OS-SANITIZER: System-wide Latent Defect Inference in Linux Applications

Addison Crump
addison.crump@cispa.de
CISPA

Sahil Sihag
sahil.sihag@cispa.de
CISPA

Florian Bauckholt
florian.bauckholt@cispa.de
CISPA

Keno Hassler
keno.hassler@cispa.de
CISPA

Thorsten Holz
thorsten.holz@mpi-sp.org
MPI-SP

Abstract

Dynamic testing has historically focused on finding situations in which software does something unwanted, typically by triggering failure or undesirable states. However, such testing is often limited to finding these scenarios by example. Can we determine that software *could* do something unwanted by inspecting *benign behavior*? In this paper, we explore this question by leveraging eBPF for *dynamic defect inference* in Linux applications. eBPF is uniquely positioned as a system introspection tool that accrues data from both user- and kernelspace events and processes them with programs in the kernel. Our prototype, OS-SANITIZER, implements such eBPF programs using heuristics which report the suspected presence of defects in *all* applications across the entire system. Conceptually, OS-SANITIZER brings the idea of code smells from static testing into dynamic testing, while simultaneously profiting from the insights of runtime events. In doing so, we infer the presence of latent *contextual* defects in software that would only induce a failure in certain environments or are otherwise difficult to test for. We consider and evaluate the strengths and weaknesses of this approach from the perspectives of performance, complexity, maintainability, and usage, differentiating the theoretical limits of eBPF versus the specific limits of our prototype. Targeting well-known types of software defects, we were able to identify more than 40 issues (including severe vulnerabilities) in widely used applications, some of which are older than a decade and present on a majority of Linux distributions. Our findings demonstrate that dynamic defect inference is both feasible and effective, highlighting opportunities for expanding this underexplored direction in software testing.

1 Introduction

All software developers are in some way familiar with dynamic testing [15]. If you want to make sure that your program works as intended, the easiest way to do so is to run it and see if it fails to do what you expect. More formally: you *hypothesize* about a particular way your program behaves, then you evaluate whether that hypothesis is *falsified* by experimenting with a test. This is the predominant way in which software is evaluated [3] both because it is simple to implement and because it is concrete: when a test failure occurs, something is definitely wrong.

We may contrast this mode of testing with typical static analysis tools [76] which look for programmer errors. Some of these are concrete (i.e., syntax or type errors), but static analysis tools are often not looking to disprove that the program behaves correctly. Instead, they *infer* that the program may not behave correctly under unknown circumstances. Static analysis tools do so by collecting context surrounding a code region, then applying a heuristic to determine from that context whether a typical error pattern is observed [73]. In this way, they can identify what code regions *may* contain defects or design issues without a corresponding testcase that produces some counterexample to expected behavior.

In principle, there is no restriction from applying the same strategy to dynamic testing. Instead of inspecting the source code itself, we may collect context from the events observed during runtime, and use these to infer the presence of a defect absent of a failure. Historically, most dynamic tooling for detecting defects has focused on detecting *faults* (that is, observed incorrect behavior) [7, 10, 55, 57, 80]. Few have inferred defects from benign behavior, and those that have are limited, focusing on only a few specific problems like specific issues in design [51, 78] or filesystem interaction [104]. It is in this gap that we find potential: we need techniques with which human investigators may look for likely defective or otherwise suspicious code regions that have not yet been observed to produce a fault, but may yet do so. We call this process *dynamic defect inference*, and recognize this as an understudied strategy for dynamic testing.

In this paper, we explore how eBPF, a monitoring subsystem of Linux, can be leveraged as an analysis tool to dynamically infer the presence of defects in Linux applications. eBPF enables fine-grained system introspection [21, 54] by collecting and processing runtime events across both user

and kernel space. eBPF has been widely adopted for system observability, with most prior work focusing on network monitoring [64, 101, 107], microservice and containerized environments [64, 105], and application-level management and sandboxing [22, 47], but never for testing.

We identify a set of runtime heuristics for common program defects and show how eBPF can be used to analyze events indicative of these defects. We then encode these heuristics into our prototype, OS-SANITIZER, which we use to evaluate eBPF and dynamic defect inference in general. By extending the notion of *code smells* [25] from static analysis to runtime observation, OS-SANITIZER uncovers latent, *context-dependent* issues in design that result in security weaknesses and code defects.

We evaluated OS-SANITIZER on several benchmarks, performed a reproduction study, and used the prototype long-term. We find that eBPF is sufficiently performant to be passively running long-term even on actively used devices. During our development and evaluation, OS-SANITIZER directly uncovered 12 defects in widely-used software projects like Golang, Docker, and Firefox that weaken security mitigations or are overtly vulnerable. By investigating additional code smells reported by OS-SANITIZER, we discovered more than 30 further defects which affect both security and correctness, highlighting that our strategy exposes long-latent issues even in the most widely used software.

Contributions. In summary, our key contributions are:

1) Prototype implementation: We design and implement OS-SANITIZER to demonstrate how eBPF programs can be leveraged for dynamic defect inference.

2) Comprehensive evaluation: We systematically assess the feasibility and practicality of using eBPF for defect inference by analyzing performance, complexity, maintainability, and usage.

3) Case studies and ecosystem implications: We present several real-world case studies that highlight the gap that dynamic defect inference addresses and discuss their implications on the software ecosystem at large.

2 Motivating Example

Consider the small Go program in Listing 1. Is this program vulnerable? Based purely on the program presented, the answer is likely "no"; this program simply deletes a specified path. Indeed, this is no different to invoking `rm -rf`.

There are other problems, though; if you compiled this program at any point before May 2024, this program would invoke a different implementation of Go's `os.RemoveAll` function. This older implementation, shown in Listing 2, first checks if a specified path is a directory, then opens it unconditionally. This code is not incorrect, but if the file

```go
package main
import "os"
func main() {
  if err := os.RemoveAll(os.Args[1]); err != nil {
    os.Exit(1)
  }
}
```

Listing 1: Go snippet which invokes `os.RemoveAll`.

```go
// Is this a directory we need to recurse into?
var statInfo syscall.Stat_t
statErr := ignoringEINTR(func() error {
  return unix.Fstatat(parentFd, base, &statInfo,
                      unix.AT_SYMLINK_NOFOLLOW)
})
/* ... */
if statInfo.Mode&syscall.S_IFMT
     != syscall.S_IFDIR {
  // Not a directory
  return &PathError{Op: "unlinkat", Path: base,
                    Err: err}
}
/* ... */
// Open the directory to recurse into
file, err := openFdAt(parentFd, base)
```

Listing 2: Snippet of old `os.RemoveAll` implementation [38].

at that position changes from a directory to a symbolic link between the check and the open, then the symbolic link will be opened and traversed.

Scenario 1: Suppose that a user Alice, acting as root, wrote, compiled, and uses this program to delete directories. Alice only ever uses this program on file trees entirely owned by root and never modifies those file trees concurrently. This program may not be written according to best practices, but it is safe; the context it is executed in never exposes it to the time-of-check to time-of-use (TOCTOU) issue above.

Scenario 2: Consider the filesystem diagram of Figure 1. Suppose that Alice now uses the same program to delete the directory `A`. User Bob has the ability to modify `B`. Because of the TOCTOU presented in Scenario 1, Bob may surreptitiously replace `C` to redirect the deletion.

Scenario 3: Consider again Figure 1, but now suppose that Alice attempts to delete the folder `A/B/C/E`. This path gets resolved to `A/D/E` via `C`, a symbolic link pointing to `D`. While `E` is only modifiable by Alice, the traversed file tree under `B` is owned by another user, Bob. If Bob replaces `B` or `C` before the program is executed, he can once again redirect the deletion intended to another directory of Bob's choice, regardless of whether there is a TOCTOU present.

