IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 27 March 2023, accepted 21 April 2023, date of publication 3 May 2023, date of current version 11 May 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3272833

TOPICAL REVIEW

When Memory Corruption Met Concurrency: Vulnerabilities in Concurrent Programs

OSCAR LLORENTE-VAZQUEZ[1], IGOR SANTOS-GRUEIRO[2,3], AND PABLO GARCIA BRINGAS[1]

[1]Deusto Institute of Technology, University of Deusto, 48007 Bilbao, Spain
[2]Faculty of Engineering, Mondragon University, 20500 Arrasate-Mondragon, Spain
[3]HP Labs, BS34 8QZ Bristol, U.K.

Corresponding author: Oscar Llorente-Vazquez (ollorente@opendeusto.es)

The work of Oscar Llorente-Vazquez was supported in part by the Basque Government under a Pre-Doctoral Grant.

ABSTRACT Concurrent programs are widespread in modern systems. They make better use of processor resources but inevitably introduce a new set of problems in terms of reliability and security. Concurrency bugs usually lead to program crashes and unexpected behavior, and are an active research topic. From a security perspective, concurrency vulnerabilities are those that exhibit harmful behavior exclusively in concurrent executions. They can take place in a diverse range of environments, such as in operating system kernels, file system operations, or general-purpose multithreaded programs. A particular characteristic of concurrency is that it not only introduces new problems, but also enables traditional vulnerabilities to be triggered in concurrent-specific ways. Those that lead to dangerous security vulnerabilities usually cause memory corruption, a strong and flexible primitive for exploitation, and are known as concurrency memory corruption vulnerabilities. In this paper, we systematically analyze concurrency vulnerabilities in C and C++ programs, their exploitation and their detection, focusing on concurrency memory corruption vulnerabilities. We organize previous work on concurrency bug characteristics and detection, and highlight the differences in relation to vulnerabilities. Then, we examine the existence of concurrency vulnerabilities in real-world programs by searching the CVE database and point out a growing trend. Further, we analyze and compare existing detection approaches towards concurrency memory corruption.

INDEX TERMS Concurrency memory corruption, concurrency vulnerabilities, race condition.

I. INTRODUCTION

Concurrency is a crucial capability in modern systems. By means of different technologies, it makes it possible to maximize the usage of computing resources and increase overall performance. However, concurrency brings in new problems and challenges as well, especially in the context of security vulnerabilities. In contrast to sequential programs, concurrent program states are not only affected by program inputs but also by thread scheduling and the different thread interleavings, making it challenging to find, reproduce, and fix errors and vulnerabilities due to non-determinism [1].

Concurrency bugs usually arise as a result of concurrent accesses to a shared resource without proper synchronization. The most representative scenarios include shared memory accesses among threads within the same process, and system resource accesses among different processes [2], [3]. As a result of improper synchronization, different concurrent accesses are executed in non-deterministic interleavings, which compromises reliability and can lead to incorrect outputs, program crashes, and so on. Different from concurrency bugs (e.g., data races), concurrency vulnerabilities exhibit harmful behavior from a security perspective. Therefore, attackers are able to exploit these vulnerabilities to escalate privileges, execute malicious code, leak sensitive data, and so on. Concurrency vulnerabilities are relatively similar in nature to concurrency errors and have been generally oriented towards the file system (e.g., to read from or write to sensitive files) [4].

In practice, a concurrency vulnerability can become apparent as a result of a concurrency error, but it is not a mandatory requirement. In fact, making thread accesses to shared

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

memory race-free does not necessarily prevent a concurrency vulnerability from taking place, even if they are correctly protected by the same lock [5]. Therefore, a potential concurrency vulnerability may appear if two or more out of a set of events (i.e., memory operations) in a given execution can have their order changed, leading to harmful program behavior.

In a different direction, memory errors are still pervasive in current software (i.e., written in low-level languages such as C and C++) and have been the focus of research for many years. By now, researchers have proposed a myriad of approaches to exploit memory corruption, protect systems from these attacks, and to enforce memory safety to prevent errors [6]. However, memory errors do not take place exclusively in their traditional form (i.e., sequentially). The role of concurrency makes it possible for memory corruption to be triggered in additional ways in concurrent executions. This gives rise to concurrency memory corruption vulnerabilities that standard detection approaches fail to unmask and to which little attention has been paid. In fact, while examining kernel commits to search for concurrency use-after-free bugs, researchers found that most of reported concurrency use-after-free errors were identified by manual inspection or runtime testing [7].

At first, it was assumed that concurrency memory corruption vulnerabilities were only a consequence of concurrency errors such as data races [8], [9], and that they were visible later in the execution flow. However, a concurrency vulnerability is more related to the orders of events (at different granularities) that can be inverted in alternative executions and exhibit harmful behavior, no matter whether the corresponding accesses can form data races [10]. A major difference between these two concepts is that conflicting accesses are usually expected to take place simultaneously in a data race, but not in concurrency vulnerabilities, as long as the harmful execution order can materialize.

The most naive method to try to find concurrency vulnerabilities would be to explore all possible thread interleavings of a program to identify harmful execution orders of operations. However, the number of possible thread interleavings grows exponentially, leading to interleaving space explosion and making this method unfeasible. Further, using data race detectors to find concurrency vulnerabilities leads to potential vulnerabilities being missed, and requires additional effort, since usually only a small portion of all reported data races are indeed harmful and even some are purposefully introduced by developers. Recently, some approaches have been proposed that focus on detecting concurrency memory corruption vulnerabilities. Some of them rely on fuzzing due to its demonstrated effectiveness [11], [12], whereas others borrow and adapt predictive detection techniques from concurrency bug detection approaches [10], [13].

Although concurrency error characteristics and detection approaches have been extensively studied previously [1], [14], concurrency vulnerabilities — especially those related to memory corruption — have not received nearly as much attention. More worryingly, an increasing number of concurrency vulnerabilities are being reported each year. Therefore, a comprehensive analysis of concurrency memory corruption vulnerabilities and known detection methods remains an important open task. A systematic study in this field will lead to a better understanding and to the development of new detection approaches. In this paper, we aim at filling this gap by highlighting the differences between bugs and vulnerabilities, revealing the existence and growing trend of reported concurrency vulnerabilities in widely used software, and systematically analyzing the state-of-the-art in concurrency memory corruption detection for C and C++ programs.

The remainder of this paper is organized as follows: Section §II depicts the general taxonomy of concurrency bugs and reviews the state-of-the-art detection approaches. Section §III details the characteristics of concurrency memory corruption vulnerabilities and measures reported vulnerabilities in the CVE database. Section §IV explores the subtleties of exploiting concurrency vulnerabilities. Section §V describes and classifies state-of-the-art approaches towards concurrency memory corruption. Section §VI discusses the most remarkable points with regard to the presence of concurrency memory corruption in real-world programs and its detection approaches. Section §VII reviews the different works that are related to this paper. Finally, Section §VIII concludes the paper.

II. CONCURRENCY ERRORS

Concurrency errors are common in multithreaded programs and lead to unintended program behavior, resulting in wrong outputs, program crashes, and security vulnerabilities, for instance. Explicitly exploring their consequences revealed that they often lead to memory corruption [8], which suggests that traditional memory error detection approaches may fail detecting these specific cases since they are usually triggered under a certain scheduling sequence.

Although a concurrency error is not a necessary requirement to trigger a concurrency vulnerability [10], [12], their characteristics are remarkably similar. Moreover, many concurrency vulnerability detection approaches rely on or adapt existing concurrency error detection methods.

A. CLASSIFICATION

According to their observable properties, concurrency errors are usually classified into deadlocks, data races, atomicity violations and order violations [1].

Deadlocks arise as a consequence of improper coordination in the use of a synchronization mechanism (i.e., mutex locks). Specifically, a deadlock materializes when a set of threads are circularly waiting to each other to obtain a resource (i.e., a lock) that is held by another thread, not being able to continue the execution.

Data Races take place when multiple memory accesses from different threads access the same memory location (i.e., a shared variable) simultaneously without proper synchronization, and at least one of them is a write operation.

As a result, memory operations may get their intended order changed, leading to inconsistent memory views from different threads, unexpected memory values, and so on. Surprisingly, data races are often intentionally introduced in programs (e.g., for optimization) [15].

It is worth noting that data races and race conditions are different concepts, even though they are often used interchangeably. While data races are exclusive to improper synchronization in concurrent memory accesses with at least one memory write, race conditions refer to timing dependent behavior in the execution.

Atomicity Violations materialize when the desired serializability among multiple operations in a concurrent execution is interleaved with operations from another thread (i.e., a code region is assumed to be atomic, but the atomicity is not enforced during execution). Atomicity violation bugs are closely connected to unserializable interleavings, which are interleavings that are not equivalent to any sequential execution of the involved operations [16], [17].

Order Violations, in a conceptually similar vein to atomicity violation errors, take place when the desired order for a set of concurrent operations is assumed but not enforced during the execution. For instance, when a memory access is expected to precede another but the order is flipped at runtime.

B. DETECTION APPROACHES

Over the years, many different approaches have been proposed towards the challenge of identifying concurrency errors in multithreaded programs. Some try to detect or predict specific bug categories whereas others work towards automatically fixing concurrency bugs. In reality, there is no absolute solution, but the different approaches have their own strengths and weaknesses in terms of false negatives, false positives, and performance.

