Analysis of Encryption Key Zeroization from System-wide Perspective

Toyofumi Sawa
Institute of Information Security
Kanagawa, Japan
dgs254101@iisec.ac.jp

Kuniyasu Suzaki
Institute of Information Security
Kanagawa, Japan
suzaki@iisec.ac.jp

***Abstract*—Cryptographic keys are required not to be duplicated unnecessarily and to be zeroized from memory after use, as mandated by ISO/IEC 19790 and FIPS 140-3. Because applications run on an operating system, verifying key zeroization solely through their interactions is difficult. Although methodologies for enforcing key zeroization have been studied, there is a lack of empirical research confirming that zeroization actually occurs in practice. To address this gap, we propose a live memory forensic technique employing an FPGA to periodically detect AES keys, which we then use to investigate the key zeroization behavior of applications that use AES under Ubuntu Linux (with a FIPS-compliant kernel) on both AArch64 and x86-64, as well as on Windows 11 (x86-64). Our investigation reveals that AES instructions hold round keys in registers, but during OS context switches these register contents are copied into memory and remain unzeroized when the process terminates. Moreover, when an application terminates abnormally, insufficient signal handling prevents key zeroization, and cryptographic keys can be included in core dumps potentially leaking them externally. We further observe that, on non-FIPS-compliant kernels, keys can persist across OS reboots. These findings provide valuable information for developers following key zeroization guidelines and for cryptographic module testers. Finally, we propose and supply developers with feasible mitigation strategies to reduce key duplication and persistence at both the application and library levels.**

***Index Terms*—AES, zeroization, key management, live memory forensics**

1. Introduction

Zeroization of secret information, including cryptographic keys, after use has long been a standard practice [1]. Key zeroization is considered important from the perspectives of defense-in-depth, forward secrecy, and certification [2]. From a defense-in-depth perspective, ensuring reliable key zeroization can reduce the likelihood of successful attacks.

Zeroization becomes a critical issue in terms of forward secrecy when a system compromise or memory disclosure occurs. If a key is not zeroized immediately and the time of zeroization is unknown, one cannot rule out that information protected by that key has been exposed when the system is compromised. Even if memory encryption is employed, from an operational perspective, the possibility of key remnants means that concerns about exploitation via other vulnerabilities remain; therefore, reliable implementation of zeroization is required. If the OS guarantees zeroization of released memory within a bounded time, then even if a process fails to zeroize its key, an upper bound on the key's lifetime can be established. Consequently, it is also beneficial to determine how long a non-zeroized key persists before it is ultimately zeroized.

In institutions that handle particularly sensitive information, such as government agencies and financial organizations, forward secrecy is highly prioritized, and standards have been established with corresponding certification schemes. For example, ISO/IEC 19790 [3] and NIST SP 800-140 [4] require zeroization of Sensitive Security Parameters (SSPs) even for Level 1 cryptographic modules, which represent the lowest security level, in order to mitigate such attacker-driven compromises. In ISO/IEC 19790:2012, three types of cryptographic modules are defined: software, hardware, and hybrid; this study focuses on software cryptographic modules.

Zeroization of cryptographic keys and secrets has been studied through static analysis at the source-code level [5], [6] and at the binary level, as well as through dynamic analysis [7]. Even if these methods guarantee key zeroization, that guarantee is limited to the period during which the application's or library's memory remains valid. Examination of the security policy documents for FIPS 140-3 certified cryptographic modules revealed that AlmaLinux 9 [8] states the following regarding abnormal termination: *"In case of abnormal termination, or swap in/out of a physical memory page of a process, the SSPs in physical memory are overwritten by the Linux kernel before the physical memory is allocated to another process."*

However, no such mention of abnormal termination was found in the FIPS 140-3 security policy documents provided by other vendors. This suggests that key zeroization when the OS terminates a process abnormally may not be adequately considered.

If cryptographic keys are duplicated in locations unknown to the application developer, zeroization of those keys

becomes difficult. It is well known that sensitive data may be copied through the interactions of various components (application, library, OS, hardware), beyond the memory explicitly allocated by the application developer. Chow et al. investigated the lifecycle of sensitive data and reported that application data is duplicated into various kernel buffers and that data not explicitly zeroized can persist for extended periods, sometimes surviving even after a reboot [9]. Dwoskin et al. used virtual machines to study the interactions among components and pointed out leaks that occur when function arguments are passed on the stack [10]. Furthermore, during interrupts, register contents may be spilled into memory, creating additional opportunities for leakage [11], [12].

Zeroization of cryptographic keys can only be considered complete once it has been verified on actual hardware that a solution incorporating various countermeasures [2], [5], [6] is operational. Indeed, under FIPS 140-3, certification is granted only after evaluating the entire system—including the Operating Environment. As noted above, although numerous scenarios can be envisioned in which an OS or hardware layer affects key handling, there is a lack of systematic research; consequently, application developers who utilize cryptography vary in their ability to address zeroization issues, leading to inconsistencies in both mitigation strategies and the quality of documentation when explaining how zeroization is enforced. To bridge this gap, we conducted a systematic empirical investigation of key zeroization failures that accounts for interactions among all system components. In this paper, we examine the real-world zeroization behavior of AES keys in applications running on Ubuntu Linux (with a FIPS-compliant kernel) on both AArch64 and x86-64 architectures, as well as in applications on Windows 11 (x86-64). To support the analysis of system behavior up to the OS reboot on physical hardware, we also developed the Periodic-AESKeyFinder, an FPGA-based tool that continuously measures the lifecycle of AES keys in physical memory. By combining this capability with various software tools, we were able to demonstrate key zeroization failures in actual deployments.

The contributions of this paper are as follows:

- Demonstration of scenarios in which immediate zeroization of keys that are no longer needed fails due to operating system behavior.
- Elucidation of the effects of both the operating system and hardware on memory zeroization during OS reboot, including kernels that comply with FIPS 140-3.
- Development of a support tool leveraging an FPGA to observe keys over the AXI bus, PCI Express, and Thunderbolt, running on both AArch64 and x86-64 owing to its instruction-set independence.

Our source code is available on GitHub: https://github.com/Tyojan/Periodic-AESKeyFinder

2. Threat Model and Assumptions

Figure 1 illustrates the threat scenarios considered in this study. In our threat model, we assume a powerful adversary

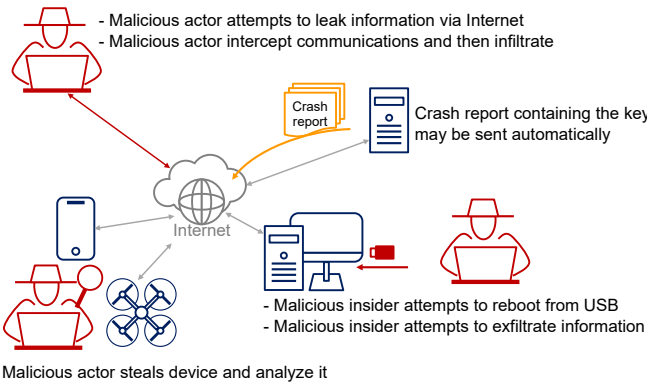


Figure 1: Threat model illustrating representative attacker scenarios, such as insider misuse, device theft, or memory-disclosure attacks. While we assume a powerful adversary, our focus is on key zeroization as a post-compromise factor essential to forward secrecy.

with physical access and the capability to execute arbitrary code, motivated to compromise the system. For example, an insider might boot from a USB device to run tools, or a stolen device could be analyzed offline.

The attacker possesses the necessary skills to exploit known vulnerabilities such as Heartbleed [13] or RAMBleed [14], and to perform various memory-disclosure attacks. Although various countermeasures can be considered for such compromises, insufficient key zeroization can leave residual keys after an attack, amplifying the scope and severity of damage. Key zeroization, therefore, plays a crucial role in estimating the impact of a compromise and guiding post-incident decisions. This study focuses on the problem of key zeroization as required by relevant standards in the context of forward secrecy.

In this study, we assume that during a process's normal lifetime, the process correctly manages its cryptographic keys. Therefore, zeroization of keys that become unnecessary during the process's lifetime is out of scope. Our focus is on interactions among system components at process termination and on key duplication by the kernel, which is difficult to detect by inspecting the process alone.

We employ an FPGA-based key scanning tool to assist our investigation; however, it should be noted that this is strictly a support mechanism. Configurations such as using an FPGA or disabling the IOMMU were performed for research purposes and do not represent assumptions about real-world attack scenarios. This study aims to elucidate the behaviors of the operating system and hardware that affect the zeroization of cryptographic keys.

3. Research Questions

In this section, we outline the research questions established to clarify the interactions that affect the lifecycle of cryptographic keys.

RQ1 When a process terminates normally, are the cryptographic keys in user space zeroized?
If the application or cryptographic library developer has implemented key zeroization correctly, keys should be zeroized upon normal termination. We will verify whether keys detected during process execution remain after the process has exited.

RQ2 When a process terminates abnormally, are the cryptographic keys in user space zeroized?
Even if key zeroization code is implemented, a process may be forcibly terminated at any time; thus, zeroization upon abnormal termination is not guaranteed. Applications that satisfy RQ**1** demonstrate developer intent to zeroize keys upon normal exit, but if a signal is received without a properly configured signal handler, zeroization may fail. Among the signals that cause process termination, all signals except SIGKILL are handleable; by implementing a corresponding signal handler, the application can perform zeroization (or other cleanup) after receiving the signal. Handleable signals include those whose default action is `terminate` (e.g., SIGTERM) and those that generate a core dump (e.g., SIGABRT). In this paper, we send SIGTERM and SIGABRT as representatives of these two categories. On Windows, since there is no built-in core-dump-generation mechanism, we exclude core-dump tests and instead attempt an immediate forced termination (analogous to SIGKILL) and a graceful forced termination with a delay (analogous to SIGTERM).

