Securing Blockchain Wallet Files Using eBPF

Jeison C. Caroly, Eder J. Scheid, Muriel F. Franco, Lisandro Z. Granville

Institute of Informatics (INF)
Federal University of Rio Grande do Sul (UFRGS)
Porto Alegre, Brazil
{jccaroly, ejscheid, mffranco, granville}@inf.ufrgs.br

***Abstract*—Blockchain (BC) and Distributed Ledger Technologies (DLT) have been widely used in various applications in different areas, from finance to healthcare. For such applications to participate and interact with BCs and DLTs, they must rely on specific software, called nodes, when providing tools and functions for BC synchronization, and called wallets when providing tools for address generation, transaction creation, and fund management. In this sense, wallets are crucial components to be secured within BC-based applications, as they hold sensitive files (*e.g.*, keystore files storing private keys used to generate addresses and sign transactions), which can be a target for attackers. Thus, we propose `Scylla`, a solution to protect wallet-related files using the extended Berkeley Filter (eBPF) that continuously monitors, at the kernel level, the system calls of processes and actively terminates unauthorized and malicious processes when accessing such files. To demonstrate the feasibility and performance of `Scylla`, a prototype was implemented and evaluated in terms of access time overhead and resource use. Such experiments show that `Scylla` is feasible and does not add significant overhead, compared to a native Linux tool dedicated to monitoring files (*i.e.*, *inotify*) while being able to terminate processes before they can read protected files.**

***Index Terms*—Blockchain, Security, eBPF**

I. Introduction

The concepts of blockchain (BC) and Distributed Ledger Technology (DLT) [22] are widely used in a myriad of applications in various fields, such as the Internet of Things (IoT) [21], healthcare, networking [8], cybersecurity [7], and finance [14]. As BC is inherently a distributed system without a central control point, each of these applications participates in this system through a node connected to the network. This node is responsible for updating and synchronizing with other nodes, managing the lifecycle of transactions (*e.g.*, creation, signing, and confirmation), performing BC wallet-related functions (*e.g.*, generating addresses and verifying balance), and including new blocks verified by the network. Therefore, a BC node plays a crucial role in any BC-based application.

However, due to this component being responsible for various tasks, a security issue arises concerning the security of these BC nodes. Examples of attacks on BC nodes include but are not limited to *(i)* denial of service attacks, *(ii)* address substitutions, and *(iii)* private key theft [16]. Considering that BC nodes may contain private keys used to manage large amounts of cryptocurrencies (*e.g.*, nodes of cryptocurrency exchanges or notary-based BC interoperability solutions [20]), these attacks, especially attack *(iii)*, are becoming increasingly common [19]. Thus, there is a need to propose new tools and approaches to protect these nodes and files against such attacks. Furthermore, data breaches and data exfiltration are of concern not only in the cryptocurrency world but also for companies as the cost of a single data breach averages at USD 4.88 million as pointed out by a recent IBM survey [9].

Although there are alternatives to improve the security of the BC wallet, such as hardware wallets and air-gapped solutions [24], [3], these also present issues such as usability as they introduce additional steps for the BC interaction process and high price as they rely on tailored and novel hardware. In this sense, one possible approach to protect these nodes is to use tools and technologies to monitor access, at the operating system level, to wallets and files related to these nodes. One such technology is the extended Berkeley Packet Filter (eBPF), which allows the code to be run at the kernel level without modifying it. eBPF has been used in various security applications, including firewalls, network monitoring, container security auditing, and policy enforcement [23].

In this context, we present `Scylla`, a solution designed to protect BC wallet-related files against unauthorized access through the utilization of eBPF. The proposed solution employs fine-grained access control mechanisms to protect critical files, such as account files, by actively monitoring the system calls of processes directed to these sensitive files. `Scylla` can be executed automatically during boot time and protects user-defined files (which might include `Scylla`'s source code files) based on their *inodes*, hence not allowing malicious actors to read `Scylla`'s code or such files. Evaluations of `Scylla` demonstrates its capability to intercept processes attempting to read or modify the content of a file without introducing significant overhead to legitimate processes. Further, `Scylla` was compared to *inotify*, which is a Linux kernel subsystem created to monitor changes in the file system but is not capable of preventing file access.

