IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received January 26, 2022, accepted February 5, 2022, date of publication February 21, 2022, date of current version March 3, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3153064

Defense and Attack Techniques Against File-Based TOCTOU Vulnerabilities: A Systematic Review

RAZVAN RADUCU, RICARDO J. RODRÍGUEZ, (Member, IEEE), AND PEDRO ÁLVAREZ
Department of Computer Science and Systems Engineering, University of Zaragoza, 50009 Zaragoza, Spain

Corresponding author: Ricardo J. Rodríguez (rjrodriguez@unizar.es)

This work was supported in part by the Spanish Ministry of Economy and Competitiveness under Grant PDC2021-121072-C22, in part by the University, Industry and Innovation Department of the Aragonese Government under Programa de Proyectos Estratégicos de Grupos de Investigación (DisCo Research Group) under Grant T21-20R, in part by the University of Zaragoza, and in part by the Fundación Ibercaja under Grant JIUZ-2020-TIC-08. The work of Razvan Raducu was supported by the Government of Aragon through the Diputación General de Aragón (DGA) Predoctoral Grant, during 2021–2025.

ABSTRACT File-based *Time-of-Check to Time-of-Use* (TOCTOU) race conditions are a well-known type of security vulnerability. A wide variety of techniques have been proposed to detect, mitigate, avoid, and exploit these vulnerabilities over the past 35 years. However, despite these research efforts, TOCTOU vulnerabilities remain unsolved due to their non-deterministic nature and the particularities of the different filesystems involved in running vulnerable programs, especially in Unix-like operating system environments. In this paper, we present a systematic literature review on defense and attack techniques related to the file-based TOCTOU vulnerability. We apply a reproducible methodology to search, filter, and analyze the most relevant research proposals to define a global and understandable vision of existing solutions. The results of this analysis are finally used to discuss future research directions that can be explored to move towards a universal solution to this type of vulnerability.

INDEX TERMS File-based race condition, TOCTOU vulnerability, avoidance techniques.

I. INTRODUCTION

Today, many applications are deployed on large-scale distributed systems and multi-core processors, which perform multiple tasks concurrently while sharing common resources such as memory, disk, or network. The intrinsic characteristics of the simultaneous execution of programs make them very difficult to write, test, and debug [1], [2], which facilitates the existence of concurrency bugs.

Concurrency bugs are caused by accesses to a shared resource between threads and processes without proper synchronization. These bugs can lead to vulnerabilities that, when triggered by adversaries, can cause a much broader impact on security, such as bypassing security checks, breaking the integrity of databases [3], hijacking the vulnerable program control flow execution, or escalating privileges [4], among others.

A common attack especially related to concurrency bugs is the privilege escalation attack, in which a malicious user gains access to other user accounts on the target system. The number of vulnerabilities related to privilege escalation has been increasing in recent years. For instance, in 2020 this type of vulnerability comprised 44% of all Microsoft vulnerabilities [5]. There are two main types of privilege escalation: *horizontal privilege escalation* attacks, in which an attacker expands their privileges by taking over another (non-privileged) user account and abusing the legitimate privileges granted to the other user; and *vertical privilege escalation* attacks, which involve increasing privileges/privileged access beyond what a user (or an application or other asset) already has.

Vertical privilege escalation attacks are commonly caused by a particular type of concurrency bug, called *race condition bugs*. The root cause of these bugs is a TOCTOU (Time-of-Check to Time-Of-Use) bug, which occurs when a program checks a particular characteristic of an object (e.g., whether the file exists), and later takes some action that assumes the

The associate editor coordinating the review of this manuscript and approving it for publication was Binit Lukose.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

checked characteristic still holds [6]. The window of opportunity that the program leaves between the time of check and the time of use is then exploited by an adversary. The adversary can increase this window by various means, such as overloading the system or creating specific inputs for the vulnerable program. In addition, TOCTOU vulnerabilities are present in different scenarios. For example, memory accesses involving the kernel [7], [8] (also known as double-fetch bugs), Remote Attestation [9], [10], Trusted Computing [11], [12], or file-based TOCTOU [6], [13], among others.

In this paper, we focus on file-based TOCTOU since they are one of the oldest known security flaws, dating back to the mid-70s [14], [15]. These types of race conditions, particularly common on Unix-like systems, occur due to the mapping from a filename to a unique inode and a device number. Although the mapping of the inode and device number to a file descriptor is race-free, the mapping of the filename to the inode and the device number is volatile since filenames and the underlying inode and device number may change on each system call invocation.

A well-known example of this kind of problem is `sendmail` [13], which used to look for a specific attribute of a mailbox file before adding new messages to it. Unfortunately, the verification and append operations are not an atomic unit. Consequently, if an adversary (the mailbox owner) replaces their mailbox file with a symbolic link to sensitive files (such as `/etc/passwd`, which contains information about system user accounts) between the verification and append operations, then `sendmail` will add email contents to `/etc/passwd`. As a result, the adversary can craft an email message to add a new user account with superuser privileges in the system.

Figure 1 illustrates this typical security flaw. The vulnerable code appears on the left side of the figure. On line 6 there is a check of the write permission on a file (identified by a string) with the `access` system call. Once the verification is successful, the file is opened (line 8) and certain data is appended to the file. If this program is run with *setuid* permission (i.e., users can run it with elevated privileges temporarily to perform a specific task), the adversary can take advantage of the race window between the operations on lines 6 and 8. An example of the exploit used by an adversary is shown on the right side of the figure. Suppose the exploit is run to write to the `/etc/passwd` file, which is a protected file in UNIX-based systems. If an adversary iteratively creates a symbolic link to `/etc/passwd` (line 14, right side) at the same time as the execution of the vulnerable program (line 12, right side), the race condition will eventually occur and the attack will succeed, appending new content to the protected file.

Specifically, file-based TOCTOU vulnerabilities[1] are file-based race conditions that occur on filesystems with *weak* synchronization mechanisms (that is, they do not provide methods to ensure that filesystem objects remain unchanged between consecutive interactions with them). Given the non-deterministic nature of race conditions, the success of an attack is highly dependent on the precise and timely actions of the attacker at any given time during the execution of the vulnerable program. Furthermore, the occurrence of this type of vulnerability also depends on certain system calls being executed in a specific order, as well as environmental conditions [13], [16]. Therefore, the reproducibility of these vulnerabilities is typically very difficult.

[1] In the rest of this paper, we refer to file-based TOCTOU vulnerabilities simply as TOCTOU vulnerabilities.

TABLE 1. Common weakness enumerations related to TOCTOU.

CWE ID	Vulnerability
CWE-59	*Improper Link Resolution Before File Access ('Link Following')*
CWE-61	*UNIX Symbolic Link (Symlink) Following*.
CWE-62	*UNIX Hard Link*
CWE-362	*Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')*
CWE-363	*Race Condition Enabling Link Following*
CWE-367	*Time-of-check Time-of-use (TOCTOU) Race Condition* (not only file-based TOCTOU)
CWE-386	*Symbolic Name not Mapping to Correct Object*
CWE-706	*Use of Incorrectly-Resolved Name or Reference*

Despite the age of this security flaw, numerous vulnerabilities are still reported each year related to TOCTOU vulnerabilities. For example, at the time of writing, a query to find TOCTOU-related vulnerabilities returns 786 results in the National Vulnerability Database [17] and 120 results in the MITRE CVE search engine [18], with the newest being only a few days old in both cases. This clearly shows that it is still a significant security problem and that the CVE release for TOCTOU vulnerabilities is common in the software industry. Furthermore, this vulnerability affects projects of any size, such as open-source projects [19], and major software vendors [20]–[22]. *The proof of the pudding is in the eating*: as shown in Table 1, there are several Common Weakness Enumeration (CWE) entries related to TOCTOU. CWEs represent a common language for discussing, finding, and addressing the causes of software security vulnerabilities, currently maintained by the MITRE Corporation. Each individual CWE represents only one type of vulnerability. This paper aims to systematically review the scientific literature in order to find techniques to mitigate TOCTOU vulnerabilities, as well as techniques to exploit these vulnerabilities. Specifically, we review the literature to find out what techniques have been proposed, how they are implemented, how they detect TOCTOU vulnerabilities, which operating system they target, and whether any source code or software tool is available to reproduce the experimental results.

```
1  // toctou.c
2  char *filename = argv[1];
3  // ...
4
5  // Check permissions
6  if(!access(filename, W_OK)){
7      // Open the file
8      file = fopen(filename, "a+");
9
10     // Write to file the user input
11     fwrite(buffer, sizeof(char), strlen(
           buffer), file);
12     fwrite("\n", sizeof(char), 2, file);
13     fclose(file);
14 }else
15     printf("No permission, exiting!\n");
```

```
1  #!/bin/bash
2  # exploit.sh
3  # (execute it as: ./exploit.sh /etc/passwd)
4  TEMPFILE="temp.file"
5  OLD_LS=`ls -l $1`
6  NEW_LS=`ls -l $1`
7
8  while [ "$OLD_LS" == "$NEW_LS" ]
9  do
10     rm -f $TEMPFILE
11     echo "From user" > $TEMPFILE
12     echo "TOCTOU success" | ./toctou
           $TEMPFILE > /dev/null &
13     unlink $TEMPFILE
14     ln -s $1 $TEMPFILE &
15     NEW_LS=`ls -l $1`
16 done
```

FIGURE 1. Example of a file-based TOCTOU vulnerability (left side) and exploit (right side).

In summary, our contributions are the following:

- We conduct a comprehensive review of the literature on defense and attack solutions against TOCTOU vulnerabilities. In particular, we found 37 articles proposing some kind of defense solution and only 4 articles proposing attacks against TOCTOU.
- We propose a taxonomy for TOCTOU defenses and attacks, according to when they perform the vulnerability detection/exploitation and at what level they operate. Furthermore, we classify TOCTOU attacks based on the attack vector they exploit.
- We highlight future research trends and directions regarding defense solutions for TOCTOU vulnerabilities. Our proposals cover modifying current operating system calls to make them race-free and security focused, modifying the kernel to avoid the use of filenames, and the use of transactional filesystems. We provide more details on this matter in Section V-B.

This paper is organized as follows. Section II briefly reviews related work. Section III presents the methodology we followed to carry out the systematic review of the literature, defining the research questions and inclusion and exclusion criteria. The results of the systematic review and the proposed taxonomy are presented in Section IV. A more detailed discussion of the results is provided in Section V, also highlighting trends and directions of future research, as well as limitations of our work. Finally, Section VI concludes the paper.

II. RELATED WORK

In this section, we review the literature related to our work.

Several different TOCTOU vulnerabilities are mentioned in other literature reviews or surveys. The survey in [23] focuses on double-fetch vulnerabilities, which is a vulnerability that occurs when data consistency between the kernel and the user space is violated in a race condition. Vulnerabilities in remote attestation in wireless sensor networks are discussed in [24]. TOCTOU vulnerabilities can also occur in this context, as attesting a node occurs at a particular point in time and does not guarantee that the node was not temporarily compromised before or that it will not be compromised right after the attestation. TOCTOU vulnerabilities due to naming collusion in Android are explored in [25], which provides a systematic review of permission-based Android security.