RQ3 Are cryptographic keys duplicated into memory regions that the process cannot read or write?
Process data may be copied to locations unknown to the application developer through interactions with the operating system. While a process holding cryptographic keys can explicitly zeroize keys in its own user space, if keys are duplicated outside of its user space—such as in kernel buffers or hardware-managed regions—the process may not be able to zeroize those copies. We will investigate whether such out-of-process copies of AES keys exist.

RQ4 Do residual keys persist after a system reboot?
According to FIPS 140-3, SSPs must not persist across a reboot [15]: *"All SSPs shall be zeroised prior to execution of the new image."* At FIPS 140-3 Level 1, procedural zeroization of software cryptographic modules—such as destruction of the module itself, e.g., by reentering the Earth's atmosphere—is permitted. If the reboot process (including firmware, operating system, and hardware actions) zeroizes memory, this is considered valid procedural zeroization. For example, Chow et al. reported that, in an emulator environment, the OS did not zeroize memory on reboot [9], but if actual hardware zeroizes memory, this would satisfy the requirement. However, the real-world behavior of hardware and OS combinations remains unknown.
When the OS shuts down, the kernel forcefully terminates processes—e.g., Linux sends SIGTERM to each process. This connects to RQ**2**: if a key persists after abnormal termination, it may survive across a reboot. Since it is not guaranteed that any component has explicit responsibility for zeroization beyond process termination, keys could survive past the reboot interval, allowing the next-run OS, firmware, or hardware to access leftover cryptographic keys. We will determine whether keys detected before shutdown persist after reboot.

4. Investigation Method

In this section, we describe the investigation methodology designed to address the research questions presented in Section 3. For RQ**1** (normal termination), conventional techniques such as taint analysis (e.g., K-Hunt [7]) are effective. However, to study RQ**2** (abnormal termination), one must first identify the memory locations of cryptographic keys and then configure debuggers within both user space and kernel space to observe whether zeroization occurs when a process is forcibly terminated.

For RQ**3**, one might consider using a system-wide emulator with taint analysis—such as TaintBochs [9]—to track all copies of sensitive data. Unfortunately, TaintBochs does not support multiple instruction set architectures (ISA), making it unsuitable for our target platforms (AArch64 and x86-64). Another option is to use virtual machines and take memory snapshots to detect kernel-level duplication, but this approach fails to reveal how duplicated keys are used after copying. Moreover, snapshot-based methods cannot observe the behavior before and after a system reboot, which is essential for RQ**4**.

Detecting key duplication in a live system is thus preferable, as it allows for the analysis of how and when duplicated data is used in real time. Additionally, because we aim to compare behavior across multiple operating systems and hardware environments, a reusable method for locating cryptographic keys in memory is desirable. To satisfy these requirements, we developed an FPGA-based live memory scanner (Periodic-AESKeyFinder) that can transparently identify AES key material on physical memory buses with minimal performance impact. This approach enables continuous monitoring of key presence and duplication across both user-space and kernel-space regions, as well as before and after system reboot, without reliance on emulators or VM snapshots.

As a method independent of ISA and operating system, one can consider leveraging an FPGA-based approach such as PCILeech [16]. Continuous key detection using an FPGA is advantageous for debugging, as it allows for the observation of key appearance and disappearance in real time. Although PCILeech can inspect physical memory and thus be used to search the entire memory for cryptographic keys duplicated by the OS, continuously outputting memory dumps and searching them for keys is inefficient.

Therefore, we propose Periodic-AESKeyFinder, which implements Halderman et al.'s AES key search tool (AESKeyFinder) [17], [18] on an FPGA and periodically scans memory to identify key addresses. Because this observation does not require the CPU, the target applications can run without interruption, and memory can be continuously monitored on actual hardware even after reboot—enabling investigation of RQ**4**. By simply executing the application, one can observe and verify in real time the generation, zeroization, and duplication traces of cryptographic keys. While Periodic-AESKeyFinder reports only the presence of keys, by combining it with CPU watchpoints, we built a complete system capable of investigating key persistence and zeroization timing at those addresses.

4.1. Overview

The overview of the proposed method is illustrated in Figure 2. In this paper, we aim to perform a comparative investigation across multiple ISAs, focusing on two architectures: AArch64 and x86-64. The AESKeyFinder IP core features an AXI master interface, enabling it to issue read requests to arbitrary physical addresses to search for keys. The blue arrows represent the path by which the CPU accesses main memory to perform AES computations, whereas the thick red arrows indicate the path by which the FPGA searches main memory for AES keys. Discovered keys are transmitted via Ethernet connected to the FPGA's pins to an analysis terminal. Periodic-AESKeyFinder detects only the presence of keys; identifying overwrites or accesses requires additional tools. In the following sections, we describe the details of Periodic-AESKeyFinder and the methods for investigating AES keys using peripheral tools.

4.2. Key Persistence Inspection Method Using Periodic-AESKeyFinder

The procedure for investigating key duplication and persistence through live memory forensics using Periodic-AESKeyFinder and peripheral tools is as follows. The ultimate objective is to identify the timing at which keys residing in physical memory are read or written.

1) Boot the target system and configure the FPGA with Periodic-AESKeyFinder.
2) After the operating system has started, launch the application under investigation.
3) Receive packets output by Periodic-AESKeyFinder (which contain key and physical address information) on an analysis terminal.
4) Based on the received packets, identify the process ID and the corresponding virtual address on the target system (details in Section 4.2.1).
5) Using the identified PID and virtual address, set watchpoints to monitor when the key is read or written (details in Section 4.2.2).

In the SoC environment, Periodic-AESKeyFinder is triggered by the bootloader. In the PC environment, it is triggered via a UEFI program, and a driver is loaded during kernel startup to ensure that PCI bus-master functionality remains enabled. Although these steps are complex, many can be executed via scripts, and we plan to fully automate the process in the future.

4.2.1. Method for Obtaining Process ID and Virtual Address from a Detected Key's Physical Address. On Linux, we identify the owner process and virtual address of a detected physical page by enumerating each candidate process's address map (`/proc/[pid]/maps`), translating those virtual addresses to physical addresses with `virt2phys` [19], and checking for a page match against the one reported by Periodic-AESKeyFinder. If the owner is unknown, we brute-force over all PIDs. When the key resides in kernel space, we recover the kernel virtual address via the direct-map offset from physical memory (e.g., by prefixing the physical address with `0xFFFF00` on our SoC, or `0xffff888` on PCs with KASLR disabled). On Windows, we use RAMMap (Sysinternals) to map a physical page back to its virtual address, owning process, and backing file.

4.2.2. Method for Determining Key Duplication and Zeroization Timing. We determine when keys are duplicated or zeroized by placing watchpoints on their virtual addresses (Section 4.2.1). For user space on Linux we use GDB; for kernel space on x86-64 we use kgdb; for AArch64 SoCs we use OpenOCD via JTAG; and on Windows we use WinDbg.

A write watchpoint indicates overwriting, which may be zeroization if the value is set to zero by functions such as `memset()`. A read watchpoint indicates that the key is accessed, either for cryptographic use or for duplication; backtraces allow us to distinguish between these cases. While a read watchpoint only captures the read side, the duplicate is subsequently detected by Periodic-AESKeyFinder, revealing the code responsible for writing it.

4.3. Design

Assumptions. Periodic-AESKeyFinder detects AES round keys derived from the original key on the FPGA; therefore, it assumes that the key schedule is laid out in memory immediately following the original key. To the best of our knowledge, regardless of whether a library uses AES instructions, key schedules are precomputed for performance and stored contiguously in memory in round order. Supporting various byte orders and alignments on the FPGA would significantly increase circuit complexity; thus, we support only 64-bit and 128-bit alignments and assume little-endian encoding in 8-byte increments. This matches the basic data handling of 64-bit CPUs.

Additionally, the key schedule may cross 4 KB page boundaries; however, those pages must be mapped to consecutive physical pages. Since such memory allocations are rare, we do not consider this case. This assumption is identical to that of the original AESKeyFinder [17], [18].

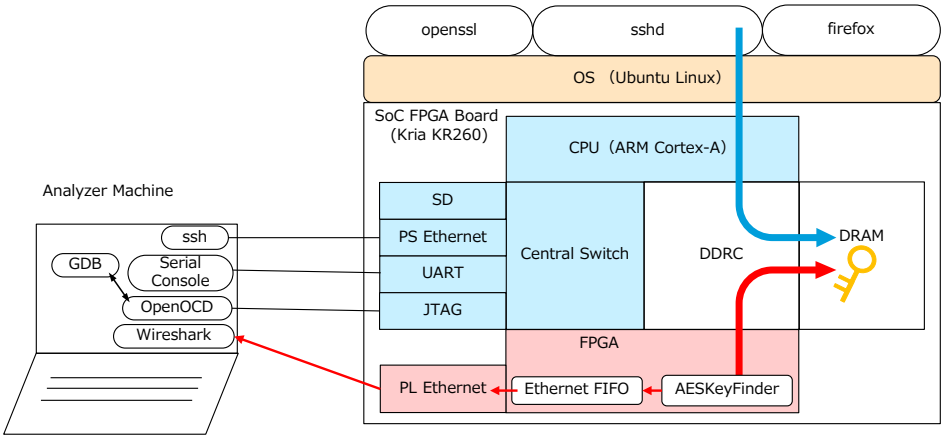


Figure 2: Detection and analysis method for AES keys using periodic live memory forensics with FPGA. Applications store keys in memory through the path indicated by the blue arrows. The AESKeyFinder implemented on the FPGA searches for keys in the memory shared with the CPU via the path indicated by the red arrows and outputs the results as packets through the PL (Programmable Logic) Ethernet on the FPGA side. On the analysis terminal, tools such as Wireshark are used to inspect the received packets to confirm the key values and physical addresses. OpenOCD or GDB can then be used to monitor the targeted keys.