The remainder of this paper is structured as follows. Section II details background information on relevant concepts and introduces approaches related to the solution proposed here. Section III presents the design and implementation of `Scylla`. Followed by Section IV, which presents the experiments performed, analyzes the outcomes of the evaluation, and discusses further applicability remarks. Finally, Section V summarizes and concludes the paper, and outlines directions for future research.

II. Background and Related Work

This section provides the necessary knowledge of the concepts involved in the solution and discusses similar approaches relying on eBPF to secure files or to secure BC wallets.

A. *extended Berkeley Packet Filter (eBPF)*

eBPF allows developers to write code that can be dynamically loaded directly into the kernel of the target Linux system without the necessity of recompiling the kernel's source code [18]. This means that novel functionalities can also be included in the operating system's kernel and modified straightforwardly. eBPF is being used by several key players, such as Google for security auditing, packet processing, and performance monitoring, Datadog networking and security in their Software-as-a-Service (SaaS) products, and Netflix for network insights and monitoring [4]. These applications highlight the versatility of eBPF, making it a compelling technology to explore in the context of securing sensitive BC files.

As eBPF programs must be attached to specific events, they are always triggered on such an event [18]. For example, within the scope of this paper, if an eBPF program is attached to the system call (*i.e.*, syscall) to open a file (*i.e.*, `sys_open`), any program that invokes such a function will trigger the eBPF program. This means that eBPF is able to provide global observability of the machine and, combined with the fact that eBPF programs run as native machine instructions in the kernel, eBPF programs are able to achieve high-performance monitoring as there is no cost of translating commands from user space to kernel space.

B. *Wallet Files*

BC node software, such as Bitcoin Core and Geth, typically includes not only functionalities for connecting and synchronizing with the BC network but also built-in wallet software. A BC wallet provides the essential functionalities to send and receive cryptocurrency [24]. This involves tasks such as creating addresses, creating transactions, and signing and later broadcasting transactions.Wallets commonly store private keys associated with addresses in standard file formats (*e.g.*, JSON), facilitating retrieval for transaction signing purposes. However, it is important to note that there is no *de jure* standard for the formatting of these files. Thus, each BC wallet employs a different approach.

1) Ethereum: In Ethereum, accounts are stored in encrypted files called "keystores". These files contain information such as the address, a ciphertext containing the encrypted 256-bit private key, a Key-Derivation Function (KDF) and its parameters (*e.g.*, salt and iteration count) and metadata.

2) Bitcoin: Bitcoin stores private keys in encrypted files named `wallet.dat`. However, these files store more than individual private keys; they also contain information regarding transactions made, user preferences, and, in the context of Hierarchical Deterministic (HD) wallets, a seed utilized for generating a master key and subsequent child keys.

C. *Similar Approaches*

There have been efforts to provide solutions that employ eBPF to protect different aspects of the operating system and its programs. For example, [6] proposes to control and monitor the access of resources by users on Secure Shell Protocol (SSH) sessions using eBPF. The solution defines different scopes of access and a *ssh-probe* is placed within the session to control syscalls and access. However, the solution is still under development and no evaluation was performed.

EZIOTracer [10] uses eBPF to provide a tracer for data-intensive applications that monitors input and output (I/O) events in both kernel- and user-space to optimize the behavior of modern storage systems. Their solution monitors file access but only using read and write probes (*e.g.*, `vfs_write` and `vfs_read` syscalls) and does not actively block access to files as `Scylla` does.

In terms of securing wallet files, the work of [2] leverages a specific technology of ARM-based processors, called TrustZones, to provide a secure lightweight Bitcoin wallet. However, their approach is not flexible, as `Scylla`, being implemented only for Bitcoin, and their approach must be executed on ARM-based processors, which limits its applicability.

In contrast, [17] proposes a multilayered architectural approach to secure BC wallets. Their solution implements three layers of protection for private keys, one that contains an offline backup wallet, one containing the main wallet, and the last one used for wallets that can spend funds. Although the authors claim that such approach provides security to private keys, it introduces additional layers in the user-wallet interaction, hence introducing complexity for the end user.

Research in related work revealed that, to the best of this paper's authors' knowledge, this is the first approach to employ eBPF in the context of BCs and DLTs to monitor and secure BC-related files, such as wallet files. Moreover, `Scylla` is not limited to its application in a single BC or wallet, does not introduce additional complexities for the user, and can be employed to secure files for different security-related use cases, such as ransomware protection.