Scenario 4: Finally, consider the opposite: Bob wrote and compiled the program in Listing 1, and Alice executes this binary as root. Because Bob owns the executable, he may surreptitiously replace this with any other executable, thereby causing Bob to execute whatever he wants as root. This is no

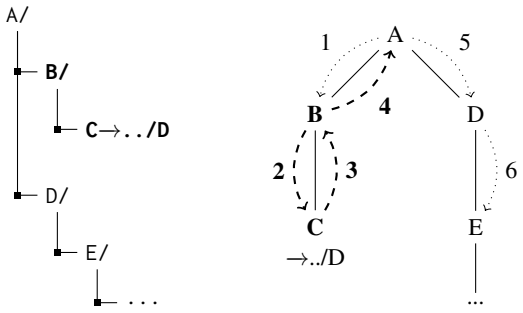

Figure 1: Example directory structure and traversal diagram resolving a path. Traversal steps are indicated with dotted lines, numbered by order of traversal. Attacker-controlled components are indicated with **bold** text and dashed lines.

longer an issue with the original program, but instead reveals something else: a usage error which induces vulnerability.

Together, these scenarios show how not following best practices can lead to vulnerability when a program is executed in a new context. Developers rarely know (or even define) exactly how their programs will be used. As a result, even seemingly benign violations of best practices can become serious vulnerabilities downstream. Indeed, this issue with `os.RemoveAll` was one of the issues found by OS-SANITIZER, and downstream users of this function were consequently affected by scenarios 1–3 in real settings. Our work defines a framework and a selection of pre-defined heuristics for detecting such issues: the contextually vulnerable, like those presented in scenarios 1 and 2, and subtle misusage and misconfiguration, like scenarios 3 and 4.

3 Background and Related Work

We begin by defining the software testing terminology used throughout this paper to ensure that there is no ambiguity in what we propose and what we achieve. Then, we provide an overview of existing work focusing primarily on methods and tools which detect defects in subjects that we target in our own work: userspace Linux applications. Finally, we review how eBPF is used in existing literature and discuss its features most relevant to our work.

3.1 Terminology

Software testing has numerous taxonomies [2, 8, 60] which sometimes conflict or have inconsistent definitions. For this paper, we discuss five classes of software issues: *errors*, *code smells*, *defects*, *faults*, and *failures*. An *error* is a mistake on the part of a developer or a user. Errors often manifest as a *defect*, or an issue in source code which leads to undesired behavior in the program. This undesired behavior is called a *fault* and can manifest either functionally (i.e., affecting the correctness of the routine) or non-functionally (not affecting correctness, but some other quality like performance). Just because a fault occurs does not mean that the program immediately terminates. In many cases, programs are surprisingly resilient to faults emerging. Instead, a fault becomes problematic when it manifests as a *failure*, either causing the program to reach an invalid state or violating a guarantee of the program.

Another common result of an error is a *code smell*, which is a design issue that typically denotes a weakness in the maintainability or design of a program. This term originates from Fowler and Beck's seminal work on the refactoring of object-oriented programs [25], in which it describes patterns that worsen the quality of code. In the years since this term was coined, it has largely expanded as a colloquialism for "benign" violations of best practices which may affect future usage and maintenance of the software; this is the meaning which we intend for this paper.

3.2 Related Work in Dynamic Defect Inference

Due to a lack of consistent language and distinction between traditional dynamic testing techniques, it is challenging to identify prior work which employs dynamic defect inference. Moreover, many of these techniques were primarily designed for mitigation, making the line between testing and mitigation blurry. Nevertheless, we survey a broad range of techniques which improve a program's ability to detect a defect, both with and without corresponding faults.

3.2.1 Program-Level Inference

Compiler sanitizers are used in myriad automated dynamic testing strategies, most often in low-level programming languages which involve error-prone resource management like C, C++, and Rust [89]. Sanitizers like Address Sanitizer [55] and Memory Sanitizer [57] (both included in LLVM) report observed memory access faults by instrumenting allocation routines and memory access operations. Similar work has been done to identify the same faults in the kernel [95] or programs where source code is not available [23]. These sanitizers are powerful, but require recompilation and often induce performance impacts that are only permissible in testing environments. Control Flow Integrity (CFI) [1, 56] terminates a program when unexpected control flow occurs (e.g., incorrect function arguments or call/return destinations), and is more widely used as it exhibits acceptable performance penalties as a mitigation technique [44]. FORTIFY [7] is a widely deployed dependency-introduced set of extra checks calls to C routines to mitigate common programming errors with effectively no performance penalty. Other simple mitigations, such as stack canaries [16], thus similarly may be considered as defect inference techniques. In the same sense, one might even argue that developer-inserted assertions are

themselves an inference technique, though certainly not an automated one.

Others look for behavior which is deemed *most likely* incorrect. Rust offers many language support features in this vein; in order of ascending performance impact, debug-assertions [82] inserts additional checks for integer overflows and other dubious behavior, cargo-careful [48] is a compiler wrapper which introduces extra checks for certain memory guarantees, and MIRI [83] runs Rust programs in an entirely different execution environment to certify the correctness of the memory model absolutely. LLVM similarly offers Undefined Behavior Sanitizer [59], which checks for situations in which the program enters a state which is undefined by the compiler.

A final group looks for behaviors which are related to defective code, but are instead indicative of poor design or suspicious behavior (a *code smell*). This category is relatively underexplored in the literature, but there are two works relevant to ours. DYNAMICS [78] dynamically identifies code smells in Android applications based on behavioral specifications, augmenting static analysis strategies with data collected at runtime to improve the precision of detection. Kumar and Chhabra [51] use a filtering and analysis strategy to identify code which exhibits the "feature envy" object-oriented code smell.

3.2.2 Interprocedural Inference

Sanitizers do not strictly refer to strategies which are compiler-based; others work by tampering with dependencies or intercepting information flow between programs. SYSTEMSAN [10] is a sanitizer which monitors when `exec()`-like calls are corrupted by attacker-controlled text, a strong indicator of command injection. Other works have applied the same concept to other forms of injection [99]. We highlight these separately as they are no longer bound to a particular programming language or level of compile-time abstraction; these instead monitor the flow of information between programs through standardized mechanisms, such as system and library calls or network protocols.

3.2.3 OS-Integrated Inference

The primary purpose of hardware or the operating system that runs on it is to allow programs to perform some desired task, and not to test those programs' correctness. As such, most such tools which infer the presence of defects at a system level are mitigatory or defensive. SELINUX [80] is a broad suite of additional checks and security controls which allows system administrators to control what behaviors programs may exhibit much more precisely, though typically for mitigating attacks. Similarly, seccomp [54] is a set of security controls for restricting program behavior, either in a rule-based pattern or by using small programs targeting the BPF instruction set architecture (ISA) to perform specialized matching on program behavior (notably separately from the eBPF subsystem itself). There are a few works which extend the Linux kernel via kernel modules or Linux Security Modules [96] to accomplish some testing goal [102, 104], but these are mostly unused in practice as they are difficult to deploy and maintain. A wide variety of security features outside of those mentioned above have been implemented for Linux, but these are almost all associated with defense, not testing goals [53, 79], or are otherwise associated with kernel security. Then, of course, there are numerous works using eBPF for security purposes which we discuss next in Section 3.3.

3.2.4 Runtime Predictive Analysis

In the field of runtime predictive analysis, the goal is to detect potential defects that have not actually manifested as faults based on observations of the program's behavior (e.g., memory access patterns). For example, jPredictor [11] can detect concurrency bugs in Java programs by analyzing execution traces. UFO [45] specializes on the detection of hard-to-observe use-after-free bugs in concurrent programs. Similar to other LLVM-based sanitizers, Thread Sanitizer [58] can be inserted into programs written in compiled languages to detect potential concurrency issues with moderate but permissible overhead. OS-SANITIZER shares the notion of predicting faulty behavior from benign executions of a program, but generalizes to different bug classes and takes a system-level perspective.

