Logan Arkema* and Micah Sherr

Residue-Free Computing

Abstract: Computer applications often leave traces or *residues* that enable forensic examiners to gain a detailed understanding of the actions a user performed on a computer. Such digital breadcrumbs are left by a large variety of applications, potentially (and indeed likely) unbeknownst to their users. This paper presents the concept of *residue-free computing* in which a user can operate any existing application installed on their computer in a mode that prevents trace data from being recorded to disk, thus frustrating the forensic process and enabling more privacy-preserving computing. In essence, residue-free computing provides an "incognito mode" for *any* application. We introduce our implementation of residue-free computing, ResidueFree, and motivate ResidueFree by inventorying the potentially sensitive and privacy-invasive residue left by popular applications. We demonstrate that ResidueFree allows users to operate these applications without leaving trace data, while incurring modest performance overheads.

Keywords: privacy; forensics; anti-forensics

DOI 10.2478/popets-2021-0076
Received 2021-02-28; revised 2021-06-15; accepted 2021-06-16.

1 Introduction

Many of the actions that users perform on their computers leave digital traces. In a computer forensics investigation, these traces form digital breadcrumbs that allow a forensic examiner to gain a fairly detailed understanding of what the user did with their computer—which applications and files were accessed, the times and potentially the duration of their accesses, and more generally, the actions performed by the user.

Computer forensics is aided by modern operating systems that tend to leave an enormous volume of these breadcrumbs. Operating systems leave traces not necessarily to ease the job of a forensic examiner, but rather to provide additional features and functionalities. For example, modern operating systems maintain access times to quickly locate the most recently modified files; these access times help an examiner determine how the computer was used. Log-based filesystems are designed to facilitate fast data recovery, but also leave remnants of erased files and more generally allow examiners to reconstruct the filesystem at various points in time.

More generally, the traces left by operating systems and applications leads to a status quo in which it is exceedingly difficult to use a computer without leaving a readily available record of one's actions. Countermeasures that enable greater privacy protections are piecemeal, and are usually constrained to a particular application—a notable example is the "incognito" or "privacy" mode of modern web browsers that are designed to remove local traces of web activity left by the browser. More general solutions do exist [10, 12], but they require modifications to the Linux kernel and fail to capture some types of application residue. (We explore the related literature in more detail in the next section.)

This paper introduces *residue-free computing*, a privacy-preserving (or anti-forensics) mode of operation for modern operating systems without requiring kernel modifications. In residue-free computing, the user uses their normal operating system and has access to their existing files. The user may optionally choose to start *any* installed program in *residue-free mode*, which enables the program to run on top of a union filesystem. The application has read access to all existing data on any installed filesystem, but file modifications (including deletions) are made to a volatile filesystem stored in memory (i.e., RAM). Upon exiting the application, the filesystem modifications are permanently erased. Conceptually, residue-free computing provides an incognito mode for *any* installed application.

Residue-free computing assumes an atypical threat model. It assumes a cooperative user who wants to preserve their privacy, and an application and operating system that do not actively attempt to prevent the application from executing in residue-free mode. That is, residue-free computing is designed to work on applications and operating systems as they currently exist. We assume as our adversary a forensic examiner who is

***Corresponding Author: Logan Arkema:** Georgetown University, Email: lja39@georgetown.edu
Micah Sherr: Georgetown University, Email: micah.sherr@georgetown.edu

able to take filesystem snapshots of the filesystem before and after the execution of an application in residue-free mode. We describe our adversary in more detail below, but in brief, we do not protect against a network adversary who learns which applications are being used by examining network traces, nor do we protect against malware running on the user's device that attempts to record actions. Our goal is to prevent our *snapshot adversary* from learning which applications were used and which files were accessed.

To be most useful, residue-free computing should not require any program modifications and should be compatible with all software already installed on the computer, including its operating system. Increased privacy should not come at a significant performance penalty; running an application in residue-free mode should incur limited overheads. Finally, residue-free computing should be user-friendly, allowing a user to easily opt to use residue-free computing mode (or not) for any application.

There already exist techniques that achieve some of the goals of residue-free computing. For example, the user can simply run an application in a virtual machine and use checkpointing to roll back filesystem and memory changes after the application exits. Or, the user could use a live CD operating system such as The Amnesic Incognito Live System (Tails) [2] that is tailored to frustrate forensic investigations by preventing traces from being recorded to non-volatile storage. However, these solutions incur high usability costs as they require the user to significantly modify their behavior (to the point of using an entirely different operating system) in order to gain some privacy protections.

We present the design and implementation of ResidueFree, an instantiation of residue-free computing that allows users to operate their existing applications (on their existing operating systems) in residue-free mode. We perform an in-depth forensic investigation in which we examine every persistent file created or modified during a run in residue-free mode, and find that ResidueFree leaks only minimal information: namely, that ResidueFree was used. The application being used and the affected files (both read and modified) are invisible to the forensic examiner. We show through extensive benchmark-based evaluation that ResidueFree incurs moderate-to-limited overhead. Finally, we provide a simple and intuitive interface for ResidueFree for the Gnome desktop: users run an application in residue-free mode by right-clicking the application icon and selecting "Run in ResidueFree."

Motivating examples. To motivate the use of residue-free computing, we consider two example scenarios.

Privacy from invasive cohabitants: A computer user shares a residence with an individual (for example, an abusive partner) who attempts to monitor the user's computer usage. This adversary may be technically sophisticated, but is not necessarily a trained forensic examiner. The user is also not especially technically sophisticated, but is aware that ResidueFree is installed on their computer.

The user uses Skype. Although the user may not be aware that Skype stores comprehensive log files that contain chat transcript and Skype call metadata (including participants and call durations), the user fortunately opted to use ResidueFree to operate Skype. Since Skype was used in residue-free mode, the cohabitating adversary can neither identify that Skype was used nor learn with whom the user communicated.

Privacy from a knowledgeable forensic investigator: An investigative journalist enters a border crossing when their computer is seized and imaged. The border crossing agent examines the journalist's computer to learn what they are investigating. Fortunately, the journalist used ResidueFree to run the VLC media player, so the log of the videos that they viewed (including some interviews streamed from the web) are not available to the agent, as they otherwise would have had ResidueFree not been activated.

Contributions. In summary, this paper makes the following contributions:

- The design of residue-free computing: a mode of operation that provides an "incognito mode" for any application;
- ResidueFree, an open-source implementation of residue-free computing. ResidueFree is available as free open-source software and is available at https://larkema.github.io/residuefree/;
- A study of the residue (i.e., forensic traces) left by popular applications, and an examination of how ResidueFree prevents the collection of these traces.

2 Related Work

The literature on privacy-enhancing methods of computing is rich and diverse. In this section, we describe how residue-free computing fits in the context of this existing literature.

Most related to residue-free computing are PrivExec [22] and TpriVexeC [10] which have the shared goal of providing an incognito-like privacy mode for any application. Although residue-free computing, PrivExec, and TpriVexeC all use a union filesystem, they differ in several important respects: unlike PrivExec and TpriVexeC, ResidueFree does not require kernel modifications and works out-of-the-box with popular Linux distributions. We achieve this by leveraging existing efforts in Linux containerization. Additionally, residue-free computing offers additional privacy guarantees over both PrivExec and TpriVexeC: residue-free computing obfuscates which applications were run and it conceals which files were *read* by the application running in privacy-mode; both are exposed by PrivExec and TpriVexeC. Like PrivExec and TpriVexeC, we also provide a performance evaluation of our approach and explore its overheads; however, unlike the prior work, we also conduct a forensics investigation using filesystem snapshots to empirically demonstrate the efficacy of our technique. Finally, ResidueFree offers a forensics mode that allows a user to explore the residues left by different applications.