We focus on non-deadlock bugs since their characteristics are the most similar to concurrency vulnerabilities, and have been shown to lead to memory corruption and even to concurrency vulnerabilities. However, we encourage the reader to refer to existing literature for further information on deadlock detection and mitigation [18], [19]. Since this work is focused on unsafe C and C++ programs, we also do not consider approaches for other languages such as Java [20].

DATA RACES

Data races have been the focus of most concurrency bug detection works [1], [2], [21], [22], [23], proposing many different static, dynamic, and systematic testing (e.g., stateless model checking) methods.

Static approaches are intended to approximate the runtime behavior of the program without it being executed, obtaining thread identifiers, identifying synchronization operations and checking memory operations that potentially access the same memory location statically. For instance, `RacerX` [24] uses flow-sensitive inter-procedural analysis to infer lock-protected operations, multithreaded contexts and dangerous shared accesses, whereas `LOCKSMITH` [21] uses context-sensitive correlation analysis to determine whether the accessed memory locations are consistently protected by locks. Unfortunately, pure static approaches usually produce a high number of false alarms, increasing manual effort.

Dynamic solutions reason about synchronization operations and memory accesses in specific execution traces and have been proven to be very effective in data race detection. Existing algorithms can be broadly classified into three groups: *sound predictive analysis*, *lock-set analysis*, and *partial order* based techniques.

Firstly, *sound predictive analyses* are intended to explore all possible feasible reorderings of a given trace, thus being sound and complete in theory. In practice, these techniques are expensive since each trace has exponentially numerous valid reorderings. Therefore, completeness is traded for performance by slicing traces into small fragments and analyzing these fragments individually. These approaches usually formulate race detection as a constraint solving problem and rely on SAT/SMT solvers [23], [25] to detect data races.

Secondly, *lock-set* based approaches track the set of locks that are held during memory accesses in the execution. They report a potential race whenever two threads access the same memory location without proper locking [15], [26].

Thirdly, *partial order* based methods formulate a partial order P on the events in a given trace so that event pairs unordered by P conform data races. The most common implementation uses the *happens-before relation* (HBR). HBR detects a data race when two conflicting accesses (i.e., accesses from different threads to the same memory location including at least one write operation) are not causally ordered so that neither is forced to happen before the other, meaning that a reordering exists. Many tools are based on HBR and have explored different ways to implement it efficiently [2], [27]. In an effort to overcome the limitations of HBR, other *partial order* techniques such as *schedulable happens-before* [28], *weak-causally-precedes* [29], and *doesn't commute analysis* [30] have been proposed in recent years.

Lastly, hybrid approaches such as `ThreadSanitizer` [22] and `Helgrind` [31] combine *lock-set* and HBR analyses to improve detection capabilities and are widely used in practice.

Other techniques aim at exposing concurrency bugs by systematically exploring different executions and multiple thread interleavings. For instance, testing methods for concurrent programs are based on some sort of controlled scheduling, since ordinary testing is inadequate due to non-determinism [3]. To this end, existing solutions modify thread priorities dynamically [11], [12], introduce delays at specific locations [3], or explore the interleaving space in a randomized way [32]. `Conzzer` [33] is a fuzzing framework for concurrent programs that uses a new context-sensitive coverage metric based on pairs of concurrently executed functions, augmented with their calling contexts (i.e., the runtime call

stack). To explore as many thread interleavings as possible, it uses an adjacency-directed mutation strategy of the already covered concurrent call pairs to infer other potential concurrent call pairs. Then, it leverages previously injected breakpoints at function entries to tamper with thread scheduling, to try to cover these new thread interleavings. To detect data races, it implements a dynamic *lock-set* analysis on the identified concurrent function pairs, and verifies them by injecting breakpoints at the racy instructions.

Moreover, controlled concurrency testing and stateless model checking are both actively and effectively applied techniques in the industry [34], [35]. To illustrate, `Nekara` [34] is an open-source library that facilitates the development of systematic testing solutions for multiple platforms. It provides an API to model the set of supported concurrency primitives, and encapsulates state-of-the-art search heuristics to explore the interleaving space from existing frameworks (e.g., in a randomized fashion, or focusing on schedules with limited context switches).

ATOMICITY VIOLATIONS

Following data races, atomicity violations are the second most discussed class of non-deadlock concurrency errors in the literature. Atomicity violation bugs widely exist because which part of the code needs to be atomic depends on programmers' intentions, who are often used to sequential thinking and assume non-atomic code regions to be atomic. Therefore, one of the most challenging tasks for the detection of atomicity violations is how to identify atomic regions.

On the one hand, some static approaches rely on user annotations and type systems to specify and verify atomic code regions (i.e., functions) [36]. On the other hand, dynamic methods such as `AVIO` [16] propose *access interleaving invariants* and build a system that extracts these invariants to detect violations at runtime, whereas `Atomizer` [37] leverages *lock-set* and reduction algorithms to verify that the execution of atomic blocks is not affected and does not interfere with other threads. Other approaches build upon the concept of *conflict serializability* of execution traces, soundly identifying traces that cannot be transformed into equivalent serial traces and detecting violations dynamically [38], [39].

From a testing perspective, frameworks such as `CTrigger` [17] and `AssetFuzzer` [40] focus on the directed exploration of specific thread interleavings inherently correlated to atomicity violations. They first perform analyses to identify candidate interleavings to then direct the execution towards them.

ORDER VIOLATIONS

In concurrency bug detection research, order violations have received the least attention, although they are widely found among concurrency errors [1]. Preventing and fixing them is particularly difficult, since even if two shared memory accesses are protected by the same lock or two conflicting code regions are atomic to each other, the execution order between them still may not be guaranteed.

Similar to atomicity violations, it is common among developers to assume an execution order but enforcing that order is notably challenging. Further, to detect order violations, one needs to be able to identify whether the executed instructions have been executed in the right order or not. Consequently, this raises the challenge of determining the expected orders between pairs of events.

Among the proposed approaches, not many target order violations specifically. Instead, there are general detectors that are not focused on a single class of concurrency bugs but target multiple classes in a more general fashion. For instance, `Falcon` [14] uses a pattern-based dynamic analysis method that identifies conflicting interleaving patterns that access shared memory to then statistically rank them based on a suspiciousness measure. `ConMem` [8] targets concurrency errors that lead to program crashes, building on bug effects instead of on specific interleaving patterns, unlike most approaches. To this end, it predicatively detects errors by identifying potentially harmful memory operations and then checking timing conditions among them. In a similar vein, `ConSeq` [41] proposes a backwards analysis approach that covers more failure patterns besides program crashes. Recently, other frameworks that target order violations specific to the Android platform have also been developed [42], [43].

BUGS IN SPECIFIC DOMAINS

Some approaches are specifically tailored to particular problems and domains such as operating system kernels, embedded systems, file systems, and so on. Therefore, they do not target general-purpose concurrent programs and are designed to overcome unique challenges.

`PASAN` [44] is one of such approaches and seeks to protect peripherals in embedded systems from concurrency issues that corrupt on-going jobs and result in erroneous sensor values, for instance. To this end, it (i) parses device documents to identify MMIO address ranges for peripherals, (ii) extracts concurrency-related functions from firmware compiled into LLVM bitcode, (iii) computes points-to information, identifies concurrently executable instructions by tracking process and thread life spans, and performs a context-sensitive *lock-set* analysis of the call stacks. Then, it (iv) identifies transaction spans (that should ideally be locked), and (v) verifies that the identified transaction spans that access the same MMIO address range are correctly locked.

Other approaches are focused on detecting data races in operating system kernels (i.e., Linux). `Razzer` [45] uses a combination of static analysis and fuzzing. It first identifies potential racy memory access instructions through context-insensitive and flow-insensitive points-to analysis, over-approximating race candidate pairs. Then, it injects breakpoints at the identified points, and performs fuzz testing seeking to execute the race candidates while controlling thread interleavings. `Krace` [46] is a fuzzing framework that targets kernel file systems. Apart from the common branch coverage metric, it uses a concurrency-coverage metric called

alias instruction pair, which describes thread interleavings by tracking memory access instruction pairs from different contexts to each memory address. It also injects random delays at memory accesses to discover thread interleavings. To check for data races, it implements offline *happens-before* and *lock-set* algorithms.

In a different direction, researchers found that jointly exploring interleavings and test inputs in kernels has received little attention due to the big search space [47]. In that context, `Snowboard` [47] is a concurrency testing framework that first executes a corpus of sequential tests generated by `Syzkaller` [48], and collects non-stack memory accesses for the relevant thread using a customized hypervisor. It then identifies and clusters pairs of tests that read and write the same memory region respectively. Next, it concurrently executes test pairs, segregating threads in separate vCPUs, and executing one vCPU at a time to enforce different interleavings. To find bugs, it relies on existing bug detectors, and ranks them by frequency, manually inspecting the highest-ranked ones.