III. Securing Wallet Files with eBPF

This section details the design and implementation of `Scylla`. Our tool is composed of two programs: a user-space program and a kernel-space program, which together are able to protect a list of files defined by the user. Additionally, `Scylla` maintains a list of permitted processes and protected *inodes* to achieve such a protection.

A. *`Scylla` Approach*

The high-level architecture of `Scylla` is depicted in Figure 1. The figure illustrates the interaction of authorized and malicious processes with `Scylla` in different layers (*e.g.*, User Space, Linux Kernel, and Hardware). In the **User Space** layer, regular user processes are executed and stored, which can be authorized BC node-related processes (*e.g.*, a Bitcoin or Ethereum node) or malicious processes (*e.g.*, an attacker). The figure depicts two examples of *Authorized Processes*, "Geth" is the command line interface for running a full Ethereum node, and "Clef" is a command line tool that provides tools to manage Ethereum accounts and sign transactions. These processes interact with the **Linux Kernel** syscalls to perform operations (*e.g.*, reading and writing files) on the file system. Further, "Malicious Process" are illustrated, which can be an application or malicious user attempting to perform read or write operations (*cf.* Table I) to access or manipulate wallet files (*cf.* Section II-B).

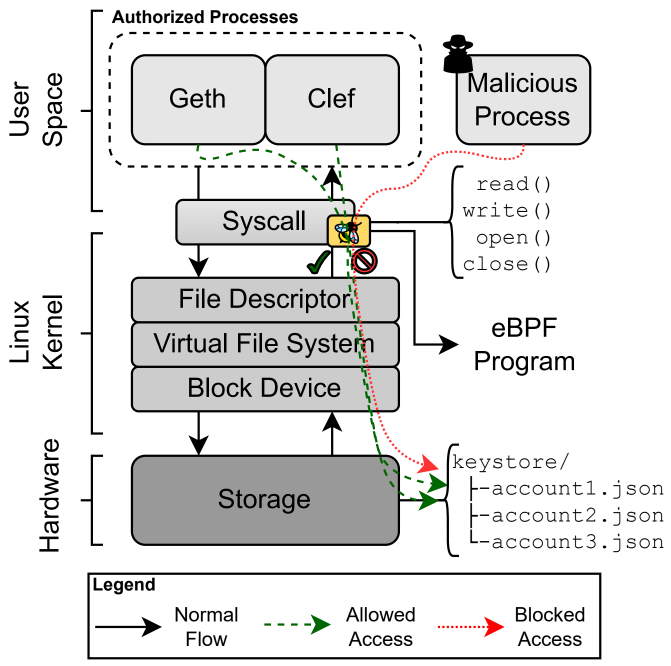

Fig. 1: High-Level Architecture of `Scylla`

The **Linux Kernel** layer comprises syscall interception, file descriptor management, the Virtual File System (VFS) layer, and block device handling, while the **Hardware** layer manages the actual storage hardware (*e.g.*, hard disks). `Scylla` is deployed as a new functionality in the **Linux Kernel** layer, intercepting syscalls, allowing or restricting commands made by processes to protect the integrity of sensitive files located in user-defined directories (*e.g.*, the "keystore" directory) based on their *inodes*.

The dashed red line represents the attempted access path of the malicious process, which was blocked by `Scylla`. `Scylla` actively monitors syscalls from processes and verifies if the process is allowed to manipulate *keystore* files, if not, `Scylla` sends a signal to the kernel to terminate (*i.e.*, sending a `SIGKILL` signal) such a process. The green dashed lines illustrate the allowed access paths from the authorized processes to the physical storage where *keystore* is located.

As `Scylla` intercepts syscalls, it is able to block several basic commands that can be used to exfiltrate BC wallet files or sensitive files outside the device or to different file systems. Table I provides basic native Linux commands, their brief description, and if they can be blocked by `Scylla` or not. In summary, every process that performs the `sys_read` syscall

TABLE I: Overview of Basic Linux Commands

Linux Command	Description	Blocked
`cat, head, tail`	Displays the content of a file	✓
`awk, cut, sort`	Process a file's content	✓
`diff`	Compares two files	✓
`cp, scp`	Copies files and directories	✓
`nano, vi`	Edits files	✓
`file`	Verifies a file's type	✓
`mv`	Moves or rename files and directories	✓✗
`ls`	Lists files and directories	✗
`ln`	Creates links between files	✗
`rm`	Deletes files and directories	✗

✓✗ `mv` is blocked when interacting between different file systems

will be blocked by `Scylla` if its Process Identifier (PID) is not in the allowed list. Thus, displaying, editing, or moving data will not be possible by unauthorized processes.

