nordic Conference on Secure IT Systems (NordSec '17), Tartu, Estonia, pp. 192–208. Springer (2017)
18. Ayub, M.A., Siraj, A., Filar, B., Gupta, M.: RWArmor: a static-informed dynamic analysis approach for early detection of cryptographic windows ransomware. Int. J. Inf. Secur. **23**, 533–556 (2023)
19. Kharraz, A., Kirda, E.: Redemption: real-time protection against ransomware at end-hosts. In: Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID '17), Atlanta, GA, USA, pp. 98–119. Springer (2017)
20. Scaife, N., Carter, H., Traynor, P., Butler, K.R.: CryptoLock (and drop it): stopping ransomware attacks on user data. In: Proceedings of the 36th International Conference on Distributed Computing Systems (ICDCS '16), pp. 303–312. IEEE (2016)
21. Mehnaz, S., Mudgerikar, A., Bertino, E.: RWGuard: a real-time detection system against cryptographic ransomware. In: Proceedings of the 21th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID '18), pp. 114–136. Springer (2018)
22. Han, J., Lin, Z., Porter, D.E.: On the effectiveness of behavior-based ransomware detection. In: Proceedings of the 16th International Conference on Security and Privacy in Communication Systems (SecureComm '20), Washington, DC, USA, pp. 120–140. Springer (2020)
23. TREND MICRO, Analysis and Impact of LockBit Ransomware's First Linux and VMware ESXi Variant. https://www.trendmicro.com/en_us/research/22/a/analysis-and-Impact-of-lockbit-ransomwares-first-linux-and-vmware-esxi-variant.html (2022). Accessed 23 Jan 2024
24. Hao, C.J.: RE Series #10: LockBit ELF. https://chanjinhao.wordpress.com/2022/01/28/re-series-10-lockbit-elf/ (2022). Accessed 23 Jan 2024
25. TREND MICRO, Ransomware Spotlight: RansomEXX. https://www.trendmicro.com/vinfo/us/security/news/ransomware-spotlight/ransomware-spotlight-ransomexx (2022). Accessed 23 Jan 2024
26. MalwareBazaar Database, RansomEXX. https://bazaar.abuse.ch/sample/cb408d45762a628872fa782109e8fcfc3a5bf456074b007de21e9331bb3c5849/ (2020). Accessed 23 Jan 2024
27. Intel, Intel®Advanced Encryption Standard Instructions (AES-NI). https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html (2012). Accessed 23 Jan 2024
28. McIntosh, T., Kayes, A.S.M., Chen, Y.P.P., Ng, A., Watters, P.: Ransomware mitigation in the modern era: a comprehensive review, research challenges, and future directions. ACM Comput. Surv. CSUR **54**(9), 1–36 (2021)
29. NIST Computer Security Resource Center, Advanced Encryption Standard. https://csrc.nist.gov/glossary/term/advanced_encryption_standard. Accessed 1 Jun 2024
30. NIST COMPUTER SECURITY RESOURCE CENTER, Rivest Shamir Adelman. https://csrc.nist.gov/glossary/term/rsa. Accessed 1 Jun 2024
31. ClamAV. https://www.clamav.net/ (2002). Accessed 23 Jan 2024
32. Kolodenker, E., Koch, W., Stringhini, G., Egele, M.: PayBreak: defense against cryptographic ransomware. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17), Abu Dhabi, United Arab Emirates, pp. 599–611. ACM (2017)

33. Huang, J., Xu, J., Xing, X., Liu, P., and Qureshi, M.K.: FlashGuard: leveraging intrinsic flash properties to defend against encryption ransomware. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17), New York, NY, USA, pp. 2231–2244. ACM (2017)
34. Baek, S., Jung, Y., Mohaisen, A., Lee, S., Nyang, D.: SSD-Insider: internal defense of solid-state drive against ransomware with perfect data recovery. In: Proceedings of the 38th International Conference on Distributed Computing Systems (ICDCS '18), Vienna, Austria, pp. 875–884. IEEE (2018)
35. Wang, X., Yuan, Y., Zhou, Y., Coats, C.C., Huang, J.: Project Almanac: a time-traveling solid-state drive. In: Proceedings of the 14th European Conference on Computer Systems (EUROSYS '19), Dresden, Germany, pp 1–16. ACM (2019)