C. FROM ERRORS TO VULNERABILITIES

Some of the approaches described above are also used to detect security vulnerabilities stemming from the aforementioned concurrency errors, usually through manual inspection [33], [45]. In this way, they have been able to identify denial of service opportunities, privilege escalation, and memory corruption instances. However, these tools are geared towards detecting certain concurrency bugs, and even discerning benign races from harmful ones is often a costly manual task. To illustrate, researchers spent approximately 80 hours on manual inspection and reproduction of 100 data races [47]. To find and verify security consequences, it requires further manual inspection and extra effort — a time consuming process that requires meticulous code analysis.

III. CONCURRENCY VULNERABILITIES

In previous works [9], [49], researchers analyzed and discussed how concurrency errors can be exploited to carry out several attacks. However, further research showed that even though vulnerabilities and errors have similar characteristics, their surrounding conditions are different and neither is a necessary requirement for the other to take place [10], [11]. For instance, concurrency vulnerabilities can exist in race-free executions [5], and triggering a concurrency bug and a vulnerability usually need different inputs and thread interleavings.

Therefore, existing bug detection approaches may fail to detect vulnerabilities on their own. In the event that a vulnerability would arise out of an error, they would be able to detect the corresponding error but not whether there could be a vulnerability or its triggering conditions, since they do not reason about harmful program behavior from a security perspective. Similarly, concurrency vulnerability detection approaches that build upon bug detectors may miss potential vulnerabilities.

```
static int nbd_handle_cmd(struct nbd_cmd *cmd, int index)
{
    struct request *req = blk_mq_rq_from_pdu(cmd);
    struct nbd_device *nbd = cmd->nbd;
    struct nbd_config *config;
    struct nbd_sock *nsock;
                                ...
    config = nbd->config;
                                ...
    nsock = config->socks[index];
    mutex_lock(&nsock->tx_lock);
                                ...
}

static int nbd_add_socket(struct nbd_device *nbd,
     unsigned long arg, bool netlink)
{
    struct nbd_config *config = nbd->config;
    struct socket *sock;
    struct nbd_sock **socks;
                                ...
    socks = krealloc(config->socks,
        (config->num_connections + 1) *
        sizeof(struct nbd_sock *), GFP_KERNEL);
    if (!socks) {
        kfree(nsock);
        err = -ENOMEM;
        goto put_socket;
    }
    config->socks = socks;
                                ...
}
```

FIGURE 1. CVE-2021-3348 in the Linux kernel.

Among the vulnerabilities that have their origin in concurrency, the most well-known include Time-Of-Check-to-Time-Of-Use (TOCTOU) vulnerabilities that target the file system [4], and double-fetches [50]. However, concurrency can also lead to other vulnerabilities that for instance enable side-channel attacks [51]. In this paper, we focus on memory corruption exclusive to concurrent executions.

A. REAL-WORLD VULNERABILITY EXAMPLES

In order to provide a practical illustration of real-world concurrency memory corruption vulnerabilities, we have manually selected two representative examples from the CVE database. Figure 1 shows a simplified version of the code in `drivers/block/nbd.c` in the Linux kernel, that is affected by CVE-2021-3348 [52], a concurrency use-after-free vulnerability.

When a Network Block Device (NBD) is being set up, the function *nbd_add_socket* gets called, which in turn calls *krealloc* to reallocate memory for the *config->socks* array to add new sockets to the configuration. Usually, *krealloc* will decide to expand or shrink the original memory block in place if possible, or to allocate a new memory block, copy the data, and free the original block. At that point, if an I/O request is received, *nbd_handle_cmd* will get called, dereferencing *config->socks* without any locking. Since this happens after *config->socks* is reallocated but before it gets assigned the new memory address, a use-after-free will take place.

To fix this vulnerability, locking accesses to *config->socks* was not a feasible solution, as it would introduce excessive

```
/* In tftpd_list.c */
struct thread_data *thread_data = NULL;

                              ...
int tftpd_list_add(struct thread_data *new)
{
    struct thread_data *current = thread_data;
    int ret;

    pthread_mutex_lock(&thread_list_mutex);

                              ...
    if (thread_data == NULL)
    {
         thread_data = new;
         new->prev = NULL;
         new->next = NULL;
    }
    else
    {
         while (current->next != NULL)
              current = current->next;
         current->next = new;
         new->prev = current;
         new->next = NULL;
    }
    pthread_mutex_unlock(&thread_list_mutex);
    return ret;
}

/* In tftpd.c */
void *tftpd_receive_request(void *arg)
{
    struct thread_data *data = (struct thread_data *)arg;

                              ...
    if (!abort)
      stats_new_thread(tftpd_list_add(data));

                              ...
}
int main(int argc, char **argv)
{

                              ...
    if (pthread_create(&tid, NULL, tftpd_receive_request,
          (void *)new) != 0)

                              ...
}
```

FIGURE 2. CVE-2019-11366 in atftp.

overhead. Instead, kernel developers opted to freeze the request queue to not receive requests when adding sockets to the configuration.

In a similar vein, Figure 2 illustrates a shortened version of the relevant code with regards to CVE-2019-11366 [53] in atftp [54], a multi-threaded implementation of the Trivial File Transfer Protocol (TFTP).

In `tftpd_list.c`, atftp uses a global linked list of server threads *thread_data*. These threads are started by the main thread when new requests arrive. To do so, the *main* function calls *pthread_create* to execute *tftpd_receive_request*, which later calls *tftpd_list_add*. This is shown in the bottom lines of the code. Inside *tftpd_list_add*, atftp initializes a local pointer to the head of the thread list *thread_data*. Then, it checks whether the thread list is empty by checking whether *thread_data* points to NULL. In that case, it inserts the first element to the list. Otherwise, it walks the list using the local pointer *current* and inserts the element at the end of the list.

Since the assignment to *current* is not lock-protected, the harmful behavior occurs when *thread_data* is NULL when assigned to *current* in a thread, but modified by another thread before it is checked. In such a case, the program will try to walk the list and dereference *current->next*, resulting in a null-pointer dereference.

B. VULNERABILITY PRESENCE

To observe and study the presence and the possible trends in concurrency vulnerabilities in real-world software, we have developed a series of scripts that automatically download and parse all CVE entries from the National Vulnerability Database (NVD) data feeds [55].

CATEGORIZATION

We classify vulnerabilities by looking at their assigned Common Weakness Enumeration (CWE) identifiers combined with pattern matching using regular expressions. Unfortunately, it is nearly impossible to obtain fully accurate results for several reasons. To begin with, descriptions and vulnerability reports do not follow a particular standard and therefore are occasionally unstructured or contain incomplete information. Many entries do not have a CWE assigned or, in the case of concurrency vulnerabilities, their assigned identifiers do not fall into the category of concurrency issues but into the categories of the consequent weaknesses (e.g., use-after-free), making it harder to identify them. In addition, searching for simple keywords such as ''thread'' or ''concurrent'' is not appropriate because they may wrongly match cases where they are used to describe software functionality or other unrelated execution aspects. Consequently, more sophisticated patterns are needed.

Bear in mind that our goal is not to develop the most accurate and comprehensive possible tool, but to obtain a more general understanding of the concurrency vulnerability landscape. Therefore, even though it may not yield the finest results and more exhaustive techniques could have been used, it still provides useful and valuable insights.

STATISTICS

Using the method described above, we then collected statistical information about the reported vulnerabilities ranging from 2002 to the end of 2021. Note that we are primarily interested in concurrency-related vulnerabilities and thus we only include other categories as a reference. There are many different categories and subclasses, as well as many possible classification criteria. For simplicity, we chose to show memory corruption since it is the sequential equivalent of the vulnerabilities we are interested in, and web-related vulnerabilities as a reference to another broad category. Within concurrency vulnerabilities, we further identify those that result in memory corruption.

Table 1 shows the number of reported vulnerabilities in the NVD database for each of the categories that we selected along with the mean and standard deviation of their Common Vulnerability Scoring System (CVSS) scores. CVSS base scores represent the innate characteristics and severity of

TABLE 1. Number of reported CVEs for the selected categories and the total, along with their mean CVSS score.

Category	# Vulnerabilities	CVSS
Concurrency memory corruption	419	7.2±1.1
Concurrency vulnerabilities	1,461	6.7±1.3
Memory corruption	35,899	7.7±1.4
Web-related vulnerabilities	46,602	6.8±1.7
Total	172,915	-

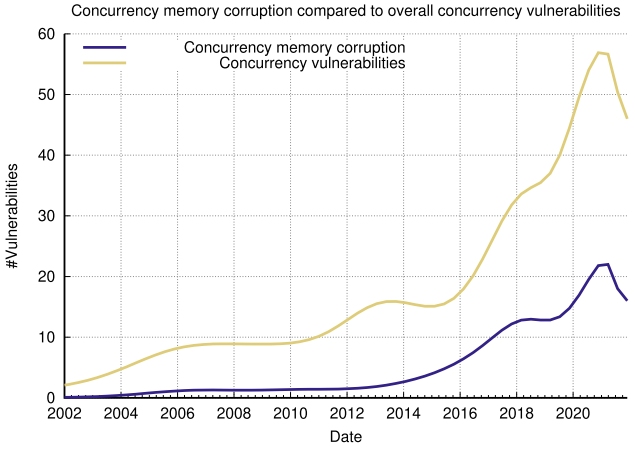

FIGURE 3. Concurrency vulnerabilities and concurrency memory corruption growth over the years.

vulnerabilities, and range from 1 to a maximum of 10. Unfortunately, not every CVE entry has a CVSS score specified. For instance, out of the 35.899 memory corruption vulnerabilities identified, only 22.292 had a CVSS score assigned.