B. *eBPF Implementation*

The source-code of the `Scylla` and the evaluations presented in Section IV are available at [1] to promote reproducibility. `Scylla` is composed of two main parts, a Python-based one that loads the eBPF code and a C-based one that contains the eBPF code that intercepts system calls and blocks access. For the Python-based one, we relied on BCC [15] to attach the eBPF part of `Scylla` in the system's kernel as a probe in the `vfs_read` event. The Python command to attach to the eBPF program is presented in Line 3 of Listing 1, where `protected_file` is the eBPF function that will be called for every `vfs_read` event.

```
1 program = "readMonitor.c"
2 b = BPF(src_file = program)
3 b.attach_kprobe(event = "vfs_read", fn_name = "
      protected_file")
```

Listing 1: Attaching `Scylla`'s as a Kernel Probe

As the *inodes* are stored in a hash table, the time complexity for *inode* lookup is $\mathcal{O}(1)$. The structure of this hash table is presented in Line 1 of Listing 2, where each *inode* is represented as a 32-bit unsigned integer in C. When retrieving the *inode* of the file that was the target of the `vfs_read` syscall (as shown in Line 3), `Scylla` performs a lookup for this *inode* in the hash table. If the *inode* is found within the hash table (defined on Lines 5 and 6), `Scylla` proceeds to verify the PID (*cf.* Listing 3). However, if the *inode* is not present in the hash table, no further verification is performed, implying that the file is not listed as a sensitive file.

```
1 BPF_HASH(inode_map, u32);
2 ...
3 data.hooked_inode = file->f_inode->i_ino;
4 ...
5 u64 *inode_ptr = inode_map.lookup(&data.
      hooked_inode);
6 if(inode_ptr != NULL) {
7     data.protected_inode = *inode_ptr;
```

Listing 2: eBPF Code for *inode* Lookup

A snippet of the process for identifying and verifying the eBPF code is presented in Listing 3. Similarly, for the identification of the wallet-related files, a mapping is defined (Line 1). Such permitted processes are identified by unsigned 64-bit integers in the hash table. `Scylla` retrieves the current PID and thread PID (Lines 3 and 4) of the process that invoked `vfs_read` syscall (*i.e.*, the potential malicious process). Then, it checks if this PID exists in the `permitted_processes_map`. If the GPID is not found (*i.e.*, process might be malicious), the program writes a "*GPID Not Authorized*" message using the `output.perf_submit` function, which can be utilized for logging or alerting purposes. Finally, to prevent the possible malicious process from accessing the file's contents, `Scylla` sends a `SIGKILL` signal (Line 11), represented by the number 9 on Unix-like systems, to terminate the unauthorized process, effectively securing the file's sensitive content.

```
1  BPF_HASH(permitted_processes_map, u64);
2  ...
3  u64 tgid = bpf_get_current_pid_tgid();
4  data.gpid = tgid >> 32;
5   ...
6  u64 *permitted_process_ptr =
       permitted_processes_map.lookup(&data.gpid);
7  if(permitted_process_ptr == NULL) {
8      char message[20] = "GPID Not Authorized";
9      bpf_probe_read_kernel(&data.message, sizeof(
       data.message), message);
10     output.perf_submit(ctx, &data, sizeof(data));
11     bpf_send_signal(9);
12 }
```

Listing 3: eBPF Code for Access Control based on PID

IV. Evaluation and Discussion

A prototype of `Scylla` was implemented and evaluated in different dimensions to show the feasibility of using eBPF to protect BC wallet files. The evaluations were performed in an 3.6 GHz Intel® Core™ i7-4790 machine running Ubuntu 22.04.1 operating system, kernel 6.5.0-15, with 16 GB of RAM and 1 TB HDD.