Modifications. In our approach, we adapt the AESKeyFinder algorithm slightly and implement it on an FPGA to scan physical memory sequentially from the beginning, performing key detection. Original AESKeyFinder [17], [18] targets all rounds of the key schedule to extract keys from memory dumps that may contain bit errors. However, if the full set of round keys is not stored contiguously, their method cannot detect the key. Moreover, in live memory forensics, bit-error tolerance is unnecessary. Therefore, as shown in line 11 of Algorithm 1, we implement matching between any expanded round key derived from a candidate original key and data read from memory. For AES-128, Periodic-AESKeyFinder takes a 16-byte block read from memory as a candidate key and computes its key schedule using the key expansion function. If any of the ten derived round keys of the candidate key schedule are found within 256 bytes following the candidate key in memory, Periodic-AESKeyFinder identifies the candidate as an original key. For example, if the first-round key and the seventh-round key both appear within a certain range, we classify the data as a valid AES key based on those partial matches. Please refer to Appendix B for this example.

Advantages of the Proposed Method. This live memory forensics technique leverages an FPGA to operate directly on physical addresses, enabling examination of nearly the entire RAM and detection of AES keys processed by the CPU without CPU involvement. Although the CPU is used to determine virtual addresses as described in Section 4.2.2, no CPU intervention is required until key detection. As a result, target applications can execute at high speed on real hardware, offering superior real-time performance compared to VM or emulator-based methods, which incur greater execution overhead. Overhead and potential missed detections are discussed in Section 4.5.

Algorithm 1 Algorithm of P(eriodic)-AESKeyFinder. This IP core reads 128 bits from memory via the AXI bus and treats it as a candidate key to compute the candidate key schedule. To improve detection rate, line 11 checks whether any round keys from the candidate key schedule are found in the recently read data.

```
1:  procedure P-AESKeyFinder(ReadAddr, ReadRange)
2:      initialize ReadQueue[0 : 15]
3:      CandAddr ← ReadAddr
4:      while true do
5:          for i = 0 to 14 do
6:              ReadQueue[i] ← ReadQueue[i + 1]
7:          end for
8:          ReadQueue[15] ← Read128bitViaAXI(CandAddr)
9:          CandKey ← ReadQueue[0]
10:         KeyScheCand[10 : 1] ← AESKeySchedule(CandKey)
11:         if KeyScheCand[10 : 1] ∩ ReadQueue[15 : 1] ≠ ∅ then
12:             output(CandKey, CandAddr)
13:         end if
14:         if CandAddr < ReadRange then
15:             CandAddr ← CandAddr + 1
16:         else
17:             CandAddr ← ReadAddr
18:         end if
19:     end while
20: end procedure
```

4.4. Experimental Setup

The target platform are shown in Table 1. In the SoC environment, we used an Ubuntu 22.04 root filesystem [20]. In the PC environment, we used both Ubuntu 22.04 and Windows 11. On the PC environment's Ubuntu, we ran both a FIPS-compliant kernel and a non-compliant kernel, but observed no differences in results.

TABLE 1: Target Platforms

Environment	**SoC**	**Intel Desktop**	**AMD Desktop**	**Laptop**
FPGA Board	Xilinx KR260	ALINX AXKU062	ALINX AXKU062	ALINX AXKU3
CPU	Cortex-A53	Core i5-14500	Ryzen 5 8500G	Core i3-1315U
Memory	4 GiB DDR4	32 GiB DDR5	32 GiB DDR5	8 GiB LPDDR5
Motherboard	—	ASUS PRIME B760M-A	ASUS PRIME B650M-A II-CSM	ThinkPad X13(Gen 4)
Firmware/BIOS ver.	U-Boot 2022.01	1002	1807	N3OET35W
FPGA Connection	On-chip AXI bus (128 bit width, up to 333 MHz)	PCIe 3.0 ×4	PCIe 3.0 ×4	Thunderbolt 4 (PCIe Tunneling)

4.5. Evaluation of Periodic-AESKeyFinder

In this section, we discuss the memory scan speed of Periodic-AESKeyFinder, detection misses, and false positives.

4.5.1. Scan Speed. When the Periodic-AESKeyFinder IP core reads memory over a 128-bit AXI bus at 250 MHz, the theoretical maximum throughput is 4 GB/s. In the SoC environment with 4 GiB of memory, the scan completes in 1.08 seconds, achieving the theoretical throughput. During this interval, multiple packets corresponding to detected keys may be generated, occasionally causing packet loss; however, this does not pose a practical problem. In the Intel Desktop environment with approximately 32.2 GiB of RAM, the scan period is 8.05 seconds, which also matches the theoretical throughput. A small portion of memory is reserved by the system and mapped to non-contiguous addresses, but these regions are not accessed. In the laptop environment, which accesses memory over Thunderbolt, approximately 5.9 GiB of memory is scanned in a 2.06 second cycle, corresponding to a throughput of 2.85 GB/s. The reduced throughput compared to the AXI bus's theoretical maximum is due to the Thunderbolt controller acting as a bottleneck. The overhead caused by high-speed memory access is shown in the Appendix C, but no significant impact was observed.

4.5.2. Detection Misses. Keys with a lifetime shorter than the scan cycle may go undetected. To evaluate this, we generated short-lived TLS sessions performing TLS 1.2 handshakes that established AES-128 session keys, and observed these keys to evaluate the detection rate.

The experiment was conducted on the SoC platform by running `openssl s_server` and establishing sessions from a separate client. We defined the key lifetime as the time between the Client Key Exchange and the final ACK of session termination. A script was created to run the client process for a specified duration: after establishing a session, it automatically disconnected after the given time. The key lifetime was varied from 0.1 to 1.5 seconds and repeated multiple times. We also tested the case where the client disconnected immediately after session establishment. The results are shown in Figure 3.

When the client disconnected immediately, the key lifetime was measured as 1.77 ms. Across 20,000 sessions,

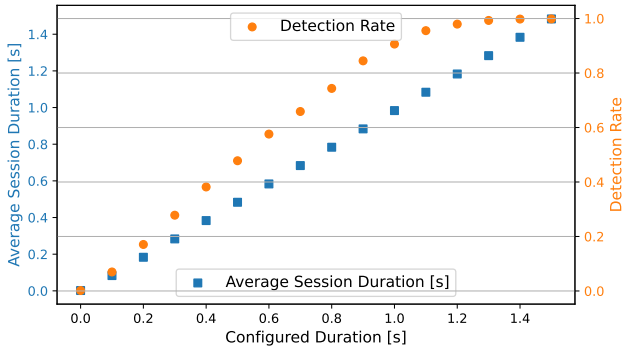


Figure 3: Key lifetime vs. detection rate in TLS. Longer key lifetimes result in higher detection rates, and detection approaches nearly 100% once the lifetime exceeds the scan cycle (about 1.08 seconds).

65 keys were detected. Since TLS uses different keys for upstream and downstream, the detection rate was 65 out of 40,000, or about 0.16%. Given the scan cycle of about 1.08 seconds in the SoC environment, the ratio of key lifetime to the scan cycle is also approximately 0.16%, showing close agreement.

In general, the detection rate can be approximated as

$$R_{\text{det}} \approx \frac{T_{\text{key}}}{T_{\text{scan}}}$$

where R_{det} denotes the detection rate, and T_{key} and T_{scan} denote the key lifetime and the scan cycle of *Periodic-AESKeyFinder*, respectively.

We also conducted supplementary experiments using the `openssl` command to encrypt and decrypt files, treating the process runtime as the key lifetime. The results followed the same proportional relationship between runtime and detection rate. Details are provided in the Appendix D.

Since our goal is to detect keys that remain in memory for a long time without being zeroized, the inability to detect short-lived keys is not a concern. Note that the scan cycle duration increases proportionally with memory size. For example, scanning 256 GB of RAM would take approximately 64 seconds. Even a several-minute scan cycle is sufficient for detecting keys that persist for hours.

4.5.3. False Positives. If memory contains a sequence like `0x0000...` followed by `0x6363...`, a false positive may occur due to AES S-Box substitution of `00` $\rightarrow$ `63`. In such cases, the key will be reported as `0x000...`, and Periodic-AESKeyFinder will include metadata such as the number of detected key schedule rounds, indicating only round 1 and its derivative. Fortunately, such false positives were rarely observed during extended experiments and were considered negligible.

4.5.4. Impact of AESKeyFinder Algorithm Modification. As described in Section 4.3, the Periodic-AESKeyFinder can detect key schedules even when only part of the derived round keys together with the original key are present, which in some cases allows it to identify more keys than the conventional tool. In the case of full schedule detection, the number of keys found by the Periodic-AESKeyFinder should match that of the original AESKeyFinder [17]. However, since partial detection accepts incomplete key schedules with missing round keys, it can succeed in cases where the original tool cannot. To evaluate this, we obtained full physical memory dumps on both SoC and Intel Desktop environments, and compared the number of unique keys (defined as the combination of memory address and key value) detected by Halderman's AESKeyFinder (`aeskeyfind` command) with those detected by the Periodic-AESKeyFinder during the memory dump acquisition.

The number of keys detected during memory dump acquisition is shown in Table 2. In this table, "Original" represents the number of keys detected from the memory dump by Halderman's AESKeyFinder. "Full" represents the number of complete key schedules detected by the Periodic-AESKeyFinder, and "Partial" corresponds to the number detected from incomplete schedules. "Measurement time" indicates the duration required to obtain the memory dump, during which the Periodic-AESKeyFinder was simultaneously running. On Ubuntu, we executed Firefox and SSH, and obtained a physical memory dump by copying `/dev/mem` with `dd`. On Windows, we executed Edge and SSH, and obtained the physical memory dump generated when a BSOD was triggered.