1,461 out of the 172,915 total reported vulnerabilities are concurrency vulnerabilities, of which 419 are concurrency memory corruption vulnerabilities. In contrast, other extensive and persistent categories such as memory corruption and web-related vulnerabilities account for 35,899 and 46,602 vulnerabilities respectively. Regarding CVSS scores, the four selected categories have similar values, resulting in a high severity rating in most of the cases. As expected, broad categories that have more diversity in the nature of the vulnerabilities show a higher standard deviation.

Figure 3 shows the number of both reported concurrency vulnerabilities and the subclass of concurrency memory corruption vulnerabilities per 3 months from 2002 to end 2021. Not surprisingly, in the early years barely any vulnerabilities were reported in either of the two cases. Out of these, the vast majority were TOCTOU and other file system related race condition vulnerabilities. Afterwards, the number of reported concurrency vulnerabilities started to gradually increase until 2016, from where it experienced a substantial and continuous growth up to 2021, reaching its peak at nearly 60 vulnerabilities per 3 months. This growth can be attributed to both the rising development of increasingly complex concurrent software and the recently proposed detection approaches. Concurrency memory corruption vulnerabilities followed a similar trend. The number of reported vulnerabilities started to steadily grow from 2014 onwards. Although it initially looked like it may have stabilized as of 2018, it then continued to grow until it peaked in 2021 as well, at over 20 reported vulnerabilities per 3 months. It is worth noting that 74% of the total number of concurrency memory corruption vulnerabilities have been reported over the period from 2017 to 2021, following an upward trend.

IV. CONCURRENCY EXPLOITATION

Some research works have been devoted to understand and improve the exploitation of concurrency vulnerabilities. `OWL` [9] was the first study that analyzed the problem of exploiting concurrency vulnerabilities stemming from data races. Their approach uses existing data race detectors and checks whether the corresponding memory region propagates to attack sites (e.g., *strcpy()* calls, NULL dereferences) through control-flow or data-flow via inter-procedural static analysis. Then, it tries to trigger the bug and execute an attack site.

They found that the duration of the vulnerable window (i.e., the timing window in which the error may materialize) is key to successfully exploit a concurrency error. While some concurrency errors have large vulnerable windows that enable their exploitation without much difficulty through triggering sequences of UI events or re-running commands a few times, those with shorter vulnerable windows are harder to exploit. Moreover, hardware cache leases and CPU time slices are often larger than these short timing windows, masking the errors.

In order to increase the probability of success of the attacks, they proposed two methods: (i) to retry multiple times until the exploit succeeds, and (ii) to provide carefully crafted inputs to enlarge the vulnerable window (e.g., by making a thread copy a large array, or triggering blocking operations such as disk I/O).

Afterwards, researchers verified that traditional race exploitation in practice relies on brute-force attacks until it succeeds [56]. Such attacks work for some kernel races and have been leveraged by most race-based privilege escalation attacks. However, brute-force is not effective at all in some cases. The exploitation complexity varies for single-variable races and multi-variable races. They also found that some multi-variable races (named non-inclusive multi-variable races) are almost impossible to successfully exploit through brute-force, since they require a unique execution order that cannot be triggered without using debugging features.

Even though the probability of successful exploitation of single-variable races and inclusive multi-variable races (i.e., when the execution time between the two writes in one task is shorter than the time taken between the two reads in the other task) through brute-force appears to be low, exploitation is indeed possible and effective with many trials. In fact, publicly available brute-force exploits succeed in between 5 to 30 seconds. However, the probability of successful exploitation of non-inclusive multi-variable races (i.e., the

execution time between the two writes in one task is greater than or equal to the time taken between the two reads in the other task) using brute-force is virtually zero due to timing constraints for the tasks to satisfy the required execution orders.

ExpRace [56] proposes a generic exploitation technique for non-inclusive multi-variable kernel data races, which cannot be exploited through brute-force. The main objective is to intentionally enlarge the execution time of the target instruction sequence of the task that performs the read accesses using interrupts, so that a non-inclusive multi-variable race turns into an inclusive multi-variable race. Since the interrupt mechanisms are not accessible from user-space, ExpRace indirectly raises interrupts using different methods to enlarge the time window that enables exploitation. In particular, it focuses on inter-processor interrupts and hardware interrupts through system calls and interrupt requests to devices. They found that hardware interrupts yield the better results in terms of exploitation success and stability.

In a different direction, COMRace [57] seeks to automatically synthesize proof-of-concept (PoC) exploits that concurrently execute unsafe method pairs with conflicting member field accesses in vulnerable Component Object Model (COM) interfaces. To do so, it finds interface methods that access member fields without synchronization through reverse engineering and static analysis. Then, given a pair of interface methods with conflicting accesses, it automatically synthesizes PoC exploits that concurrently invoke these methods in a loop, expecting to trigger the vulnerable interleaving.

V. DETECTION APPROACHES

Memory errors are widespread in current software and pose a major threat from a security perspective since they enable several attacks [6]. In consequence, numerous detectors and mitigations have been proposed in over three decades of research. Unfortunately, concurrency introduces new ways in which memory corruption may take place, giving rise to the need for new detectors that can find vulnerabilities that traditional tools cannot identify.

In this section, we bring together the relevant proposed concurrency memory corruption detectors in the literature. As in previous sections, we only consider approaches that target C and C++ programs. Thus, we do not include frameworks such as ExceptioNULL [58], specific to Java programs. The individual approaches are described in their own sections below.

A. PREDICTIVE DETECTION ALGORITHMS

Tools within this category seek to predict vulnerabilities that can take place from an observed correct execution. They are usually based on existing concurrency bug detection algorithms that have been adapted to vulnerability detection.

Table 2 summarizes proposed predictive detection algorithms for concurrency memory corruption detection in the literature. It details the underlying detection method of each tool. We also specify the vulnerability subclasses targeted by the different tools. In addition, we point out the output scheme of the reported vulnerabilities since it is a relevant aspect in the reproduction and verification of findings. The validation method refers to whether the proposed approaches try to trigger the identified vulnerabilities to verify their execution. Although this is intrinsic to testing approaches such as fuzzing, tools based on predictive detection may also benefit from it to avoid false positives.

UFO

UFO [13] formulates concurrency use-after-free detection as a constraint solving problem and builds upon a maximal causality model with the intention to predict the maximal set of vulnerabilities from an observed concurrent execution trace. Since the original maximal causal model [23] is not aware of the semantic properties that define a use-after-free, UFO first extends it by including allocation, use, and free events to then encode the model and use-after-free violations as first order logical constraints to be solved using a SMT solver.

To this end, UFO first modifies ThreadSanitizer to produce the execution trace that contains occurrences of all the previously defined events in the model. This data is encoded and written to disk for offline analysis for correctness and efficiency. Afterwards, UFO builds the model's constraints from the traces and finds allocation and free operation pairs along with the conflicting memory accesses. In order to detect use-after-frees, it then carries out a *happens-before analysis* for each pair of use and free events on overlapping memory ranges. If the HBR is not satisfied either way by inter-thread synchronization, then UFO builds the conjunction of the model and the use-after-free constraints and invokes Z3 [59] to solve it. If it produces a solution, UFO reports the specific scheduling that triggers the use-after-free.

CONVUL

In ConVul [10], the authors highlight the differences between data races and concurrency memory corruption vulnerabilities, and the ineffectiveness of data race detectors in finding concurrency vulnerabilities.

They propose the concept of *exchangeable events* to refer to pairs of events whose orders can be reversed. In its strict version, when for a pair of events both execution orders are observed, they are considered to be exchangeable. However, that approach is not practical. In consequence, they define a relaxed model that heuristically predicts whether two events are exchangeable based on their *synchronization distance*. The *synchronization distance* is the minimal number of synchronization edges that order the two events, being synchronization edges sequences of either lock acquire and release operations in the same thread, or release and later acquire operations by different threads. Taking this definition into account, they determine that, for a pair of events, if their *synchronization distance* is 3 or less, there is a high probability to reverse their execution order. Further, if there is a third event

TABLE 2. Predictive detection approaches for concurrency memory corruption vulnerabilities along with their main characteristics.

	UFO	ConVul	ConVulPOE
Technique	Maximal Causal Model + Constraint Solving	Exchangeable Events Tracking	Trace Construction via Partial Orders
Targeted vulnerabilities	Use-After-Free	Use-After-Free, NULL-Pointer Dereference, Double-Free	Use-After-Free, NULL-Pointer Dereference, Double-Free
Output	Conflicting Call Stacks	Conflicting Call Stacks	Vulnerability-Exposing Traces
Validation Method	-	Enforced Scheduling	-

such that the distances from both events to it are close to 0, then there will be an increased probability.

Given the above definition, they implement a detection system on top of `Pin` [60] dynamic binary instrumentation framework that instruments synchronizations and memory accesses to track exchangeable events. Whenever two conflicting operations, as defined by the different algorithms for each targeted concurrency memory corruption vulnerability (see Table 2), are found to be exchangeable, a potential vulnerability is reported.