A. Access Time Overhead

The overhead introduced by each approach, namely `Scylla` and *inotify*, was evaluated in two scenarios: without verifying which file is being accessed (unprotected) and with verification of the file access (protected). This was achieved by measuring the file access time and comparing it to the baseline access time (*i.e.*, access without any protective approach in place). Figure 2 shows the overhead for each approach along with the average baseline access time. The normal time required to access a file was measured at 23 717 nanoseconds, or 2.3717×10^{-5} seconds. In the protected scenario, `Scylla` incurred a higher overhead, increasing access time by 32%, compared to the increase 27% observed with *inotify*. Despite the 32% increase in access time when

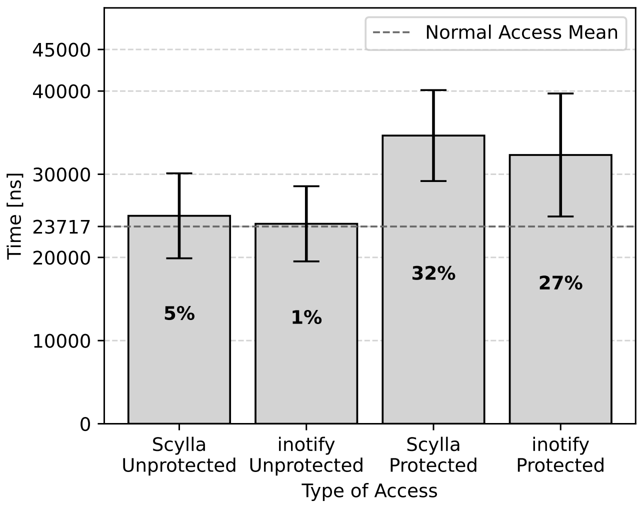

Fig. 2: Overhead Comparison of the Different Approaches

using `Scylla` compared to the baseline, this increase is negligible, since the times are still measured in nanoseconds.

In addition, it should be mentioned that *inotify* does not offer functionalities to block system calls. Therefore, it cannot achieve the goal of blocking access to wallet-related files. This limitation is evident in the small difference, approximately 5%, between protected and unprotected access times when using *inotify*. In the protected scenario, *inotify* only verifies the *inode* of the file. In contrast, `Scylla` verifies and actively blocks access by sending a signal to terminate the process that sent the syscall. Therefore, it results in a higher overhead compared to the *inode* verification of *inotify*.

B. Resource Usage

Another important aspect evaluated was the resource consumption introduced by `Scylla` compared to *inotify* and the normal file access. For this evaluation, two tests were conducted: *(i)* the total CPU usage of the machine was measured over a duration of 150 seconds during which one access (*i.e.*, a `cat` command) to a protected file was performed, and *(ii)* the total CPU usage was measured over 5 minutes (300 seconds) during continuous access to the protected file.

Figure 3 illustrates the results of the first test. The y-axis represents CPU usage in percentage, ranging from 0% to 20% and the x-axis represents time in seconds. It is observed that both `Scylla` and *inotify* do not introduce significant CPU usage overhead when idle. However, upon the triggering of a `sys_read` syscall, the CPU usage peaks at approximately 20% for `Scylla`, 17.5% for *inotify*, and 15% for normal file access. `Scylla` introduces a small CPU usage overhead of about 5% compared to normal access, which is likely attributable to the verification process and the command execution to terminate the malicious process. Thus, given the fact that *inotify* only verifies the file being accessed and does not block access to the file, the overhead of `Scylla` is acceptable.