In all cases, the original AESKeyFinder did not produce any partial detections. All duplications caused by context switches, as shown in Section 5, were discovered through partial detection, demonstrating the effectiveness of the modification.

In the case of Intel Desktop running Windows, the original tool achieved better results in detecting full AES-256 schedules. This is because the Periodic-AESKeyFinder assumes 8-byte and 16-byte alignments, while the original AESKeyFinder can also detect schedules in other alignments. The 13 keys responsible for the difference were found at offsets not assumed by the Periodic-AESKeyFinder. On Ubuntu, only 8-byte or 16-byte aligned keys were present.

In the SoC environment, memory dump acquisition required a long time, leading to longer execution times. When memory dumps are obtained live while the system is running, keys with shorter lifetimes than the dump process can be missed. In contrast, on Windows systems, where dumps are captured during a BSOD hang, the keys remain stable, resulting in higher full-schedule detection rates by the original tool.

4.6. Limitations

To decrypt encrypted data, an initialization vector (IV) is required. However, Periodic-AESKeyFinder does not retrieve the IV, so decryption cannot be performed. Nevertheless, because implementations typically place the IV near the key schedule in memory, once the key's address is known, it may be possible to obtain the IV by calculating its relative offset.

On systems with an IOMMU, the IOMMU must be disabled via OS or UEFI firmware settings. Memory encryption and PCIe data scrambling features must also be disabled. However, since these configuration changes do not affect the zeroization behavior, they are not considered problematic.

Although one might argue that keys other than AES should also be supported, the objective of this study is to investigate the impact on zeroization, and thus we consider it appropriate to focus solely on the widely used AES. Halderman et al. also proposed an RSA key detection method called RSAKeyFinder [17], [18]. RSAKeyFinder detects RSA keys in the PKCS#1 (RFC 8017) format [21] by identifying the ASN.1 Distinguished Encoding Rules (DER) representation of the key format, searching for the hexadecimal sequence at the beginning of the key structure. In addition to the PKCS#1 format detected by RSAKeyFinder, we extended support to other DER-encoded formats, namely the ECDSA key format of RFC 5915 [22] and the Asymmetric Key Packages of PKCS#8 (RFC 5958) [23], but were unable to detect as many keys as with AES. McLaren et al. have proposed a ChaCha20 key detection method [24]. Furthermore, for Base64-encoded keys (such as those using ASCII Armor), one could search for high-entropy encoded data to locate the keys.

Asymmetric ciphers such as RSA are considered difficult to implement in our framework due to both the circuit scale required and the diversity of implementations. Further implementation details, including FPGA resource utilization, scalability analysis, and engineering effort, are provided in Appendix E.

5. Empirical Demonstration of AES Key Zeroization Failures

In this section, we conduct experiments on common applications to investigate scenarios in which key zeroization fails in order to derive insights for mitigation strategies. Table 3 summarizes the target applications and the persistence or duplication of cryptographic keys at the time each process terminates. In the table, ✘indicates a failure case in which a key was detected (i.e., zeroization failure), and

TABLE 2: Detection results of AES key schedules using the original AESKeyFinder [17] "Original" and the modified Periodic-AESKeyFinder. "Full" indicates full schedule detection, while "Partial" allows detection from incomplete schedules. The rightmost column shows the measurement time in seconds.

	AES-128			AES-256			
		Periodic-AESKeyFinder			Periodic-AESKeyFinder		Measurement time [s]
Environment	Original [17]	Full	Partial	Original	Full	Partial	
SoC (Ubuntu)	169	655	3	35	93	15	420
Intel Desktop (Ubuntu)	105	164	0	142	177	0	33
Intel Desktop (Windows)	22	22	0	105	92	3	43

✔indicates that no key was detected after process termination. The applications were selected based on availability and the diversity of their dependent cryptographic libraries. Below, we describe the results corresponding to the research questions presented in Section 3, using the OpenSSH case study illustrated in Figure 4 and Figure 5 as examples.

5.1. Key Zeroization upon Normal Application Termination (RQ1)

Figure 4 shows the timeline of session-key generation to zeroization observed by Periodic-AESKeyFinder for `sshd` in the SoC environment. The two keys highlighted against a white background correspond to the user-space keys of `sshd`, and they are confirmed to vanish at the EXIT event. In the OpenSSH experiment, we launched an SSH session specifying AES-128-CTR. After typing "`exit`" in the terminal to terminate the session, the user-space keys were no longer observed. For the OpenSSL command, we performed file encryption and decryption and waited for process completion to constitute a normal termination. To determine whether the phenomenon observed in OpenSSH was application-specific, we also investigated the OpenSSL command, which uses the same cryptographic library. In the browser experiment, we browsed a streaming video site to generate continuous encrypted traffic and observed the session key used by TLS with AES-128-GCM. For all of these applications, once a virtual address corresponding to an AES key was identified via Periodic-AESKeyFinder, a watchpoint was set at that address; in each case, zeroization occurred at process termination.

Among the applications under investigation, only 7-Zip failed to zeroize its key. Although we identified the key's address and set a watchpoint, the watchpoint was not triggered upon process exit, and the AES key continued to be detected afterward. Since 7-Zip is the only application in our set that implements encryption natively, it is likely that its developers did not consider zeroization.

5.2. Key Zeroization upon Abnormal Application Termination (RQ2)

Figure 5 shows the key detection timeline when SIGABRT is sent to `sshd`. The two keys labeled Addr1 and Addr2 on a white background correspond to `sshd`'s user-space keys. Because these keys persisted after the process terminated due to SIGABRT, we conclude that they were not zeroized by the signal handler (RQ**2**). Key persistence after abnormal termination was observed in all combinations of target applications and setups. In the timeline shown in Figure 5, the key at Addr1 remained in memory for up to 1635 seconds after process termination before being zeroized. Multiple experiments revealed no consistent pattern in persistence time, which ranged from a few seconds to several hours.

5.3. Duplication of Detected Cryptographic Keys (RQ3)

During the above investigation, two types of duplication were observed. The key appearing at Addr3 in Figure 4 and Addr5 in Figure 5 (highlighted in yellow) was determined, using the method described in Section 4.2.1, not to be a user-space key of `sshd`. This key was found to have been duplicated into kernel space via the context switch described in Section 5.3.1. Additionally, the keys at Addr3 and Addr4 in the middle of Figure 5 (highlighted in gray) began to be observed at the time of SIGABRT. These correspond to keys included in the core dump, as discussed in Section 5.3.2.

5.3.1. Duplication of Keys via Vector Register Context Switch. When AES keys are stored in vector registers and a context switch occurs, the AES keys are saved from the CPU's vector registers into kernel memory areas that maintain the process context. Using the method described in Section 4.2, we investigated the keys that appear at process creation in the lowest panels of Figures 4 and 5. Vector registers hold cryptographic keys, plaintext/ciphertext data, and results of AES instructions. Upon a scheduler-initiated context switch on AArch64 Linux, the values in vector registers are saved in `thread_struct.user_fpsimd_state.vregs` within the `task_struct` structure. On x86-64 Linux, they are saved starting at `task_struct.thread_struct.fpu.__fpstate.regs.xsave.i387.xmm_space` and onward. The keys duplicated by context switching were found in these memory regions.

Note that, although we did not observe key duplication due to context switches on the PC environment for our test

TABLE 3: Target Software and Key Management Results

Project	Library	OS	Version	Ex	KILL	TERM	CORE	Ctx	Dump
OpenSSH	OpenSSL	Ubuntu 22.04 (AArch64)	8.9p1	✔	✘	✘	✘	✘	✘
OpenSSH	OpenSSL	Ubuntu 22.04 (x86-64)	8.9p1	✔	✘	✘	✘	✔	✘
OpenSSH	LibreSSL	Windows 11 (x86-64)	9.6p1	✔	✘	-	-	✔	-
OpenSSL	OpenSSL	Ubuntu 22.04 (AArch64)	3.0.2	✔	✘	✘	✘	✘	✘
OpenSSL	OpenSSL	Ubuntu 22.04 (x86-64)	3.0.2	✔	✘	✘	✘	✔	✘
Firefox	LibNSS	Ubuntu 22.04 (AArch64)	110.0.1	✔	✘	✘	✘	✘	✘
Firefox	LibNSS	Ubuntu 22.04 (x86-64)	130.0.1	✔	✘	✘	✘	✔	✘
Chromium	BoringSSL	Ubuntu 22.04 (AArch64)	129.0.6668.89	✔	✘	✘	✘	✔	-
Chromium	BoringSSL	Ubuntu 22.04 (x86-64)	131.0.6778.139	✔	✘	✘	✘	✔	-
Edge	Schannel	Windows 11 (x86-64)	131.0.2903.112	✔	✘	✘	-	✔	-
7-Zip	Original	Windows 11 (x86-64)	24.09	✘	✘	✘	-	✔	-

✔ indicates that the key was zeroized and not detected. ✘indicates that the key was not zeroized and was detected. –: Not applicable. Ex: normal exit; KILL: immediate non-catchable termination; TERM: graceful termination; CORE: termination with core dump; Ctx: duplication via context switch; Dump: key in core dump. No difference in key remanence regardless of whether the OS's FIPS mode was enabled or disabled.