3.3 eBPF

eBPF itself is, first and foremost, an ISA with a very limited number of operations, which can be targeted by several widely-used compilers like GCC and LLVM [12, 26]. When we refer to eBPF in this paper, we refer *specifically* to the eBPF subsystem of Linux accessed by the `bpf()` system call [54]. This subsystem, as described by the Linux kernel BPF group, allows developers to "dynamically program the kernel for efficient networking, observability, tracing, and security" [21]. More concretely, this subsystem allows for version-agnostic loading of small programs into the kernel at predefined *attachment points*, where the kernel will execute these programs when the given attached point is reached. These programs may internally process information about the event by inspecting the provided context, reading kernel and user memory, querying process state, or manipulating persistent data sandboxed to eBPF, and may emit information to cooperative userspace processes for reporting [12].

eBPF's design affords extensive observability into the system's state and behavior. This capability has been used with great success in a variety of applications in previous work. Historically, eBPF has focused on network operation monitoring and manipulation; as such, most previous works use

eBPF to implement network monitoring, acceleration, or other related applications [5, 63, 64, 87, 98, 101, 107]. Beyond networking, eBPF is often considered for the monitoring of *microservices*, which is a strategy for deployment of services as smaller units which individually scale, typically used within the context of cloud computing [52, 64, 105]. eBPF is growing as an industry standard for this purpose [65], as microservices are often implemented within *containers*, an isolation strategy that separates programs and resources while retaining the same (eBPF-instrumented!) kernel. Finally, eBPF is employed for monitoring, management, and sandboxing of user applications [19, 22, 47].

Beyond its usage, eBPF itself is also a common subject of *inspection* by academic works. The eBPF verifier serves as the first line of defense to ensure that programs loaded into the kernel do not corrupt kernel data or induce undesired behavior. To maintain performance, eBPF programs are often Just-in-Time (JIT) compiled within the kernel [12]. As a result, not only is the correctness of verification critical, but miscompilation thereafter may itself induce unintended behavior into the kernel. These two aspects of eBPF have been frequently studied in academic work, both due to impact and the inherent interestingness of the problems therein from a software testing and verification perspective [68, 72, 90, 103].

In this paper, we principally utilize three attachment types of eBPF: uprobes and uretprobes, fprobes and fretprobes, and LSM programs. These represent the attachment of eBPF programs at the entry and exit of userspace functions and kernelspace functions, and the implementation of Linux Security Modules (LSMs) [96], respectively. u(ret)probes insert breakpoints into all memory mappings of a given userspace library function's entry or exit, which induces a context switch into the eBPF program in kernelspace. These probes are given the context of the function, including arguments and limited access to the active memory of the process, among other details. f(ret)probes do the same for kernel functions, but with "trampolines" which redirect control flow to the corresponding eBPF programs before restoring to the original function. When these kernel functions are called via system calls by processes, they can provide greater context about the program's execution than the program itself. Finally, LSM eBPF programs allow eBPF programs to act as security modules, which allow eBPF programs to control whether certain security-relevant operations are permitted. These attachment points offer special access to program information, allowing for even greater context.

eBPF programs execute concurrently, atomically, and in isolation, so even though these different attachment types offer different contexts, one might imagine they cannot work in tandem. eBPF maps [94] allow for data to flow between not only executions of the same program, but of different programs and attachment types. In this way, one may accrue context at different points in order to achieve some combined goal. This is the principle around which OS-SANITIZER is designed, as we explain further in the next section.

4 Defect Inference with eBPF

Our goal in this work is to identify the capacity of eBPF to infer the presence of defects in userspace Linux applications. eBPF is not unique to Linux [32] or the eBPF subsystems implemented in Linux already [97], nor is it technically limited to userspace defects. We choose userspace Linux applications as our principal subject because they are widely used, widely studied in the literature, and are simple to implement, debug, and inspect. We start by illustrating a concrete set of eBPF programs (which we collectively term as a "pass") designed to detect potentially unsound file handling. Afterwards, we generalize to all passes we implemented and classify the defects they aim to detect. Lastly, we compare eBPF's theoretical limits to that of other tools.

4.1 Example Pass: `intercept_path`

Consider again Scenarios 2 and 3 from Section 2. It would not be sufficient to simply check if the target file has the correct permissions; intermediary path components may also be intercepted. Checking these individually would require checking each path component separately, which is race-prone and expensive. In order to detect a vulnerable path traversal at runtime, OS-SANITIZER attaches six eBPF programs, collectively referred to as the `intercept_path` pass, to different kernel functions responsible for resolving paths and checking file permissions on the way. This is illustrated in Figure 2.

The first program, `filp_open_fentry`, attaches to the start of the `do_filp_open` kernel function and records the userspace pointer referring to the requested path. A corresponding cleanup program, `filp_open_fexit` (bottom left in Figure 2), cleans this up at the end by attaching to the return of the same kernel function. The second program, `openat_fentry`, attaches to the start of `path_openat`, which is invoked by every open-like operation just before the path traversal actually begins. This zeroes out existing information that we collected in other `intercept_path` programs for this application thread before, guaranteeing that the information we report is specific to this opening operation. The kernel then begins traversing the path with `link_path_walk`, each execution of which will trigger our next program, `inode_perm_lsm` as an LSM [96]. With this, we accumulate the list of users and groups capable of modifying each component in an eBPF map like before. At the end of this sequence, we have the file which will actually be accessed—at which point, the kernel function `may_open` is invoked to check if the user may open the given file with the requested permissions, which will later be used to determine report severity (e.g., execution is higher severity than reading). We attach a program, `may_open_fentry`, to the start of this function and extract the requested permissions, saving that information to a map as well. Finally, we attach

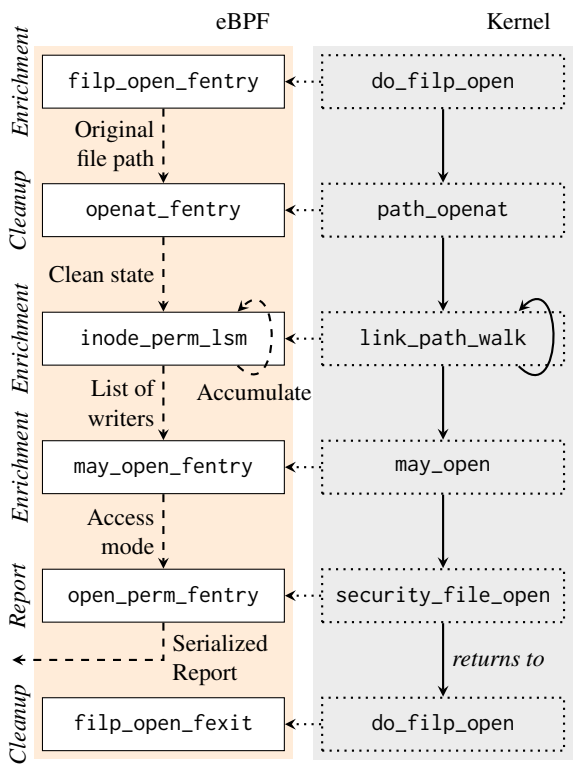

Figure 2: Diagram of information flow in the `intercept_path` pass. Dotted arrows refer to eBPF attachment points, dashed lines refer to information flow, and solid lines refer to control flow.

`open_perm_fentry` to `security_file_open`, a special function in which the helper function `bpf_d_path` is available. This helper function allows us to get the canonicalized path of the requested file. Once the canonicalized path is read, a report is sent to userspace along with all collected information if it is deemed potentially unsafe.

This example demonstrates how each pass is designed. First, we identify the information the pass needs to collect to make a decision. Then, we find corresponding kernel or userspace events from which we collect that data, implementing eBPF programs to attach to each and record that information in a map. With these programs defined and data collected, the final program in this sequence may inspect the data and conditionally send a report to userspace if the accrued information is deemed to represent undesired behavior.