Techniques that use a separate privacy-preserving environment. As detailed in Garfinkel's survey of anti-forensic techniques, there are many techniques a user could employ to disrupt a potential future forensic investigation [14]. Residue-free computing belongs to a class of techniques that attempt to "minimize the footprint[s] [14]" available to computer forensic tools. In Garfinkel's survey, he considered two footprint minimization techniques: the use of live CDs—or operating systems that boot from a removable device (e.g., a CD)—and virtual machines (VMs). In particular, Tails [2] is a live CD operating system designed to provide strong privacy protections by avoiding any writes to non-volatile storage and routing all network communication anonymously via Tor [9]. Similarly, using a virtual machine allows a user to *potentially* destroy traces by reverting to an earlier snapshot. This latter approach however entails significant risk, since it relies on the secure erasure of the unsaved machine state; in practice, it is unclear that such information is actually unrecoverable.

Similarly, so-called lightweight VMs such as containers (with Docker being a noteworthy example) can provide strong application isolation and ensure that filesystem modifications are not directly applied to the system's underlying filesystem. However, these containers do not eliminate traces; they merely shift the trace from traditional files stored on the main filesystem to containerized filesystems, which themselves are stored on the main filesystem. In short, containers provide at best a level of indirection, but do not significantly obfuscate filesystem changes. A knowledgeable forensic examiner could easily examine containerized filesystems. (We address the additional difficulty of securely deleting these containerized filesystems below.)

Live CDs achieve many of the same privacy goals as residue-free computing, which we describe in more detail in §3. However, both live CDs and VMs incur large usability costs, as they require the user to maintain at least two computing environments: one for ordinary use, and one for more private computing. Live CDs and VMs prevent the user from making a quick impromptu decision to run an existing application (along with its existing configuration) in a more privacy-preserving mode, since configuring the live CD or the VM requires a non-trivial amount of work. In contrast, residue-free computing enables the user to run any application in a privacy-preserving mode *on demand*, at the user's whim. This allows the user to run any installed application as already configured, without leaving a digital trace for a potential future forensics investigation.

Private web browsing modes. Residue-free computing is partially inspired by the private web browsing modes of modern web browsers. These modes prevent websites from accessing cookies stored in non-private mode, and it erases browsing history, search history, and cookies upon exit. However, private browsing modes have historically not removed all forensic traces [27, 28]. Prior work has also shown that many users are unaware of private browsing modes, and many of those who are aware do not understand the actual privacy protections offered by these modes [13]. In particular, Habib et al. show that many users of private browsing modes overestimate their protections from targeted advertisements and user tracking [16].

Residue-free computing can be viewed as a form of private or incognito mode for any application. Architecturally, however, it is very dissimilar to browser-based privacy modes. As discussed in more detail in §5, residue-free computing uses redirect-on-write filesystems and volatile filesystems stored on RAM-disks to ensure that all file modifications are transient and removed after the application exits. Private web browsing modes require specific functionality in the application (i.e., the web browser) to track file modifications and ensure that they are removed after the browser exits its privacy mode. We argue that residue-free

computing offers stronger guarantees than application-specific solutions since the former operates at the operating system/filesystem level, ensuring that any and all filesystem modifications made by a program running in residue-free mode are transient. In contrast, application-specific solutions require careful attention by the programmer to identify and revert all modifications.

Steganographic file systems. The goals of residue-free computing are also similar to those of steganographic filesystems (also called *plausibly deniable storage systems*) [4]. Steganographic filesystems typically have multiple layers of files, with a given layer being exposed by providing its corresponding key while all other layers remain hidden. Conceptually, steganographic filesystems provide a mechanism where a coerced individual—for example, a detainee at a border crossing—could provide a key that reveals benign content while obscuring the existence of the more sensitive covert data. There are several deployed steganographic filesystems, with notable examples including StegFS [20] and (the now defunct) TrueCrypt [1]. However, these systems do not hide the fact that the steganographic filesystem is installed. Czeskis et al. show that an operating system's regular functionality may betray the existence of the hidden volume, and demonstrate such attacks against TrueCrypt [8]. Chen et al. extend the notion of plausible deniability to include *invisibility*, which also conceals evidence of the steganographic file system [7].

In theory, steganographic filesystems enable users to conceal the digital breadcrumbs left by their applications by obscuring the existence of the entire filesystem on which the application resides. In practice, however, steganographic filesystems require maintaining multiple computing environments—one for an overt "cover" OS, and a separate environment for covert application usage. This entails a significant and persistent effort by the user. Residue-free computing attempts to lighten this workload by enabling on-demand privacy-preserving operation of existing applications. We do not require the user to maintain multiple copies of data or applications, and only require that the user proactively starts an application in residue-free mode when they want enhanced privacy protections.

Secure deletion. Residue-free computing requires that any filesystem modifications are undone after a program running in residue-free mode terminates. On modern log-based and versioning filesystems, this can be (perhaps) surprisingly difficult [23]. Reardon et al. provide a comprehensive survey of the challenges and techniques for securely removing data [26]. Much of the literature on secure deletion focuses on mobile devices (cf. [3] and [11]) and their NAND-based flash file systems.

Our implementation (see §6) sidesteps the difficulties of erasing data on flash-based storage media and versioning filesystems by storing all file modifications on volatile memory.

Redirect-on-write filesystems. Residue-free computing makes extensive use of redirect-on-write filesystems. A redirect-on-write filesystem makes a virtual copy of an underlying base filesystem. Read operations are usually directed to the underlying base filesystems, while writes are made to a modified copy stored in a separate storage area. Subsequent reads to modified files are then directed to this separated storage area. Popular implementations of redirect-on-write filesystems include UnionFS [25], aufs [18], and overlay2 [6].

Designed primarily to support efficient snapshotting (also called checkpointing) rather than anti-forensics, redirect-on-write filesystems are by themselves insufficient for achieving the goals of residue-free computing. The storage area that records filesystem modifications constitutes a (fairly comprehensive) digital trace of application and data use.

3 Threat Model and Goals

Our system model includes the *user*, who is the operator of the machine; the *operating system*; the *application* to be run in residue-free mode; the *filesystem* that contains the operating system and the application[1]; and the *adversary*, who conducts a forensic investigation.

Goals. The primary goals of residue-free computing are to prevent the adversary from (1) identifying which application was used in residue-free mode and (2) identifying any files that were modified (including file creation and deletion) or accessed by an application operating in residue-free mode. For ease of exposition, we consider a single use of residue-free mode in this paper, but our goals naturally extend to sequential or concurrent use of residue-free computing: for example, the adversary should not be able to tell which applications were run in residue-free mode during their investigation.

1 In practice, the user's computer can have multiple filesystems. For ease of exposition, we consider the filesystem to be one logical volume in this paper.

An important secondary goal of residue-free computing is ease-of-use. Although this is far more difficult to quantify, residue-free computing should not require significant work or expertise on the part of the user. Residue-free computing should be compatible with the user's existing operating system and applications, and all of the user's data (i.e., all data stored on their filesystem) should be available to an application operating in residue-free mode. The behavior of an application running in residue-free mode should be *identical* to that application running outside of residue-free mode, excepting for the existence of trace data not being available after the program exits. Starting a program in residue-free mode should be intuitive and not require a high level of technical expertise.

Finally, operating in residue-free mode should not incur significant overhead. The performance of a program operating in residue-free mode should be on par with its operation outside of residue-free mode.

Threat model and adversary capabilities. We consider an adversary who has access to snapshots of the filesystem before and after an application runs in residue-free mode. The adversary has knowledge of residue-free computing, and can apply any forensic tools or analyses to the snapshots. We assume the adversary has, at minimum, knowledge of publicly documented forensic artifacts in the operating system, such as those presented by Nishida [21].

The adversary is not permitted to eavesdrop on the computer's network communication or monitor the computer while an application is running in residue-free mode. In general, "live" or real-time forensics is considered out-of-scope for residue-free computing. The goal of residue-free computing is to remove the residue left by applications on the filesystem in order to subvert a forensic investigation. It does not protect against adversaries who use other means (e.g., network eavesdropping) to ascertain a user's computer usage.