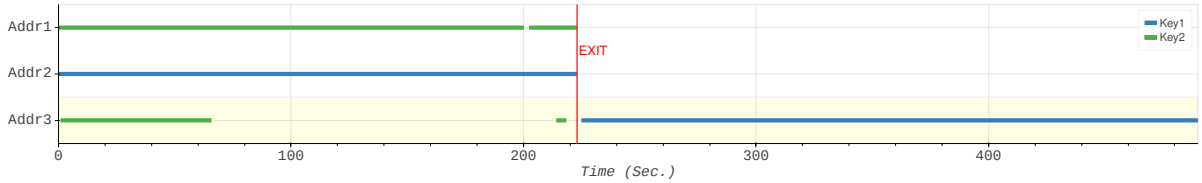


Figure 4: Session keys (Addr1 and Addr2) detected during execution of `sshd`. Addr3 (yellow background) was detected in the kernel context switch save area, indicating duplication.

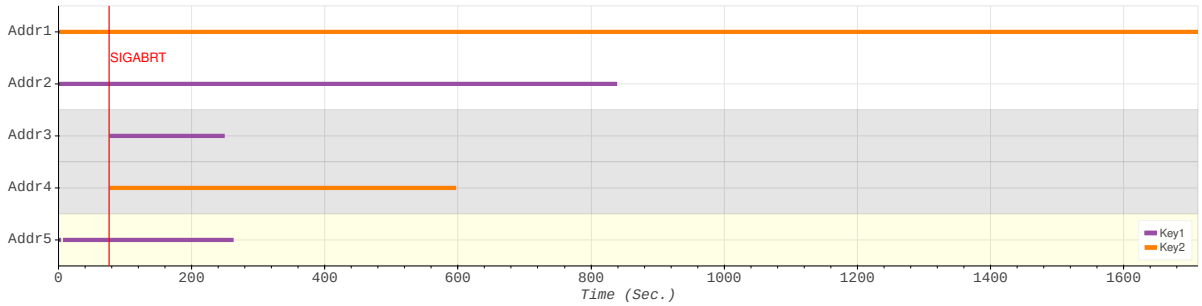


Figure 5: Key detection timeline after sending `SIGABRT` to `sshd`. Addr3 and Addr4 (gray background) appeared only after the signal and correspond to keys extracted from core dumps. Addr5 (yellow background) was duplicated via context switch.

applications, such duplication can potentially occur. On x86-64, prior to AVX2, there are only 16 vector registers, so registers must be reused, causing the original key to be overwritten before it can be duplicated, making detection impossible. On AArch64, there are 32 vector registers, so no reuse is necessary, and keys can be duplicated without being overwritten, allowing detection. In AArch64 Chromium, however, the register holding the original key is overwritten by subsequent round keys, preventing detection.

After a key is duplicated into the `task_struct`, we set a watchpoint on the corresponding kernel-space address and terminated the process to check for zeroization. Because the watchpoint was not triggered at process exit, we conclude that keys duplicated via context switching are not zeroized upon process termination. The keys duplicated by context switching persisted for some time even after the EXIT event, as indicated by the red vertical line.

5.3.2. Reading Core Dumps via Crash-Reporting Tool. When a process terminates abnormally, the Linux kernel generates a core dump and writes it to the destination specified by `/proc/sys/kernel/core_pattern`. On Ubuntu, this output is piped by default into `apport`'s standard input. `apport` is Ubuntu's crash-reporting tool. Crash reports include core dumps to provide information to distribution developers. During crash report generation, the core dump file was memory-mapped and detected by Periodic-AESKeyFinder. Although it is known that core

TABLE 4: Persistence of Keys After Reboot in Various OS Environments

OS	WB	KC	RS
Linux-5.15.0-1038-xilinx-zynqmp (SoC)	✘	✘	✘
Linux-5.15.0-130-fips (Intel Desktop)	✔	✔	✔
Linux-6.8.0-50-generic (Intel Desktop)	✘	✔	✔
Linux-6.8.0-50-generic (AMD Desktop)	✔	✔	✔
Linux-6.8.0-50-generic (Laptop)	✘	✔	-
Windows 11 Build 22621 FIPS (Intel Desktop)	✘	✔	✔
Windows 11 Build 22621 (Intel Desktop)	✘	✔	✔

WB: Warm Boot, KC: After Kernel Crash, RS: Reset Switch. ✔: Memory was zeroized after reboot. ✘: Keys persisted in memory after reboot. FIPS-compliant Linux zeroizes memory during reboot; other systems do not. The AMD desktop environment did zeroize memory on reboot.

dumps can contain sensitive data, this finding captures part of the process by which keys in the core dump are automatically converted into the crash report, highlighting the risk that they may be transmitted externally. Chromium and the Windows applications did not exhibit crash-report generation behavior, so they were excluded.

5.4. Key Persistence after OS Reboot (RQ4)

In this section, we reboot each target OS to investigate whether cryptographic keys persist after reboot. Rather than specifying a particular key for observation, we monitored whether keys detected during OS operation could still be observed post-reboot. The operating systems under test are listed in Table 4.

We employed three reboot methods. The "Warm Boot" shown in Table 4 refers to a standard reboot initiated via GUI actions or the `reboot` command in a terminal. For "After Kernel Crash," on Linux we induced a kernel panic by writing to `/proc/sysrq-trigger`, and on Windows we triggered a BSOD by forcibly terminating `wininit`. Additionally, we performed a reboot by pressing the Reset Switch.

Our investigation revealed that only the FIPS-compliant kernel zeroized memory during a normal reboot. In Table 4, ✔ indicates that no key was detected by Periodic-AESKeyFinder after reboot, confirming memory zeroization in those cases. The laptop does not have a reset switch, so it is excluded from consideration. In the SoC environment, keys persisted regardless of the reboot method, whereas on the AMD desktop environment, memory was zeroized for every reboot method. We verified memory zeroization by using the FPGA's Integrated Logic Analyzer (ILA) to confirm that data on the AXI bus was all zeros when Periodic-AESKeyFinder read from memory. ✘ indicates that a key was detected by Periodic-AESKeyFinder.

6. Discussion

6.1. Causes of Zeroization Failures

The fact that Linux does not zero out memory belonging to terminated processes has remained unchanged since the study by Chow et al. [9]. In contrast, there have been divergent claims regarding Windows: Malliaros et al. [25] assert that Windows immediately zeroizes process memory upon termination, whereas Li et al. [7] report the opposite. In our investigation of Windows environments, we observed that keys persist in memory even after process termination. We attribute the differing conclusions in prior work to variations in memory availability. In experiments with 7-Zip, even when encrypting and decrypting a file large enough to fill memory, most of the memory was occupied by memory-mapped files, leaving little free space; consequently, freed pages were promptly reused and zeroization occurred.

Sometimes it is assumed that an operating system immediately zeroizes memory once it is no longer needed, but in reality this is not the case. Freed pages are managed through "zeroed" and "free" page lists, and the timing of zeroization is not deterministic. If zeroization were performed at every deallocation, additional CPU cycles would be consumed and overall system performance could degrade. Moreover, during the zeroization process the target pages are brought into the cache, incurring extra performance costs due to cache pollution. Modern CPUs provide instructions that accelerate memory zeroization, but these cannot eliminate the cache-related overhead. Thus, immediate zeroization of all freed pages is not practical from a performance standpoint. For this reason, many operating systems adopt a lazy zeroing strategy, in which pages are zeroized only when they are reallocated. This approach represents a trade-off between performance and security, but it also means that sensitive data may remain in memory for a certain period of time. In our experiments, we observed cases where watchpoints did not hit until the page was reused, and only then did we detect the zeroization.

Among abnormal termination scenarios, zeroization failures when a process ends due to a handleable signal (e.g., SIGTERM) are the responsibility of the application, since the operating system provides an opportunity for key zeroization before process exit. In the threat model of the applications we studied, developers may trust the OS and not implement zeroization in signal handlers; demonstrating that missing signal handlers result in keys not being zeroized is thus a valuable finding.

Under FIPS 140-3 certification, it is required that keys not persist after an OS reboot; hence, in FIPS-certified systems, the kernel is presumed to include special routines to zeroize memory. In PC environments, we found it surprising that, although non-FIPS kernels do not zero out memory during a normal reboot, memory was zeroized after a crash-induced reboot and after pressing the reset switch. This suggests that memory may be zeroized either by hardware reset or by firmware interventions.

Even in operating systems configured for FIPS-compliant mode, we observed differing behaviors: on Ubuntu Linux, memory was zeroized after reboot, whereas on Windows, AES keys could still be detected. On Ubuntu Linux, we confirmed that the shutdown process employs a simple strategy of overwriting all memory with zeros in its final stage. Conversely, on Windows, keys persisted; with FIPS mode disabled, keys were detected across approximately 25 pages, whereas with FIPS mode enabled, they were detected across approximately 15 pages. Repeated reboot experiments showed a tendency for fewer pages to contain residual keys when FIPS mode was active. We therefore infer that keys from applications using non-FIPS-compliant cryptographic libraries persist in memory. Identifying all such applications remains a considerable task and is left for future work.

6.2. Mitigations for Zeroization Failures

We describe mitigations for each cause of key persistence.

6.2.1. Mitigation for Key Persistence upon Signal-Induced Termination. SIGKILL cannot be caught, so when a process is terminated by SIGKILL, memory contents may persist, and this cannot be addressed at the user-space level. The kernel should be responsible for immediately zeroizing the memory used by a process terminated by SIGKILL. As the occurrence of SIGKILL is not expected to be frequent, we believe it is acceptable to prioritize security over performance and perform immediate zeroization in this case. For handleable signals, the kernel provides the application with a grace period before termination; therefore, the application should implement a signal handler to perform zeroization in response to such signals.