4.1.1 Example Report

The `intercept_path` pass is able to detect Scenarios 2–4 of Section 2; we present an example report for Scenario 2 in Listing 3. This report includes a brief description of what occurred, the involved users, and a backtrace, and is issued with a severity level. Users may optionally translate executable offsets to source code locations with a secondary tool pro-

```
[WARN  os_sanitizer] listing-1 (pid: 204448),
    acting as root:root, attempted to access
    /tmp/tmp.3pRdIBT1bl/A/B (originally as B) with
    ["MAY_READ"] (0x4), which may be intercepted
    by uids [bob] and gids []; stacktrace:
 #0 0x40286e  (/tmp/tmp.3pRdIBT1bl/listing-1+0x286e)
 #1 0x45fab8  (/tmp/tmp.3pRdIBT1bl/listing-1+0x5fab8)
 #2 0x469e1b  (/tmp/tmp.3pRdIBT1bl/listing-1+0x69e1b)
 #3 0x46db9e  (/tmp/tmp.3pRdIBT1bl/listing-1+0x6db9e)
 #4 0x46d5ca  (/tmp/tmp.3pRdIBT1bl/listing-1+0x6d5ca)
 #5 0x46cf45  (/tmp/tmp.3pRdIBT1bl/listing-1+0x6cf45)
 #6 0x46e72f  (/tmp/tmp.3pRdIBT1bl/listing-1+0x6e72f)
 #7 0x43121b  (/tmp/tmp.3pRdIBT1bl/listing-1+0x3121b)
 #8 0x459a21  (/tmp/tmp.3pRdIBT1bl/listing-1+0x59a21)
```

Listing 3: Unsymbolized report for Scenario 2 of Section 2.

vided in the artifact, when debug information is available. The severity level is assigned in the userspace program and can be adjusted to the objectives of the user. In our artifact, the severity is determined by the *kind* of access that occurs. For the read/write operations of Scenarios 2 and 3, we assign the "warning" level because it is potentially quite dangerous, but has benign usages (e.g., indexing programs which run as root and scan user files). For execute operations like that of Scenario 4, it is assigned the "error" level because it is both very dangerous and rarely intended. Similar logic is used to define severity levels for other passes; for example, the TOCTOU pattern of Scenario 1 is assigned the "info" level because, without attacker modification, it is benign, but still not best practice. This severity classification allows users to avoid sifting through reports they deem irrelevant at the cost of false negatives.

4.2 eBPF Passes

For all of the mentioned strategies of Section 3.2, we highlight especially that an *overwhelming* majority of these tools simply augment a system's ability to detect a *fault*. They do not detect the defect itself nor the *capacity* of a program to enter a faulty state. While fault detection is an extensively explored topic, dynamic defect inference without fault detection is exceedingly rare. Because of the unique high-level nature of eBPF's observability, the defects we detect are primarily those which do not exhibit faults when subjected to normal inputs; they must instead be found heuristically with extensive program context.

For each type of defect we detect, we develop an OS-SANITIZER pass. Programs in a pass have one of four purposes: *reporting* suspicious code regions, *enriching* the report context by making additional observations, *cleaning up* outdated context, or *filtering* known false positives. In terms of the testing goals of these passes, we build passes for four general classes of behavior based on previously observed errors and warnings from the manual. We provide a brief summary

of passes below and include full description in our artifact. In total, including Section 6.3, these passes are implemented in around 2.8k lines of Rust code.

The first class is *Fault-Prone System Interactions* that indicate programmer errors in interactions with the kernel. Here we implement three passes: `access`, which corresponds to the use of the access or stat syscalls before an opening operation as extensively warned against in the manual page [54]; `fixed_mmap`, which detects erroneously overlapping memory maps [54]; and `rwx_mem`, the allocation of simultaneously readable, writable, and executable memory that often signifies high complexity self-modifying code. These passes were implemented to determine eBPF's efficacy in finding faulty interactions with the kernel.

Fault-Prone Library Usage is the second category, in which passes report erroneous usage of userspace library functions. We implement five such passes: `filep_unlocked`, for unsynchronized accesses to file references across multiple threads [54]; `gets`, the universally unsafe usage of the gets function [54]; `snprintf`, for information leakage issues associated with a footgun of the library function of the same name [35]; `printf_mut` and `system_mut`, which correspond to the use of mutable string data in template and command arguments, respectively (both of which are severe security errors [66]); and `system_abs`, which identifies cases when the developer has not specified the full path to an executable when launching it with the system function [100]. These passes evaluate eBPF's ability to find defects in misuses of userspace APIs.

Third are *Environmental Defects* which indicate that a program's behavior could be made faulty by the environment. The most typical form of this defect class relates to filesystem operations; we implement two passes in this category: `sec_file_open`, which identifies numerous issues associated with opening a given file, such as excessive file permissions; and `intercept_path`, which identifies if the path specified by the user could be intercepted and redirected by an attacker. On their surface, the two passes sound similar, but have different implications. `sec_file_open` checks if an opened file is incorrectly protected; if the targeted file is opened with unsafe permissions, the state of the file cannot ever be trusted even if the program later corrects those permissions. `intercept_path` checks if any member of a path (i.e., the targeted file or the intermediary directories) has improper permissions which may allow local attacker to redirect the open only at the time the file is initially accessed [14]. These passes focus on situations where the environment induces the undesired behavior rather than the program itself.

Finally, we write passes which track *Unsafe Memory-Manipulating Function Usage*, using numerous heuristics to identify when likely memory-unsafe usage of certain library functions occurs. These are the remaining passes: `memcpy` and `strncpy` check for likely overflowable buffers based on the usage of input-dependent copy sizes [54]; and `strcpy` and `sprintf` check functions which copy untrusted input without length checks [54]. This class focuses on library functions which deal with low-level memory manipulation. By attaching to these functions, we get a better sense of eBPF's ability to detect low-level errors rather than the high-level errors represented by the previous classes.

4.3 Prototype Infrastructure

To start, OS-SANITIZER implements a launcher which attaches eBPF programs to their desired points, awaits the corresponding reports regarding potential defects, and formats them for interpretation by a human reviewer. The reports, read from buffers shared by the eBPF programs and the launcher, are processed in five stages:

1) Deserialization The reports are first read and deserialized from the shared buffers such that they may be interpreted.

2) Stacktrace Mapping Each report contains a process ID and thread ID and a stacktrace collected by eBPF [54].

3) Filtering Some report types cannot be filtered in eBPF alone and require additional local tests, so the launcher performs some basic checks before emitting the final reports. In addition, OS-SANITIZER classifies reports by severity. Any reports under the user-specified severity level are discarded.

4) Reporting Once the reports are processed and filtered, they are emitted in a human-readable but standardized format that may later be "symbolized".

5) Symbolization A second utility, the *symbolizer*, parses the formatted reports emitted by the launcher and maps the observed file offsets to source code information. This is *not* performed continuously as this task is computationally expensive when not performed in bulk.

The launcher itself is typically executed as a systemd unit, running in the background continuously, and is comprised of about 2.5k lines of Rust code.

4.4 eBPF vs. Other Tools

We reviewed a large number of methods and tools to determine which would be most appropriate for dynamic defect inference. In Section 4.1, checking for path interception out-of-band (i.e., not during the original path traversal) would be raceful and lossy. To implement the `fixed_mmap` pass, one must monitor and record mappings of a process continuously and check when they overlap, requiring lots of bookkeeping that may lead to scalability issues. Finally, if we tried to implement *any* pass involving userspace monitoring from kernelspace, we would effectively need to reimplement the breakpoint strategy already provided by eBPF. Table 1 presents

Table 1: Pass's ability to be implemented by static analysis, compiler instrumentation, userspace interception, kernel (or kernel module) monitoring, various auditing tools, or eBPF, respectively. Citations are known implementations and ✘, ✓, and ∼ indicate infeasibility, feasibility, and severe tradeoffs or limitations, respectively.

Pass	Statically?	Compiler?	User?	Kernel?	strace [50]	ptrace [54]	ltrace [9]	seccomp [54]	Falco [91]	eBPF?
`access`	∼	∼	✘	[104]	∼	∼	✘	∼	✓	✓
`fixed_mmap`	✘	✘	∼	✓	∼	∼	✘	∼	∼	✓
`rwx_mem`	∼	✘	∼	[85]	✓	✓	✘	✓	✓	✓
`filep_unlocked`	∼	[58]	✓	∼	✘	∼	∼	✘	✘	✓
`gets`	[34]	[42]	✓	∼	✘	∼	✓	✘	✘	✓
`snprintf`	[35][4]	[55][1]	∼	✘	✘	∼	∼	✘	✘	∼
`printf_mut`	∼	[18][1]	∼	∼	✘	✘	✘	✘	✘	✓
`system_mut`	∼	✘	[10][2]	∼	∼	[10][2]	∼	∼	∼	✓
`system_abs`	[33][3]	✓	✓	∼	✘	∼	✓	✘	✘	✓
`sec_file_open`	✘	✘	∼	[104]	✘	∼	✘	∼	✓	✓
`intercept_path`	✘	✘	✘	✓	✘	✘	✘	∼	✘	✓
`memcpy`	∼	[55][2]	∼	✘	✘	∼	∼	✘	✘	∼
`strcpy`	∼	[55][2]	∼	✘	✘	∼	∼	✘	✘	∼
`strncpy`	∼	[55][2]	∼	✘	✘	∼	∼	✘	✘	∼
`sprintf`	∼	[55][2]	∼	✘	✘	∼	∼	✘	✘	∼

[1] Reports whenever a non-literal argument is used (FP-prone).
[2] Only reports when a corresponding fault occurs as a result.
[3] Only reports when a relative path is provided in a string literal.
[4] Only reports when the return value is unchecked.

an overview of this dilemma, indicating the capabilities of general testing strategies and representative tracing utilities to implement our passes.