Unlike these works, our work focuses exclusively on file-based TOCTOU vulnerabilities. Furthermore, the previous works do not provide an in-depth analysis of how this vulnerability is exploited or of existing defense and offensive techniques. To the best of our knowledge, we present the first systematic literature review of file-based TOCTOU vulnerabilities.

III. METHODOLOGY OF THE SYSTEMATIC LITERATURE REVIEW

We conduct a systematic review of the literature following the recommendations given in [26] to find detection, prevention, avoidance or exploitation techniques that are related to TOCTOU vulnerabilities. Systematic literature reviews are methodical, complete, transparent, and replicable studies that allow the compilation of results following reproducible and bias-free research carried out by consulting the main scientific and academic search engines [27].

Next, we explain in detail the methodology that we have followed. We first state the research questions and the search strategy used. We then present the criteria used to select studies for quantitative analysis. Finally, we summarize the number of articles obtained in each execution phase of our review protocol.

A. RESEARCH QUESTIONS

The main objective of this research is to review the literature in the field of prevention, detection, and mitigation mechanisms for TOCTOU vulnerabilities, as well as related exploitation techniques. In particular, we want to know the underlying principles behind mitigating and attacking file-based race conditions, how they affect the host operating

system, whether they are located at the user or kernel-space level, and whether any tool or source code exists to replicate the experimental results. More formally, we formulate the following research questions (RQ):

RQ1.- How do defensive and offensive techniques of file-based TOCTOU vulnerabilities work?
RQ2.- In which regions of the memory do they reside?
RQ3.- When is the vulnerability detected or exploited?
RQ4.- In which operating system is the technique implemented?
RQ5.- Is there any tool or source code available to validate or replicate the experimental results?

B. SEARCH STRATEGY

We consulted various scientific databases that allow the results to be exported for later analysis. In particular, we considered IEEE Xplore, ScienceDirect, Scopus, and ACM since together they cover the main journals and conferences in the field of interest.

The advanced search relied on keywords that were carefully selected and modified throughout the review process to improve the results and fulfill the purpose of our review. Specifically, we started from scratch by conducting a preliminary search with the term "*TOCTOU*" and adding those terms that help us narrow down the results (for instance, synonyms have also been contemplated). The final search string is: (*TOCTOU* **OR** *TOCTTOU* **OR** "*time of check to time of use*") **AND** *file* **AND** (attack* **OR** exploit* **OR** abus***OR** defen* **OR** mitigat* **OR** fix*). We looked for items until the year 2021, without setting any initial year.

As we are only interested in scientific/academic works that have been published in peer-reviewed scientific journals and conferences, other works such as gray literature, books, standards, or patents are discarded from our results. In addition, we carried out a complementary manual search by reviewing the title of the works presented in the Tier-1 and Tier-2 conferences of computer security, according to [28]. This search consisted of checking whether the titles of the publications contained at least one of the following keywords: *file*, *race*, *time* or *toc**. A total of 470 conferences (216 from Tier-1 and 264 from Tier-2) have been verified and all editions of each conference have been reviewed. For example, regarding the *IEEE Symposium on Security and Privacy*, 25 editions have been reviewed, from 1995 to 2020. In particular, the following conferences have been consulted: *IEEE Symposium on Security and Privacy*, *ACM Conference on Computer and Communications Security*, *USENIX Security Symposium*, *Network and Distributed System Security Symposium*, *Annual International Cryptology Conference*, *International Conference on the Theory and Application of Cryptographic Techniques*, *European Symposium on Research in Computer Security*, *International Symposium on Recent Advances in Intrusion Detection*, *Annual Computer Security Applications Conference*, *Dependable Systems and Networks*, *ACM Internet Measurement Conference*, *ACM Asia Conference on Computer and Communications Security*, *International Symposium on Privacy Enhancing Technologies*, *IEEE European Symposium on Security and Privacy*, *IEEE Computer Security Foundations Symposium*, *International Conference on Theory and Application of Cryptology and Information Security*, *Theory of Cryptography Conference*, and *Workshop on Cryptographic Hardware and Embedded Systems*.

After thoroughly reviewing the proceedings of these conferences, 13 articles were selected according to their titles for further study. In addition, we also carried out snowballing (i.e., reference inspection) on all selected articles. This process allowed us to find 11 additional relevant articles. We provide more details on the number of articles selected during the review process in Section IV.

C. STUDY SELECTION CRITERIA

After finding these initial articles, we used the `StArt` tool to better perform the research and article selection processes. `StArt` [29], [30] is a tool that helps researchers define and execute the systematic review protocol. This tool automatically detects duplicate results, rates them based on predefined keywords, and provides visualizations of the current review status, among other features.

The scoring metric provided by `StArt` was used as the first filter. The `StArt` scoring system allows the user to rate each article based on the appearance of certain keywords in its title, list of keywords, or abstract. The rating system we have used is simple, but allows us to really focus on the relevant articles. For each term in the keyword bag, the score value is obtained as follows:

- Add 5 points if the term appears in the article title.
- Add 3 points if the term appears in the article abstract.
- Add 2 points if the term appears in the article's keywords.

The keyword bag comprises the following main terms, as well as their synonyms and plurals: *TOCTOU*, *attack*, *concurrency*, *defense*, *exploit*, *filesystem*, *interference*, *mitigation*, *race condition* and *data race*. We also consider variations of these terms. The term TOCTOU, for example, has different forms throughout the literature such as *TOCTTOU*, *time-of-check-time-of-use*, or *time of check to time of use*. We set a minimum score of 15 to select an article. Note that articles relevant to our research should easily exceed that minimum value, given the scoring scheme described above.

All articles above the threshold are considered for further inspection and selected or excluded based on the criteria defined in Table 2. These criteria help us select and focus on the most relevant articles in relation to the proposed RQs. After applying the criteria filter, the selected articles are studied in depth to answer each RQ indicated in Section III-A.

D. ARTICLES COLLECTED AND REVIEWED

After running the search protocol, we collected 563 articles. 66 of them have been discarded for being duplicates. After applying the scoring threshold, only 126 remained, which were further analyzed to apply the selection and

TABLE 2. Inclusion (IC) and exclusion (EC) criteria.

Type	Criterion
IC1	The article focuses on concurrency attacks.
IC2	The main contribution of the article is a defense against TOCTOU or an attack to exploit it.
EC1	The article is a short introductory paper, early access, or conference abstract.
EC2	The article is not written in English.
EC3	The article is duplicated.
EC4	The article focuses on other types of TOCTOU rather than file-based TOCTOU.
EC5	The article is not available for downloading or reading.

exclusion criteria. In addition, 13 articles were collected after manually reviewing the Tier-1 and Tier-2 computer security conferences, according to [28]. 6 of them were also discarded because they were duplicates with regard to the previously considered corpus. These manually-included articles have not undergone the scoring system, but were reviewed immediately. Again, 6 of them were discarded because they did not focus on file-based race conditions. This process resulted in a total of 41 articles that we considered for our quantitative synthesis analysis.

This information is combined and summarized in Figure 2. The PRISMA diagram summarizes the execution phases of our review protocol (*identification*, *screening*, *eligibility*, and *inclusion*) and the articles obtained in each phase.

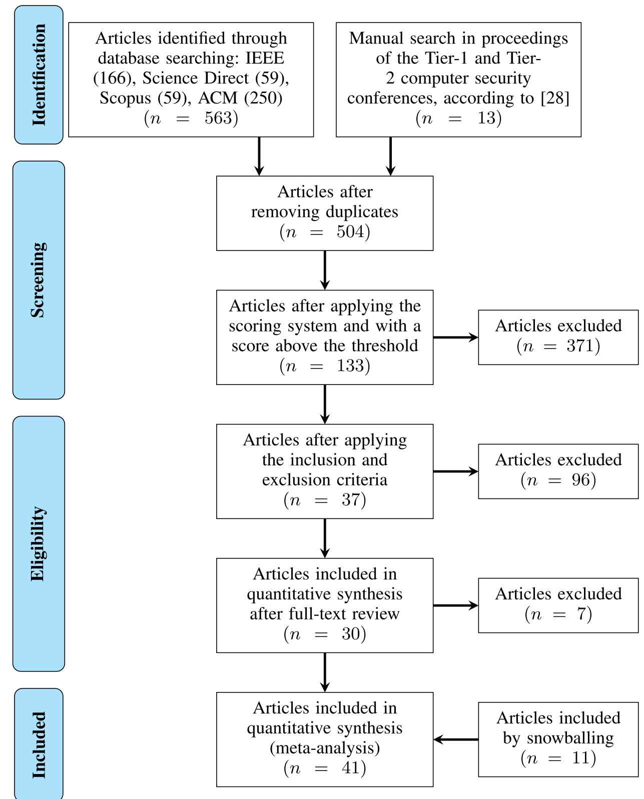

FIGURE 2. PRISMA diagram of our review protocol.

IV. ANALYSIS OF RESULTS

This section presents the results of the systematic review of the literature. We first propose a taxonomy for current TOCTOU defense and attack mechanisms that collects the main insights drawn after the systematic review of the literature and responds to the research questions established in Section III-A. We then explain the different categories into which TOCTOU defenses can be classified, and then we explain the articles in each category in more detail. Finally, we follow the same narrative to explain the studies found on attack methods against TOCTOU vulnerabilities.

A. TOWARDS A TAXONOMY FOR TOCTOU DEFENSE AND ATTACK MECHANISMS

Figure 3 illustrates the classification of TOCTOU defense and attack mechanisms resulting from responding to the research questions established in Section III-A.

A TOCTOU defense or attack mechanism can be categorized considering two aspects: *memory region*, which indicates at which level the TOCTOU defense or exploitation occurs; and *time of detection/exploitation*, which means the time when the TOCTOU vulnerability is detected or exploited. Memory regions can be divided into *user-space level*, which includes solutions that run in the same memory area as the vulnerable application (and some drivers), and *kernel-space level*, which comprises solutions that run in the same memory area where the operating system kernel runs (as well as kernel extensions and most device drivers). Detection/exploitation times can be divided into *static*, which comprises defense solutions in which the vulnerability is detected without the vulnerable application running or after it has been run (in other words, the detection of the vulnerability occurs before or after the execution); and *dynamic*, which includes proposals capable of detecting or exploiting the vulnerability when the vulnerable program is running.

Regarding memory regions, as shown in Figure 4a, 60% of the defense solutions are located in kernel-space, while 40% are located in user-space. As for offensive techniques, all of these are attacks from the user-space level. Recall that this vulnerability allows the attacker to gain system privileges. If the attacker is able to execute the attack from the kernel-space level, there is no real gain in exploiting the vulnerability. We give a more detailed discussion on this matter below in Section V-A.