To verify reported vulnerabilities, `ConVul` tries to trigger the harmful execution order by suspending the thread that should execute the last operation (e.g., the use in a use-after-free) right before its execution.

CONVULPOE

In a similar vein to `ConVul`, `ConVulPOE` [61] focuses on exchangeable event pairs because they are more likely to cause concurrency memory corruption. In this approach, two events are considered exchangeable if two valid traces exist in which the execution order of these events is different.

The main objective of `ConVulPOE` is to construct feasible traces that manifest vulnerabilities from an observed execution trace. To that end, it leverages partial orders that represent the execution orders of events. In the first place, it needs to identify vulnerability-potential event pairs from an observed trace, in accordance with the targeted vulnerabilities (see Table 2). Next, it assembles a feasible subset of events. These feasible event sets are the events that should be included in the new trace (i.e., the event pairs that can lead to a vulnerability if reordered) along with the other events that make the execution correct. To prove that a potential vulnerability may exist, it tries to generate a new feasible trace by building a partial order over the feasible event set to add ordering between events while meeting all necessary constraints.

`ConVulPOE` is composed of a trace recorder and a vulnerability predictor component. The trace recorder is implemented on top of `Pin` to instrument threading and dynamic memory allocation operations, as well as pointer dereferences heuristically. Taking the generated traces as input, the vulnerability predictor works as an offline analysis to find potential event pairs and construct partial orders incrementally.

B. CONCURRENCY TESTING

Testing methods specific to concurrency randomly or systematically explore different thread interleavings and program states. For instance, systematic state space exploration by model checking is an effective verification method. Since traditional model checkers are not practical for concurrent programs, stateless model checkers systematically explore traces of shared memory accesses without capturing all program states. A custom scheduler controls the non-determinism in the program and enumerates all execution paths.

Furthermore, dynamic analyses through testing such as fuzzing and controlled concurrency testing aim at effectively triggering concurrency memory corruption vulnerabilities in actual executions. Their main objectives and challenges include the exploration of the thread interleaving space and the orchestration of thread scheduling.

Table 3 encapsulates concurrency testing approaches towards concurrency memory corruption detection. It specifies the underlying detection technique of each tool, and the vulnerability subclasses targeted by the different tools. In addition, we point out the output scheme of the reported vulnerabilities since it is a relevant aspect in the reproduction and verification of findings.

CONAFL

`ConAFL` [11] is an heuristic framework composed of a static analysis phase and thread-aware fuzzing using a priority-based strategy. In the first place, the static analysis locates sensitive concurrent operations on shared memory (e.g., calls to *memcpy()*) and categorizes them into a potential type of vulnerability from the targeted concurrency vulnerabilities (i.e., buffer overflow, use-after-free and double-free). Then, a custom fuzzer built upon `AFL` [62] adjusts thread priorities through instrumentation and executes the target program.

The static analysis is implemented upon the concurrency bug detector `LOCKSMITH` and can be divided into four steps: (i) discover shared memory by recording pointers that are either passed through *pthread_create* or point to a global

TABLE 3. Proposed concurrency testing approaches towards memory corruption vulnerability detection along with their main characteristics.

	ConAFL	MUZZ	AutoInter-fuzzing	PERIOD	GenMC
Technique	Static Sensitive Operation Finding + Fuzzing	Thread-Aware Grey-Box Fuzzing	Static Operation Pair Finding + Fuzzing	Controlled Concurrency Testing	Stateless Model Checking
Targeted vulnerabilities	Buffer Overflow*, Use-After-Free*, Double-Free	General Memory Corruption	Use-After-Free, Double-Free, NULL-Pointer Dereference	Use-After-Free, Double-Free, NULL-Pointer Dereference	Use-After-Free, Double-Free, Uninitialized Memory Access
Output	Crashing Input and Interleaving	Crashing Input and Interleaving	Crashing Input and Interleaving	Buggy Schedule	Offending Execution Graph

* Although buffer overflow and use-after-free are specified as targeted vulnerability classes in ConAFL, it could not detect any in the experiments.

variable in the different threads, (ii) mark sensitive operations by building a Data-Flow Graph (DFG) for the identified shared variables and labeling sensitive operations (e.g., calls to *free()*) on that DFG, (iii) find pairs of concurrent sensitive operations by merging data-flows that share a preceding node and verifying that the operation pairs are concurrent by checking the Control-Flow Graph (CFG), and (iv) categorize each sensitive operation pair into a potential vulnerability type (e.g., two calls to *free()* to a double-free).

To trigger potential vulnerabilities, the thread-aware fuzzer inserts an assembly code snippet around the identified operations from a sensitive operation pair. This code adjusts the priority of the thread so that the two threads would likely be executed in the order that would trigger the vulnerability.

MUZZ

General-purpose grey-box fuzzers are usually tied to single-threaded testing, and thus they are not aware of multithreading-specific contexts. To overcome these limitations on fuzzing multithreaded programs, MUZZ [12] proposes three new thread-aware instrumentation algorithms on top of AFL that provide useful runtime feedback to optimize seed generation and execution strategies to explore execution states originated from different thread interleavings.

Prior to instrumentation, MUZZ builds a thread-aware interprocedural CFG (ICFG) that encodes threading operations information such as thread creation and exit, lock acquires and releases, and shared variables. Given that ICFG, it identifies code sequences relevant to multithreading that may interleave during execution (i.e., suspicious interleaving scope) to later focus instrumentation on them. The objective is to emphasize multithreading-relevant paths during fuzzing and to reduce the amount of instrumentation required.

- Coverage-oriented instrumentation aims at tracking thread-interleaving induced coverage. It does so by extending a common coverage approach [62] to instrument basic blocks that include suspicious interleaving scope as well as specific instructions inside with certain probability, and to instrument fewer blocks that do not.
- Threading context instrumentation collects the thread context (i.e., a deputy instruction and thread ID tuple) at calls to lock, unlock and join. When execution ends, MUZZ computes a context signature via hashing that defines the overall thread context.
- Schedule-intervention instrumentation seeks to diversify thread interleavings by assigning a random priority value to threads on thread creation. It also stores a thread ID for the calling thread to maintain a structure.

During the fuzzing process, MUZZ prioritizes seeds that exercise new coverage as done in AFL, but giving precedence to seeds that cover new thread contexts by checking the context signatures. In addition, it also adjusts the number of repeated executions for seeds that produce different context signatures in order to discover new thread interleavings.

AUTOINTER-FUZZING

Similar to other approaches, AutoInter-fuzzing [63] combines static analysis and fuzz testing. It leverages static points-to analysis on top of LLVM to extract operation pairs that access the same memory region and are executed in different threads, in a similar way to Razzer. These operations include load and store operations, external function calls using local pointers, and memory allocation and deallocation functions. To filter out pairs that can never execute concurrently, it builds the control-flow graph to check reachability for the identified pairs afterwards. To control the scheduling, it instruments the detected operations with synchronization barriers to check and enforce an execution order for the given operation pair.

During fuzzing, AutoInter-fuzzing uses the number of created threads, the number of operations executed per thread, and the number of covered operation pairs as a metric to determine the number of test cases to be created from a given seed. Whenever it encounters any operation pairs during regular fuzzing, it executes the program again and forces the execution of the opposite interleaving if it has not been tested yet. In this way and by leveraging AddressSanitizer [64], it is able to detect use-after-free, NULL-pointer dereference, and double-free vulnerabilities.

PERIOD

PERIOD [65] is one of the most recent controlled concurrency testing (CCT) approaches and is built around three components: (i) a schedule generator, (ii) a periodical executor, and (iii) a feedback analyzer.

The schedule generator systematically models program execution as a series of execution periods, forming a schedule. An execution period represents the number of key points (i.e., relevant instructions) executed by a given thread in that period. Concatenating periods results in a schedule and depicts how different threads are interleaved. Given a slice of key points and a maximum number of execution periods (starting from 2), it generates schedule patterns that order periods in lexicographical order of thread identifiers. Then, it systematically generates all schedules in order by assigning key points to the corresponding periods for each pattern.

The periodical executor takes the generated schedules and controls thread interleavings. To do so, it leverages Deadline Task Scheduling [66], a scheduling policy that assigns deadlines to tasks and picks the task with the earliest scheduling deadline to be executed next. PERIOD sets all threads to the same period length and start time, and instruments key points to halt execution of threads until the next period when necessary. In addition, it gathers information about any errors that may be triggered during execution with the help of AddressSanitizer, and the activated slice of key points, so that it is passed to the feedback analyzer.

During execution, the executed slice may be different to the one that generated the schedule (i.e., new key points are executed) because different interleavings may result in the program taking different paths. If a slice obtained that way is previously uncovered, the feedback analyzer creates a new schedule job for it. To this end, it creates a schedule prefix, a partial schedule that contains everything from the previous schedule up to the first different key point. Finally, it adds a pattern period after it for the corresponding thread.

PERIOD explores all possible schedules for the given number of periods. Then, it increments the period number to explore new schedules. This process is repeated until it reaches a preset bound.

GENMC

Verification of concurrent programs is challenging because of the huge number of interleavings of the threads comprising a concurrent program, that leads to a massive number of program states. GenMC [67] is a state-of-the-art stateless model checker for C and C++ concurrent programs built upon the LLVM toolchain. It is able to verify concurrent programs according to different memory models.