We assume that the operating system (including the implementation of the filesystem) and any applications do not attempt to detect and thwart residue-free computing. While applications should be agnostic to (and even ignorant of) residue-free computing, we do not attempt to conceal its existence from an application that actively tries to detect it. Similarly, residue-free computing does not protect against spyware or other forms of malware that may be running on the user's computer.

Security properties. Residue-free computing meets the above two goals: a forensic examination of the second snapshot (taken after the program operating in residue-free mode exits) and optionally the first filesystem snapshot (taken before the program spawned in residue-free mode begins execution) reveals neither which application was run in residue-free mode nor which files were accessed (read, modified, created, deleted, etc.) by that application.

We emphasize that residue-free's security properties are met *after* the program running in residue-free mode terminates; during that program's operation, we do not attempt to hide that it is running or what files it accesses.

The adversary is permitted to learn that the computer supports residue-free mode (e.g., that ResidueFree is installed) and the time that the user launched residue-free mode; which specific application(s) were run remains hidden.

4 Motivation: Case Studies of Popular Application Residues

To motivate the need for residue-free computing, we first investigate the frequency and degree to which popular applications leave residues. We also examine these residues to attempt to gauge the level of privacy loss or information leaked to forensic examiners. Conceptually, this provides a baseline of how much an examiner can learn without the protections offered by residue-free computing.

We selected applications across a wide range of categories, based on their ranking in Ubuntu's Software application. We focused our selections on popular applications, where popularity was based primarily on our own experiences with these applications and the ease at which we could find independent information that hinted at the application's popularity (e.g., blogs and YouTube videos made by users). While we attempted to identify quantitative application metrics for Ubuntu desktop applications, we could not identify a definitive metric similar to an Apple App Store or Google Play download metric. The applications, which are listed in Table 1 in the Appendix, were installed either using the default packages from Ubuntu Software on Ubuntu 18.04 (if available) or the version available on the application's web site (if not).

To perform our investigation, we used the *forensic mode* of ResidueFree, which we describe more fully in §6. In brief, we used a redirect-on-write filesystem to isolate all files that the application modified while in use. We also provided unique inputs, which we call *ca-*

nary strings (e.g., "gorgonzola penguins") to the applications to determine whether the applications' residues captured these inputs. Their presence in an application's residue suggests that a forensic examiner would have detailed information about the user's inputs to that application.

We manually inspected the application residues to search for both our canary strings as well as other obvious information about the user's session (e.g., timestamps, chat logs, names of transferred files, log entries, etc.).

We emphasize that our manual analyses do not allow us to exhaustively determine the information contained within an application's residue. The presence of canary strings tells us definitively that the user's inputs are captured, but the absence could indicate that the information is somehow otherwise recorded (e.g., in an encoded form). As a result, the residues we identify are a subset of all the potential residues that could be uncovered by a trained forensic examiner (especially one well-versed in a particular application), and should be taken as conservative estimates of the trace data left by popular applications.

Our findings are summarized in Table 1 in the Appendix. We found that these common user applications left a significant amount of residue on the filesystem that, if left on disk for an attacker to read, could reveal significant data on how the user used the application. For example, Spotify leaves all songs the application displayed to the user and the user's social media information on the filesystem; Skype records detailed information from the user's audio, video and instant message conversations, including message contents, recipient usernames, and timestamps; Discord stores the channels a user participated in; and VLC tracks and updates timestamps on recently played media files.

Privacy-focused applications, such as Signal, Telegram, Brave Web Browser, Tor Web Browser, and other web browsers operating in "private" or "incognito" mode do not record detailed data on the user's activity, but they sufficiently modify the filesystem to clearly denote that the user ran the applications and when the user ran them. Unfortunately, designing these private browsing modes is difficult since developers need to track (and subsequently, eliminate) any state naturally kept by the browser. Subtle errors can lead to tracking. For example, Solomos et al. recently showed that malicious websites could bypass incognito mode on all major browsers using favicons [27]. ResidueFree avoids such tracking by eliminating all persistent data.

Most of this data is stored without the user's knowledge or explicit consent, which is not to say that these applications are categorically flawed. Much of the residue applications leave is used to provide additional features or performance benefits to users, and these applications do not consider an attacker with future access to the user's machine in their threat models. However, a user seeking to manually remove the details from their session would be hard-pressed to do so for every application. Most of the residue we found is stored in hidden directories (i.e., directories whose names begin with '.'), in files multiple directory levels deep, and/or in files without descriptive names. Rather than task a user with finding every file that may contain information from their session, we provide a solution that automatically discards *all* changes to the filesystem made during their session.

5 Architecture

Residue-free computing is composed of three main components: a completely volatile RAM disk, a union filesystem [25], and an isolated execution environment. These components *must* not write to the user's on-disk filesystem nor modify any filesystem metadata and *should* sufficiently interact with the user's system to provide a nearly indistinguishable user experience from running the application outside of residue-free mode. This latter interaction requires special care, since the application's functionality may require it to communicate with other processes, but such interprocess communication (IPC) should not result in leaked information about the application running in residue-free mode. As a result, system processes may need to be modified during residue-free computing to eliminate sensitive logging and protect against information the operating system may otherwise leave on the user's disk.

The execution environment and RAM disk will both contain information from the user's residue-free session that must be kept private, while the union filesystem is a logical component that facilitates file access without moving data to a new physical device. At the same time, residue-free must have access to the physical computer hardware necessary to run user programs without having the system log sensitive information. Figure 1 visualizes this architecture.

We define a *completely volatile RAM disk* as a filesystem physically hosted on RAM that can reliably have all of its contents made irrecoverable. To securely

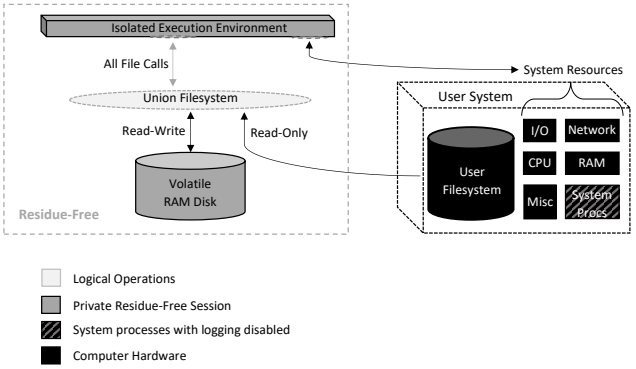

Fig. 1. The architecture of residue-free computing.

erase the contents of RAM disk, an implementation could either overwrite every bit of the disk or encrypt the contents of disk in real time with an irrecoverable key (we discuss the associated overhead costs of this implementation choice in §7.3). Any data written to the RAM disk should be made unrecoverable quickly after a residue-free computing session ends and must be completely erased after the user's system powers off or enters sleep mode.

The union filesystem takes the volatile RAM disk and merges it with the user's filesystem to create a logically unified filesystem interface while accessing two different physical devices. The union filesystem must have read-write access to the RAM disk and read-only access to the user's filesystem. Conceptually, this allows the application running in residue-free mode to *appear* to have full (i.e., read-write) access to the user's regular filesystem. Every write a process using the union filesystem makes must be written to the RAM disk and these writes must be readable to processes that use the union filesystem (so long as the RAM disk is still intact). The union filesystem can read from the user's filesystem for data that has not been created or changed in the isolated execution environment, and these reads must not modify any file data or filesystem metadata on the user's regular filesystem. The union filesystem does not store any data; it merely mediates access between the two storage devices.

Finally, the isolated execution environment must run on top of the union filesystem and should provide a nearly indistinguishable user experience from the user's system. The environment must ensure that all calls to the filesystem are made to the union filesystem and should be sufficiently isolated to minimize interaction with user processes running outside of residue-free mode. However, this isolation must be balanced with sufficient access to system resources, including system processes, to provide a functional user experience with as few performance costs as possible. In addition, system processes may be modified to provide additional privacy guarantees without limiting their functionality.

In summary, a user should not notice that they are using residue-free computing while the underlying operations do not modify the user's filesystem.