Regarding time of detection/exploitation, static proposals are exclusively defense approaches, and can be further divided into *source code detection* approaches, which analyze the source code of the vulnerable program [6], [31], [32], and *post-mortem detection* approaches, which detect the TOCTOU vulnerability after the exploitation attempt has already occurred [33]–[40]. Unlike post-mortem detection approaches, source code detection approaches find the

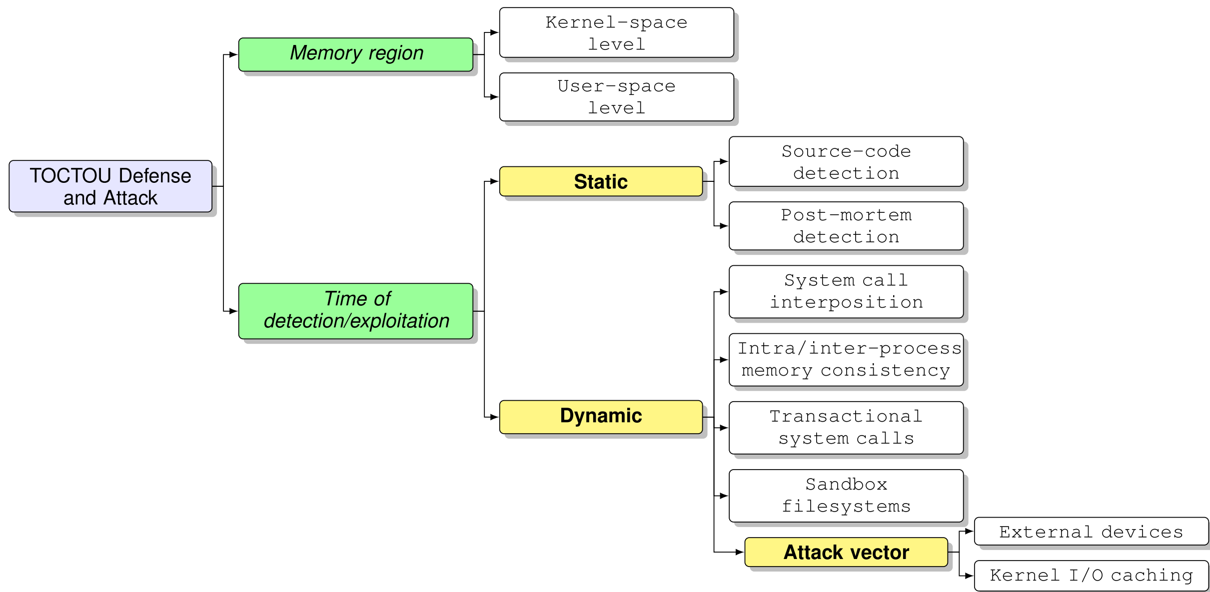

FIGURE 3. Proposed taxonomy for TOCTOU defense and attack mechanisms.

TOCTOU vulnerability before it is exploited, which is often preferable in certain systems such as critical infrastructures or systems with highly sensitive information.

Dynamic proposals are more diverse, based on a multitude of runtime analysis techniques. Some defense approaches use *system call interposition*, monitoring the behavior of the programs by intercepting their system calls. This monitoring can occur at either the user-space level [41]–[46] or the kernel-space level [13], [16], [47]–[59]. A few defense approaches propose *intra-process* or *inter-process techniques for memory consistency* (both at kernel-space level [60], [61]) to guarantee the consistency of variables shared across threads. Other kernel-level defense approaches propose *transactional system calls* [62] as an alternative to traditional filesystems to prevent race conditions from occurring in system resources, while others propose *sandbox filesystems* [63], [64] to protect against unauthorized file modifications caused by file-based race condition vulnerabilities.

In particular, as shown in Figure 4b, 71.4% of the defense solutions are dynamic, while only 28.6% are static. Note that we use the same terms as in program binary analysis (static and dynamic), but we refer to the time of detection rather than how the program is analyzed (not running or while running).

The attack techniques are all dynamic since the vulnerable program must be running to exploit it. Attack mechanisms can be further classified according to the *attack vector*, which defines the path or means that an attacker takes to exploit a vulnerability. Regarding TOCTOU vulnerabilities, we have found two different attack vectors. The first is the *external devices* vector, which consists of abusing the trust that the system places in external devices (i.e., USB sticks or SD cards) during the installation process of a given application. During the installation process, the system can use external devices to store sensitive data that can be altered or manipulated by an attacker, since the external device is under their control. This attack vector is used in [65], [66]. The second is *kernel Input/Output (I/O) caching*, which involves attacks that abuse the kernel's I/O caching mechanism to deliberately increase the window of the vulnerability. If the attacker can tamper with the kernel cache, they will force I/O operations so that the kernel resolves the specified pathnames. These I/O operations take time to complete, which broadens the vulnerability window and facilitates exploitation. This attack vector is used in [67], [68].

B. ON TOCTOU DEFENSES

Figure 5 shows a timeline of the articles studied in this work focused on defense solutions against TOCTOU vulnerabilities. Although the first references to TOCTOU vulnerabilities are approximately 50 years old [14], [15], [69], the first defense solution was not proposed until 1994 [33]. Defense solutions then extend over the years until 2019, the date of the last solution we found.

Figures 4a and 4b show a graphical summary of defense solutions according to *memory region* and *time of detection*, respectively. Regarding detection location, there is no clear or predominant choice among the proposed solutions analyzed in this systematic literature review, although kernel-level solutions represent slightly more than half. As for the moment when TOCTOU is detected, almost three-quarters of the defense solutions are dynamic (specifically, 71.4%).

Finally, it is worth mentioning the trend of detection techniques chosen by the proposed defenses and their level of execution. The timeline in Figure 5 clearly indicates that the first solutions were based on static user-space detection, beginning in 1994. In the early 2000s, the first solutions

TABLE 3. Overview of TOCTOU defenses, sorted by publication year.

Publication	Detection location	Detection time	Operating System	Reproducible	Items used to identify file objects
[33]	User-space	Static	Sun Solaris	✗	File path, permission mode, owner ID, GID
[31]	User-space	Static	Unix-like	✗	(n/a)
[6]	User-space	Static	SunOS and Solaris	✗	(n/a)
[47]	Kernel-space	Dynamic	Linux	✗†	Filename
[48]	Kernel-space	Dynamic	Red Hat Linux 6.2	✗	(n/a)
[34]	User-space	Static	Unix-like	✗	(n/a)
[49]	Kernel-space	Dynamic	OpenBSD	✗	File path, PID, current time, file operation, inode
[32]	User-space	Static	Linux 2.4.18, FreeBSD 4.7, Solaris 8, SunOS 4.1.4	✓	Inode, device ID, generation number (if available)
[50]	Kernel-space	Dynamic	Linux 2.4.20	✗	Filename, inode
[51]	Kernel-space	Dynamic	Red Hat Linux 7.3 (kernel 2.4.18)	✗	Filename, inode
[16]	Kernel-space	Dynamic	Red Hat Linux 9 (kernel 2.4.20)	✗	File path, arguments, PID, filename, UID, GID, EUID, EGID
[52]	Kernel-space	Dynamic	Red Hat Linux 7.3	✗	File path, PID, inode, # of processes accessing the file
[41]	User-space	Dynamic	Linux	✓	Filename
[60]	Kernel-space	Dynamic	Linux 2.4.28	✗	File path, # of processes accessing the file, UID
[35]	User-space	Static	Red Hat Linux 7.3	✗	File path, UID, inode
[42, 43]	User-space	Dynamic	Solaris 8, AIX 5.3 and Linux 2.4.26, 2.6.20 and 2.6.22	✓	Filename, inode
[53]	Kernel-space	Dynamic	POSIX	✓	Device ID, inode
[62]	Kernel-space	Dynamic	Linux 2.6.22	✓∗	Inode
[44]	User-space	Dynamic	POSIX	✓⋆	File path, UID
[13]	Kernel-space	Dynamic	Linux 2.4.28	✗	File path, logical disk block
[54]	Kernel-space	Dynamic	Linux	✗	(n/a)
[36, 37]	Kernel-space	Static	Linux 2.6.35	✓†	Inode, file handlers, UID, PID, PPID
[55]	Kernel-space	Dynamic	Linux 2.6.35	✗	Inode
[45]	User-space	Dynamic	Unix-like	✗†	Device ID, inode, parent directory
[56]	Kernel-space	Dynamic	Linux 3.2.0	✗	Inode
[38]	User-space	Static	Linux 2.6.15	✗	File descriptors, inode, PID
[63]	Kernel-space	Dynamic	Linux 3.2.0-36 and 3.8.10	✓	(n/a)
[57]	Kernel-space	Dynamic	Linux 2.6.35 and 3.2.0	✓‡	Device ID, inode
[58]	Kernel-space	Dynamic	Linux 3.2	✗	Inode
[61]	Kernel-space	Dynamic	Linux	✗	(n/a)
[59]	Kernel-space	Dynamic	Linux	✗	Device ID, inode,
[46]	User-space	Dynamic	POSIX.1-2008 compliant	✗†	Inode
[39]	User-space	Static	Any on Simics	✗	PID, File descriptors, inode
[64]	Kernel-space	Dynamic	Linux 4.10	✓	File path, inode
[40]	User-space	Static	(n/a)	✗	(n/a)

(n/a): Not available; †: No longer available; ‡: Found in the related article material, such as the conference presentation; ∗: Found by searching the Internet; ⋆: Lack of details to fully replicate it.

based on dynamic kernel-space detection emerged. Dynamic detection at the user-space level began in 2006. Additionally, 19 dynamic kernel-space solutions were published between 2001 and 2014, thus averaging more than one publication per year. In contrast, during our research we found only one defense mechanism that relies on a static kernel-based solution.

Table 3 summarizes our findings on defensive techniques, answering research questions RQ2 through RQ5. A detailed discussion answering these questions is provided in Section V. For each study, we indicate in the table the detection location and the detection type. We also indicate the operating system on which it is evaluated (the specific operating system and version if indicated in the publication, or otherwise the generic operating system) and if the proposed solution is reproducible (that is, if a prototype tool or source code is provided for download). In this regard, we consider reproducibility to be an important issue in terms of

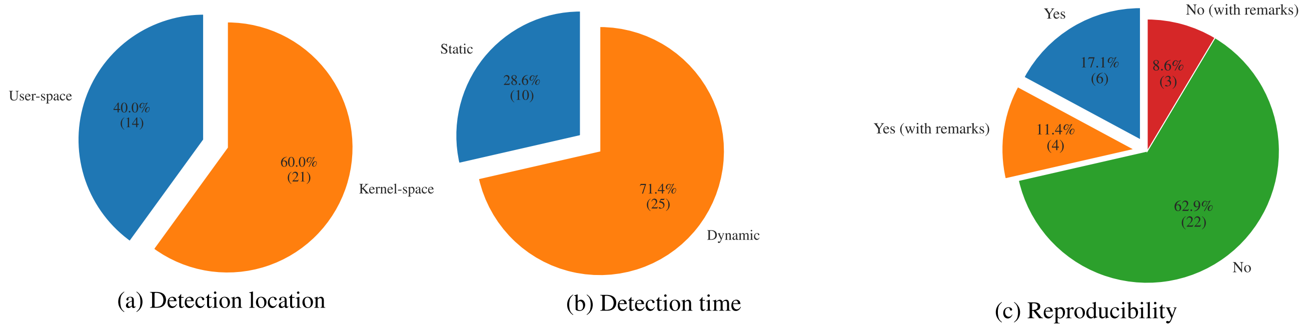

FIGURE 4. Graphical summary of defense solutions (detection location, detection time, and reproducible).

scientific rigor and its contribution to open science. Figure 4c summarizes the reproducibility of the techniques proposed by the works analyzed in this systematic literature review. In addition, we indicate the items each technique uses to identify (uniquely) a filesystem object and detect external manipulation. Figure 7 shows in a bar graph how many defense techniques use each item identified in the literature review. We describe each of these works in more detail below to answer RQ1.