The listed utilities have a variety of tracing strategies, but are not as effectively positioned as eBPF: strace [50] and ltrace [9] are userspace utilities for intercepting system calls and library calls, respectively. They can be used to implement some of the passes, but face issues with scalability, are not applied system-wide, and must be applied per-process. ptrace [54] is a Linux tracing utility which can manipulate and monitor the behavior of other processes on the system, and can effectively monitor many behaviors. Like strace and ltrace, ptrace requires userspace helpers that do not easily scale and cannot be applied system-wide. seccomp [54] is a monitoring and system call filtering tool which can even include small programs that decide what action is taken when certain system interactions occur. Unlike eBPF, there is no data persistence mechanism, so these programs cannot idiomatically accrue information for heuristic-based decision making. Finally, Falco [91] is an eBPF-based monitoring language and utility which allows to write configurations that enable monitoring of some system behaviors. It is unfortunately not general enough to implement many passes and lacks the ability to monitor userspace behavior. We considered additional tools for tracing and monitoring, but faced similar limitations. The observability of eBPF uniquely fills this gap in testing.

The detection of TOCTOU vulnerabilities is a particularly relevant case, as this is a very common latent defect for which inference has been implemented in previous works. The extensive body of research on detection and prevention strategies for such defects include both kernel- and userspace-based dynamic analysis and static analysis [79]. Static analysis tools often search for *potentially* race-prone function calls in the source code, but as we illustrated in our motivating example (Section 2), accesses may be hidden in a library dependency or totally benign in the intended use case, and the severity depends on the execution context. Existing dynamic tools either monitor file access logs, interpose or modify the system call interface, or provide a sandboxed filesystem [79]. eBPF puts us in the position that neither kernel customization nor target program patching (e.g., using binary rewriting [77]) are necessary to detect TOCTOU bugs. Indeed, eBPF trivializes the process of TOCTOU detection on two fronts: the detection of raceful behavior (the `access` pass) and the file operations which involve potentially unsafe permissions (`sec_file_open` and `intercept_path`).

5 Limitations

We provide an overview in Table 1 of how existing strategies compare with eBPF's capabilities in implementing a detection mechanism for the passes we specify in Section 4.2. Naturally, we cannot implement passes which cannot be implemented in eBPF, so these are absent. Before continuing into the evaluation, we must discuss what cannot be done *well* and what cannot be done *at all*.

5.1 Hard Limits

The eBPF subsystem has several limitations by nature of its design. Its primary purpose is in attaching small programs

to various points in a running Linux system; eBPF is, by definition, a dynamic tool. Static analysis is an orthogonal strategy, and we cannot expect eBPF to assist in such tasks.

While we can (and do!) monitor system interactions of programs written in *any* programming language, we cannot attach to userspace functions of programming languages for which the compilation target is not an executable. As an example, we can observe that a Java application performs a fault-prone interaction with the filesystem, but we cannot write passes which observe misuse of Java APIs. This is a hard limitation for which there is no remediation in eBPF, and other strategies for dynamic defect inference must be used for these languages and testing goals instead.

The limitation most affecting eBPF's applicability is the verifier itself. The eBPF verifier ensures both the termination and safety of a loaded eBPF program. As such, there is a strict upper bound on the complexity of programs we can load and attach. To compensate for this, we write our programs in Rust with the aya library [92], which ensures via Rust's memory model that our programs are more easily verified. Though we face theoretical limitations in complexity, we did not struggle with this in our prototype.

5.2 Soft Limits

The ways in which eBPF programs are attached and executed are fixed, and our attachment types are limited. Most relevant, the u(ret)probe attachment points introduce breakpoints that cause the application to context switch into the corresponding eBPF program in kernel space before switching back into the application. As a consequence, this type of attachment faces significant overhead beyond the execution of the eBPF program alone. We evaluate the severity of this limitation with the `memcpy`, `strcpy`, `strncpy`, and `sprintf` passes, all of which attach to userspace functions used in performance-critical code.

Because we process many events in isolation and our reporting is performed per observed instance of potentially defective behavior, we expect to observe duplicated events. This is itself a limitation of all dynamic strategies, but is amplified in our case as we do not terminate programs when we generate a report. We evaluate the duplication rate and its impact on investigation of reports to determine the severity of this limitation. Similarly, because we use heuristics to overapproximate defective behavior, some reports will be false positive findings. We discuss their prevalence and example false positives in Section 6.4.1 and Section 7, respectively.

6 Evaluation

For our evaluation, we perform a benchmarking study, a reproduction study, and a long-term usage study. The purpose of our benchmarking study in Section 6.2 is simple: to determine the raw performance impacts of our monitoring. We conduct the reproduction study in Section 6.3 to inspect the extensibility of eBPF by reimplementing an existing work entirely in OS-SANITIZER, quantifying the difficulty of writing new passes by example.

The purpose of our long-term study in Section 6.4 is more subtle. We want to understand the implications of deploying our tool in terms of its ability to find new defects in Linux applications. To do so, we employ OS-SANITIZER on actively used Linux desktop computers, giving a profile of running the tool constantly on personal devices.

6.1 Experimental Setup

We use two systems as different workloads have different expected execution environments, including different system configurations, hardware, and other installed software on the system. We denote the system used during each evaluation accordingly. System A is a Fedora Linux 39 Server VM with 4 vCores and 24 GiB RAM on an Intel Xeon Gold 6248R CPU. Multiple instances of System A were used on a single KVM host exclusively used for this evaluation, and care was taken not to overbook cores. System B is a Fedora Linux 39 Desktop baremetal installation on a Lenovo ThinkPad X1 Carbon Gen 10 with Intel Core i7-1260P CPU and 32 GiB RAM.

6.2 Benchmarking

To assess the performance impacts of eBPF when used for defect inference, we conduct two forms of benchmarking: microbenchmarks, which measure the wall-time impact of our passes via programs which *only* implement the defective code, and benchmarks of real programs, which measure the impact of our passes on real programs. Microbenchmarks indicate the performance impact of the pass absolutely, allowing us to investigate the raw overhead of our passes. Real program benchmarking, on the other hand, indicates the impact of the pass as one might expect to observe in practice.

6.2.1 Microbenchmarks

We design a suite of microbenchmarks implemented in C, one for each pass implemented in OS-SANITIZER. These synthetic programs represent "true positive" defects that should be detected by OS-SANITIZER if they occur in real programs. Each microbenchmark exactly implements the sequence of events under which the pass should issue a report. This sequence is executed in a tight loop, incrementing a counter for every execution of the loop. Immediately before the loop begins, a timer is registered for 10 seconds, which, upon completion, triggers a signal handler that prints the execution counter and terminates the process. In this way, we can measure the exact overhead for a given operation induced by OS-SANITIZER. Each of the scenarios was evaluated for ten

Table 2: Results of microbenchmarks.

Pass	Iteration Time [μs]		Ratio
	Baseline	OS-SAN.	
`access`	1.98	7.33	3.70
`fixed_mmap`	7.19	27.05	3.76
`rwx_mem`	2.55	15.22	5.96
`filep_unlocked`	3.55	6.07	1.71
`gets`	54.10	60.69	1.12
`snprintf`	0.28	6.81	24.04
`printf_mut`	0.18	6.33	34.94
`system_mut`	4571.43	4905.57	1.07
`system_abs`	4465.28	4916.42	1.10
`sec_file_open`	1.38	3.22	2.33
`intercept_path`	1.53	6.13	4.00
`memcpy`	0.07	9.96	140.19
`strcpy`	0.04	4.68	106.38
`strncpy`	0.09	9.46	103.90
`sprintf`	0.10	4.77	46.56

trials and the results are summarized in Table 2. System A was used for this evaluation. As expected, the cost of eBPF programs is non-zero, and, in the case of userspace function probes, passes induce a context switch. While double- or triple-digit overhead ratios exhibited by some passes might seem prohibitive at first glance, keep in mind that these artificial benchmarks are *contrived worst-case* workloads. To get a more realistic picture, in the next section, we test the passes on real-world benchmarks. We discuss the special cases of `memcpy`, `strcpy`, `strncpy`, and `sprintf`, passes which attach to functions that do not involve system calls themselves, separately in Section 6.2.3.