6 Implementation

We implement the residue-free computing architecture using an encrypted RAM disk, the unionfs copy-on-write filesystem, and a Docker container. We carefully configure these components to enable access to the user's system resources and additionally modify select system processes. Our proof-of-concept implementation, which we refer to as ResidueFree, implements the basic requirements of residue-free computing while providing additional features to the user that go beyond architectural requirements. While this implementation incurs moderate overhead and performance costs, particularly on file operations, it overwhelmingly meets the architecture's privacy goals.

ResidueFree is released as free open-source software and is available to download at https://larkema.github.io/residuefree/.

ResidueFree operates on Linux, and has been tested with Ubuntu 18.04 LTS with kernel 5.4.0.48. As noted in §3, ResidueFree is designed for standard system configurations and cannot provide privacy guarantees when system administrators install enhanced logging capabilities that are specifically designed to capture and record all computer activity (e.g., kernel-level provenance tracking [5, 24]). We emphasize that the "stock" versions of popular distributions (e.g., Ubuntu, RedHat/Fedora, Debian, Mint, etc.) should be largely compatible with ResidueFree, but may require small changes to how ResidueFree modifies system processes. [2] Finally, ResidueFree requires administrative privileges (e.g., `sudo`) to switch into residue-free mode; the application itself runs with the user's regular (non-administrative) privileges.

We implement the volatile RAM disk by encrypting files as they are written to RAM. We mount a temporary filesystem (tmpfs), then mount an encrypted filesystem (ecryptfs [17]) on top of it. ResidueFree initializes

2 For example, our implementation targets logs written by the Gnome desktop environment, so implementations targeted at KDE distributions would have to account for KDE-specific logs.

and mounts the encrypted file system with a random key, and adds the key and its cryptographic hash to the keyring. ResidueFree enables access to the files in the tmpfs filesystem only while the ecryptfs filesystem remains mounted. The tmpfs filesystem is initially set to one gigabyte, but the size can be configured by the user. In edge-cases where the tmpfs filesystem is larger than available RAM, data in the tmpfs filesystem may be written to swap space. Though this is not ideal, the encryption provided by the ecryptfs filesystem prevents any meaningful residue from being retrieved by an adversary.

We construct the union filesystem using the unionfs package [25]. Unfortunately, Docker currently does not allow root (i.e., /) filesystem mounts, which is necessary to ensure that the application running in residue-free mode functions as it would if it were not in residue-free mode. We sidestep this constraint by separately mounting the computer's top-level directories. The union mounts have the copy-on-write option set (among other options) so that files modified in the Docker session are copied in their entirety to the RAM disk. Once a modified or newly created file is on the RAM disk, future read operations read from RAM rather than the old version on the user's filesystem. After mounting, ResidueFree sets the permissions of the directories in the union filesystems to match the permissions in the user's filesystem.

A particularly challenging aspect of ResidueFree is to provide an isolated execution environment while simultaneously allowing the application running in residue-free mode to interact with the necessary system processes/daemons to ensure the program operates correctly. Rather than construct our own isolated execution environment, we use Docker, which provides strong isolation and has a mature codebase. We mount the `/dev` filesystem inside the Docker container and run the container as privileged to allow the application to interface with the system's devices. However, simply mounting the union filesystems and `/dev` into a Docker container still does not provide sufficient access to the user's system resources for a functional user experience. To ensure sufficient access, we bind-mount critical unix sockets from the host filesystem into the union filesystems and keep ResidueFree in the same network namespace as the host. All sockets in `/tmp/.X11-unix/`, `/tmp/.ICE-unix/`, and the `/run/user/<UID>/` directories, as well as the `/run/lock` and `/run/dbus` sockets, are bind-mounted into their respective union filesystem directories. In addition, running ResidueFree in the same network namespace as the host allows applications to access abstract unix sockets set up by system processes.[3] These steps allow processes running inside ResidueFree to have transparent access to daemons, GUI components, and other critical system resources.

While these steps prevent *applications* running inside ResidueFree from writing data to disk, operating system features outside of ResidueFree may still write sensitive data to disk, especially given that ResidueFree intentionally provides access to system processes. To address this challenge, ResidueFree modifies the main filesystem's mount options, mlocate, the system journal daemon, the syslog daemon, the system apport service, the user's pulse audio daemon, the user's keyring, and the user's Gnome shell. Before starting the Docker container, ResidueFree remounts the main filesystem with the noatime (no access time) option to ensure that execute and access operations are not tracked inside ResidueFree. ResidueFree updates mlocate's configuration file to not update file names under a ResidueFree mount point, restarts the system journal daemon with all journal logs written to none (instead of the default disk), and stops the syslog and apport services from running. ResidueFree also stops the user's pulse audio and keyring daemons and then starts them again inside the ResidueFree container.[4] Finally, ResidueFree sets the current Gnome shell to stop recording application use and sets the file where Gnome stores application use to read-only.

After ResidueFree exits, the ecryptfs is immediately unmounted. At this point, none of the data in ResidueFree's write-cache is logically recoverable, as the key for its encrypted contents is not stored in any process and ecryptfs clears the keyring when the corresponding filesystem is unmounted. The temporary filesystem unmount follows, and all directories that ResidueFree creates as well as the RAM disk's (encrypted) contents are removed. ResidueFree then

3 The most important of these is the user's session dbus. Certain applications will only interact with the abstract socket connection to the user's dbus and refuse connections to the filesystem socket connection.

4 Pulse audio, by default, tracks the names of applications that produce sound in a file. Applications can store user information, such as their login state, in the keyring. When these applications restart inside ResidueFree, these file changes are written to the RAM disk.

restores the seven system modifications to their original state.[5]

To address unexpected exits and ensure the proper removal of any residue, ResidueFree handles interrupt and termination signals by executing this cleanup process. ResidueFree also launches a detached background process that will execute this cleanup process if ResidueFree exits and cannot do so itself (i.e., after receiving a kill signal). In a scenario where the entire computer unexpectedly shuts down, the encrypted filesystem and RAM disk unmount - leaving the residue unrecoverable.

As we describe in more detail in §7.1, our evaluation of the user's filesystem contents reveal that no residue exists from the application run in residue-free mode, nor is there any evidence that the particular application was run in residue-free mode (or at all).

Docker-based isolated execution environment. We use a Docker container to construct our isolated execution environment because of Docker's well-documented, well-maintained features that allow us to incorporate our filesystem modifications without re-implementing Linux's isolation capabilities. Docker's popularity makes user-customization and additional open-source contributions to ResidueFree much simpler while allowing us to make modular backend filesystem changes. While we could likely cut performance costs by only using the containerization features necessary for ResidueFree, these extensive modifications are outside the scope of our proof-of-concept implementation.

Notably, our ResidueFree container is far outside of Docker's typical use case. While Docker is typically used to run lightweight, single applications securely separated from the host operating system, our "container" provides access to every file on the host file system, and does not provide traditional Docker security assurances. A remote attacker who gains access via a process running inside ResidueFree will not only have full read-access to the entire filesystem, but the numerous system resources and privileged configurations granted to the Docker container make container-escaping trivial.

Forensic mode. To increase ResidueFree's usefulness, our implementation also includes a "forensic" mode in which, instead of writing all modified files to RAM, the user can opt to write all modified files to their filesystem. Conceptually, forensic mode allows the user to determine exactly what residue is left after running an application. This mode operates identical to normal ResidueFree, except that it does not mount a tmpfs nor ecryptfs filesystem. Instead, ResidueFree's forensic mode mounts the read-write portion of the unionfs filesystems in a folder of their choosing. When forensic mode exits, the write-cache is unmounted and all its files and directories have their owner set to the user. To reduce the risk of any potentially malicious files added during the ResidueFree session, ResidueFree removes the execute permissions from executable files. The residue described in Table 1 was collected using forensic mode.

Persistent Files. Conceivably, a user may want to store select files from their ResidueFree session, such as a word processing document or a PDF file downloaded via web browser. To enable these use cases, ResidueFree creates a specific folder in the user's home directory for files the user does not want deleted. When ResidueFree exits, these files are transferred to the user's desktop. To prevent the inadvertent deletion of desired files, ResidueFree displays a brief message when launched reminding the user that all files not stored in the persistent folder will be deleted at the end of their ResidueFree session.