DESCRIPTION OF TOCTOU DEFENSE SOLUTIONS

In this section we describe each of the works according to the research questions established in Section III-A. The studies, which are presented in chronological order, have been grouped according to the detection location (user-space level versus kernel-space level) and the detection time (static versus dynamic).

1) BASED ON STATIC USER-SPACE DETECTION

The first work we found is [33]. The authors proposed a defense solution that detects TOCTOU exploitation attempts after the vulnerable program has been executed. The detection process is mainly based on the analysis of execution traces. We refer to this type of static detection as *post-mortem detection*, since the detection is made *after* the exploitation attempt has been carried out and the vulnerable program has finished its execution. The authors' solution monitors the execution of privileged programs, auditing certain sequences of unwanted actions and then checking them against expressions described by the logic of predicates and regular expressions. The authors also presented a software prototype that runs on the Sun Solaris operating system and is capable of detecting TOCTOU vulnerabilities in three widely used programs (specifically, `fingerd`, `rdist`, and `sendmail`). However, no reference to the source code or to the tool itself is provided to facilitate reproduction of the experiments.

File-based race conditions on Unix-like operating systems were first discussed in detail in [31], concluding that kernel modifications are required to eliminate file-based race conditions. In addition, a lexical source code scanner is proposed to detect the vulnerabilities related to file access. Although the presented prototype successfully discovered new instances of TOCTOU, the tool or its source code is not available.

In a later work, Bishop and Dilger [6] demonstrated that privilege escalation attacks that exploit TOCTOU vulnerabilities only occur when filesystem objects are referenced by their names and not by file descriptors. Again, a software prototype is developed to (lexically) parse C source code files and detect file-based race conditions. Detection is based on pattern matching techniques and dependency and data-flow graph analysis. Unfortunately, the prototype is not accessible.

Goyal *et al.* [34] proposed an algorithm that is evaluated on a Unix-like system to detect TOCTOU attacks based on the analysis of execution traces (that is, it performs a post-mortem detection). A set of predefined rules is verified against execution traces to detect successful exploitation of TOCTOU vulnerabilities. As the authors stated, this solution is incomplete as the attack patterns must be known beforehand. No reference is provided to the availability of the prototype that implements the algorithm.

A probabilistic solution was proposed in [32], in which the source code of the vulnerable program is modified to reduce the probability of success of an attack. This solution replicates an arbitrary number of times the execution of the original sequence of potential vulnerable actions, verifying afterwards if the accessed file is changed. Since the authors provided examples on how to modify the source code, we consider this to be a reproducible work.

The solution proposed by Bhatkar *et al.* [35] was also based on post-mortem detection as it parses execution traces to build a control and data-flow graph which is then verified against a set of learning temporal properties representing TOCTOU vulnerabilities. The solution is implemented in a software tool for Red Hat Linux 7.3 that is not available.

Yu *et al.* [38] proposed a virtualization-based solution dubbed `SimRacer`, which tests the occurrence of certain types of race conditions by replaying event traces of the executions of the program. The authors test it against a set of vulnerable programs, successfully detecting all TOCTOU vulnerabilities. By construction, `SimRacer` is compatible with any operating system that runs on top of the full-system `Simics` simulator. Unfortunately, neither the tool nor its source code is available.

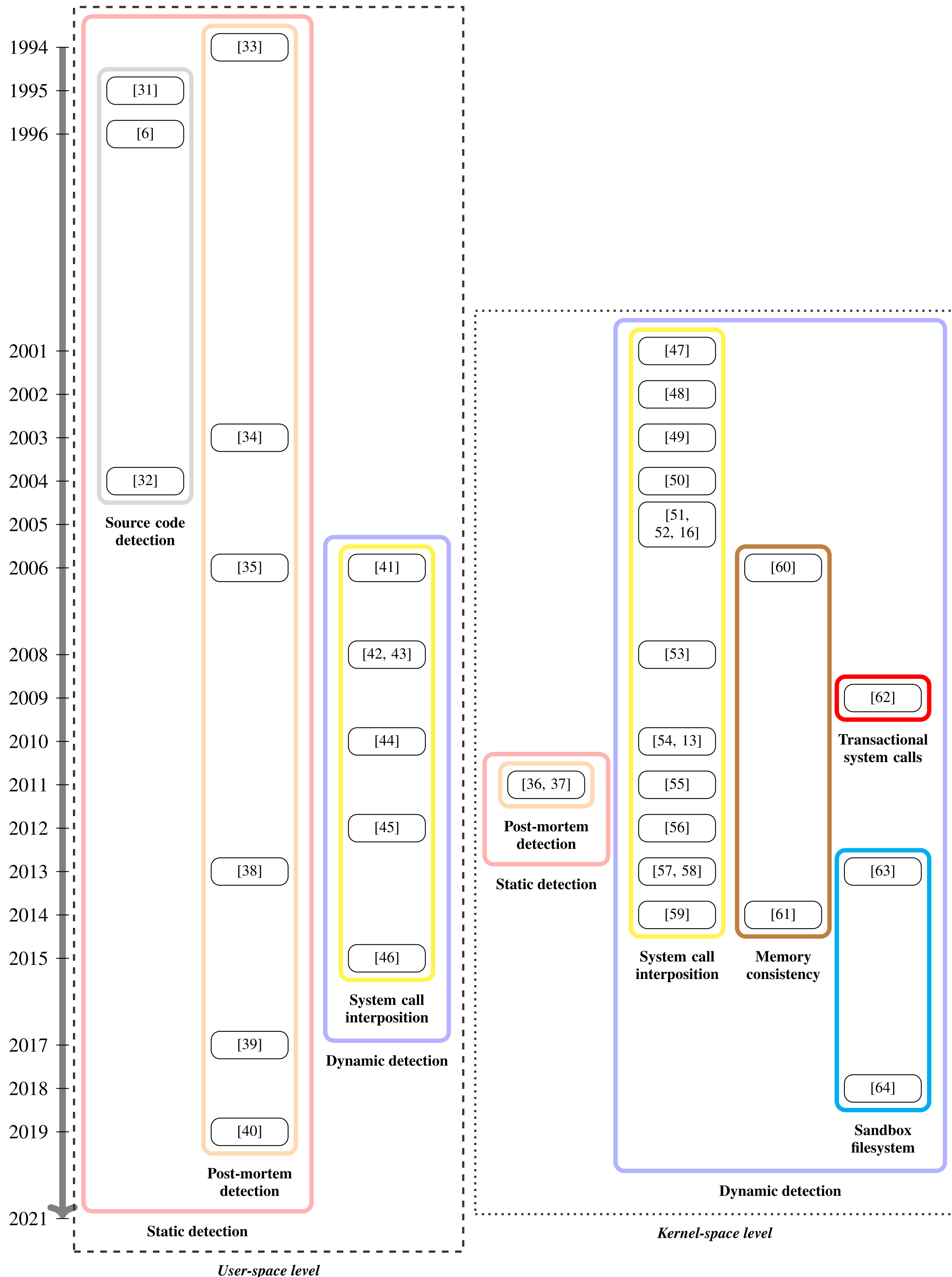

FIGURE 5. Evolution of TOCTOU defenses over the years.

Yu *et al.* [39] introduced SIMEXPLORER, an improved version of SimRacer. This solution extends the detection algorithms of SimRacer to consider hardware interruptions and signal handlers. Tested with 24 programs, it detected 36 out of 41 previously known vulnerabilities. Like SimRacer, SIMEXPLORER can run

on any OS that runs on top of Simics and is also not available.

The latest static user-space solution also relies on post-mortem detection. Capobianco *et al.* [40] provided a solution for detecting TOCTOU vulnerabilities by calculating the attack graph of the vulnerable program and analyzing it to detect sequences of events that may end up exploiting the vulnerability. These attack graphs allow the user to find the attack surface used by the adversaries and how they effectively elevate privileges. While the authors conduct detection statically, they explore how it can be applied at runtime and discuss the research challenges, as their primary goal is to improve the capabilities of intrusion detection systems. Unfortunately, neither the prototype nor the source code is available.

2) BASED ON DYNAMIC USER-SPACE DETECTION

Aggarwal and Jalote [41] proposed the first solution based on dynamic user-space detection. In particular, the solution relies on a software agent integrated in the vulnerable program to control its execution while detecting common vulnerabilities. System calls are monitored by the agent and sent to another process in charge of real-time analysis of the behavior of the vulnerable program. A software prototype for Linux is provided and evaluated, which succeeds in stopping file-based TOCTOU exploits.

A new standard function was provided in [42], [43] to avoid the TOCTOU vulnerability window between the system call sequence `access` and `open`. This new feature is an enhancement of a previous version introduced in [32] to defend against complex attacks such as filesystem mazes [67]. More details on this type of attacks are given in Section IV-C. Although the solution provides good results, it still has some drawbacks, such as the difficulty of deployment in production, defending against circular symbolic links, or multi-threaded applications, among others. Source code is provided by the authors. Since [43] is the full report of [42], we consider them as a single solution.

Chari *et al.* [44] proposed a set of secure calls for POSIX-compliant operating systems to prevent privilege escalation attacks based on TOCTOU. These secure calls overlap actual system calls, monitoring invocations of certain file-based system calls for unwanted inputs. However, the solution does not work with statically-linked programs since it is provided as a software library. Unfortunately, the work in [44] only shows a subset of the proposed secure calls, leaving the reader without full knowledge to fully reproduce their work.

Likewise, Payer and Gross [45] also proposed a Unix-based software library dubbed `DynaRace`. This binary-instrumentation solution is based on the state-machine formalism: it maintains a state machine for each file used by the vulnerable program, updated upon a sequence of certain file-based system calls, to detect unwanted behavior. When detected, it issues a warning and aborts the vulnerable program. The tool was available on the author's website.