6.2.2 Real Program Benchmarks

To understand the performance impact of OS-SANITIZER on heavy workloads, we utilized two recognized existing performance benchmarks: SPEC CPU 2017 [6] and Browserbench Speedometer 3.0 [13]. We selected these two benchmarks as they represent common usage of Linux systems, and thus may be used as a reference for actual impact on user applications during realistic heavy tasks. The SPEC CPU benchmark was executed on System A and the Browserbench benchmark was executed on System B. We evaluated each benchmark with 18 scenarios: one for each of the 15 passes, one for all non-memory-safety passes, one for all passes, and one baseline with no passes enabled. Each of the scenarios was evaluated for ten trials for both benchmarks with the results summarized in Table 3. On benchmarks which have low reliance on C APIs or system interactions, we see very little impact whatsoever (e.g., *mcf*, *xalancbmk*, *deepsjeng*, and *exchange2*). Benchmarks which involve significant amounts of data processing that rely on glibc's extremely efficient implementation of these routines incur severe penalties (see the `memcpy`, `strcpy`, and `strncpy` results for *perlbench*, *gcc*, *omnetpp*, *x264*, *leela*, and *xz*), confirming our expectations from Section 5. Finally, the *gcc* benchmark's `filep_unlocked` result shows significant and relatively large overhead due to its sole use of unlocked file operations [27]. Other than the latter two cases, no statistically significant result had a performance overhead greater than 7%.

6.2.3 Performance of Memory-Safety Passes

Consistent with our expectations in Section 5, our artificial microbenchmarks show that all four memory safety passes induce a very high overhead ratio. The real-world benchmarks paint a more nuanced picture; while `memcpy` and, at least in some benchmarks, both `strcpy` and `strncpy` come with a prohibitive overhead, there is no significant slowdown for `sprintf`. Further investigation reveals that this is a direct consequence of how frequently these calls are used: `memcpy`, `strcpy`, and `strncpy` are all used frequently with their optimizations more clearly affecting program performance, whereas `sprintf` is used less often and is not as performance critical. Consequently, while we include these passes in the benchmarks to demonstrate the feasibility of harnessing eBPF for these low-level operations, we conclude that this application is not appropriate for eBPF in practice.

> **Takeaway #1:** In general, eBPF probes for dynamic defect inference are performant, but suffer when applied to userspace functions that do not interact with the system.

6.3 Reproduction Study

Wei and Pu introduced the "CUU model" (check, use, use), a methodology for passively identifying filesystem-based TOCTOU defects [104]. The paper categorizes various filesystem-specific syscalls into sets describing their actions, with specific sequences of observations implying the existence of such defects. This model was then implemented with kernel modifications which log all file interaction events and a set of userspace programs to extract and inspect said log. We requested a copy of the source code from the authors, but did not receive a response. We note that the paper's underlying premise is quite similar to our own: *inferring* the presence of defects in Linux applications based on runtime observations.

To demonstrate the complexity of implementing a new pass as described in Section 4.2, we implement a pass for the CUU model as described in Section 2 of the aforementioned work [104]. This was completed within 4 hours: 2 hours for the initial implementation and 2 hours for debugging and finalization. The final implementation was written in the same style as the original passes and is composed of 260 lines of code change (written according to Rust style conventions [81]) and measured by diff. Of these 260 lines, 151 are located within eBPF programs, 74 are associated with

Table 3: Results of standardized benchmarks expressed as ratio over baseline[1].

		SPECspeed 2017 Integer [6]										
	Pass	perlbench	gcc	mcf	omnetpp	xalancbmk	x264	deepsjeng	leela	exchange2	xz	Browser[2]
Overhead Ratio	`access`	1.03	1.01	0.99	1.01	**1.06**	1.01	1.01	1.00	1.00	0.96	0.95
	`fixed_mmap`	1.00	1.00	1.00	0.99	1.02	1.00	0.99	1.00	1.01	1.02	1.01
	`rwx_mem`	1.00	1.02	1.00	0.97	1.00	1.00	1.01	1.00	1.00	1.00	0.96
	`filep_unlocked`	1.01	**1.32**	0.99	1.01	1.07	1.01	1.01	1.01	1.00	0.99	1.01
	`gets`	1.02	0.99	0.98	0.98	1.04	1.01	1.01	1.01	1.00	0.94	0.95
	`snprintf`	1.02	1.06	1.00	1.03	1.05	**1.03**	**1.05**	**1.01**	1.01	**1.07**	1.00
	`printf_mut`	1.00	1.04	0.99	1.03	1.04	1.02	**1.04**	**1.02**	**1.01**	1.02	1.02
	`system_mut`	1.01	0.98	0.99	0.97	0.99	1.01	1.02	1.01	1.00	0.99	1.00
	`system_abs`	0.99	0.99	1.00	1.00	1.01	1.00	0.99	1.00	1.00	1.01	1.02
	`sec_file_open`	1.02	1.00	1.00	0.98	0.99	1.00	1.01	1.00	1.01	1.07	0.99
	`intercept_path`	1.01	1.04	1.00	0.97	0.98	1.00	0.99	1.00	**1.01**	1.02	1.04
	All Previous	0.96	**1.36**	0.99	1.01	1.01	1.00	1.01	1.00	1.00	1.05	1.00
	`memcpy`	**6.33**	**2.18**	0.92	**4.37**	1.02	**6.01**	0.97	**1.83**	**1.04**	**1.95**	**8.38**
	`strcpy`	1.08	**1.17**	0.99	**5.56**	1.02	1.00	1.02	1.00	1.00	0.97	**1.25**
	`strncpy`	1.08	**1.19**	0.98	**4.57**	1.00	1.01	1.02	1.01	1.00	0.96	**1.23**
	`sprintf`	1.01	0.99	0.99	1.01	1.03	1.01	1.00	1.00	1.00	1.00	1.00
	All Passes	**6.46**	**2.56**	0.96	**5.50**	1.03	**5.98**	0.98	**1.81**	**1.03**	**1.95**	**8.62**
Baseline Time [*s*]		396.42	600.33	885.33	680.82	287.30	184.35	395.64	456.88	193.16	1070.77	45.92

[1] Significant results ($p < 0.05$) by one-sided Welch's t-test are **bolded**. [2] Browserbench Speedometer 3.0 [13] on Google Chrome.

launching the programs and formatting the reports, and 35 are serialization routines. We tested this on the same system as the one described in Section 6.4 and indeed found several programs that violated the CUU model, though none proved to be defective.

Takeaway #2: (Re-)Implementing complex dynamic defect inference works is massively simplified with eBPF.

6.4 Usage Case Study

One of the primary advantages of using eBPF is that we may discover latent defects outside of a testing environment, as there is no need to recompile or modify applications. To evaluate this, one of the authors ran the tool on the aforementioned System B for one month while using the device for typical software development workplace activities during 8-hour workdays. This usage primarily consisted of graphical IDEs, browsers[1], compilers, and containerization software. We found a large number of defects during this period; a representative selection of case studies are provided in Section 7. Below, we analyze the aggregate reporting data to provide a general sense of data volume and a brief discussion of the subjective user experience. We do *not* make the raw data for this segment of the evaluation available as the report logs contain private information collected during the tool's execution.

[1] Segments of this paper were written with Firefox [69] during this period. Two of the entries in Table 5 were discovered by this.

Over the one month period, all passes were run continuously, excluding the `memcpy`, `strcpy`, and `strncpy` passes which induced prohibitive performance impact and were instead only run during investigation periods. The remaining passes resulted in negligible performance impact under normal workloads and did not perceivably affect user experience. Reports were restricted to those of high confidence and moderate to high impact based on the perceived strength of the heuristic and historical impact of related defects, respectively.