Optimizations and additional features. ResidueFree includes a number of options during installation and at run-time for the user to tailor their use of ResidueFree. When running ResidueFree for the first time, users have the option to enable the Docker daemon to startup with system boot. If enabled, it will reduce the costs of launching ResidueFree the first time after system boot, but will stand out on many client machines and may not be ideal for users wishing to raise as little attention as possible to the fact that ResidueFree is on the system (we discuss associated performance costs in §7.3).

A core design goal of ResidueFree is to make the interface as intuitive, unobtrusive, and user-friendly as possible. The user can opt to add a right-click option for every desktop application to run in ResidueFree mode, as shown in Figure 2.

Once installed, users can run ResidueFree in either privacy (the default) or forensic mode. In privacy mode, users can specify the size of the RAM disk. In

5 The Gnome shell updates application use every five minutes, so ResidueFree launches a background process that only restores Gnome application use recording five minutes after ResidueFree exits. All other system configurations are restored on ResidueFree's exit.

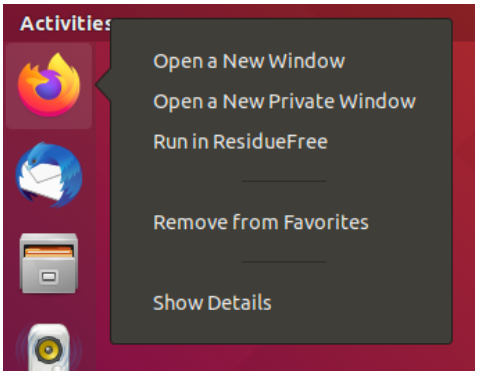

Fig. 2. Application right-click menu with an option to run the application in residue-free mode.

forensic mode, the user can specify the output directory's location and whether to compress the output file.

Operating system portability. Our initial implementation of ResidueFree operates on Linux. Because it uses Docker to help enforce sandboxing and Docker itself uses a Linux kernel to drive its containers, ResidueFree cannot straightforwardly be modified for other operating systems. (Although Docker can run on OSX, the kernel used inside containers is still Linux, which would not permit local OSX applications to be run.) However, much of the containerization enabled by Docker can be duplicated on OSX, and we have made significant progress in doing so towards a future OSX-compatible release of ResidueFree. ecryptfs is also Linux-dependent, but ResidueFree is agnostic to the cryptographic filesystem that protects the ramdisk, and could easily be adapted to support a FUSE-based solution (e.g., EncFS[6]). We are planning to fully support OSX in a future releases of ResidueFree, which we hope will lead to more widespread adoption.

Like OSX, Windows has support for FUSE filesystems. FUSE-based copy-on-write filesystems exist, and coupled with FUSE-based encrypted filesystems, many of the logical components of ResidueFree are currently available for Windows. However, Windows presents unique challenges such as the common use of shared keystores (e.g., the Windows registry) as well as the use of isolation features that significantly differ from those of Linux and OSX. Consequently, enabling Windows support remains a more distant future goal.

7 Evaluation

We conducted both forensic and performance analyses to evaluate our proof-of-concept implementation. For our forensic evaluation, we compared the filesystems of two different virtual machines across three different states: before running ResidueFree, immediately after running a series of applications within ResidueFree, and ten minutes after ResidueFree exits. For our performance analyses, we used the Iozone[7] and Phoronix[8] performance benchmark suites and timed tasks in common user applications to analyze the filesystem and general system performance of ResidueFree. Finally, we timed ResidueFree's startup and shutdown overhead times.

We ran our performance analyses on an Ubuntu 18.04 machine running directly on a Toshiba Satellite E45W-C4200X with a two-core 2.10 GHz Intel i3-5015U processor, 6 GB of RAM, and a 500 GB HDD. Which is to say, we ran our performance tests on a (metaphorical) potato to demonstrate its functionality for users with access to limited hardware resources.

7.1 Forensic Evaluation

The primary goal of ResidueFree is to eliminate any filesystem changes a user or process makes while using ResidueFree. Our implementation should leave no residue on the filesystem, and ideally nowhere on disk, that could be useful in a forensic examination. Our forensic analysis shows ResidueFree leaves no information about what processes the user ran inside ResidueFree and what file(s) or filesystem data those processes may have accessed. While an examiner can determine that ResidueFree is installed on the computer and the times a user ran ResidueFree, no further information is left in the filesystem.

For our analysis, we compared the filesystems of two virtual machines (VMs) across three states. First, we created an Ubuntu 18.04 virtual machine (VM), downloaded a series of applications and installed ResidueFree, and then cloned the VM to ensure the two VMs would be as close to identical as possible. While one VM, which we refer to as the "baseline VM," remained active in the background, we used the other VM, or "ResidueFree VM", to launch ResidueFree and in-

6 https://github.com/vgough/encfs

7 http://www.iozone.org/

8 https://www.phoronix-test-suite.com/

teract with several applications inside ResidueFree. After exiting ResidueFree, we immediately froze both VMs and made copies of their virtual disks. We then restored the ResidueFree VM, let it run for an additional ten minutes, and then froze the VM and copied its virtual hard disk again.[9] We then mounted read-only copies of these filesystems in a separate machine, generated hashes of each file in all three states (including the virtual disks' swap files), and manually inspected the contents of any files whose hashes differed between the baseline VM and either of the ResidueFree VM's filesystem snapshots.

Within the ResidueFree VM, we first connected to ProtonVPN using an OpenVPN configuration file. We then opened LibreOffice Writer to create and save a short document, ran Firefox to generate a favicon-based "supercookie" from https://supercookie.me/, and played an entire Youtube video. While Firefox was running, we intentionally caused Chromium to crash. We then opened Chromium successfully and downloaded a PDF, ended ProtonVPN, and opened Nautilus to select the PDF and read it using Evince. Next, we opened Telegram and sent a message, then downloaded a simple `.txt` file from another computer on the local network using Filezilla, and finally used apt to install a new application (gnome-gmail). We then exited ResidueFree and immediately froze the VM.

Thirty-nine files differed between the baseline VM and the ResidueFree VM both immediately after ResidueFree ran and after ten minutes idling. We manually analyzed each file to determine that no sensitive information from the ResidueFree session was part of these differences. We determined the differences were caused by either routine filesystem activity (e.g., timestamps differing by a few seconds or different unique identifiers for system resources) or indications that ResidueFree ran (e.g., Docker and Containerd logs noting ResidueFree's container invocation or log files that tracked the ResidueFree launch command). In the latter case, none of the files indicated which application was run in residue-free mode.

Notably, ResidueFree removes common residue identified by trained forensic examiners. Nishida [21] describes locations in the Ubuntu operating system for reliably identifying forensic artifacts left by the operating system and the Gnome Desktop Environment.[10] For application downloads and use, he points to system log files, application installation files, journal files, Gnome's file for tracking application use, Gnome's file for tracking recently opened files, and a handful of additional miscellaneous files in the user's cache or that are application-specific. ResidueFree ensures that none of the described forensic artifacts are written to disk.

In addition, ResidueFree did not preserve any files from Firefox or Chromium, including favicons where the supercookie attempts to persist between sessions. As Solomos et al. [27] note, these favicon-based supercookies persisted past all major browsers' "private" or "incognito" modes. While these incognito modes are generally effective, they are vulnerable to novel persistent storage techniques like this one. By ensuring that *all* file modifications never touch the user's disk, no matter an application's settings, ResidueFree protects against such user-tracking attacks.