36. PPark, J., Jung, Y., Won, J., Kang, M., Lee, S., Kim, J.: RansomBlocker: a low-overhead ransomware-proof SSD. In: Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC '19), Las Vegas, NV, USA, pp 1–6. ACM/IEEE (2019)
37. Splunk Technology, Gone in 52 seconds...and 42 minutes: a comparative analysis of ransomware encryption speed. https://www.splunk.com/en_us/blog/security/gone-in-52-seconds-and-42-minutes-a-comparative-analysis-of-ransomware-encryption-speed.html (2022). Accessed 23 Jan 2024
38. MalwareBazaar Database, Conti. https://bazaar.abuse.ch/sample/95776f31cbcac08eb3f3e9235d07513a6d7a6bf9f1b7f3d400b2cf0afdb088a7/ (2022). Accessed 23 Jan 2024
39. MalwareBazaar Database, DarkSide. https://bazaar.abuse.ch/sample/984ce69083f2865ce90b48569291982e786980aeef83345953276adfcbbeece8/ (2021). Accessed 23 Jan 2024
40. MalwareBazaar Database, HelloKitty. https://bazaar.abuse.ch/sample/8f3db63f70fad912a3d5994e80ad9a6d1db6c38d119b38bc04890dfba4c4a2b2/ (2021). Accessed 23 Jan 2024
41. MalwareBazaar Database, REvil. https://bazaar.abuse.ch/sample/ea1872b2835128e3cb49a0bc27e4727ca33c4e6eba1e80422db19b505f965bc4/ (2021). Accessed 23 Jan 2024
42. man7.org, ptrace(2) - Linux manual page. https://man7.org/linux/man-pages/man2/ptrace.2.html (2023). Accessed 23 Jan 2024
43. Schallner, M.: Beginners guide to basic linux anti anti debugging techniques. https://api.semanticscholar.org/CorpusID:58341211 (2006). Accessed 23 Jan 2024
44. Security Intelligence, RansomExx upgrades to rust. https://securityintelligence.com/x-force/ransomexx-upgrades-rust/ (2022). Accessed 19 May 2024
45. Trend Micro, Erebus Resurfaces as Linux Ransomware. https://www.trendmicro.com/en_nl/research/17/f/erebus-resurfaces-as-linux-ransomware.html (2017). Accessed 19 May 2024
46. man7.org, hexdump(1) - Linux manual page. https://man7.org/linux/man-pages/man1/hexdump.1.html (2023). Accessed 23 Jan 2024
47. The UPX Team, UPX: the Ultimate Packer for eXecutables. https://upx.github.io/ (2016). Accessed 23 Jan 2024
48. Nginx Inc., Advanced Load Balancer, Web Server; Reverse Proxy - NGINX. https://www.nginx.com/ (2004). Accessed 23 Jan 2024
49. The Apache Software Foundation, ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.4/programs/ab.html (2023). Accessed 23 Jan 2024
50. OpenSSL Foundation, Inc. OpenSSL: cryptography and SSL/TLS toolkit. https://www.openssl.org/ (1999). Accessed 26 May 2024
51. Redis Labs. https://redis.io/ (2010). Accessed 23 Jan 2024
52. Redis Labs, How fast is Redis? https://redis.io/topics/benchmarks (2023). Accessed 23 Jan 2024
53. The Samba Team, Samba. https://www.samba.org/ (1998). Accessed 23 Jan 2024
54. Oracle, MySQL. https://www.mysql.com/ (1999). Accessed 23 Jan 2024
55. akopytov, sysbench. https://github.com/akopytov/sysbench (2017). Accessed 23 Jan 2024
56. Fraunhofer FKIE, Malpedia. https://malpedia.caad.fkie.fraunhofer.de (2020). Accessed 19 May 2024
57. ABUSE, MalwareBazaar Database. https://bazaar.abuse.ch/ (2020). Accessed 19 May 2024
58. Conti (Malware Family), Malpedia. https://malpedia.caad.fkie.fraunhofer.de/details/elf.conti (2020). Accessed 26 May 2024
59. DarkSide (Malware Family), Malpedia. https://malpedia.caad.fkie.fraunhofer.de/details/elf.darkside (2020). Accessed 26 May 2024
60. Erebus (Malware Family), Malpedia. https://malpedia.caad.fkie.fraunhofer.de/details/elf.erebus (2020). Accessed 26 May 2024
61. HelloKitty (Malware Family), Malpedia. https://malpedia.caad.fkie.fraunhofer.de/details/elf.hellokitty (2020). Accessed 26 May 2024
62. Hive (Malware Family), Malpedia. https://malpedia.caad.fkie.fraunhofer.de/details/elf.hive (2020). Accessed 26 May 2024