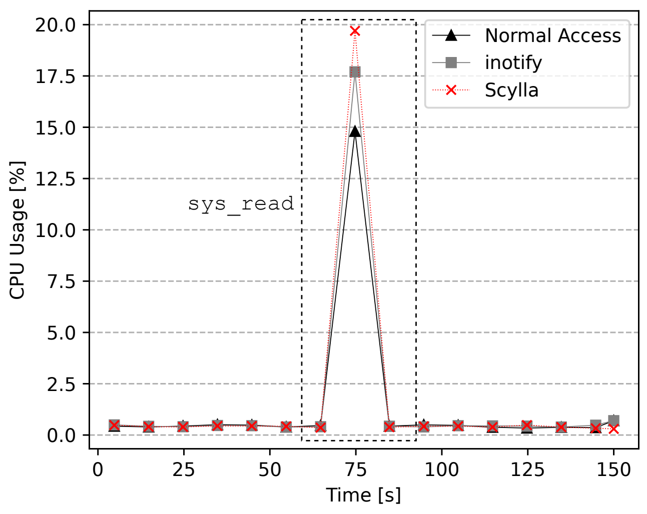

Fig. 3: CPU Usage During a `sys_read` Syscall

The results of the second test are depicted in Figure 4, where the y-axis represents the CPU usage, ranging from 0% to 30%, and the x-axis represents the time in seconds. In this test, all solutions were subjected to a high-stress scenario to analyze their behavior during intense activity, which was processing a file access call every 5 microseconds over a period of 5 minutes. The graph indicates that both `Scylla` and *inotify* exhibit similar levels of CPU usage under these conditions. Additionally, it is important to note that, apart from the initial usage peak, there were no significant increases in CPU usage for either solution. This suggests that `Scylla` is stable and does not incur additional CPU load even when handling a high rate of sequential file access requests.

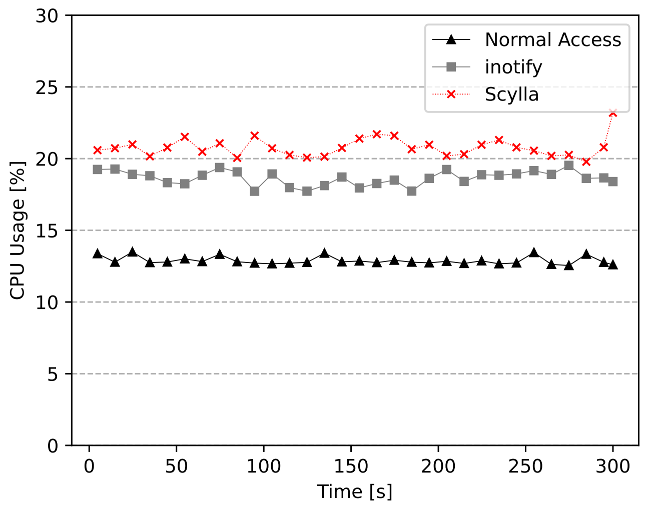

Fig. 4: CPU Usage under File Access Stress

C. Applicability Remarks

During the development of `Scylla`, eBPF has been shown to be a powerful tool for fine-grained control over the operating system's functionalities and for extending them without introducing complex interactions with kernel compilation. Furthermore, given eBPF's flexibility, programs can be injected and modified at run-time, offering adaptability to rapidly evolving system requirements, such as the case of novel BC applications. This feature is crucial for `Scylla` as it can be adapted to new challenges and changes in BC protocols, *e.g.*, new Ethereum Improvement Proposals (EIP) [5].

Moreover, considering that most Unix and Linux-based systems follow the principle "*everything is a file*" [12], in which resources (*e.g.*, documents, disks, peripherals, and network devices) are handled as streams of bytes read and written directly as files using the native file system, the proposed solution in this paper is not bound to the BC context. Other approaches may rely on our approach to develop tools that can be used in different contexts, such as protection against ransomware, sensitive file access, and *keyring* security. Thus, showing that `Scylla` is an interesting and robust approach to secure several aspects of the operating system.

Security implications are also a concern for the usage of eBPF. Threat modeling has to be conducted to verify how the usage of eBPF can be susceptible to cyberattacks and, consequently, impact the overall security proposed by `Scylla`. Besides conduct a threat modeling (*e.g.*, MITRE ATT&CK, STRIDE, and OWASP Threat Dragon) [25] against the eBPF itself, specific tools can be used to verify eBPF implementations [11] and find errors that may generate potential security issues [13].

Finally, while `Scylla` was initially used to protect Ethereum-related wallet files in this paper, its application is not restricted to Ethereum alone. It can be used effectively to protect the wallets of other blockchains, such as Bitcoin (*cf.* Section II-B). This generic applicability is possible because `Scylla` handles *inodes* and process PIDs, and does not require any BC-specific code or adapters. Therefore, it is flexible enough to be used on servers hosting a single BC node or multiple nodes from different BCs.