A software library solution that detects file-based race conditions is also provided in [46]. This solution, though, puts all the responsibility on the team of software developers, as they must use the secure system calls provided by the software library rather than those of the operating system. In addition, it can also generate false positives, and leaves the vulnerable program in an unknown state after detecting an exploitation attempt. The source code was available on the website of the author's research group. Unfortunately, it is no longer online.

3) BASED ON STATIC KERNEL-SPACE DETECTION

In [36], [37], the authors introduced a system dubbed `RacePro` capable of detecting different types of race conditions, including TOCTOU. The system monitors program executions and audits system calls that access shared kernel objects. These audit records are then verified against benign and harmful race models, which are known in advance. `RacePro` was tested on Linux Kernel version 2.6.35 and found 4 unknown bugs in common Linux tools such as `make` and `locate`. The source code for `RacePro` is freely available at [70], although it is not explicitly mentioned in the paper. Since [36] is the preliminary work of [37], we consider them a single solution.

4) BASED ON DYNAMIC KERNEL-SPACE DETECTION

`RaceGuard` is a Linux kernel modification proposed in [47] to detect race conditions when creating temporary files. Internally, it keeps track of filenames created through certain system calls, which are then checked for race conditions. However, this solution is incomplete as it only monitors the creation of new files, regardless of existing ones. Although the source code was originally available as a kernel patch for Immunix, this commercial operating system was discontinued in 2003.

Similarly, Ko and Redmond proposed in [48] a kernel module for Red Hat Linux 6.2 that monitors system calls made during the execution of a privileged program, deliberately performing them ahead of the execution of system calls in non-privileged programs. However, the availability of the prototype is not mentioned.

In [49], Tsyrklevich and Yee also proposed a kernel module for OpenBSD to detect sequences of system calls that can lead to race conditions. The module removes the sharing property of some file objects, making their accesses mutually exclusive. The solution is successfully evaluated in four attack scenarios, detecting and stopping all exploitation attempts. However, as the authors admit, this solution is not free of race conditions, as the interception of system calls implicitly generates another race condition vulnerability window. Although the authors state that their proposal is portable to other operating systems (not only Unix-like), the source code is not provided.

`Race-attack Prevention System` is a system proposed by Park *et al.* [50] that also intercepts system calls and checks the consistency between them. Built on top of

RaceGuard [47], it verifies if shared file objects are manipulated from a transactional point of view to avoid race conditions. This solution is implemented for Linux kernel 2.4.20, but unfortunately there is no mention of where or how it can be obtained.

Lhee and Chapin [51] proposed another software library that intercepts system calls to detect TOCTOU vulnerabilities, detecting inconsistencies in file objects by means of their binding information (specifically, the inode and the filename). This solution is implemented as a kernel module for Red Hat Linux 7.3 running on top of Linux kernel version 2.4.18. Although the authors implemented, tested, and evaluated a simplified prototype of their defense proposal, it is not available.

Uppuluri *et al.* [52] defined a set of security policies, specified using a behavior modeling specification language, that can be compiled and integrated into different detection engines. A prototype engine is implemented as a kernel module for the Red Hat 7.3 operating system. As before, this solution is not free of race conditions, as it relies on the interception of system calls. In addition, the prototype is not available either.

Wei and Pu [16] proposed a model for TOCTOU vulnerabilities in Unix filesystems, called *CUU model*. This model consists of pairs of system calls that can lead to a TOCTOU vulnerability. In addition, they propose different tools that are based on this model to monitor and detect TOCTOU vulnerabilities in Linux systems at the kernel level. These tools were successfully tested in version 2.4.20 of the Red Hat Linux 9 kernel, in approximately 130 utility programs. This solution, though, is only suited for single core processors. However, none of the tools are publicly available.

A defense solution called *Event Driven Guarding of Invariants* (EDGI), based on the CUU model, is presented in [60]. Vulnerable pairs of system calls are translated into invariants, which are used as sophisticated locks with a time-out mechanism. This solution is implemented in version 2.4.28 of the Linux kernel, but neither the tool nor its source code is available.

Kupsch and Miller proposed in [53] a set of functions to create and manipulate files in a secure way, replacing standard C functions such as `creat`, `open`, or `fopen`, to name a few, to eliminate TOCTOU race conditions. A working implementation of these functions is publicly available at [71].

Porter *et al.* [62] introduced a variant of Linux 2.6.22, dubbed TxOS, which incorporates system call transactions, allowing software developers to perform operations on system resources guaranteeing ACID properties (atomicity, consistency, isolation and durability) of the underlying system calls. Furthermore, the vulnerable program is blocked when an exploitation attempt is detected. TxOS is open source and publicly available at [72].

Wei and Pu extend their CUU model in [13] by proposing the *Stateful TOCTOU Enumeration Model*. This model lists all the pairs of system calls that can lead to a TOCTOU vulnerability on a Linux and POSIX system (224 and 285 pairs, respectively). To the best of our knowledge, this study is the most comprehensive characterization of the system calls leading to the TOCTOU vulnerability to date. EDGI is also extended to incorporate this model in version 2.4.28 of the Linux kernel.

Rouzaud-Cornabas *et al.* [54] formalized the concept of race conditions and provided a framework for defining security properties to prevent them. These properties are specified and integrated into a Linux kernel module, implemented on top of SELinux. It is based on information flow graphs to represent the temporal relationships between processes and system resources. This solution was tested in production systems for six months with successful results. However, it is not publicly available.

Vijayakumar *et al.* [55] introduced a software prototype that stops attacks targeting vulnerabilities based on name resolution (such as TOCTOU) by combining four incomplete defense techniques (specifically, system resource restrictions, capabilities, namespace management, and program resource restrictions) to build a complete solution. It is implemented as a SELinux module in version 2.6.35 of the Linux kernel. Neither the source code nor the tool is available.

Vijayakumar *et al.* [56] presented a software engine, dubbed STING, that prevents name resolution attacks. In particular, it analyzes system calls at runtime and creates test cases that are then used to replicate an adversary's behavior and thereby detect exploitation attempts. STING is implemented as a Linux security module in Linux kernel 3.20, and tested with different operating systems, discovering 26 race condition vulnerabilities (21 of them were previously unknown). However, as the authors warn, it can produce false positives under certain running conditions. Unfortunately, the tool is not available for public use.

Different policies are given in [58] to determine which files can be retrieved using the name resolution process in system calls. These policies control file access at run-time in the context of a system call. A prototype that enforces these policies is deployed on top of the SELinux access control module and tested on the Ubuntu 12.04 operating system. The experimental results show that all exploitation attempts were successfully stopped. However, the policy enforcement prototype is not available.

Vijayakumar *et al.* [57] also proposed Process Firewall, a Linux security module that analyzes system calls and restricts access to resources depending on the current state of the process. These constraints are modeled as Linux IPTable rules and are interpreted by a rule processing mechanism designed for system calls. Tested on Ubuntu 10.04, nine resource attacks (including TOCTOU-based attacks) were detected and blocked successfully. Although it is not mentioned in the article, Process Firewall is publicly available at [73].

Kim and Zeldovich [63] introduced a new Linux-based sandboxing mechanism called Mbox that interposes on system calls. Mbox creates a layered sandbox filesystem on top

of the host filesystem where all the operations take place, preventing the latter from being manipulated. The user can then browse the sandbox filesystem, committing the modifications to the host filesystem or discarding them accordingly. The interposition of system calls is carried out using the `seccomp/BPF` facility. `Mbox` is open source and available at [74].

Zhou *et al.* [61] proposed `SHIELD`, a software that uses deterministic multithreading techniques to guarantee that variables shared across threads are consistent. To do this, when it detects a memory modification, it executes a memory propagation mechanism that extends the modification to the virtual memory of other threads within the program. Like other solutions, `SHIELD` is not available.

Vijayakumar *et al.* [59] presented `Jigsaw`, a defense mechanism against resource access attacks based on the interception of system calls. It works in two phases: first, it parses the program using graph-based formalisms to find system calls on shared resources; second, it uses `Process Firewall` [57] to enforce the invariants that avoid the vulnerability. This solution is implemented as a kernel module and tested on Ubuntu 10.04, identifying two unknown vulnerabilities. Unlike `Process Firewall`, neither the program application nor its source code is publicly available.

A lightweight Unix-based filesystem sandboxing mechanism is proposed in [64]. The mechanism is dubbed `SandFS` and is designed as an extensible kernel filesystem that intercepts all filesystem requests. It works with low-level kernel objects and provides a C-like API for the developers to implement their own security extensions. Acting as an interposing layer between the filesystem and the user-defined security extensions, it does not perform any filesystem operations, but instead compares them to the extensions. Allowed operations are tracked to the filesystem, whereas denied operations are canceled with the corresponding error number. `SandFS` leverages the `eBPF` framework to achieve safety guarantees and is publicly available at [75].

C. ON TOCTOU ATTACKS

Figure 6 shows the timeline of the articles studied in this work that focus on attacks that try to exploit TOCTOU vulnerabilities. At a glance, the literature on attacks is very scarce. We have found only 4 works that propose new ways to abuse TOCTOU vulnerabilities, in addition to the seminal work of [33] that first introduced specific examples on how to exploit TOCTOU vulnerabilities and gain elevated privileges. Regarding these offensive works, the oldest and newest of them date from 2005 and 2017, respectively. The remaining are dated from 2009 and 2012.

As commented above, all of them are attacks from the user-space level and dynamic. In summary, we conclude that there is not much innovation in the ways of abusing TOCTOU vulnerabilities and few authors are interested in it. Regarding the attack vectors, the offensive techniques that we found take advantage of: the *trust in external devices* and the *I/O caching mechanism of the system kernel* (explained in Section IV-A).

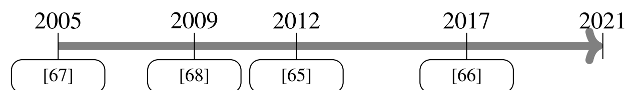

FIGURE 6. Evolution of TOCTOU attacks over the years.

DESCRIPTION OF TOCTOU ATTACKS

In this section, we classify the articles found during our systematic review of the literature that contribute to new ways of abusing TOCTOU according to the attack vector they exploit. We describe them according to the research questions set out in Section III-A. As before, they are presented in chronological order. Let us remark that none of these works provide source code or software tools, but instead they provide a detailed explanation of how the attacks work.

1) ATTACK VECTOR BASED ON EXTERNAL DEVICES

Mulliner and Michéle proposed in [65] another novel attack called *Read It Twice*, focused on consumer electronics and embedded devices. This attack takes advantage of the installation and update processes of these devices, which normally depend on external devices and are carried out in two steps (not atomic): one to verify and the other to install/update. This attack has been successfully tested on Linux-based Samsung TVs. In addition, the authors develop a hardware board to determine whether a device is vulnerable to these attacks.