6.4.1 Investigation Process

Over the month in which we ran OS-SANITIZER, we spent approximately two hours per week investigating reports for a total time of about eight hours. We begin our inspection using `journalctl` to read the reports. A small subset of reports were clearly false positives and were recorded to be ignored in future iterations of OS-SANITIZER as filtering programs. The remaining reports required greater inspection and consideration.

Using debuginfod [61] and the `debuginfo-install` dnf plugin in combination with our symbolization utility, we can easily identify the source code region(s) associated with a report. When the binaries were not distributed by Fedora but provided by other vendors, it was necessary to use online references and manually identify version info, which could be quite difficult when source code was not available. Even more rarely, binaries were configured in a way that prevented

complete stacktraces from being reported due to a known limitation of eBPF stacktrace recording [17]. Nevertheless, in an overwhelming majority of the reports investigated, we found the corresponding responsible source regions in a matter of a few minutes at most. Once a corresponding cause was determined, we could directly test whether it was correct by simply performing an action with the corresponding application and seeing if the same report was produced. In this way, we were able to uncover a number of issues in Linux applications present in our evaluation system which we describe further in Section 7.

Takeaway #3: eBPF is sufficiently performant to be passively used for defect inference in long-term scenarios, even on actively used devices.

6.4.2 Report Volume

We select the five days of highest report volume from the trial month to reduce the effect of low usage days. During these five days, a total of 276MiB was generated (originally 7.2MiB as compressed by journald [30]), containing approximately 679K log entries. Across these entries, 2,784 unique binary offsets appeared in stacktraces, including 127 unique binaries. We consider the approximate upper bound for the number of unique reports to be 2,784 as deeper stack trace entries are less likely to appear in multiple reports with separate causes. This gives a *lower-bound* observation frequency of 244; that is, for every unique report, there are on average 244 corresponding log entries. This is, admittedly, a usability limitation of our prototype. With additional engineering effort, one could address this issue by storing reports into a database structure where we can analyze similar events. That said, this limitation did not meaningfully impact our ability to inspect events. Only two hours were actually needed to inspect these reports per week, as the plaintext reports were easily filtered with shell utilities like grep, awk, and sort. Reports occurring in the same code locations were processed into their own files and inspected together, allowing us to relate multiple events and deduplicate easily. While this manual effort could certainly be automated to improve our tool for production use, it did not meaningfully affect our ability to find defects nor demand too much additional time.

Measuring a false-positive rate regarding the reports described above would be necessarily misleading, as we found that the determination of a particular code region as "defective" was strongly dependent on the execution context; we discuss this at length in Section 8.2. We provide some representative case studies in Section 7 which highlight this conundrum.

Takeaway #4: Reports are often duplicated. Manual deduplication of these reports is simple due to consistent stack traces, but automated aggregation is possible.

Table 4: Breakdown of events and their report severity, if any report was emitted.

	Report Severity			
Pass	ERROR	WARN	INFO	Events
`access`	0	0	839k	13.5M
`fixed_mmap`	0	0	707k	2.5M
`rwx_mem`	0	62.9k	0	4.6M
`filep_unlocked`	0	0	0	417M
`gets`	0	0	0	0
`snprintf`	0	11.3k	0	34.3M
`printf_mut`	0	0	0	13.5M
`system_mut`	0	0	0	0
`system_abs`	0	0	0	0
`sec_file_open`	0	1	345	6.2M
`intercept_path`	1	275	1545	7.9M

6.4.3 Prevalence Study

As we discussed earlier and in-depth in Section 8.2, determining the false positive rate for our approach is non-trivial. We *can* measure how many times each pass was executed and how many reports were produced at each severity, giving us the *prevalence* of events detected by our passes. This provides an indication as to how greatly the passes reduce the set of events to sort through and the frequency of such events on the evaluated system. We recorded the prevalence of reported events on the same system over a separate 24-hour period and present the results in Table 4.

The results demonstrate, as one might expect, that events happen occur at different frequencies. Passes which monitor events that happen very frequently in many processes trigger the most often, but most of these events are considered low severity or benign by our heuristics. Recall that only ERROR and WARN level reports were analyzed during the long-term study, since these are the events for which we had a higher confidence that a vulnerability was present. Other events never occur at all; both `system` and `gets` are known to be difficult or impossible to use safely, and thus rarely appear in practice. Conversely, there are many reports for the `access` pass, indicating that many processes are still racefully accessing files. Otherwise, what remains are a few benign, but highly duplicated reports found by `fixed_mmap` (pre-allocation in `ld`, which is invoked by every dynamic executable), `rwx_mem` (RWX mappings in Firefox, Java, and PCRE2 mentioned in Section 7.4), and `snprintf` (a known benign case in libmutter that cannot be filtered in eBPF).

Takeaway #5: The prevalence of events which are actually reported is relatively low. Users can adjust severity rankings to filter for relevant information.

7 Defect Case Studies

Below, we provide a representative sample of the identified defects from which we determine takeaways about dynamic defect inference specifically with eBPF and in general. A summary of all confirmed issues is presented in Table 5. By nature of the passes we developed, most of our true positive findings were security-relevant. In addition to our own investigations, we privately shared OS-SANITIZER with Linux quality assurance teams before paper submission to find defects beyond those we were able to find ourselves. We did not request any bug information as to maintain responsible disclosure, but received notice that OS-SANITIZER was indeed able to find similar defects in other Linux distributions.

7.1 C API Misuse leading to Buffer Overflow

During development, OS-SANITIZER flagged speech-dispatcher [29], a text-to-speech utility, because `strcpy` was called with a stack buffer destination without a prior `strlen` check on the source buffer. Upon symbolization, we were able to identify the code region responsible for the behavior in dotconf [46], a dependency of speech-dispatcher, and confirmed by source code analysis that the `strcpy` call was indeed vulnerable to buffer overflow. Any attacker able to cause the loading of a malicious configuration file with dotconf immediately gains code execution capabilities. This defect had been present for at least two decades and is installed by default on many Linux distributions. Treating this detection as a code smell, we developed a fuzzer to exercise this code region further. Though unable to produce an input triggering the original defect due to a precise size requirement, this fuzzer uncovered three additional bugs.

Takeaway #6: When dynamic defect inference is deployed system-wide with eBPF, the *system* is evaluated for defects, uncovering long-undiscovered defects in programs which expose the system to undesired effects.

Takeaway #7: Traditional software testing techniques such as fuzzing complement OS-SANITIZER, finding defects that traditional testing cannot and guiding testers to code regions that require additional testing.

7.2 Unsafe Language-System Interaction

Docker [20] is a very widely used development and hosting software. It is an integral part of many developer pipelines, both on development systems and in continuous integration (CI) services, including on the machine used for long-term usage described in Section 6.4. Docker offers the ability for developers and server administrators to execute code within a "container" that shares the kernel, but isolates access to resources on the host.

During the long-term evaluation, OS-SANITIZER discovered an issue within Docker, caused by the Go [36] standard library. This issue causes the process to delete a directory unsafely and has been present in Go since the first release of the programming language, having previously been identified in 2022 [86]. This is caused by the issue described in the motivating example of Section 2. More precisely, this is exactly Scenario 2: a TOCTOU which is used in a privileged context on files owned by an unprivileged user. In practice, this allows local attackers to delete arbitrary targeted directories recursively when a container is being deleted. Investigating further, we found that two other major containerization systems, Kubernetes [93] and runc [74], were affected by the same issue. We reported this issue first to the Go security team, who did not recognize it as a defect and indicated that this was a result of developer error. We subsequently reported the issue to the affected containerization systems who then submitted a fix to Go and coordinated disclosure.

Later, OS-SANITIZER identified another similar issue in Go which exposed Docker to an undesired *copy* of files instead of deletion. This issue could be used to copy arbitrary files from the host system into a container, potentially disclosing sensitive information. Further inspection found that Kubernetes was also affected. Go acknowledged this as a defect as it was inconsistent with documentation that indicated that such behavior should not occur [71]. This issue could not be completely fixed because the function's specification made it impossible to implement correctly. Instead, its documentation was updated to reflect its limitations for privileged operations. Given that now two such issues had been found, we and the Docker development team investigated further and found a few additional issues, which prompted extensive hardening efforts in both Docker and Go [70].