6.2.2. Mitigation for Core Dump Leakage. The duplication observed in memory due to the crash-report tool reading the core dump is caused by the inclusion of keys in the core dump. The risks associated with core dumps are well-known [26], and system administrators can mitigate this by suppressing core dumps. However, since administrators may not always configure such settings, it is preferable for applications or libraries to ensure that memory regions containing keys are not included in core dumps. Specifically, this can be achieved by specifying `MAP_NOCORE` in the `mmap` system call or `MADV_DONTDUMP` in the `madvise` system call. These flags force the operating system to exclude the designated memory from core dumps. OpenSSH adopts the `mmap` approach [27]. On Windows, core dumps are referred to as crash dumps, and the function `WerRegisterExcludedMemoryBlock` can be used to prevent keys from being included.

6.2.3. Mitigation for Duplication via Context Switch. From the perspective of cryptographic library users, portability across diverse environments implies that developers may not have the freedom to choose a specific kernel or CPU. Therefore, mitigation should be implemented at the library level. One straightforward approach is to XOR-zeroize the vector registers that hold the AES round keys immediately after each cryptographic operation. This functionality has already been studied [2], and compiler options such as `-fzero-call-used-regs` that zeroize registers are also available.

On the kernel side, one could consider encrypting the register context during every context switch and zeroizing it upon process termination. However, encrypting the register context on each switch imposes significant overhead. Xu et al. [12] encrypt the contents of vector registers on interrupts by introducing and utilizing dedicated instructions. In x86-64 environments, context switching is performed with a single `xsave` instruction; supporting encryption within `xsave` would require modifications by the CPU vendor. Implementing this mitigation at the ISA level is therefore non-trivial and not easily achievable. Intel Key Locker [28] provides hardware support to store AES round keys as encrypted data in memory, decrypting them internally when AES instructions are executed. While Key Locker effectively protects AES round keys, it is limited to AES and requires specific CPU models, making it unavailable on many platforms.

Since the kernel is responsible for duplicating the keys during context switches, one might argue that it should also be responsible for zeroizing those duplicates. Implementing register-context zeroization at process exit within the kernel is relatively straightforward, and since the task structure and related information are often already loaded into the cache at that point, adding zeroization of the register context area would not impose a significant overhead. As the target region is limited in size, the performance impact is small, making this a practically feasible countermeasure.

6.3. Can these findings be applied to other keys?

Although this study focused on AES keys, we believe that similar results apply more broadly. Abnormal process termination and the absence of signal handlers are not problems unique to AES. The issue of core dumps being transmitted externally as crash reports likewise extends beyond AES keys; users may unknowingly allow core dumps containing various sensitive data—such as credit card numbers—to be sent. Duplication via context switches may be easier to detect for short symmetric keys (e.g., AES) than for longer asymmetric keys (e.g., RSA). Because these considerations affecting zeroization have not been clearly documented, developers have likely remained unaware of them.

7. Related Work

The tools CryptoGuard [29], CRYLOGGER [30], and CrypTody [31] monitor parameters passed to cryptographic APIs to detect cryptographic misuse, such as the use of insecure parameters. They target misuses within the scope of application developers, for example, employing outdated

algorithms like SHA-1. However, they do not focus on key zeroization, which demands specialized knowledge of cryptographic internals and system-wide behavior.

Research aimed at detecting cryptographic implementations—such as Aligot [32] and Crypto-DFG [33]—is driven primarily by malware analysis and binary reverse engineering. Tools like Cryptohunt [34] and Dispatcher [35] do not perform key detection. CipherXRay [36] and Kerckhoffs [37] perform key detection, and K-Hunt can additionally detect key zeroization. K-Hunt [7] employs dynamic analysis to identify fundamental cryptographic properties—namely randomness and repetition—from binaries, thereby recognizing various cryptographic implementations and flagging insecure keys. Although K-Hunt is a powerful analysis tool, its scope is limited to the cryptographic implementation itself, observing only memory behavior within the application's user space. In contrast, our study focuses on examining system-wide interactions that occur during live cryptographic operations.

AES key forensics is based on the work of Halderman et al. with AESKeyFinder [17]. By slightly modifying the algorithm to target partial key schedules, we were able to detect context held in vector registers. As a countermeasure to AESKeyFinder, Mohamed et al. proposed a technique to avoid detection by scattering the AES key schedule across memory [38]. However, even if the key schedule is scattered, if AES operations utilize vector registers, one can still detect the key schedule from the vector register context, as we have demonstrated.

Gross et al. were the first to implement AESKeyFinder on an FPGA [39]. They obtained a one-time dump of a protected memory region within the SoC FPGA's TrustZone and searched for the key schedule therein to extract the AES key, using this method to bypass secure boot. In contrast, the FPGA implementation of Periodic-AESKeyFinder continuously scans the entire physical memory and outputs detected keys via Ethernet, enabling investigation of the key's lifecycle.

In the field of forensics, there are scenarios where one may want to extract cryptographic keys used by ransomware [40] or decrypt malware communications [41]. Our findings show that AES keys can remain in memory even after a process terminates and that partial key schedules can persist in vector register contexts, providing valuable information for forensic analysts. Moreover, because Periodic-AESKeyFinder operates on an FPGA, it has the advantage of being undetectable by malware anti-analysis techniques when used as a support tool for ransomware analysis.

Finally, Table 5 compares our approach with cryptographic misuse detection and system analysis tools. Most taint analysis tools, such as K-Hunt [7] and Secretgrind [42], can only analyze an application's user space. A major strength of K-Hunt is that it does not require cryptographic templates (e.g., the algebraic relationships or encodings of the AES key schedule). In terms of observing system-wide interactions between applications and the OS, TaintBochs [9] is one example; however, by monitoring physical memory, we also observe reset-related behavior on actual hardware. Methods that use Pin for taint analysis, like K-Hunt, can only be applied to Intel CPUs, making them unsuitable for comparative studies across multiple platforms. Our method is limited to AES, but it is instruction-set independent. Comparing the operation of the entire system—including hardware—and the zeroization results across multiple platforms represents a novel contribution.

TABLE 5: Comparison with the related works

Tools	C1	C2	C3	C4	C5	C6	C7
K-Hunt [7]	✔	✔	✔	✘	✘	✘	✘
Secretgrind [42]	✘	✘	✔	✘	✘	✘	✘
TaintBochs [9]	✘	✘	✔	✔	✔	✘	✘
Cold boot [17]	✘	✔	✘	✘	✔	✔	✔
Ours	✘	✔	✔	✔	✔	✔	✔

C1: No need of crypto template **C2**: Detecting crypto key
C3: Detecting keys not zeroized at process termination
C4: Detecting keys overwritten after process termination
C5: Detecting keys after OS reboot **C6**: Extracting keys after reboot on real machine **C7**: Supporting multiple ISA

8. Conclusion

Cryptographic keys that have exceeded their intended usage period must be zeroized. Even if developers handle cryptographic keys with the explicit intention of zeroizing them, in practice, interactions among various system components can cause keys to be duplicated or remain in memory. The results of our investigation using the proposed method demonstrate concrete examples of key duplication and persistence that developers may not anticipate.

Responsible Disclosure

We reported the observed zeroization issues to the developers of three major projects. For OpenSSH and OpenSSL, we used their email contact points, and for Firefox, we reported through the Mozilla forum. The issues we disclosed concerned residual keys caused by inadequate signal handling and keys embedded in core dumps.

All three projects responded to our reports. Both OpenSSH and Firefox addressed the problem of cryptographic keys remaining in core dumps [27]. Firefox also fixed a similar issue in its minidump crash reporting feature [43]. Regarding signal handling, OpenSSH replied that it is difficult to cover all cases, while OpenSSL stated that the issue is beyond the scope of the library and should be handled by applications. The OpenSSL team did not comment on the behavior of the `openssl` command-line tool, and Firefox did not address the signal-related issue in their response.

Acknowledgement

This work was supported by JSPS KAKENHI Grant Number JP23H03373.

References

[1] P. Gutmann, "Secure Deletion of Data from Magnetic and Solid-State Memory," in *6th USENIX Security Symposium (USENIX Security 96)*, 1996.

[2] S. Arranz Olmos, G. Barthe, R. Gonzalez, B. Grégoire, V. Laporte, J.-C. Léchenet, T. Oliveira, and P. Schwabe, "High-assurance zeroization," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, no. 1, pp. 375–397, Dec. 2023.

[3] International Organization for Standardization, "ISO/IEC 19790:2012," https://www.iso.org/standard/52906.html, 2012.

[4] K. Schaffer, "FIPS 140-3 derived test requirements (DTR): CMVP validation authority updates to ISO/IEC 24759," National Institute of Standards and Technology, Tech. Rep. NIST Special Publication (SP) 800-140, Mar. 2020.

[5] Z. Yang, B. Johannesmeyer, A. T. Olesen, S. Lerner, and K. Levchenko, "Dead Store Elimination (Still) Considered Harmful," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. USA: USENIX Association, Aug. 2017, pp. 1025–1040.

[6] L.-A. Daniel, S. Bardin, and T. Rezk, "Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure," *ACM Trans. Priv. Secur.*, vol. 26, no. 2, pp. 1–42, Apr. 2023.

[7] J. Li, Z. Lin, J. Caballero, Y. Zhang, and D. Gu, "K-Hunt: Pinpointing Insecure Cryptographic Keys from Execution Traces," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 412–425.

[8] Cloudlinux Inc. , TuxCare division, "OpenSSL FIPS provider for AlmaLinux 9 FIPS 140-3 non-proprietary security policy."

[9] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, Mendel Rosenblum, "Understanding Data Lifetime via Whole System Simulation," in *Proceedings of the 13th USENIX Security Symposium*. USENIX, Aug. 2004, p. 22.