Similarly, Lee at al. studied the installation process of of Android applications in [66], finding TOCTOU vulnerabilities in all its phases. As a result, the authors present a novel attack called *Ghost Installer Attack* (GIA), as well as defense solutions against it. We have categorized this paper exclusively as TOCTOU attack because the proposed defense solutions are designed for the GIA.

2) ATTACK VECTOR BASED ON THE KERNEL I/O CACHING

Borisov *et al.* presented in [67] a novel technique to exploit race conditions when the defense solution proposed by [32] is working. This attack, carried out by means of three software tools, relies on a deliberate increase in input/output filesystem operations, as they are likely to force the preemption of the running thread due to memory cache buffering issues.

Subsequently, in [68] Cai *et al.* presented a novel attack that defeats the solutions proposed by [32] and by [42]. This attack is based on collision attacks targeting the kernel's filename resolution algorithm. As a result, the filesystem operations of the vulnerable program are slowed down and thus the window of vulnerability increases.

V. SYNTHESIS

In this section we first present a detailed analysis of the results of our systematic literature review to answer research questions RQ2 through RQ5. We then highlight the future research trends and directions that we envision and finally discuss the limitations of our work.

A. DISCUSSION OF RESULTS

Below, we address and answer each of the research questions established in Section III-A.

1) RQ2: MEMORY REGIONS OF DEFENSIVE AND OFFENSIVE TECHNIQUES

After performing the systematic literature review, we have found that defense mechanisms reside either in user-space or in kernel-space. As shown in Figure 4a, 60% of the defense techniques reside in kernel-space.

User-space techniques can be applied to a wide variety of programming languages, compilers, and interpreters and are easier to debug. However, they cannot access kernel-level information or mechanisms such as the system's cache, the scheduler, hardware, input/output (I/O), or the inode generation algorithm, which is a major limitation in terms of the scope of the solution.

Alternatively, kernel-space solutions have access to all system components and information. Solutions that work in kernel-space benefit from lower latency when performing certain operations, such as system calls. Despite all this, the kernel-space approaches may require modifying the kernel or adding modules, which is not always a viable option. If the kernel can be modified, serious backward compatibility issues can arise, even rendering older software or kernel versions unusable. Furthermore, implementing and debugging solutions in kernel-space is not only more difficult than its user-space counterpart, but is also limited to the kernel language. Moreover, kernel-level errors are likely to crash the entire system.

As for offensive techniques, all of them are attacks from the user-space level. This result would be expected, because otherwise, if the attacker can already execute code in the kernel, there is no motivation to exploit a file-based TOCTOU vulnerability.

2) RQ3: TIME OF VULNERABILITY DETECTION OR EXPLOITATION

According to the moment at which the vulnerability detection is carried out, the defense solutions are broadly divided into static and dynamic techniques. Static detection comprises solutions in which the vulnerability is detected without the vulnerable application running or after it has been run (that is, the detection happens before or after the execution), while dynamic detection includes techniques capable of detecting the vulnerability when the vulnerable program is running. As shown in Figure 4b, there is a clear predominance of dynamic techniques, with 71.4%.

Static techniques have proven useful in detecting and correcting the vulnerability before it occurs (for instance, during the development phase of the software system). Some of these techniques require the source code of the program to be executed, which is unlikely to happen in most cases. Their main advantages include the fact that they are simple to implement, they do not require modification of the runtime environment, and that the overall system performance is not affected as a result of code analysis. Unfortunately, these techniques only propose solutions to known attacks (that is, to specific vulnerable instruction sequences) which limits their effectiveness. Furthermore, even when the attack is detected, it is not always easy to figure out how to modify the source code of the vulnerable program to avoid the vulnerability, especially in multi-threaded programs.

Other static detection approaches rely on log analysis or execution traces. Therefore, in this case the vulnerability is detected when it has already occurred. In addition, these detection techniques are often unsound because there is a wide variety of factors that influence the exploitability of the vulnerability, such as environment variables or system load.

On the other hand, dynamic defenses protect systems in real time and can thwart exploitation attempts as they occur. The defense is typically performed by an external agent that is integrated into the runtime environment, and hence these solutions tend to incur performance overheads for the system. Alternatively, other solutions require running the vulnerable application in a controlled environment to perform the analysis. Regardless of the type of approach, dynamic techniques are useful for detecting well-known attacks, although they have the ability to store information about the program execution that could be used to identify new attacks. The main advantage of these techniques is that it is not necessary to have or modify the source code of the program, facilitating their adoption in a more general way.

As for the offensive techniques, they are all dynamic since the vulnerable program must be running to exploit it. These techniques use two different attack vectors: external devices (such as USB sticks or SD cards) and the I/O caching mechanism of the system kernel. External devices are abused during the installation process of the vulnerable application, as the attacker can alter or manipulate the sensitive data that can be stored on these devices. In contrast, abuse of the system kernel I/O caching mechanism is done by third-party programs, which force the kernel to perform more I/O operations and thus increase the vulnerability window, which facilitates the occurrence of race conditions.

3) RQ4: OPERATING SYSTEM USED BY THE TECHNIQUES

All the defense solutions focus on Unix-like operating systems. In addition, 3 out of the 4 attack techniques are also directed at these operating systems (the remaining attack proposal focuses on Android, which uses Linux as its kernel). The prevalence of the Unix-like operating system is due to the fact that other operating systems, such as Windows, manage references to files through internal structures similar to file descriptors (for instance, via *handles* in Windows [76]). This means that these operating systems are free of file-based TOCTOU vulnerabilities, although other types of race conditions are still possible (which are beyond the scope of this paper).

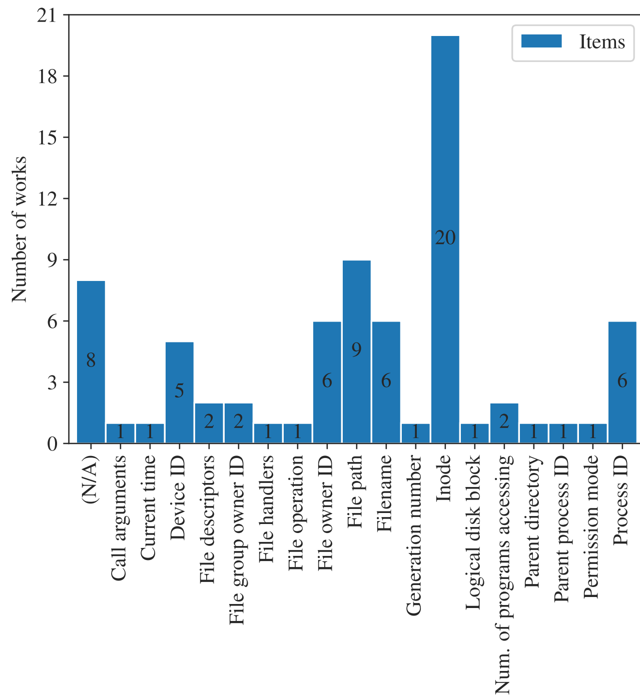

FIGURE 7. Prevalence of filesystem objects' metadata to detect any changes.

RQ5: REPRODUCIBILITY OF SOURCE CODE OR TOOLS

Figure 4c shows the reproducibility of defense techniques. Almost three-quarters of the articles studied are not reproducible, either because no source code or tool is provided or because details are lacking to fully replicate them. In the spirit of open science, experimentation on any proposal should be reproducible to allow others to evaluate, compare, and improve the proposal.

Most defense solutions use some kind of history tracking of filesystem objects' metadata to detect any changes. When they are detected, the solutions apply their logic to decide whether the modification is legitimate or corresponds to an attempted exploitation. Although none of the articles studied specify what underlying filesystem they used to test their proposed defenses, commonly used metadata includes inode, device ID, filename, or file path, among others. Figure 7 shows the prevalence of each metadata used by the defense solutions analyzed in this systematic literature review. However, the accuracy of these metadata for detecting file-based TOCTOU vulnerabilities is highly dependent on the underlying filesystem.

For instance, a common metadata is the inode, used by 20 of the 35 defense solutions. An inode (*index node*) is a data structure that defines a file or a directory on a Unix-style filesystem and is stored in the directory entry. To determine if inodes are unique per file, we have studied the behavior of inodes on the main filesystems in the Unix universe [77]. In particular, we have empirically tested if, when an inode is freed (that is, without hard or soft links that point to it), the filesystem eliminates it and never uses it again or if it is free to reuse it when necessary. We have found that the uniqueness of an inode depends to a large extent on the underlying filesystem and, therefore, inodes cannot be assumed to be an item for single distinction. The results of our tests are shown in Table 4.

TABLE 4. Reutilization of inode according to the filesystem.

Filesystem	Reutilization of inode
BTRFS	No
EXT2	Yes
EXT3	Yes
EXT4	Yes
FAT16	No
FAT32	No
NTFS	No
HFS+	No
JFS	No
NILFS2	Yes
REISERFS	Yes
XFS	Yes
RAMFS	No
TMPFS	No

Turning to offensive techniques, we consider two of them are partially reproducible and the other two are no longer reproducible. Based on their level of detail, we consider that [65] and [66] are partially reproducible because, although they do not provide any source code or tool to perform the attack, both works are detailed enough to be replicated. On the other hand, we consider both [67] and [68] as no longer reproducible, as both articles link to their respective source code repositories which are unfortunately no longer available.

B. FUTURE RESEARCH TRENDS AND DIRECTIONS

Most defense solutions protect against specific cases of TOCTOU vulnerabilities, but incur large impacts on performance or make strong assumptions about the behavior of the underlying filesystem. In summary, no defense solution is universal, as reflected in the fact that, until now, no solution has been officially adopted to prevent TOCTOU vulnerabilities.

In our opinion, it is unlikely that a universal solution will be found, given the non-determinism of the TOCTOU vulnerability and the influence of external factors (such as the environmental variables, among others). In any case, the best solution we envision is a mixture of some of the approaches mentioned above. In particular:

- *A new API (or modification of the current one) to provide a race-free, security-focused API*. A good option to avoid TOCTOU vulnerabilities is to use an API based on file descriptors rather than filenames. However, legacy software would still be vulnerable. In addition, the burden falls on software developers, who must know and use this race-free, security-focused version of the API.
- *Modification of the kernel to always work with file descriptors*. Modifying the kernel to work exclusively

with file descriptors instead of filenames can also be a good solution. However, this solution implies a drastic modification of the kernel and therefore it is likely to cause serious backwards compatibility issues.
- *Transactional filesystems*. Another good option can be to use transactional filesystems. A transactional filesystem allows the files and directories to be created, modified, renamed, and deleted atomically, protecting the consistency of their filesystem structure. A good solution can rely on this type of filesystem to verify that the file objects do not change between pairs of TOCTOU vulnerable system calls.

C. LIMITATIONS

Like any other systematic review of the literature, we have defined a search protocol that is reproducible and its results are free of bias. Note that we have considered articles written in English, without considering articles potentially relevant written in any other language. In addition, as we have used the scoring system of the `StArt` tool, our results are tied to that particular scoring system.