V. Summary, Conclusions, and Future Work

With the increasing number of applications based on blockchain (BC) and their use in different areas, securing the underlying server that hosts BC-related software (*e.g.*, BC nodes and wallets) used to connect and participate in BC networks becomes increasingly crucial. Moreover, not only do exchanges hold a significant amount of cryptocurrencies, but also BC-based applications need to possess cryptocurrencies to cover mining and interaction fees; hence, BC-based exchanges and applications are targets for cyberattacks aiming to exfiltrate private keys of BC accounts to steal their cryptocurrency.

To address such an issue, this paper detailed the design, implementation, and evaluation of a full-fledged solution, called `Scylla`, which uses extended Berkeley Packet Filter (eBPF) technology to protect BC wallet files against malicious unauthorized access. Scylla acts directly at the kernel level to intercept syscalls from malicious processes and is capable of terminating processes that attempt to access user-defined sensitive files automatically. Further, Scylla maintains a list of authorized processes (*e.g.*, Geth and Clef in Ethereum) that are able to read such files (*e.g.*, BC nodes and wallets). Evaluations of Scylla demonstrated its feasibility and acceptable overhead compared to a solution (*i.e.*, *inotify*) that is not capable of terminating the process. Furthermore, Scylla has been shown to be stable when handling a high rate of sequential file access.

In conclusion, eBPF was successfully employed in Scylla to provide fine-grained access control to BC wallet-related files without additional layers of interaction for system administrators by adding additional functionality in the Linux kernel. Scylla's employment is not restricted to the BC context, but can be effectively applied to other contexts, such as ransomware protection, showing the high flexibility of Scylla.

Future work regarding Scylla includes but is not limited to, *(i)* design a Graphical User Interface (GUI) for user interaction, *(ii)* research on Machine Learning (ML) models to determine the behavior of legitimate processes, *(iii)* further testing of Scylla across different environment and hardware platforms, and *(iv)* investigating its use in the ransomware context with further threat modelling analysis and discussion on security considerations.

Acknowledgements

This work was partially supported by the São Paulo Research Foundation (FAPESP) under grant number 2020/05152-7, the PROFISSA project, and is part of CNPq process 316662/2021-6. It is also part of the INCT of Intelligent Communications Networks and the Internet of Things (ICoNIoT), funded by CNPq (proc. 405940/2022-0) and the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) Finance Code 88887.954253/2024-00.

References

[1] J. C. Caroly and E. J. Scheid, "Scylla Source Code," 2024, https://github.com/ComputerNetworks-UFRGS/scylla.

[2] W. Dai, J. Deng, Q. Wang, C. Cui, D. Zou, and H. Jin, "SBLWT: A Secure Blockchain Lightweight Wallet Based on Trustzone," *IEEE Access*, vol. 6, pp. 40 638–40 648, Jul. 2018.

[3] A. Davenport and S. Shetty, "Air Gapped Wallet Schemes and Private Key Leakage in Permissioned Blockchain Platforms," in *IEEE International Conference on Blockchain (Blockchain 2019)*, Atlanta, USA, Jul. 2019, pp. 541–545.

[4] eBPF.io authors, "eBPF Case Studies," 2023, https://ebpf.io/case-studies/.

[5] Ethereum Community, "Ethereum Improvement Proposals," Feb. 2024, https://eips.ethereum.org/.

[6] G. Fournier, "Monitoring and Protecting SSH Sessions with eBPF," in *Symposium sur la Sécurité des Technologies de lÍnformation et des Communications (SSTIC 2021)*, Renner, France, Jun. 2021.

[7] M. F. Franco, N. Berni, E. J. Scheid, B. Rodrigues, C. Killer, and B. Stiller, "SaCI: a Blockchain-based Cyber Insurance Approach for the Deployment and Management of a Contract Coverage," in *Lecture Notes in Computer Science (LNCS)*, no. 13072. Virtual: Springer, Sep. 2021, pp. 79–92.

[8] M. F. Franco, E. J. Scheid, L. Granville, and B. Stiller, "BRAIN: Blockchain-based Reverse Auction for Infrastructure Supply in Virtual Network Functions-as-a-Service," in *IFIP Networking (Networking 2019)*. Warsaw, Poland: IEEE, May 2019, pp. 1–9.

[9] IBM Security, "Cost of a Data Breach Report 2024," 2024, https://www.ibm.com/reports/data-breach.