7.2 Filesystem Costs

Since ResidueFree effectively intercepts file operations to prevent application residues, we performed a detailed evaluation of ResidueFree's file I/O overheads. That is, we sought to answer the question, *what is the performance penalty of using* ResidueFree*?* We used the Iozone Filesystem Benchmark suite to evaluate 13 different filesystem operations on a baseline standard mode without ResidueFree, on ResidueFree without the ecryptfs mount on top of the RAM filesystem, and on the fully-fledged (with ecryptfs) ResidueFree. We ran the suite 30 times for each category, and took an average across block sizes and file sizes for each file operation in each run. Figure 3 shows the average of those 30 averages for each file operation, with error bars denoting the standard distribution. For the ResidueFree tests, we ran the Iozone Benchmark 30 times in a row in the same ResidueFree session.

On average, ResidueFree file operations run at 30.1% the speed of standard mode file operations: 636.9 vs. 2113.8 MB/s. Without ecryptfs, ResidueFree runs 40.6% as fast as standard mode, averaging 858.9 MB/s. ResidueFree runs 74.1% as fast as ResidueFree with encryption disabled; the difference between them almost entirely comes from write operations (when the en-

9 We took the ResidueFree VM's state twice to ensure that system processes with delayed writes, namely Gnome's application-use tracking described in §6, did not generate any residue.

10 While Nishida [21] evaluates Ubuntu 20.04 and we evaluate Ubuntu 18.04, almost all forensic artifacts remain the same.

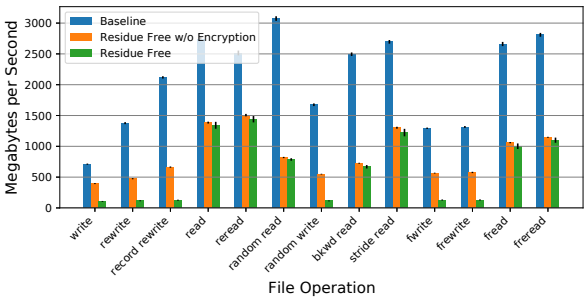

Fig. 3. Filesystem operation speeds from Iozone Benchmark tests.

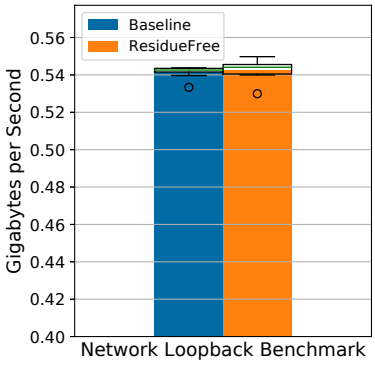

Fig. 4. Transfer speed for 10 GB of data over the loopback interface from the Phoronix network-loopback benchmark.

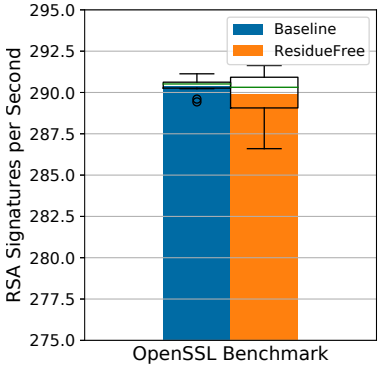

Fig. 5. RSA signing speeds from the Phoronix OpenSSL benchmark.

cryption layer actually operates on the files). ResidueFree runs at 22.3% the speed of ResidueFree without encryption on write operations (120 vs. 537 MB/s), but 95.1% as fast on read operations (1000 vs. 1134.8 MB/s). Finally, we note that the performance costs shown in Figure 3 are not weighted by the frequency of the individual file operations.

7.3 Additional Performance Considerations

Impacts on network, CPU, and RAM speed. In addition to the filesystem costs, we evaluated ResidueFree's impacts on other key system performance measures. We used the Phoronix Test Suite to evaluate network device, processing, and RAM speeds, with and without ResidueFree.

We ran all Phoronix Benchmarks 10 times consecutively in both standard mode and in ResidueFree. We ran all 10 ResidueFree runs in the same ResidueFree session. Unlike our other tests, the Phoronix runs did not fall into a uniform distribution. In particular, tests running in ResidueFree were more prone to variance. However, the results show ResidueFree performs approximately the same as standard mode in these non-filesystem benchmarks.

To measure performance on network devices, we used Phoronix's network-loopback benchmark[11] to measure data transfer speeds over the loopback interface. Transfer speeds for both ResidueFree and standard mode are practically equivalent, with ResidueFree averaging 11 MB/s faster than standard mode. Since ResidueFree shares a network namespace with the host, these results are expected. We show these results in Figure 4.

To measure processing performance, we used Phoronix's OpenSSL[12] and API-Test[13] benchmarks. The OpenSSL benchmark measures how fast the system calculates RSA signatures, while the API benchmark measures how many frames per second the system can render using various graphics API operations.

On average, ResidueFree performs 4096-bit RSA signatures at 99.8% the rate of standard mode, calculating 289.9 vs. 290.4 signatures per second. We show these results in Figure 5.

We ran all 22 of the API benchmark's per-API tests, though only 20 of them successfully ran on our system (both inside and outside of ResidueFree). We ran the tests at a 1024x768 resolution. ResidueFree performs graphics API operations at 98.5% the rate of standard

11 https://openbenchmarking.org/test/pts/network-loopback

12 https://openbenchmarking.org/test/pts/openssl

13 https://openbenchmarking.org/test/pts/apitest

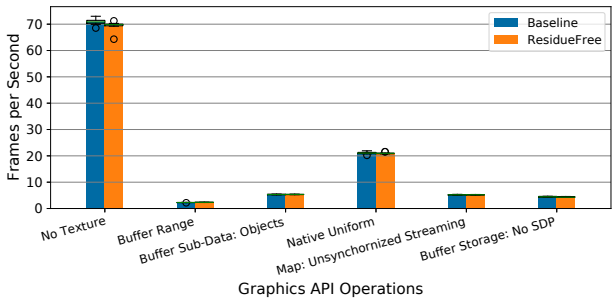

Fig. 6. Subset of Graphics API Tests from Phoronix API-Test Benchmark

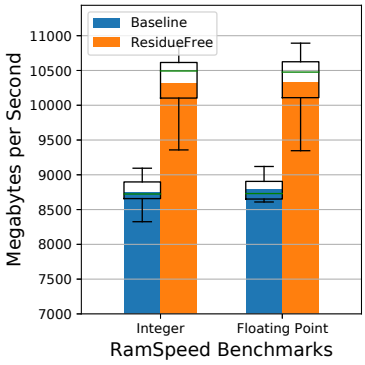

Fig. 7. RAM speeds from Phoronix RamSpeed benchmark.

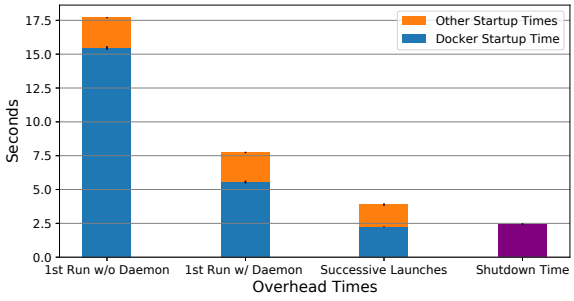

Fig. 8. ResidueFree startup and shutdown overhead times.

mode, registering an average of 13.83 vs. 14.04 frames per second across the 20 different benchmarks. As with our filesystem benchmarks, this average is not weighted by the frequency of the operations in actual software. We provide a representative set of the API benchmarks in Figure 6.

Finally, we use Phoronix's RamSpeed benchmark[14] to measure how fast the system transfers data into and out of RAM when the processor is performing various calculations on a large set of either integer or floating point numbers. We ran both integer and floating point tests on an average of the copy, scale, add, and triad operations. ResidueFree's RAM speed is notably faster than standard mode, transferring data through RAM at an average of 117.7% of standard mode, 10325 vs. 8771 MB/s, between integer and floating point storage. We have not isolated a cause for either ResidueFree higher average performance over the baseline nor decreased performance in the first two outlier runs. We posit that ResidueFree's use of a RAM disk may yield positive caching effects. These results are shown in Figure 7.