Its execution exploration algorithm leverages execution graphs where nodes represent memory access events, and edges relations among them, such as the order of the events in a thread, and write-to-read relations. From an empty graph, an interpreter executes the program and adds the next event found to the graph, while checking whether the graph is consistent with respect to the memory model. In the process of generating a full graph (i.e., a complete execution), it also finds and stores alternative exploration options such as, given a read event, the different writes to the same location from which it can read. When the found event is a write, it also considers other read events in the graph that could read from it. In this way, GenMC generates and explores every possible program execution.

Moreover, whenever a new event is added to the current graph, it also checks whether the event entails an assertion violation or an error (i.e., from the error classes that GenMC is able to detect). Thus, it detects concurrent use-after-free, double-free, and accesses to uninitialized memory.

C. STATIC ANALYSIS

Pure static analysis of concurrent programs such as standard data-flow analysis poses additional challenges due to the numerous possible thread interleavings, intensifying the usual trade-off between precision and performance. However, researchers actively devise new methods to effectively find vulnerabilities through static analysis.

CANARY

Canary [68] is a value-flow analysis framework built upon LLVM that tracks how values are stored and loaded via data and interference dependence relationships, making it possible to check diverse multi-threaded software safety properties.

In the first place, it constructs a value-flow graph (VFG) for the given program. This process is driven by an intra-thread data dependence analysis and an inter-thread interference dependence analysis. The data dependence analysis seeks to resolve the data dependence relations to represent variables and the statements that define or use them as nodes, along with the execution constraints as edges to represent value-flow relations. Then, the interference dependence analysis adds new dependence edges to the graph on pairs of load and store nodes from different threads to shared memory objects. These edges are also annotated with the constraints for the corresponding value flow.

Then, Canary checks for errors by verifying the reachability of source-sink paths in the VFG, where the source is a free statement and the sink any use statement. Specifically, it checks for value-flows that connect source to sink across different threads. To validate whether a given value-flow path conforms a feasible interleaving execution, the formula that encodes the aggregated constraints is passed on to the Z3 SMT solver. In this way, Canary is able to detect concurrency use-after-free vulnerabilities.

DCUAF

DCUAF [7] aims to detect use-after-free errors in Linux device drivers. To identify driver function pairs that may be concurrently executed, it analyzes the lock usage of each driver individually as local information, and combines the information from all drivers to perform a global statistical analysis.

During the first phase, it gathers lock-related calls in each driver source file and performs an alias analysis to check

whether two different calls have an aliased lock variable. In these cases, the callers are considered as possible concurrent functions. Then, it checks the ancestors of each candidate function pair in the call graph and filters out those that have a common ancestor to avoid potential false positives. Finally, it gets the set of driver interfaces that call each function, and computes the Cartesian product of the sets of driver interfaces, discarding pairs where both driver interfaces are the same.

In the second phase, `DCUAF` takes each concurrent interface pair from the previous phase and calculates the percentage of source files that contain these driver interfaces that have the selected concurrent interface pair. If that percentage is greater than or equal to a given threshold, the given interface pair is considered global concurrent. Then, given a pair of global concurrent interfaces in a driver, the functions linked to these interfaces are annotated as a concurrent function pair for that driver.

To find use-after-free bugs, `DCUAF` uses a summary-based *lock-set* analysis for efficiency. For each concurrent function pair, it collects the lock set of each variable access for both functions and leverages function summaries to handle called driver functions (i.e., function name, source file, accessed variables, their lock sets, code path to each access, and location). Then, it analyzes pairs of variable accesses of function pairs to verify whether the variable is the same, the lock set intersection is empty, and one of the accesses is a call to a memory freeing function. In these cases, a concurrency use-after-free is reported.

D. OTHER APPROACHES

There are other approaches also found in the literature that are related to the problem of concurrency memory corruption detection. The main reasons for not including them in the above systematization are because their core objective is different, they target non C/C++ programs, they are too specific, or reimplement and integrate existing approaches.

For instance, `EBF` [69] combines standard Bounded Model Checking (BMC) and fuzzing techniques. It implements an open-source fuzzer that implements standard techniques: random delay injection at instruction-level to explore thread interleavings and branch coverage to guide the mutation process. To find potential bugs and vulnerabilities, it uses common sanitizers such as `AddressSanitizer` and `ThreadSanitizer`. `EBF` separately performs BMC and fuzzing and aggregates the results. Also, whenever the BMC reports a failed verification outcome, it initializes the fuzzer seeds with the produced counterexample.

Another example is `COMRace` [57], which explores the Windows registry to find registered Component Object Model (COM) objects, and uses common reverse engineering methods to decompile and reconstruct virtual function tables from binary files. Then, it statically analyzes each interface method implementation to detect unsafe methods that access member fields without synchronization.

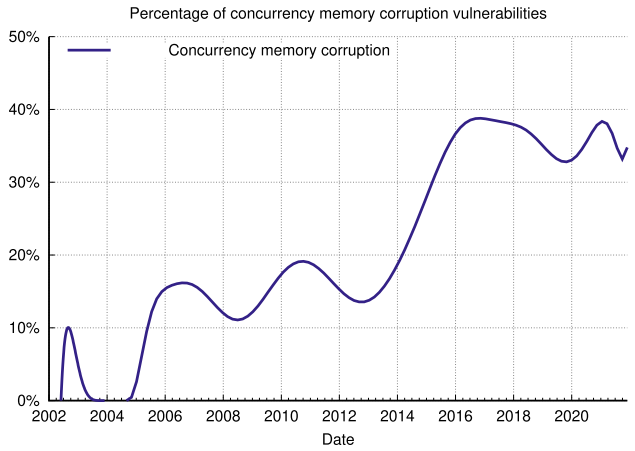

FIGURE 4. Concurrency memory corruption vulnerabilities as a percentage of total reported concurrency vulnerabilities.

VI. DISCUSSION

Concurrency not only introduces new potential security vulnerabilities but also enables traditional ones to be triggered in different ways. In previous sections, we have explored concurrency issues from bugs to vulnerabilities, with a strong focus on concurrency memory corruption vulnerabilities. Having analyzed their characteristics, their presence in real-world programs and main detection approaches, in this section we discuss the most relevant points and observations.

A. REPORTED CONCURRENCY VULNERABILITIES

Evidence from the CVE database shows that there is a low number of reported concurrency memory corruption vulnerabilities and concurrency vulnerabilities in general when compared to other more established categories (e.g., web, memory corruption). Comparing to these categories is not necessarily fair since they cover a much larger variety of vulnerabilities, are not specific to concurrent programs, and have been the focus of research and bug hunting for many years. However, the reduced amount of reported concurrency vulnerabilities can be attributed to: (i) they have received little attention in both industry and academia, (ii) concurrency vulnerabilities are harder to find and exploit since they require not only specific inputs to the program, but specific thread interleavings to be triggered, and (iii) unlike sequential detection tools, there are substantially fewer and less mature detection systems designed towards concurrency vulnerabilities. However, there has been a significant growth in reported concurrency vulnerabilities in the last few years, in line with the development of increasingly complex concurrent programs, the recent security research, and the newly proposed detection approaches.

Most of the concurrency memory corruption vulnerabilities have been reported in recent years. Specifically, 74% of them are found from 2017 onwards and approximately half off the total from 2019 to 2021, which indicates that they are an emerging threat that is also gaining attention, reflecting the number of proposed approaches in this period. Figure 4

outlines the relative number of concurrency memory corruption vulnerabilities in relation to the total number of reported concurrency vulnerabilities. As of 2021, approximately one third of the aggregated concurrency vulnerabilities actually lead to memory corruption. To illustrate, there are 85 out of 216 in 2021. Not surprisingly, we can observe that memory errors are prevalent and still a major source of vulnerabilities even in multithreaded programs.

It is worth to note that the number of reported vulnerabilities could vary to some extent due to the conservative approach we used to identify them.

With regard to CVSS scores, concurrency memory corruption vulnerabilities show rather similar values to those of traditional memory corruption, entailing a high severity threat in most cases. However, it is noteworthy that they are slightly lower in overall. This can be caused by the fact that they are harder to exploit, increasing the attack complexity and often requiring additional conditions to be met, making it necessary to repeatedly execute the exploit until it succeeds [9], [49], for example.

B. VULNERABILITY DETECTION

With regards to concurrency memory corruption detection, it is remarkable that virtually all approaches have been proposed in recent years. Given that this topic has been gaining attention lately, there are not many research works yet.

Proposed solutions use static, predictive, or testing algorithms, and are mostly focused on detecting temporal memory errors (e.g., use-after-free, double free) because they are considered to be directly caused by event orders. Therefore, their characteristics are inherently compatible with concurrency and are likely to be triggered in concurrent executions. On the contrary, spatial memory errors (e.g., buffer overflow) pose a greater challenge since there is a wider variety of operations involved and their triggering conditions are usually complex. In fact, `MUZZ` is the only approach capable of detecting new concurrency spatial memory errors, although it has difficulties to find temporal memory corruption.