[10] M. Islam Naas, F. Trahay, A. Colin, P. Olivier, S. Rubini, F. Singhoff, and J. Boukhobza, "EZIOTracer: Unifying Kernel and User Space I/O Tracing for Data-Intensive Applications," *ACM SIGOPS Operating Systems Review*, vol. 55, no. 1, pp. 88–98, Jul. 2021.

[11] J. J. L. Jaimez and M. Inge, "Introducing a New Way to Buzz for eBPF Vulnerabilities," May 2023, https://security.googleblog.com/2023/05/introducing-new-way-to-buzz-for-ebpf.html.

[12] J. Lessem, "Everything is a File - Understanding Unix Concepts," Jul. 1996, http://ibgwww.colorado.edu/~lessem/psyc5112/usail/concepts/filesystems/everything-is-a-file.html.

[13] M. H. N. Mohamed, X. Wang, and B. Ravindran, "Understanding the Security of Linux eBPF Subsystem," in *14th ACM SIGOPS Asia-Pacific Workshop on Systems*, Seoul, South Korea, Aug. 2023, pp. 87–92.

[14] A. A. Monrat, O. Schelén, and K. Andersson, "A Survey of Blockchain From the Perspectives of Applications, Challenges, and Opportunities," *IEEE Access*, vol. 7, pp. 117 134–117 151, Aug. 2019.

[15] I. V. Project, "BCC - Tools for BPF-based Linux IO Analysis, Networking, Monitoring, and More," 2024, https://github.com/iovisor/bcc.

[16] QuickNode, "How to Secure Your Node against Common Blockchain Attacks & Vulnerabilities," Feb. 2013, https://bit.ly/3AhaIwU.

[17] H. Rezaeighaleh and C. C. Zou, "Multilayered Defense-in-Depth Architecture for Cryptocurrency Wallet," in *IEEE International Conference on Computer and Communications (ICCC 2020)*, Chengdu, China, Dec. 2020, pp. 2212–2217.

[18] L. Rice, *Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Networking, and Security*, 1st ed. O'Reilly Media, Apr. 2023.

[19] M. Saad, J. Spaulding, L. Njilla, C. Kamhoua, S. Shetty, D. Nyang, and D. Mohaisen, "Exploring the Attack Surface of Blockchain: A Comprehensive Survey," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 3, pp. 1977–2008, Mar. 2020.

[20] E. J. Scheid, T. Hegnauer, B. Rodrigues, and B. Stiller, "Bifröst: a Modular Blockchain Interoperability API," in *IEEE Conference on Local Computer Networks (LCN 2019)*, Osnabrück, Germany, Oct. 2019, pp. 332–339.

[21] E. J. Scheid, A. Knecht, T. Strasser, C. Killer, M. Franco, B. Rodrigues, and B. Stiller, "Edge2BC: a Practical Approach for Edge-to-Blockchain IoT Transactions," in *IEEE International Conference on Blockchain and Cryptocurrency (ICBC 2021)*, Sydney, Australia, May 2021, pp. 1–9.

[22] E. J. Scheid, B. Rodrigues, C. Killer, M. Franco, S. Rafati, and B. Stiller, "Blockchains and Distributed Ledgers Uncovered: Clarifications, Achievements, and Open Issues," in *Advancing Research in Information and Communication Technology*, ser. IFIP AICT Festschrifts. Cham, Switzerland: Springer, Aug. 2021, vol. 600, pp. 289–317.

[23] H. Sharaf, I. Ahmad, and T. Dimitriou, "Extended Berkeley Packet Filter: An Application Perspective," *IEEE Access*, vol. 10, pp. 126 370–126 393, Dec. 2022.

[24] S. Suratkar, M. Shirole, and S. Bhirud, "Cryptocurrency Wallet: A Review," in *International Conference on Computer, Communication and Signal Processing (ICCCSP 2020)*, Chennai, India, Sep. 2020, pp. 1–7.

[25] J. Von Der Assen, M. F. Franco, C. Killer, E. J. Scheid, and B. Stiller, "CoReTM: An Approach Enabling Cross-Functional Collaborative Threat Modeling," in *IEEE International Conference on Cyber Security and Resilience (CSR 2022)*, Virtual, Jul. 2022, pp. 189–196.

All links were last visited on August 8, 2024