Startup and shutdown costs. Finally, we calculated the overhead costs of starting up and shutting down ResidueFree. As noted in §6, ResidueFree takes a noticeably longer time to launch when the Docker daemon is not configured to startup with system boot. Even with the daemon enabled, the first ResidueFree launch after system boot is still slower than subsequent launches. We have not noticed any variables that impact shutdown times.

We define the startup time to be the time between the invocation of ResidueFree and the start of the application running in residue-free mode. To measure the time required to launch the first ResidueFree invocation after system boot, we restarted our machine, waited until all boot processes finished running and CPU usage was consistently below .2% (approximately five minutes), and repeated the above procedure. We collected startup times 10 times for the initial ResidueFree execution without and with the Docker daemon enabled. When measuring successive launches, we ran our modified ResidueFree 30 times.

Similarly, we define the exit time to be the time between the application's exit and ResidueFree's termination. We ran this version of ResidueFree 30 times. All of our overhead times were tightly distributed and showed little variance.

As shown in Figure 8, the first ResidueFree session that is launched after system boot without the use of the Docker daemon requires 17.7 seconds, on average. The first launch with the daemon enabled takes 7.7 seconds, and successive launches require 3.9 seconds. Of those times, the Docker container takes 15.5, 5.6, and 2.2 seconds to launch; and all operations outside of Docker (i.e., creating the encrypted RAM disk, mounting the union filesystems, and modifying system processes) take 2.2 seconds or less.

14 https://openbenchmarking.org/test/pts/ramspeed-1.4.2

Importantly for our privacy goals, ResidueFree exits quickly, averaging 2.4 seconds to shutdown among the 30 exits we tested. The quick exit is important for a user who needs to quickly remove access to filesystem residue, and we prioritized the quick exit time by amortizing the costs of erasing residue as files are encrypted and written to a RAM disk. In our implementation, ResidueFree only needs to destroy the encryption key to ensure residue is unrecoverable. Alternative implementations that remove file residue by overwriting all of the RAM disk when the user exits a ResidueFree session would improve filesystem performance but noticeably increase exit times.

7.4 Use-Case Performance Measurements

In addition to our (micro)benchmarking experiments, we also measured the more "holistic" cost of using ResidueFree by considering the total overhead (measured in time) of performing simple tasks within ResidueFree, relative to these tasks' execution without ResidueFree. We designed our tasks around typical user applications—a subset of those discussed in §4—and executed these tasks ten times in both ResidueFree and standard mode. The results of our use-case-based measurements are shown in Figure 9 and described in more detail below.

LibreOffice. We timed how long it took LibreOffice to convert five 1 MB open document (.odt) files to pdfs. On average, it took standard mode 12.6 and ResidueFree 19.7 seconds (↑56.59%) to convert the files.

SFTP. To test how long it would take an application to read a file from disk and send it over the network, we timed how long it took SFTP to send an 81 MB file to another computer on the local network. On average, it took standard mode 5.4 seconds and ResidueFree 4.9 seconds (↓9.8%) to transfer the file.

VLC. To test how long it would take a video to load from disk, we timed how long it took VLC to load a 24 second, 21 MB, 1080p mp4 file from disk and completely play it. On average, it took standard mode 26.1 seconds and ResidueFree 26.8 seconds (↑2.68%) to load and play the video.

wget. To test file download speeds, we timed how long it took `wget` to download a 1 GB file from a computer on the local network. On average, it took standard mode 40.8 seconds and ResidueFree 39.9 seconds (↓2.06%) to download the file.

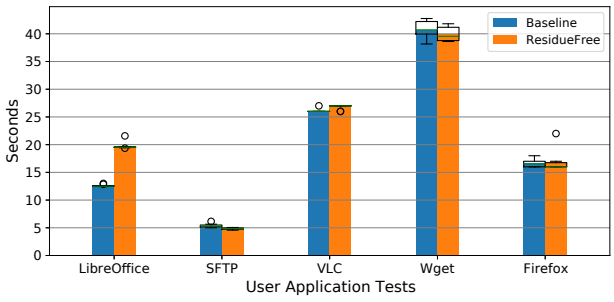

Fig. 9. Runtimes to complete tasks on typical user applications.

Firefox. To test internet browsing speeds, we used Selenium to automate web browser interaction. We timed how long it took Firefox, driven by Selenium, to visit 10 different Wikipedia pages followed by youtube.com. On average, it took standard mode 16.6 seconds and ResidueFree 16.8 seconds (↑1.20%) to visit the pages.

Performance summary and experiential assessment. In summary, ResidueFree incurs moderate performance penalties when conducting file-heavy operations, while graphic-, network-, and RAM-based operations are mostly cost free.

Though there is a significant cost to file operations, these operations make up a small enough portion of a typical user application's workload, outside specifically file-heavy tasks, that they create no more than a few seconds delay for small tasks.

8 Limitations

While we include a number of user options and though many sophisticated applications run seamlessly in ResidueFree, there is room for improvement. Some straightforward improvements include allowing multiple ResidueFree sessions at the same time, deleting additional system logs that track ResidueFree runs and Docker container changes, and adding options to obfuscate the ResidueFree runtime or delete itself after running.

More challenging limitations are imposed by the applications that do not run inside ResidueFree. During our testing, we found that applications packaged as "snaps" do not run in ResidueFree due to the snap daemon's inability to either connect into ResidueFree via a socket bind-mount or run independently inside of ResidueFree. Similarly, applications that re-

quire "daemonizing" or running in the background in conjunction with multiple other system processes, such as specific VPN solutions or server applications (e.g., Plex Media Server), do not run inside the ResidueFree container. Determining appropriate levels of interaction with background system processes while maintaining a lightweight, containerized implementation of the ResidueFree execution environment will be the focus of future work as we release our current ResidueFree implementation as open-source code.

In addition, our contributions and evaluation focus on the technical aspects of residue-free computing and our ResidueFree implementation. Though we designed ResidueFree with common user applications, restraints faced by different user communities, and overall usability in mind, we did not perform any usability reviews or consult with different user communities. We believe ResidueFree makes significant usability contributions, but recognize this has not been validated by lay users. Our proof-of-concept runs on Linux. As we discuss in §6, while our implementation does not straightforwardly map to Windows, the overall architecture may still serve as an effective roadmap towards achieving future Windows support.

Finally, residue-free computing defines residue as data left on the filesystem. We do not take steps to ensure that data left in RAM by processes run in residue-free mode is removed, nor does ResidueFree prevent processes' memory from being swapped to disk. We are aware of services that address these limitations (for example, Linux's grsecurity extensions [15] or the use of encrypted swap).

9 Conclusion

This paper presents residue-free computing, an anti-forensics privacy-enhancing technology that enables an "incognito"-like privacy mode for existing applications, without requiring kernel modifications. We motivate residue-free computing by exploring the residue (digital breadcrumbs) left by popular Linux applications, and show that such residue can contain highly sensitive information, potentially unbeknownst to the applications' users.

Our implementation of residue-free computing, ResidueFree, enables users to run their existing applications in a mode that avoids leaving residue. We explore the effectiveness of ResidueFree by performing a forensic investigation on both unprotected and ResidueFree-protected invocations of an application, and show that ResidueFree effectively eliminates application residue.

We evaluate the performance costs of ResidueFree, and determine that they are moderate. Our benchmark-based evaluation reveals that the privacy protections provided by ResidueFree come at the cost of slower file operations, although common user tasks did not result in a significant or intolerable slowdown.

Our overarching aim in developing ResidueFree was to provide a user-friendly mechanism that enables users to use existing applications on their current computer platforms in a more privacy-preserving and safe manner. Towards that goal, ResidueFree is released as free and open-source software and is available for download at https://larkema.github.io/residuefree/.

Acknowledgments

We thank the reviewers for their insightful comments and suggestions, and Clay Shields for several fruitful conversations that helped inspire this work. This work is partially funded by the National Science Foundation under grant CNS-1718498 and the Callahan Family Professor of Computer Science Chair Fund.

References

[1] TrueCrypt. http://truecrypt.sourceforge.net/.