[10] J. S. Dwoskin, M. Gomathisankaran, Y.-Y. Chen, and R. B. Lee, "A framework for testing hardware-software security architectures," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: Association for Computing Machinery, Dec. 2010, pp. 387–397.

[11] E. N. Kärna, A. Härska, and Ö. D. Alla, "iOS kernel PAC, one year later," https://bazad.github.io/presentations/BlackHat-USA-2020-iOS_Kernel_PAC_One_Year_Later.pdf, 2020.

[12] J. Xu, H. Lin, Z. Yuan, W. Shen, Y. Zhou, R. Chang, L. Wu, and K. Ren, "RegVault: Hardware Assisted Selective Data Randomization for Operating System Kernels," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, Aug. 2022, pp. 715–720.

[13] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman, "The Matter of Heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC '14. New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 475–488.

[14] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading Bits in Memory Without Accessing Them," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2020, pp. 695–711.

[15] National Institute of Standards and Technology, "Implementation guidance for FIPS 140-3 and the cryptographic module validation program," https://csrc.nist.gov/csrc/media/Projects/cryptographic-module-validation-program/documents/fips%20140-3/FIPS%20140-3%20IG.pdf, Sep. 2020.

[16] U. Frisk, "pcileech-fpga," https://github.com/ufrisk/pcileech-fpga, Oct. 2021, accessed: 2023-10-24.

[17] J. Alex Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest We Remember: Cold Boot Attacks on Encryption Keys," in *17th USENIX Security Symposium (USENIX Security 08)*, 2008.

[18] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten., "Memory research project source code," https://citp.princeton.edu/our-work/memory/code/, Jul. 2008, accessed: 2024-5-23.

[19] K. Suzuki, "virt2phys: A tool of translating virtual address of user space to physical address," https://github.com/katsuster/virt2phys, Jul. 2017, accessed: 2024-4-22.

[20] Canonical Ltd., "Install ubuntu on AMD," https://ubuntu.com/download/amd, 2024, accessed: 2024-5-9.

[21] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch, "RFC 8017: PKCS #1: RSA cryptography specifications version 2.2," https://datatracker.ietf.org/doc/html/rfc8017.

[22] D. R. L. Brown and S. Turner, "RFC 5915: Elliptic curve private key structure," https://datatracker.ietf.org/doc/html/rfc5915.

[23] S. Turner, "RFC 5958: Asymmetric key packages," https://datatracker.ietf.org/doc/html/rfc5958.

[24] P. McLaren, W. J. Buchanan, G. Russell, and Z. Tan, "Deriving ChaCha20 key streams from targeted memory analysis," *Journal of Information Security and Applications*, vol. 48, p. 102372, Oct. 2019.

[25] S. Malliaros, C. Ntantogian, and C. Xenakis, "Protecting sensitive information in the volatile memory from disclosure attacks," in *2016 11th International Conference on Availability, Reliability and Security (ARES)*. IEEE, Aug. 2016, pp. 687–693.

[26] The Mitre Corporation, "CWE - CWE-316: Cleartext storage of sensitive information in memory (4.14)," https://cwe.mitre.org/data/definitions/316.html, Jul. 2006, accessed: 2024-4-30.

[27] D. Miller, "openssh-portable@cc048ca," https://github.com/openssh/openssh-portable/commit/cc048ca536d6?diff=split&w=1, Aug. 2024, accessed: 2024-10-11.

[28] Intel Corporation, "Intel key locker specification," https://www.intel.com/content/www/us/en/content-details/671438/intel-key-locker-specification.html, Sep. 2020, accessed: 2025-1-8.

[29] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. d. Yao, "CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, Nov. 2019.

[30] L. Piccolboni, G. D. Guglielmo, L. P. Carloni, and S. Sethumadhavan, "CRYLOGGER: Detecting Crypto Misuses Dynamically," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2021, pp. 1972–1989.

[31] J. Wang, S. Guo, W. Diao, Y. Liu, H. Duan, Y. Liu, and Z. Liang, "CrypTody: Cryptographic Misuse Analysis of IoT Firmware via Data-flow Reasoning," in *The 27th International Symposium on Research in Attacks, Intrusions and Defenses*. New York, NY, USA: ACM, Sep. 2024, pp. 579–593.

[32] J. Calvet, J. M. Fernandez, and J.-Y. Marion, "Aligot: cryptographic function identification in obfuscated binary programs," in *Proceedings of the 2012 ACM conference on Computer and communications security*. New York, NY, USA: ACM, Oct. 2012.

[33] P. Lestringant, F. Guihéry, and P.-A. Fouque, "Automated Identification of Cryptographic Primitives in Binary Code with Data Flow Graph Isomorphism," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. New York, NY, USA: ACM, Apr. 2015.

[34] D. Xu, J. Ming, and D. Wu, "Cryptographic Function Detection in Obfuscated Binaries via Bit-Precise Symbolic Loop Mapping," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2017, pp. 921–937.

[35] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering," in *Proceedings of the 16th ACM conference on Computer and communications security*. New York, NY, USA: ACM, Nov. 2009, pp. 621–634.

[36] X. Li, X. Wang, and W. Chang, "CipherXRay: Exposing cryptographic operations and transient secrets from monitored binary execution," *IEEE Trans. Dependable Secure Comput.*, vol. 11, no. 2, pp. 101–114, Mar. 2014.

[37] F. Gröbert, C. Willems, and T. Holz, "Automated Identification of Cryptographic Primitives in Binary Programs," in *Recent Advances in Intrusion Detection*, R. Sommer, D. Balzarotti, and G. Maier, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 41–60.

[38] F. Mohamed, B. AlBelooshi, K. Salah, C. Y. Yeun, and E. Damiani, "A Scattering Technique for Protecting Cryptographic Keys in the Cloud," in *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. IEEE, Sep. 2017, pp. 301–306.

[39] M. Gross, N. Jacob, A. Zankl, and G. Sigl, "Breaking TrustZone memory isolation and secure boot through malicious hardware on a modern FPGA-SoC," *Journal of Cryptographic Engineering*, vol. 12, no. 2, pp. 181–196, Jun. 2022.

[40] S. R. Davies, R. Macfarlane, and W. J. Buchanan, "Evaluation of live forensic techniques in ransomware attack mitigation," *Forensic Science International: Digital Investigation*, vol. 33, p. 300979, Jun. 2020.

[41] P. McLaren, G. Russell, W. J. Buchanan, and Z. Tan, "Decrypting live SSH traffic in virtual environments," *Digital Investigation*, vol. 29, pp. 109–117, Jun. 2019.

[42] lmrs, "secretgrind: Secretgrind: a valgrind analysis tool to detect secrets in memory," Jan. 2017.

[43] "minidump-writer," https://github.com/rust-minidump/minidump-writer/issues/130, Aug. 2024, accessed: 2025-9-22.

Appendix

A Target Platform

Figure 6 shows the experimental equipment used in the Intel Desktop environment, and Figure 7 shows the experimental equipment used in the Laptop environment.

B Detection of Non-Contiguous Round Keys

Figure 8 shows a memory dump of a complete key schedule within the address space of an ssh process. Figure 9 shows a memory dump containing a partial key schedule duplicated by a context switch. The values highlighted in yellow represent the original AES-128 key, while those highlighted in gray represent the derived round keys. In Figure 8, the key schedule is laid out sequentially from line 2 to line 11, covering all 10 derived round keys. This format can also be detected by the original AESKeyFinder.

In Figure 9, the first round key derived from the original key appears in line 2, while lines 5 to 8 contain the seventh to tenth round keys as shown in Figure 8 (counting line 1 as the first round). When reading line 1 of Figure 9, Periodic-AESKeyFinder treats it as a candidate key, computes the candidate key schedule, and checks whether subsequent candidate round keys appear in the following lines. In addition, lines 3, 9, and 10 of Figure 9 show incrementing values, which we interpret as IVs.



Figure 6: In the PC setup, the FPGA board is connected via PCI Express. The board installed in the motherboard is AXKU062. The white cable from the right side of AXKU062 is Ethernet, used for sending detected keys. The red USB cable at the bottom-left is for JTAG-based debugging.

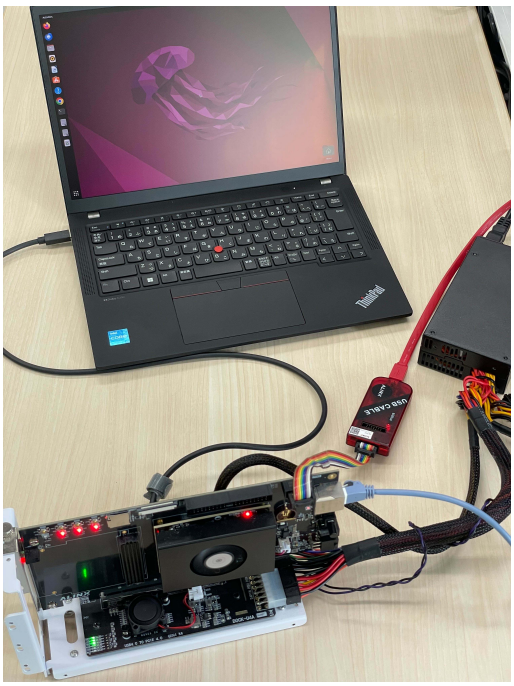


Figure 7: In the laptop setup, PCI Express is tunneled via Thunderbolt. The DOCK-U4A board in front converts PCIe to Thunderbolt and connects to the laptop.

```
845e74b00: 0fc3b56c309d162d 55cbdd1966fd3458
845e74b10: eba2bc81e46109ed d89455c08d5f88d9
845e74b20: b5a297925e002b13 e0694a8b38fd1f4b
845e74b30: d643455363e1d2c1 0ed71093eebe5a18
845e74b40: 69099950bf4adc03 8960d3db87b7c348
845e74b50: 6fe4952506ed0c75 613385b6e853566d
845e74b60: 27e65ae74802cfc2 ae86893ccfb50c8a
845e74b70: 8400d1c2a3e68b25 e53354744bb5dd48
845e74b80: b53f9947313f4885 1bb9107bfe8a440f
845e74b90: a5af871310901e54 409cd3675b25c31c
845e74ba0: 303647179599c004 2b8f576c6b13840b
845e74bb0: 0000000000000000 0000000000000000
845e74bc0: 0000000000000000 0000000000000000
```