Our results are also limited to the keyword bag that we have defined to perform the search. We can also improve these terms to expand our search results. Finally, we have excluded gray literature (e.g., blogs or repositories) as we are exclusively interested in scientific contributions. However, gray literature is an important source of knowledge about issues related to security. For instance, the Openwall kernel patch [78] is a collection of security hardening patches for various versions of the Linux kernel posted on a website.

VI. CONCLUSION

Although file-based TOCTOU vulnerabilities were first mentioned in the mid-1970s, they began to be studied in more detail twenty years later. Despite this vulnerability being almost 50 years old, it remains unresolved. In this paper, we have presented a systematic review of the literature up to 2021 on defense solutions, as well as related attack techniques, against this type of race condition vulnerability. In particular, we found 41 articles of interest in different scientific databases (in particular, IEEE Xplore, ScienceDirect, Scopus, and ACM).

Our results indicate that a large majority of research efforts have been directed towards defense mechanisms (37 out of 41), whereas a small fraction of works focuses on offensive techniques (the remaining 4). The defense solutions proposed in the literature can be classified into *source code detection*, *post-mortem detection*, *system call interposition*, *memory consistency*, *transactional system calls*, and *sandbox filesystems*. As for the offensive solutions, half of them deliberately force more time-consuming input/output operations, while the rest focus on exploiting the installation of programs from external storage devices.

We found defense solutions that reside in the kernel (slightly above half, 21 out of 35) and at the user-space level (the remaining 14). However, all the attack techniques are carried out from the user-space level. Most of the defense solutions proposed are dynamic (25 out of 35), while the others are static solutions. Static solutions detect TOCTOU vulnerabilities in source code or at the binary level, while dynamic solutions execute, monitor, and verify execution at runtime or after program execution, analyzing logs and audit trials. All the defense techniques are developed for Unix-like operating systems. Similarly, 3 out of the 4 attack solutions focus on Unix-like systems, while the remaining attack focuses on Android.

Finally, we discovered that almost all the software tools developed to defend or exploit TOCTOU vulnerabilities are not available. Few give access to the source code or the tool itself, or give enough details to code it ourselves, making it difficult to replicate experiments later. For the sake of open science and reproducibility, any contribution that introduces new software tools or new methods should be accessible to the public and other scientific researchers.

REFERENCES

[1] C. E. McDowell and D. P. Helmbold, "Debugging concurrent programs," *ACM Comput. Surveys*, vol. 21, no. 4, pp. 593–622, Dec. 1989.
[2] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, May 2006.
[3] T. Warszawski and P. Bailis, "ACIDRain: Concurrency-related attacks on database-backed web applications," in *Proc. ACM Int. Conf. Manage. Data*, New York, NY, USA, May 2017, pp. 5–20.
[4] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan, "Concurrency attacks," in *Proc. 4th USENIX Conf. Hot Topics Parallelism (HotPar)*, New York, NY, USA, 2012, p. 15.
[5] BeyondTrust. (Mar. 2021). *Microsoft Vulnerabilities Report 2021*. Accessed: May 30, 2021. [Online]. Available: https://www.beyondtrust.com/assets/documents/BeyondTrust-Microsoft-Vulnerabilities-Report-2021.pdf
[6] M. Bishop and M. Dilger, "Checking for race conditions in file accesses," *Comput. Syst.*, vol. 9, no. 2, pp. 131–152, 1996.
[7] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro, "How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the Linux kernel," in *Proc. 26th USENIX Secur. Symp. (USENIX Security)*, Vancouver, BC, Canada, Aug. 2017, pp. 1–16.
[8] P. Wang, K. Lu, G. Li, and X. Zhou, "DFTracker: Detecting double-fetch bugs by multi-taint parallel tracking," *Frontiers Comput. Sci.*, vol. 13, no. 2, pp. 247–263, Apr. 2019.
[9] M. Ambrosin, M. Conti, R. Lazzeretti, M. M. Rabbani, and S. Ranise, "Collective remote attestation at the Internet of Things scale: State-of-the-art and future challenges," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 4, pp. 2447–2461, Jul. 2020.
[10] O. Arias, D. Sullivan, H. Shan, and Y. Jin, "LAHEL: Lightweight attestation hardening embedded devices using macrocells," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, Dec. 2020, pp. 305–315.
[11] S. Bratus, N. D'Cunha, E. Sparks, and S. W. Smith, "TOCTOU, traps, and trusted computing," in *Proc. Int. Conf. Trusted Comput.*, vol. 4968. Berlin, Germany: Springer-Verlag, Mar. 2008, pp. 14–32.
[12] X. Chang, B. Xing, and Y. Qin, "Formal analysis of a response mechanism for TCG TOCTOU attacks," in *Proc. 4th Int. Conf. Multimedia Inf. Netw. Secur.*, Nov. 2012, pp. 19–22.
[13] J. Wei and C. Pu, "Modeling and preventing TOCTTOU vulnerabilities in unix-style file systems," *Comput. Secur.*, vol. 29, no. 8, pp. 815–830, Nov. 2010.
[14] W. S. McPhee, "Operating system integrity in OS/VS2," *IBM Syst. J.*, vol. 13, no. 3, pp. 230–252, 1974.
[15] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and A. andD Webb, "Security analysis and enhancements of computer operating systems," Inst. Comput. Sci. Technol., Nat. Bureau Standards, Gaithersburg, MD, USA, Tech. Rep. NBSIR 76-1041, Apr. 1976.

[16] J. Wei and C. Pu, "TOCTTOU vulnerabilities in unix-style file systems: An anatomical study," in *Proc. 4th USENIX Conf. File Storage Technol. (FAST)*, San Francisco, CA, USA, Dec. 2005, p. 12.

[17] (Jan. 2022). National Vulnerability Database. *NVD—TOCTOU Search Results*. Accessed: Jan. 13, 2022. [Online]. Available: https://nvd.nist.gov/vuln/search/results?form_type=Advanced&results_typ%e=overview&search_type=all&cwe_id=CWE-59&isCpeNameSearch=false

[18] (Jan. 2022). MITRE. *MITRE CVE—TOCTOU Search Results*. Accessed: Jan. 13, 2022. [Online]. Available: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=file-based+TOCTOU]

[19] (Jun. 2021). Flysystem. *Time-of-Check Time-of-Use (TOCTOU) Race Condition in League/Flysystem*. Accessed: Jan. 13, 2022. [Online]. Available: https://github.com/thephpleague/flysystem/security/advisories/GHSA-9f46% -5r25-5wfm

[20] (Oct. 2020). VMWare. *VMSA-2020-0023.3*. Accessed: Jan. 13, 2022. [Online]. Available: https://www.vmware.com/security/advisories/VMSA-2020-0023.html

[21] (May 2021). Red Hat. *CVE-2021-30465—Red Hat Customer Portal*. Accessed: Jan. 13, 2022. [Online]. Available: https://access.redhat.com/security/cve/cve-2021-30465

[22] (Mar. 2020). Adobe. *Adobe Security Bulletin*. Accessed: Jan. 13, 2022. [Online]. Available: https://helpx.adobe.com/security/products/creative-cloud/apsb20-11.html%

[23] P. Wang, K. Lu, G. Li, and X. Zhou, "A survey of the double-fetch vulnerabilities," *Concurrency Computation: Pract. Exper.*, vol. 30, no. 6, p. e4345, Mar. 2018.

[24] R. V. Steiner and E. Lupu, "Attestation in wireless sensor networks: A survey," *ACM Comput. Surveys*, vol. 49, no. 3, pp. 1–31, Dec. 2016.

[25] Z. Fang, W. Han, and Y. Li, "Permission based Android security: Issues and countermeasures," *Comput. Secur.*, vol. 43, pp. 205–218, Jun. 2014.

[26] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering—A systematic literature review," *Inf. Softw. Technol.*, vol. 51, no. 1, pp. 7–15, 2009.

[27] A. P. Siddaway, A. M. Wood, and L. V. Hedges, "How to do a systematic review: A best practice guide for conducting and reporting narrative reviews, meta-analyses, and meta-syntheses," *Annu. Rev. Psychol.*, vol. 70, no. 1, pp. 747–770, Jan. 2019.

[28] (2015). G. Gu. *Computer Security Conference Ranking and Statistic*. Accessed: Feb. 10, 2020. [Online]. Available: https://people.engr.tamu.edu/guofei/sec_conf_stat.htm

[29] A. Zamboni, A. Thommazo, E. Hernandes, and S. Fabbri, "StArt uma ferramenta computacional de apoioà revisão sistemática," in *Congresso Brasileiro de Softw. (CBSoft)*, Salvador, Brazil, 2010, pp. 91–96.

[30] E. Hernandes, A. Zamboni, S. Fabbri, and A. Di Thommazo, "Using GQM and TAM to evaluate StArt—A tool that supports systematic review," *CLEI Electron. J.*, vol. 15, no. 1, Apr. 2012.

[31] M. Bishop, "Race conditions, files, and security flaws; or the tortoise and the hare redux," Univ. California Davis, Davis, CA, USA, Tech. Rep. CSE-95-9, 1995.

[32] D. Dean and A. J. Hu, "Fixing races for fun and profit: How to use access(2)," in *Proc. 13th USENIX Secur. Symp. (USENIX Security)*, San Diego, CA, USA, Aug. 2004, pp. 195–206.

[33] C. Ko, G. Fink, and K. Levitt, "Automated detection of vulnerabilities in privileged programs by execution monitoring," in *Proc. 10th Annu. Comput. Secur. Appl. Conf.*, 1994, pp. 134–144.

[34] B. Goyal, S. Sitaraman, and S. Venkatesan, "A unified approach to detect binding based race condition attacks," in *Proc. Int. Workshop Cryptol. Netw. Secur. (CANS)*, 2003, p. 16.

[35] S. Bhatkar, A. Chaturvedi, and R. Sekar, "Dataflow anomaly detection," in *Proc. IEEE Symp. Secur. Privacy (S&P)*, May 2006, p. 15.

[36] O. Laadan, C.-C. Tsai, N. Viennot, C. Blinn, P. S. Du, J. Yang, and J. Nieh, "Finding concurrency errors in sequential code: OS-level, *in-vivo* model checking of process races," in *Proc. 13th USENIX Conf. Hot Topics Operating Syst. (HotOS)*, Napa, CA, USA, May 2011, p. 20.

[37] O. Laadan, N. Viennot, C.-C. Tsai, C. Blinn, J. Yang, and J. Nieh, "Pervasive detection of process races in deployed systems," in *Proc. 23rd ACM Symp. Operating Syst. Princ. (SOSP*, New York, NY, USA, 2011, pp. 353–367.

[38] T. Yu, W. Srisa-an, and G. Rothermel, "SimRacer: An automated framework to support testing for process-level races," in *Proc. Int. Symp. Softw. Test. Anal.*, Jul. 2013, pp. 167–177.