[2] Tails Portable Operating System. https://tails.boum.org/, 2020.

[3] Pietro Albano, Aniello Castiglione, Giuseppe Cattaneo, and Alfredo De Santis. A Novel Anti-forensics Technique for the Android OS. In *International Conference on Broadband and Wireless Computing, Communication and Applications*, 2011.

[4] Ross Anderson, Roger Needham, and Adi Shamir. The Steganographic File System. In *International Workshop on Information Hiding*, 1998.

[5] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. Trustworthy Whole-system Provenance for the Linux Kernel. In *USENIX Security Symposium (Security)*, 2015.

[6] Neil Brown. Overlay Filesystem. https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html.

[7] Chen Chen, Anrin Chakraborti, and Radu Sion. INFUSE: Invisible Plausibly-deniable File System for NAND Flash. *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2020(4):239–254, 2020.

[8] Alexei Czeskis, David J St Hilaire, Karl Koscher, Steven D Gribble, Tadayoshi Kohno, and Bruce Schneier. Defeating Encrypted and Deniable File Systems: TrueCrypt v5. 1a and the Case of the Tattling OS and Applications. In *USENIX Workshop on Hot Topics in Security (HotSec)*, 2008.

[9] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium (USENIX)*, August 2004.

[10] Judicael Briand Djoko, Brandon Jennings, and Adam J Lee. Tprivexec: Private Execution in Virtual Memory. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2016.

[11] Christian DOrazio, Aswami Ariffin, and Kim-Kwang Raymond Choo. iOS Anti-forensics: How can we Securely Conceal, Delete and Insert Data? In *Hawaii International Conference on System Sciences (HICS)*, 2014.

[12] Tariq Elahi, George Danezis, and Ian Goldberg. PrivEx: Private Collection of Traffic Statistics for Anonymous Communication Networks. In *ACM Conference on Computer and Communications Security (CCS)*, November 2014.

[13] Xianyi Gao, Yulong Yang, Huiqing Fu, Janne Lindqvist, and Yang Wang. Private Browsing: An Inquiry on Usability and Privacy Protection. In *Workshop on Privacy in the Electronic Society (WPES)*, 2014.

[14] Simson Garfinkel. Anti-forensics: Techniques, Detection and Countermeasures. In *International Conference on i-Warfare and Security*, 2007.

[15] grsecurity. https://grsecurity.net/.

[16] Hana Habib, Jessica Colnago, Vidya Gopalakrishnan, Sarah Pearman, Jeremy Thomas, Alessandro Acquisti, Nicolas Christin, and Lorrie Faith Cranor. Away from Prying Eyes: Analyzing Usage and Understanding of Private Browsing. In *Symposium on Usable Privacy and Security (SOUPS)*, 2018.

[17] Michael Austin Halcrow. eCryptfs: An Enterprise-class Encrypted Filesystem for Linux. In *Linux Symposium*, 2005.

[18] Junjiro R. Okajima. Aufs: Another Union Filesystem. http://aufs.sourceforge.net/.

[19] Samy Kamkar. Evercookie – Virtually Irrevocable Persistent Cookies. https://samy.pl/evercookie/.

[20] Andrew D McDonald and Markus G Kuhn. StegFS: A Steganographic File System for Linux. In *International Workshop on Information Hiding*, pages 463–477, 1999.

[21] Brian Nishida. Ubuntu Artifacts Generated by the Gnome Desktop Environment. Technical report, SANS Institute, 2020.

[22] Kaan Onarlioglu, Collin Mulliner, William Robertson, and Engin Kirda. PRIVEXEC: Private Execution as an Operating System Service. In *IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[23] Zachary NJ Peterson, Randal C Burns, Joseph Herring, Adam Stubblefield, and Aviel D Rubin. Secure Deletion for a Versioning File System. In *USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[24] Devin J Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. Hi-Fi: Collecting High-fidelity Whole-system Provenance. In *Annual Computer Security Applications Conference (ACSAC)*, 2012.

[25] David Quigley, Josef Sipek, Charles P Wright, and Erez Zadok. Unionfs: User-and Community-oriented Development of a Unification Filesystem. In *Linux Symposium*, 2006.

[26] Joel Reardon, David Basin, and Srdjan Capkun. SoK: Secure Data Deletion. In *IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[27] Konstantinos Solomos, John Kristoff, Chriss Kanich, and Jason Polakis. Tales of favicons and Caches: Persistent Tracking in Modern Browsers. In *Network and Distributed Systems Security Symposium*, 2021.

[28] Meng Xu, Yeongjin Jang, Xinyu Xing, Taesoo Kim, and Wenke Lee. UCognito: Private Browsing without Tears. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

A Application Residues

Table 1 presents the residues of popular applications when not running in residue-free mode.

Table 1. Residues of popular applications running on Ubuntu Linux 18.04.

Application	Category	Residue
0AD	Gaming	0AD records detailed logs of gameplay (i.e., every command issued alongside a timestamp), system performance logs, and timestamps for the user's most recent session.
Discord	Comm.	Discord leaves the names of the channels that the user viewed, usernames (both the user's and others in the channel that the user messaged), the user's login email, various application settings, message drafts and, in some cases, recent messages.
Eclipse	Dev. Tool	Eclipse leaves configuration settings, and user and program metadata.
Evince	Productivity	Evince, Ubuntu's default PDF reader application, does not specifically track which files it accessed. However, Gnome tracks the number of times each desktop application, including Evince, opens a file and the most recent time each application opened the file.
Filezilla	Productivity	After connecting to an FTP server, Filezilla leaves as residue the server's IP address, the login username, and the user's base-64 encoded password (using default settings), and the most recent local directory used.
LibreOffice	Productivity	LibreOffice tracks the name of each file it opens, even if the file is deleted, and updates the modified timestamp of those files. In addition, Gnome tracks that LibreOffice opened or created a given file like it does with Evince. Further, LibreOffice logs timestamps for each time LibreOffice starts.
Minetest	Gaming	When playing as a client on another server, Minetest stores the IP address of the server and the usernames of all players on the server. When playing locally, Minetest stores user settings and game save data.
ProtonVPN	Comm.	ProtonVPN, when connecting to VPN servers using OpenVPN, modifies an OpenVPN configuration file and the operating system logs the creation of a new IP interface. ProtonVPN uses both OpenVPN and a command line tool for connecting to its servers. As the command line tool is currently in beta, our analysis is based on the connection process described at https://protonvpn.com/support/linux-vpn-setup/.
Signal	Comm.	Signal did not leave any identifiable sensitive information from the user's session, but left detailed logging information in dedicated files that reveal when the user ran Signal.
Skype	Comm.	Skype records retain a significant amount of information including timestamps for when a call started, when the user joined the call, when the call ended, whether or not they hosted the call, and chat messages.
Slack	Comm.	Like Skype, Slack records the workspaces the user was logged into, the channels the user had access to in those workspaces, user information - such as names, emails, and links to profile pictures - for all workspace members, and the contents and timestamps for channel and draft messages.
Spotify	Aud./Vid.	Spotify leaves residue data (e.g. title, artist, album) for all songs presented to the user, timestamps for the song the user was playing, and social media (e.g., Facebook) account information (login username and friend activity).
Telegram	Comm.	Telegram did not leave any identifiable sensitive information from the user's session, but left detailed logging information in dedicated files that reveal when the user ran Telegram.
VLC	Aud./Vid.	VLC updates the modified timestamp of files it accesses and tracks recently played media in a separate file.
Web Browsers Brave Firefox Chromium Opera Tor	Comm.	Firefox, Chromium, and Opera running in their default operating mode left considerable, but well-known, residue such as cookies and cached data, making it trivial to determine the user's web activity. However, we were unable to determine user activity in these browser's "private" or "incognito" modes, nor we were able to identify user activity from standard Tor and Brave sessions. Our analyses included searches for data from specific sites visited, images loaded, and cached data stored using evercookies [19]. All browsers left sufficient logging information in application-specific log files to reveals when the user ran the browser.