63. Kuiper (Malware Family), Malpedia. https://malpedia.caad.fkie.fraunhofer.de/details/elf.kuiper (2020). Accessed 26 May 2024
64. LockBit (Malware Family), Malpedia. https://malpedia.caad.fkie.fraunhofer.de/details/elf.lockbit (2020). Accessed 26 May 2024
65. Monti (Malware Family), Malpedia. https://malpedia.caad.fkie.fraunhofer.de/details/elf.monti (2020). Accessed 26 May 2024
66. RansomEXX (Malware Family), Malpedia. https://malpedia.caad.fkie.fraunhofer.de/details/elf.ransomexx (2020). Accessed 26 May 2024
67. RansomEXX2 (Malware Family), Malpedia. https://malpedia.caad.fkie.fraunhofer.de/details/elf.ransomexx2 (2020). Accessed 26 May 2024
68. RedAlert Ransomware (Malware Family), Malpedia. https://malpedia.caad.fkie.fraunhofer.de/details/elf.red_alert (2020). Accessed 26 May 2024
69. REvil Ransomware (Malware Family), Malpedia. https://malpedia.caad.fkie.fraunhofer.de/details/elf.revil (2020). Accessed 26 May 2024
70. arm, Arm A-profile A64 Instruction Set Architecture. https://developer.arm.com/documentation/ddi0602/2022-06/SIMD-FP-Instructions/AESE--AES-single-round-encryption- (2022). Accessed 23 Jan 2024
71. Amazon Web Services, What is AWS Graviton? https://docs.aws.amazon.com/whitepapers/latest/aws-graviton-performance-testing/what-is-aws-graviton.html (2021). Accessed 23 Jan 2024
72. Google Cloud, Arm VMs on Compute. https://cloud.google.com/compute/docs/instances/arm-on-compute (2023). Accessed 23 Jan 2024
73. Cortex-A7 MPCore Technical Reference Manual r0p3, Breakpoints and watchpoints. https://developer.arm.com/documentation/ddi0464/d/Debug/Debug-register-interfaces/Breakpoints-and-watchpoints?lang=en (2013). Accessed 23 Jan 2024
74. Github, Linux kernel source tree. https://github.com/torvalds/linux/blob/master/arch/arm/kernel/hw_breakpoint.c (2023). Accessed 23 Jan 2024
75. Github, freebsd/freebsd-src. https://github.com/freebsd/freebsd-src/blob/main/sys/amd64/include/pcb.h (2023). Accessed 23 Jan 2024
76. Github, freebsd/freebsd-src. https://github.com/freebsd/freebsd-src/blob/main/sys/amd64/include/cpufunc.h (2023). Accessed 23 Jan 2024
77. Microsoft, PsSetCreateProcessNotifyRoutine function. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-pssetcreateprocessnotifyroutine (2022). Accessed 23 Jan 2024
78. Process Hacker, KProcessHacker/include/ntfill.h File Reference. https://processhacker.sourceforge.io/doc/ntfill_8h_source.html#l00308. Accessed 23 Jan 2024

79. CVE, CVE-2021-41773. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-41773 (2021). Accessed 23 Jan 2024
80. EVP Asymmetric Encryption and Decryption of an Envelope, OpenSSL Wiki. https://wiki.openssl.org/index.php/EVP_Asymmetric_Encryption_and_Decryption_of_an_Envelope (2017). Accessed 28 May 2024
81. The GnuTLS Transport Layer Security Library, GnuTLS. https://www.gnutls.org/ (2000). Accessed 28 May 2024
82. Crypto++ Library 8.9, Crypto++ project. https://cryptopp.com/ (2015). Accessed 28 May 2024
83. 11.5 Cryptographic Backend, GnuTLS. https://www.gnutls.org/manual/html_node/Cryptographic-Backend.html (2024). Accessed 28 May 2024
84. Crypto++ Library | 5.6.1 Release, Crypto++ project. https://cryptopp.com/release561.html (2010). Accessed 28 May 2024
85. Cozzi, E., Graziano, M., Fratantonio, Y., Balzarotti, D.: Understanding linux malware. In: Proceedings of the 39th IEEE Symposium on Security and Privacy (SP '18), pp. 161–175. IEEE (2018)
86. Ami, O., Elovici, Y., Hendler, D.: Ransomware prevention using application authentication-based file access control. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18), ACM, pp. 1610–1619 (2018)
87. TRAX, Data Center Storage, Capacity Planning and Requirements. https://www.traxindprod.com/data-center-storage-and-capacity-planning/ Accessed 06 Jun 2024