In the field of concurrency issues, bug and vulnerability reproduction is also an important and challenging task due to non-determinism. Being able to consistently reproduce a vulnerability to some extent reduces false positives and greatly helps developers to understand and fix it properly. Approaches built upon fuzzing techniques already provide the program input that crashed the application in the generated report. However, verifying vulnerabilities reported by tools based on predictive algorithms poses a major challenge. In that context, `ConVul` is the only predictive approach that tries to actually trigger vulnerabilities by manipulating thread scheduling to execute the harmful execution order.

There is no single approach that performs absolutely better than the rest, but each one comes with its own strengths and limitations. For instance, `ConAFL` is limited by its thread-aware static analysis because it does not scale to large programs (i.e., with more than 10K lines of code). Similarly, its fuzzer component does not explore new paths or thread interleavings, it tries to enforce the vulnerable order for a given pair of sensitive operations, limiting the findings to the results of the static analysis.

`UFO` relies on a constraint solver to determine the feasibility of different schedulings. Since it usually generates long traces with a high number of constraints, it has to trade detection capability for efficiency in order to solve the constraints in reasonable time. Similar to other predictive approaches, it may miss vulnerabilities that would materialize in a different execution trace given a different program input.

`ConVul` leverages the *happens-before relation* to identify exchangeable events. It performs practical prediction and practical validation methods, which may result in missing potential vulnerabilities. Further, it uses a heuristic approach to predict exchangeable events that limits the *synchronization distance* to 3, missing potential exchangeable events with longer distances, and thus missing potential vulnerabilities.

`ConVulPOE` constructs feasible traces that expose vulnerabilities from partial order graphs. However, it cannot construct a trace when there are unordered conflicting event pairs that cannot be resolved, missing potential vulnerabilities. Moreover, if it is not able to identify pairs of events in the trace according to its definitions of vulnerability-potential event pairs, it does not detect vulnerabilities in that execution. Further, if the target program crashes, it does not detect any vulnerabilities even if it was caused by a vulnerability.

Prior to fuzzing, `MUZZ` performs static analysis to extract a suspicious interleaving scope to later focus coverage-oriented instrumentation on it. However, it does not include lock-protected operations, which have been proven to lead to concurrency vulnerabilities [10], [61]. Therefore, it may intensely exclude statements that involve interesting interleavings. Moreover, controlled concurrency testing approaches such as `PERIOD` do not deal with program inputs, i.e., do not generate or try to discover new inputs but rely on a predefined set. Hence, they potentially benefit from other methods that seek to find new test cases, such as fuzzing.

In general, many approaches rely on points-to static analysis, which is associated to high false positive rates as precise and concrete control-flow and data-flow information can only be known at runtime. In addition, it also requires precise concurrency information, increasing the complexity of the analyses.

To sum up, temporal memory errors have been the focus of most existing concurrency vulnerability detectors due to their close alignment with concurrency, whereas spatial errors seem to remain as a greater challenge. Among the proposed tools, those based on predictive analysis showed a better performance in detecting temporal errors, whereas fuzzing performed better for spatial memory corruption. There is room for further exploration in both cases to develop new approaches to effectively detect the already addressed error classes, or to target those that have received less attention.

VII. RELATED WORK

There are no existing works that explicitly focus on measuring and analyzing concurrency vulnerabilities and how they are detected. However, other measurement papers are closely related to individual parts of this paper.

Concurrency bugs have been a topic of research for several years [2], [17], [28]. Researchers have previously studied concurrency bug patterns, their manifestation conditions, and fix strategies [1]. From the analyzed errors, they found that around one third were caused by different execution orders from those intended by the developers, and that most concurrency bugs can be reliably triggered, among other findings.

Given that memory corruption is still one of the major concerns in the security community, researchers have thoroughly analyzed how it is used to launch several attacks as well as the main mitigation and protection strategies [6], also from a binary-only perspective [70]. Part of these approaches are also applicable or could be adapted to the concurrent domain.

Fuzzing has been shown to be very effective in finding vulnerabilities even in multithreaded programs, and thus several different algorithms that improve or specialize this process have been proposed in recent years [12], [46], [62]. In consequence, researchers have analyzed the different techniques and designs implemented by fuzzers to build a taxonomy [71], and examined experimental methodologies in the literature to define a proper evaluation methodology to be carried out by fuzzers [72]. Concurrency vulnerability detection will also benefit from advances in fuzzing.

VIII. CONCLUSION

To take advantage of the increased processing capacity of modern processors, concurrent programs have become increasingly pervasive. Unfortunately, concurrency introduces new problems in the form of bugs and security vulnerabilities. Furthermore, it also enables traditional vulnerabilities such as memory corruption to be triggered in alternative ways specific to concurrent executions.

In this paper we have reviewed the particular properties of concurrency bugs, how they are commonly classified according to their characteristics, and the main detection algorithms and tools proposed in the literature. In addition, we have pointed out how concurrency bugs and vulnerabilities are related but not required for each other to take place. We have also analyzed the presence and trends of reported concurrency and concurrency memory corruption vulnerabilities in the CVE database, and showed that they are an emerging threat. Lastly, we have systematically surveyed, classified, and discussed the most relevant detection approaches that target concurrency memory corruption vulnerabilities.

We hope that this paper will draw attention towards concurrency vulnerabilities and serve as a basis for future research to explore new detection methods and perspectives, as well as further security issues.

REFERENCES

[1] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proc. 13th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2008, pp. 329–339.
[2] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection," in *Proc. 30th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2009, pp. 121–133.
[3] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye, "Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing," in *Proc. 27th ACM Symp. Oper. Syst. Princ. (SOSP)*, Oct. 2019, pp. 162–180.
[4] X. Cai, Y. Gui, and R. Johnson, "Exploiting unix file-system races via algorithmic complexity attacks," in *Proc. 30th IEEE Symp. Secur. Privacy (SP)*, May 2009, pp. 27–41.
[5] *NVD—CVE-2010-5298*. Accessed: Feb. 27, 2022. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2010-5298
[6] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2013, pp. 48–62.
[7] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu, "Effective static analysis of concurrency use-after-free bugs in Linux device drivers," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2019, pp. 255–268.
[8] W. Zhang, C. Sun, and S. Lu, "ConMem: Detecting severe concurrency bugs through an effect-oriented approach," in *Proc. 15th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2010, pp. 179–192.
[9] S. Zhao, R. Gu, H. Qiu, T. O. Li, Y. Wang, H. Cui, and J. Yang, "OWL: Understanding and detecting concurrency attacks," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2018, pp. 219–230.
[10] Y. Cai, B. Zhu, R. Meng, H. Yun, L. He, P. Su, and B. Liang, "Detecting concurrency memory corruption vulnerabilities," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, Aug. 2019, pp. 706–717.
[11] C. Liu, D. Zou, P. Luo, B. B. Zhu, and H. Jin, "A heuristic framework to detect concurrency vulnerabilities," in *Proc. 34th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, Dec. 2018, pp. 529–541.
[12] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu, "MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs," in *Proc. USENIX Secur. Symp. (USENIX Secur.)*, 2020, pp. 2325–2342.
[13] J. Huang, "UFO: Predictive concurrency use-after-free detection," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng. (ICSE)*, Mar./Jun. 2018, pp. 609–619.
[14] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: Fault localization in concurrent programs," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng. (ICSE)*, May 2010, pp. 245–254.
[15] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov. 1997.
[16] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: Detecting atomicity violations via access interleaving invariants," in *Proc. 12th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2006, pp. 37–48.
[17] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing atomicity violation bugs from their hiding places," in *Proc. 14th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2009, pp. 25–36.
[18] H. Jula, D. M. Tralamazza, C. Zamfir, and G. Candea, "Deadlock immunity: Enabling systems to defend against deadlocks," in *Proc. 8th USENIX Conf. Oper. Syst. Design Implement. (OSDI)*, 2008, pp. 295–308.
[19] Y. Cai and W. K. Chan, "MagicFuzzer: Scalable deadlock detection for large-scale applications," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 606–616.
[20] Y. Cai, H. Yun, J. Wang, L. Qiao, and J. Palsberg, "Sound and efficient concurrency bug prediction," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, Aug. 2021, pp. 255–267.
[21] P. Pratikakis, J. S. Foster, and M. Hicks, "LOCKSMITH: Context-sensitive correlation analysis for race detection," in *Proc. 27th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2006, pp. 320–331.
[22] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data race detection in practice," in *Proc. Workshop Binary Instrum. Appl. (WBIA)*, 2009, pp. 62–71.
[23] J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Jun. 2014, pp. 337–348.

[24] D. Engler and K. Ashcraft, "RacerX: Effective, static detection of race conditions and deadlocks," in *Proc. 19th ACM Symp. Oper. Syst. Princ. (SOSP)*, 2003, pp. 237–252.

[25] M. Said, C. Wang, Z. Yang, and K. Sakallah, "Generating data race witnesses by an SMT-based analysis," in *Proc. NASA Formal Methods Symp. (NFM)*, 2011, pp. 313–327.

[26] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: A race and transaction-aware Java runtime," in *Proc. 28th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Jun. 2007, pp. 245–255.

[27] M. D. Bond, K. E. Coons, and K. S. McKinley, "PACER: Proportional detection of data races," in *Proc. 31st ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2010, pp. 255–268.