[39] T. Yu, W. Srisa-an, and G. Rothermel, "An automated framework to support testing for process-level race conditions," *Softw. Test., Verification Rel.*, vol. 27, nos. 4–5, p. e1634, Jun. 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1634

[40] F. Capobianco, R. George, K. Huang, T. Jaeger, S. Krishnamurthy, Z. Qian, M. Payer, and P. Yu, "Employing attack graphs for intrusion detection," in *Proc. New Secur. Paradigms Workshop*, New York, NY, USA, Sep. 2019, pp. 16–30.

[41] A. Aggarwal and P. Jalote, "Monitoring the security health of software systems," in *Proc. 17th Int. Symp. Softw. Rel. Eng.*, Nov. 2006, pp. 146–158.

[42] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva, "Portably solving file TOCTTOU races with hardness amplification," in *Proc. 6th USENIX Conf. File Storage Technol. (FAST)*, New York, NY, USA, 2008.

[43] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva, "Portably preventing file race attacks with user-mode path resolution," IBM Res., Yorktown Heights, NY, USA, Tech. Rep. RC24572 (W0806-008), 2008.

[44] S. Chari, S. Halevi, and W. Z. Venema, "Where do you want to go today? Escalating privileges by pathname manipulation," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, Mar. 2010, pp. 1–16.

[45] M. Payer and T. R. Gross, "Protecting applications against TOCTTOU races by user-space caching of file metadata," in *Proc. 8th ACM SIGPLAN/SIGOPS Conf. Virtual Execution Environ. (VEE)*, New York, NY, USA, 2012, pp. 215–226.

[46] X. Cai, R. Lale, X. Zhang, and R. Johnson, "Fixing races for good: Portable and reliable UNIX file-system race detection," in *Proc. 10th ACM Symp. Inf., Comput. Commun. Secur.*, New York, NY, USA, Apr. 2015, pp. 357–368.

[47] C. Cowan, S. Beattie, C. Wright, and G. Kroah-Hartman, "RaceGuard: Kernel protection from temporary file race vulnerabilities," in *Proc. 10th USENIX Secur. Symp. (USENIX Security)*, Washington, DC, USA, Aug. 2001, pp. 165–176.

[48] C. Ko and T. Redmond, "Noninterference and intrusion detection," in *Proc. IEEE Symp. Secur. Privacy*, May 2002, pp. 177–187.

[49] E. Tsyrklevich and B. Yee, "Dynamic detection and prevention of race conditions in file accesses," in *Proc. 12th Conf. USENIX Secur. Symp. (SSYM)*. Washington, DC, USA, Aug. 2003, p. 17.

[50] J. Park, G. Lee, S. Lee, and D.-K. Kim, "RPS: An extension of reference monitor to prevent race-attacks," in *Proc. Adv. Multimedia Inf. Process. (PCM)*. Berlin, Germany: Springer, 2004, pp. 556–563.

[51] K.-S. Lhee and S. J. Chapin, "Detection of file-based race conditions," *Int. J. Inf. Secur.*, vol. 4, nos. 1–2, pp. 105–119, Feb. 2005.

[52] P. Uppuluri, U. Joshi, and A. Ray, "Preventing race condition attacks on file-systems," in *Proc. ACM Symp. Appl. Comput. (SAC)*, New York, NY, USA, 2005, pp. 346–353.

[53] J. A. Kupsch and B. P. Miller, "How to open a file and not get hacked," in *Proc. 3rd Int. Conf. Availability, Rel. Secur.*, 2008, pp. 1196–1203.

[54] J. Rouzaud-Cornabas, P. Clemente, and C. Toinard, "An information flow approach for preventing race conditions: Dynamic protection of the Linux OS," in *Proc. 4th Int. Conf. Emerg. Secur. Inf., Syst. Technol.*, 2010, pp. 11–16.

[55] H. Vijayakumar, J. Schiffman, and T. Jaeger, "A rose by any other name or an insane root? Adventures in name resolution," in *Proc. 7th Eur. Conf. Comput. Netw. Defense*, 2011, pp. 1–8.

[56] H. Vijayakumar, J. Schiffman, and T. Jaeger, "STING: Finding name resolution vulnerabilities in programs," in *Proc. 21st USENIX Secur. Symp. (USENIX Security)*, Aug. 2012, pp. 585–599.

[57] H. Vijayakumar, J. Schiffman, and T. Jaeger, "Process firewalls: Protecting processes during resource access," in *Proc. 8th ACM Eur. Conf. Comput. Syst. (EuroSys)*, New York, NY, USA, 2013, pp. 57–70.

[58] H. Vijayakumar and T. Jaeger, "The right files at the right time," in *Proc. 5th IEEE Symp. Configuration Analytics Autom. (SafeConfig)*. Cham, Switzerland: Springer, Oct. 2013, pp. 119–133. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-01433-3_7

[59] H. Vijayakumar, X. Ge, M. Payer, and T. Jaeger, "JIGSAW: Protecting resource access by inferring programmer expectations," in *Proc. 23rd USENIX Secur. Symp. (USENIX Security)*, San Diego, CA, USA, Aug. 2014, pp. 973–988.

[60] C. Pu and J. Wei, "A methodical defense against TOCTTOU attacks: The EDGI approach," in *Proc. Int. Symp. Secure Softw. Eng.*, May 2006, pp. 1–9.

[61] X. Zhou, G. Li, K. Lu, and S. Wang, "Enhancing the security of parallel programs via reducing scheduling space," in *Proc. IEEE 12th Int. Conf. Dependable, Autonomic Secure Comput.*, Aug. 2014, pp. 133–138.

[62] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel, "Operating System Transactions," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Princ. (SOSP)*, New York, NY, USA, 2009, pp. 161–176.

[63] T. Kim and N. Zeldovich, "Practical and effective sandboxing for non-root users," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, San Jose, CA, USA, Jun. 2013, pp. 139–144.
[64] A. Bijlani and U. Ramachandran, "A lightweight and fine-grained file system sandboxing framework," in *Proc. 9th Asia–Pacific Workshop Syst.*, New York, NY, USA, Aug. 2018.
[65] C. Mulliner and B. Michéle, "Read it twice! A mass-storage-based TOCTTOU attack," in *Proc. 6th USENIX Workshop Offensive Technol. (WOOT)*, E. Bursztein and T. Dullien, Eds. Bellevue, WA, USA: USENIX Association, Aug. 2012, pp. 1–8.
[66] Y. Lee, T. Li, N. Zhang, S. Demetriou, M. Zha, X. Wang, K. Chen, X. Zhou, X. Han, and M. Grace, "Ghost installer in the shadow: Security analysis of app installation on android," in *Proc. 47th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2017, pp. 403–414.
[67] N. Borisov and R. Johnson, "Fixing races for fun and profit: How to abuse atime," in *Proc. 14th USENIX Secur. Symp. (USENIX Security)*, Baltimore, MD, USA, Jul. 2005, pp. 1–12. [Online]. Available: https://www.usenix.org/conference/14th-usenix-security-symposium/fixing% -races-fun-and-profit-how-abuse-atime
[68] X. Cai, Y. Gui, and R. Johnson, "Exploiting unix file-system races via algorithmic complexity attacks," in *Proc. 30th IEEE Symp. Secur. Privacy*, May 2009, pp. 27–41.
[69] R. Bisbey and D. Hollingsworth, "Protection analysis project final report," USC/Inf. Sci. Inst., Marina del Rey, CA, USA, Tech. Rep. ISI/RR-78-13, DTIC AD A, 1978, vol. 56816. [Online]. Available: https://csrc.nist.gov/csrc/media/publications/conference-paper/1998/10/08/proceedings-of-the-21st-nissc-1998/documents/early-cs-papers/bisb78.pdf
[70] O. Laadan, C.-C. Tsai, N. Viennot, C. Blinn, P. S. Du, J. Yang, and J. Nieh. (2011). *RacePro*. Accessed: Oct. 28, 2021. [Online]. Available: https://github.com/columbia/racepro
[71] J. A. Kupsch and B. P. Miller. (2008). *Safefile Library and Documentation*. Accessed: Oct. 28, 2021. [Online]. Available: https://research.cs.wisc.edu/mist/safefile/
[72] A. Dunn and D. Porter. (2017). *Operating System Demonstrating System Transactions*. Accessed: Oct. 28, 2021. [Online]. Available: https://github.com/ut-osa/txos
[73] H. Vijayakumar, J. Schiffman, and T. Jaeger. (2014). *Process Firewall*. Accessed: Oct. 28, 2021. [Online]. Available: https://github.com/siis/pfwall
[74] T. Kim and N. Zeldovich. (2013). *Mbox*. Accessed: Oct. 28, 2021. [Online]. Available: https://pdos.csail.mit.edu/archive/mbox/
[75] A. Bijlani and U. Ramachandran. (2019). *SandFS. A File System Sandboxing Framework*. Accessed: Oct. 28, 2021. [Online]. Available: https://sandfs.github.io/
[76] P. Yosifovich, A. Ionescu, M. E. Russinovich, and D. A. Solomon, *Windows Internals, Part 1: System Architecture, Processes, Threads, Memory Management, and More*, 7th ed. Redmond, WA, USA: Microsoft Press, 2017.
[77] L. Lu, A. C. Arpaci-dusseau, R. H. Arpaci-dusseau, and S. Lu, "A study of Linux file system evolution 1 introduction," in *Proc. 11th USENIX Conf. File Storage Technol. (FAST)*, San Jose, CA, USA, Feb. 2013, pp. 31–44.
[78] A. Peslyak. (2002). *Kernel Patches From the Openwall Project*. Accessed: Oct. 28, 2021. [Online]. Available: https://www.openwall.com/linux/

RAZVAN RADUCU received the B.Sc. degree in computer science and the M.Sc. degree in cybersecurity research from the University of León, Spain, in 2017 and 2019, respectively. He is currently pursuing the Ph.D. degree in computer science with the University of Zaragoza, Spain. His main research interests include program binary analysis, concurrency issues, and offensive security.

RICARDO J. RODRÍGUEZ (Member, IEEE) received the M.Sc. and Ph.D. degrees in computer science engineering from the University of Zaragoza, Spain, in 2010 and 2013, respectively. His Ph.D. dissertation was focused on performance analysis and resource optimization in critical systems, with special interest in Petri net modeling techniques. He is currently an Associate Professor at the University of Zaragoza. His research interests include performability analysis, program binary analysis, and memory forensics. He has been involved in reviewing tasks for international conferences and journals.

PEDRO ÁLVAREZ received the Ph.D. degree in computer science engineering from the University of Zaragoza, Spain, in 2004. Since 2000, he has been working as a Lecturer at the University of Zaragoza. He has participated in more than 30 research and innovation projects and is the author of more than 25 articles in various high-impact international journals. His current research interests include the problems of integration of network-based systems and the use of novel techniques and methodologies to solve them, as well as on the application of formal analysis techniques to mine event logs and databases. The results of his research work have been applied to different application domains, such as business intelligence, cybersecurity, health and sports, and e-learning.