Figure 8: AES-128 full key schedule, with the original key in yellow and subsequent keys in gray.

```
8033e9d90: 0fc3b56c309d162d 55cbdd1966fd3458
8033e9da0: eba2bc81e46109ed d89455c08d5f88d9
8033e9db0: 05f006f2f39d44f3 2bf60ccac89f43a6
8033e9dc0: 17f6ea6224e7872e cad13992c8c8f466
8033e9dd0: 27e65ae74802cfc2 ae86893ccfb50c8a
8033e9de0: 8400d1c2a3e68b25 e53354744bb5dd48
8033e9df0: b53f9947313f4885 1bb9107bfe8a440f
8033e9e00: a5af871310901e54 409cd3675b25c31c
8033e9e10: 05f006f2f39d44f3 2cf60ccac89f43a6
8033e9e20: 05f006f2f39d44f3 2df60ccac89f43a6
8033e9e30: d823a4ef35a44bdc 99101d42a81c5753
8033e9e40: d3e21b5769b7e96a 89ab4b04e6efd5ae
8033e9e50: 0000000000000000 0000000000000000
```

Figure 9: AES-128 key schedule duplicated in the kernel's save area, with the original key in yellow and subsequent keys in gray.

C Overhead

In this section, we discuss the overhead imposed on applications. Table 6 presents the evaluation results. To the best of our knowledge, we are the first to extend forensic tools like AESKeyFinder into FPGA-based implementations for measuring the in-memory lifetime of cryptographic keys on actual hardware. As such, it is difficult to identify a direct comparison target for Periodic-AESKeyFinder. However, we discuss the overhead introduced to applications.

Although the method used by K-Hunt [7] to measure overhead is unclear, in our evaluation, we compared the performance of the `7z a` command when compressing and encrypting 28 files, each approximately 380MB in size, into a ZIP archive. Additionally, the `7z b` command refers to the built-in benchmark utility provided by 7-Zip. K-Hunt reports an average overhead of 7.6× in key tracking, varying by application. Specifically, 10.1× overhead was observed for 7-Zip and 11.5× for OpenSSL-RSA. In contrast, Periodic-AESKeyFinder introduced negligible differences in execution times for these applications.

D Supplementary Experiment with the `openssl` Command

We measured detection rates by encrypting and decrypting files using the `openssl` command. We treated the process runtime as the key lifetime and varied file sizes to control it. The experiment was conducted on the Intel Desktop platform. Due to unstable I/O performance from the SD card on the SoC platform, it was excluded.

File sizes ranged from 500 MB to 1.25 GB in 500 MB increments. For each size, encryption and decryption were repeated at least 200 times, with runtimes measured using the `time` command. The results are shown in Figure 10.

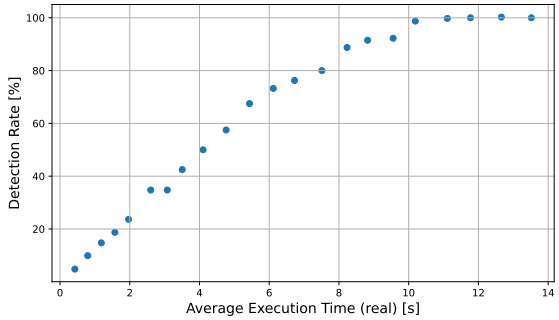


Figure 10: Detection rate of Periodic-AESKeyFinder. Processes with longer execution times have higher detection rates. Near 100% detection was achieved for runtimes exceeding the 8-second scan cycle.

E FPGA Resource and Development Analysis

E.1 FPGA Resource Utilization. Table 7 shows the logic resource utilization for the FPGA on the KR260, Table 8 shows the logic resource utilization for the FPGA on the AXKU062, and Table 9 shows the logic resource utilization for the FPGA on the AXKU3. The resources used by Periodic-AESKeyFinder include not only AES key schedule detection but also DER format detection.

DSPs are employed because the matching logic required to implement line 12 of Algorithm 1 is large in terms of circuit size, so 48 bits of each 128-bit comparison are offloaded. The FPGA's DSP primitive, DSP48E2, is configured to XOR two 48-bit inputs and then perform pattern detection to verify whether all bits are zero. This approach reduces the number of LUT6s by 16 and the number of CARRY8s by 2 for each 128-bit comparison, thereby easing placement and routing.

On the KR260, additional pipeline registers are inserted to support operation at 333 MHz, resulting in variation in register usage across different FPGA boards. It was confirmed that the resulting circuit size is implementable on all FPGA boards.

TABLE 6: Performance Comparison between KR260 and B760+AXKU062

Software	KR260		B760+AXKU062		Unit
	NA	250MHz	NA	250MHz	
7z a command	-	-	47.237 98	47.182 37	s
7z b Compressing	-	-	98.316	98.958	MB/s
7z b Decompressing	-	-	1381.259	1355.246	MB/s
openssl aes 200MiB encrypt	0.587 72	0.596 66	0.074 86	0.074 66	s
openssl rsa 8192bit decrypt	0.465 14	0.465 27	0.066 35	0.066 36	s
openssl rsa 4096bit decrypt	0.089 00	0.089 15	0.039 35	0.039 34	s

TABLE 7: FPGA resource usage on KR260

Category	LUTs		Registers		Block RAM		DSPs	
	Usage	(%)	Usage	(%)	Usage	(%)	Usage	(%)
Available	117 120	-	234 240	-	144	-	1248	-
AESKeyFinder	26 209	22.37	45 659	19.49	42	29.17	654	52.40
Ethernet (Total)	7947	6.78	11 959	5.10	16	11.11	0	0.00
Others (Total)	21 839	18.65	35 834	15.30	32	22.22	0	0.00
Total	55 995	47.81	93 452	39.89	90	62.50	654	52.40

Others includes interconnect and debug logic.

TABLE 8: FPGA resource usage on AXKU062

Category	LUTs		Registers		Block RAM		DSPs	
	Usage	(%)	Usage	(%)	Usage	(%)	Usage	(%)
Available	331 680	-	663 360	-	1080	-	2760	-
AESKeyFinder	26 217	7.90	29 800	4.49	42	3.89	654	23.70
Ethernet (Total)	4397	1.33	6259	0.94	14	1.30	0	0.00
Others (Total)	40 519	12.22	63 462	9.57	178.5	16.53	0	0.00
Total	71 133	21.45	99 521	15.01	234.5	21.71	654	23.70

TABLE 9: FPGA resource usage on AXKU3

Category	LUTs		Registers		Block RAM		DSPs	
	Usage	(%)	Usage	(%)	Usage	(%)	Usage	(%)
Available	162 720	-	325 440	-	360	-	1368	-
AESKeyFinder	26 213	16.11	29 800	9.16	42	11.67	654	47.74
Ethernet (Total)	4113	2.53	6071	1.87	11.5	3.19	0	0.00
Others (Total)	56 385	34.65	86 201	26.49	198.5	55.14	0	0.00
Total	86 711	53.29	122 072	37.52	252	70.00	654	47.74

E.2 Complexity Analysis of Extending to Other Algorithms. The detection logic for AES requires a large number of bitwise comparators. A 128-bit bitwise comparator takes two 128-bit inputs and outputs a single-bit match result, consuming 28 LUTs, 6 FFs, 3 CARRY8s, and 1 DSP. For AES-128, considering 10 rounds of round keys with two alignment assumptions, a total of 290 comparators are required. For AES-256, with 13 rounds, 364 comparators are required. In total, 654 bitwise comparators are needed, each utilizing a DSP.

Furthermore, if the alignment assumptions are removed, the required circuit scale increases by about 8×, making it impractical to fit on an FPGA. If byte-order assumptions are also removed, additional patterns such as 4-byte little endian, 8-byte little endian, and big endian must be considered, further multiplying the resource consumption. The AES key schedule computation circuit itself is also of significant scale. In the case of AES-128, the key schedule logic requires 1,345 LUTs, 1,408 FFs, and 9 Block RAMs.

For public-key cryptography, such as RSA, the situation becomes even more complex. For example, targeting RSA-2048 would require a 2048-bit bitwise comparator.

Compared to the 128-bit comparator for a single round key in AES, a single candidate pattern match in RSA requires approximately 16× more circuit area. Moreover, multiprecision integers are often stored in structures or arrays, and their implementation differs across libraries. This makes it difficult to place general assumptions about alignment or byte order, requiring support for a wide variety of layout patterns. Therefore, when targeting RSA or other public-key cryptography, it is expected that the FPGA resource requirements will be orders of magnitude greater than those for AES.

E.3 Engineering Effort. To implement the IP core for AES detection, we first analyzed a running process with GDB to extract the key schedule, which was necessary to design the FPGA testbench. Creating the initial detection logic—limited to detecting the original key together with one subsequent round key—took about one week, including testbench development and debugging. Extending the logic from this partial detection to cover the entire key schedule and to add pipelining required roughly two additional weeks.

Beyond the AES detection logic, handling board-specific issues such as Ethernet/PCIe control, overall timing constraints, and supporting different hardware boards required additional engineering effort, taking more than half a year in total. Detection of DER-encoded keys was simple and took only three days.