[28] U. Mathur, D. Kini, and M. Viswanathan, "What happens-after the first race? Enhancing the predictive power of happens-before based dynamic race detection," in *Proc. ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang., Appl. (OOPSLA)*, 2018, pp. 1–29.

[29] D. Kini, U. Mathur, and M. Viswanathan, "Dynamic race prediction in linear time," in *Proc. 38th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Jun. 2017, pp. 157–170.

[30] J. Roemer, K. Genç, and M. D. Bond, "High-coverage, unbounded sound predictive race detection," in *Proc. 39th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Jun. 2018, pp. 374–389.

[31] *Helgrind: A Thread Error Detector*. Accessed: Mar. 1, 2022. [Online]. Available: https://valgrind.org/docs/manual/hg-manual.html

[32] S. Nagarakatte, S. Burckhardt, M. M. K. Martin, and M. Musuvathi, "Multicore acceleration of priority-based schedulers for concurrency bug detection," in *Proc. 33rd ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Jun. 2012, pp. 543–554.

[33] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, "Context-sensitive and directional concurrency fuzzing for data-race detection," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2022.

[34] U. Agarwal, P. Deligiannis, C. Huang, K. Jung, A. Lal, I. Naseer, M. Parkinson, A. Thangamani, J. Vedurada, and Y. Xiao, "Nekara: Generalized concurrency testing," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2021, pp. 679–691.

[35] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, J. Van Geffen, and A. Warfield, "Using lightweight formal methods to validate a key-value storage node in Amazon S3," in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ. CD-ROM*, Oct. 2021, pp. 836–850.

[36] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, May 2003, pp. 338–349.

[37] C. Flanagan and S. N. Freund, "Atomizer: A dynamic atomicity checker for multithreaded programs," in *Proc. 31st ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, 2004, pp. 256–267.

[38] C. Flanagan, S. N. Freund, and J. Yi, "Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs," in *Proc. 29th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2008, pp. 293–303.

[39] U. Mathur and M. Viswanathan, "Atomicity checking in linear time using vector clocks," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, Mar. 2020, pp. 183–199.

[40] Z. Lai, S.-C. Cheung, and W. K. Chan, "Detecting atomic-set serializability violations in multithreaded programs through active randomized testing," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng. (ICSE)*, May 2010, pp. 235–244.

[41] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps, "ConSeq: Detecting concurrency bugs through sequential errors," in *Proc. 16th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2011, pp. 251–264.

[42] X. Fu, D. Lee, and C. Jung, "NAdroid: Statically detecting ordering violations in Android applications," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, Feb. 2018, pp. 62–74.

[43] D. Wu, J. Liu, Y. Sui, S. Chen, and J. Xue, "Precise static happens-before analysis for detecting UAF order violations in android," in *Proc. 12th IEEE Conf. Softw. Test., Validation Verification (ICST)*, Apr. 2019, pp. 276–287.

[44] T. Kim, V. Kumar, J. Rhee, J. Chen, K. Kim, C. H. Kim, D. Xu, and D. J. Tian, "PASAN: Detecting peripheral access concurrency bugs within bare-metal embedded applications," in *Proc. USENIX Secur. Symp. (USENIX Secur.)*, 2021, pp. 249–266.

[45] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 754–768.

[46] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1643–1660.

[47] S. Gong, D. Altinbüken, P. Fonseca, and P. Maniatis, "Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis," in *Proc. 28th ACM Symp. Operating Syst. Princ. (SOSP)*, 2021, pp. 66–83.

[48] Google. *Syzkaller—Kernel Fuzzer*. Accessed: Mar. 1, 2022. [Online]. Available: https://github.com/google/syzkaller

[49] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan, "Concurrency attacks," in *Proc. s4th USENIX Workshop Hot Topics Parallelism (HotPar)*, 2012, p. 15.

[50] 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. USENIX Secur. Symp. (USENIX Secur.)*, 2017, pp. 1–16.

[51] M. Wu and C. Wang, "Abstract interpretation under speculative execution," in *Proc. 40th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Jun. 2019, pp. 802–815.

[52] *NVD—CVE-2021-3348*. Accessed: Mar. 3, 2022. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-3348

[53] *NVD—CVE-2019-11366*. Accessed: Mar. 3, 2022. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-11366

[54] Lefebvre, Jean-Pierre. *Atftp*. Accessed: Mar. 3, 2022. [Online]. Available: https://sourceforge.net/projects/atftp/

[55] *NVD—Data Feeds*. Accessed: Mar. 3, 2022. [Online]. Available: https://nvd.nist.gov/vuln/data-feeds

[56] Y. Lee, C. Min, and B. Lee, "ExpRace: Exploiting kernel races through raising interrupts," in *Proc. USENIX Secur. Symp. (USENIX Secur.)*, 2021, pp. 2363–2380.

[57] F. Gu, Q. Guo, L. Li, Z. Peng, W. Lin, X. Yang, and X. Gong, "COMRace: Detecting data race vulnerabilities in COM objects," in *Proc. USENIX Secur. Symp. (USENIX Secur.)*, 2022, pp. 3019–3036.

[58] A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino, "Predicting null-pointer dereferences in concurrent programs," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng. (FSE)*, Nov. 2012, pp. 1–11.

[59] L. D. Moura and N. Bjørner, "z3: An efficient SMT solver," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.*, 2008, pp. 337–340.

[60] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Jun. 2005, pp. 190–200.

[61] K. Yu, C. Wang, Y. Cai, X. Luo, and Z. Yang, "Detecting concurrency vulnerabilities based on partial orders of memory and thread events," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, Aug. 2021, pp. 280–291.

[62] M. Zalewski. *American Fuzzy Lop*. Accessed: Mar. 1, 2022. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[63] Y. Ko, B. Zhu, and J. Kim, "Fuzzing with automatically controlled interleavings to detect concurrency bugs," *J. Syst. Softw.*, vol. 191, Sep. 2022, Art. no. 111379.

[64] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2012, pp. 309–319.

[65] C. Wen, M. He, B. Wu, Z. Xu, and S. Qin, "Controlled concurrency testing via periodical scheduling," in *Proc. 44th Int. Conf. Softw. Eng. (ICSE)*, May 2022, pp. 474–486.

[66] The Linux Kernel User's and Administrator's Guide. *Deadline Task Scheduling*. Accessed: Feb. 14, 2023. [Online]. Available: https://www.kernel.org/doc/html/latest/scheduler/sched-deadline.html

[67] M. Kokologiannakis and V. Vafeiadis, "GenMC: A model checker for weak memory models," in *Proc. 33rd Int. Conf. Comput.-Aided Verification (CAV)*, 2021, pp. 427–440.

[68] Y. Cai, P. Yao, and C. Zhang, "Canary: Practical static detection of inter-thread value-flow bugs," in *Proc. 42nd ACM SIGPLAN Int. Conf. Program. Lang. Design Implement. (PLDI)*, Jun. 2021, pp. 1126–1140.

[69] F. K. Aljaafari, R. Menezes, E. Manino, F. Shmarov, M. A. Mustafa, and L. C. Cordeiro, "Combining BMC and fuzzing techniques for finding software vulnerabilities in concurrent programs," *IEEE Access*, vol. 10, pp. 121365–121384, 2022.

[70] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (State of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 138–157.

[71] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2312–2331, Nov. 2021.
[72] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, Oct. 2018, pp. 2123–2138.

OSCAR LLORENTE-VAZQUEZ received the B.Sc. degree in computer engineering from the University of Deusto, Bilbao, Spain, in 2016, and the M.Sc. degree in information and communications security from Alfonso X El Sabio University, Madrid, Spain, in 2017. He is currently pursuing the Ph.D. degree in computer science in the field of system security with the University of Deusto. From 2015 to 2017, he was a Research Intern with the University of Deusto, where he has been a Research Assistant, since 2017. His research interests include software and system security, vulnerability analysis, and the development of detection and mitigation methods for software vulnerabilities.

IGOR SANTOS-GRUEIRO received the Ph.D. degree in computer science from the University of Deusto, Bilbao, Spain, in 2011. From 2011 to 2020, he was an Assistant Professor and an Associate Research Professor with the University of Deusto, leading the Information and Systems Security Research Group. Since 2020, he has been a Lecturer and a Researcher with the Cybersecurity Research Group, Mondragon University. He is currently a Security Research Scientist with HP Labs, conducting industrial cybersecurity research. His research interests include the area of systems security and, in particular, the areas of program analysis, including malware analysis and operating systems security; web security, including online malicious content detection and online privacy; and network analysis.

PABLO GARCIA BRINGAS received the Ph.D. degree in computer science, specializing in the application of artificial intelligence to the field of security, from the University of Deusto, Bilbao, Spain, in 2007. He has been the Founder and the Director of the Digital Industry Chair, University of Deusto. He has worked decisively in the constitution of the DeustoTech Technology Center, in which he has held various positions of responsibility, including general management. He has also directed the Deusto Master's Degree in information security for 11 years. He is also directing the new Executive Program in Industry 4.0. He is currently a Full Professor of engineering with the University of Deusto, where he leads the nationally recognized Deusto for Knowledge—D4K Research Group. He is also the Vice-Dean of external relations, continuous education, and research with the Faculty of Engineering, University of Deusto.