Takeaway #8: Certain passes observe system interactions through which we can detect defects in arbitrary programming languages. This includes defects introduced by the programming languages themselves.

Takeaway #9: Dynamic defect inference concretely identifies that software is potentially defective at runtime. In this case, the privilege boundary and TOCTOU together indicated the presence of the vulnerability.

7.3 User-Owned File Executed as Root

As the long-term usage study was performed on a developer machine, many reports related to unsafe usage of programs by direct act of the user. In one such case, a script was distributed to build another ongoing research project. When executed within a Docker container, the script invoked build commands as root. The project itself was contained within a user directory in the container, and as such the file was owned by a local

Table 5: Overview of confirmed issues.

Weakness (CWE [67])	Case Study	Affected Software	Discovering Pass(es)	Status
Stack Buffer Overflow (121)	7.1	dotconf [46]	`strcpy`	Fixed
Arbitrary File Deletion (59)	7.2	Golang [36]	`intercept_path`, `access`	Fixed
Arbitrary File Read (59,552)	7.2	Golang [36]	`intercept_path`, `access`	Can't Fix
Root Execution of User File (471)	7.3	-	`intercept_path`	Fixed (Locally)
RWX Memory (732)	7.4	PCRE2 [40]	`rwx_mem`	See Section 7.4
User-modifiable Command (732)	7.5	modprobe [54]	`system_mut`	Works as Intended
Execution of Globally RWX File (281)	-	Chrome [37]	`sec_file_open`	Fixed
Globally RWX File Creation (281)	-	Firefox [69]	`sec_file_open`	Fixed
`stat`/`open` TOCTOU (367)	-	Firefox [69]	`access`,`sec_file_open`	Won't Fix[1]
RWX Memory (732)	-	Firefox [69]	`rwx_mem`	Works as Intended
Root Execution of User File (471)	-	GDB [28]	`intercept_path`	Confirmed
RWX Memory (732)	-	mesa [62]	`rwx_mem`	Fixed[2]
Arbitrary File Read (59,552)	-	NetworkManager [31]	`intercept_path`	Fixed
Self-resettable Lockout (307)	-	pam_faillock [54]	`sec_file_open`	Won't Fix[2]
Distribution of Globally RWX Files (281)	-	Rust [84]	`intercept_path`	Fixed
2× Memory Safety Issue (121)	7.1	dotconf [46]	*secondary investigation*	Fixed
Improper Escaping (150)	7.1	dotconf [46]	*secondary investigation*	Fixed
(Several) Arbitrary File Read (59,552)	7.2	Docker [20]	*secondary investigation*	Confirmed[1]
12× Memory Safety Issue (121,122,908)	7.4	PCRE2 [40]	*secondary investigation*	Fixed[3]
22× Differing Implementation (474)	7.4	PCRE2 [40]	*secondary investigation*	Fixed[3]

[1] Privately reported. [2] Not reported by us, but discovered independently. [3] Incomplete list; several additionally privately reported.

user but executed by root. The execution of the user-owned script by root was flagged by the `intercept_path` pass at maximum severity, as the file permissions requested included execution permissions. A local attacker capable of replacing the file may have trivially gained root privileges within the container.

Takeaway #10: eBPF can be used to identify erroneous program *usage* which leads to undesired effects.

7.4 Just-In-Time Compilers

OS-SANITIZER issued reports that indicated that a number of applications were allocating memory which was simultaneously readable, writable, and executable (RWX). This is not itself a defect; while RWX memory is generally warned against, this warning is primarily a security hardening concern [88]. Just-in-time (JIT) compilers often use such memory, are deeply specialized, and very difficult to implement, often holding numerous subtle defects which require specialized verification and search-based testing techniques to uncover [4,39,72,106]. Three such applications exhibited this behavior most: Java and Firefox, which emit JIT compiled code representing Java and Javascript code, respectively; and PCRE2, a regular expression library. Though we had unrelated findings in the former two, we focus on the latter.

PCRE2 [40] is one of the most widely-used regular expression libraries available, with its bindings affording search capabilities to a wide number of languages. One of the reasons for its widespread adoption is speed: it very efficiently expresses and executes regular expressions against even large strings. This is due in part to its early adoption of JIT compilation [41] with the introduction of the sljit compiler [43] in 2011. When investigating the JIT compiler, we found that this component of the library was not undergoing automated testing. We introduced testing infrastructure for this component with which we have since uncovered 12 additional security-relevant defects and 22 correctness issues related to the consistency between the JIT implementation and the interpreter.

Takeaway #11: Dynamic defect inference finds behaviors which are not indicative of just defects, but of a code smell. These *behavioral code smells* highlight code regions which require additional investigation.

7.5 Mutable `system` Argument in `modprobe`

`modprobe` [54] is a system utility which loads kernel modules. When called on a particular kernel module, the module may specify "install" and "remove" commands which execute upon the loading and unloading of the module, respectively. These commands are read and copied to a mutable memory region, causing the `system_mut` pass to report that a potentially untrusted string was executed as a command [100]. Given the purpose of `modprobe`, this report is a false positive; yes, an input-controlled string is executed as a command, but the input itself is code to be loaded into the kernel. In other words,

the command being executed is already trusted by nature of the input.

Takeaway #12: Using heuristics to infer the presence of defects necessarily leads to false positives, as the otherwise unsafe code may be benign in certain contexts.

8 Discussion

Before concluding, we present a number of thoughts regarding our paper's findings and its implications.

8.1 Threats to Validity

Given the small effect size and the number of evaluated scenarios, we infer that some statistically significant findings from Section 6.2.2 may be spurious as a result of multiplicity [49], where the likelihood of spurious results increases with the number of experimental scenarios. Numerous methods exist to control for the effects of multiplicity, but are prone to inducing false negatives. In our case, the issue emerges predominantly from the high variance of our benchmarks; in one case, we observe a median execution time of up to 8% *less* than baseline, which is not reasonable. Taking an order of magnitude more samples to account for this observed variance is computationally prohibitive, as existing evaluation already takes multiple days. As such, we suggest that the system-induced variance is too high to claim reliably significant performance impact for effect sizes of less than 10%.

The reproduction study of Section 6.3 is only one such example of a pass one could implement. We do not concretely know the complexity of the original implementation, and can only infer that it was of significantly greater complexity due to its implementation's description. Moreover, our reported times are that of the only expert developers in using OS-SANITIZER: ourselves. Conducting a human expert study was deemed inappropriate as this paper is primarily concerned with the technical implications of using eBPF for dynamic defect inference. For this reason, we emphasize that the results presented in Section 6.3 (and, similarly, Section 6.4.1) are not the results of formal human research study, and are included only to give a general sense of OS-SANITIZER usage.

8.2 False Positives

In none of our evaluations do we compute any false positive rates. This was a choice; we found early on in our long-term evaluation that evaluating the outcome of a given report is deeply non-trivial. Consider the case study presented in Section 7.4, where we used RWX memory as a strong indicator for self-modifying code. In all reports we processed, the report correctly identified the presence of a JIT compiler which managed memory in a potentially unsafe manner. Can we truly classify these as true positives, when this behavior is intended but merely indicative of a particular class of high code complexity? Indeed, there is disagreement on what a false positive in the context of code smells even is, both in theory [24] and in practice [75]. Similarly, we had many detections where developers had chosen a simple, but potentially defective implementation based on assumptions of how the code was to be used. Consider the case study in Section 7.2; the original implementation was indeed against best practices for implementing these filesystem accesses. If we had detected this in a scenario where there was not an expected privilege boundary, would it still be considered a true positive? Many reports issued by OS-SANITIZER during our evaluation force us to ask such questions. Clear best practice recommendations are often not followed when there is a perception that the recommendations are irrelevant to the intended use case, but these use cases are often poorly communicated to downstream developers and users. As such, we find that most "true positive" findings encountered during our long-term evaluation are a result of this inconsistency in expectations, and that most "false positive" findings reveal the potential for such inconsistencies in the future.

9 Conclusion

In this paper, we demonstrated that eBPF is an effective tool for inferring the presence and location of defects based on runtime-observed behavior. By targeting a set of Linux application defects which are well-understood in practice, we were able to evaluate the applicability and performance impact of our approach. Our analysis revealed a significant number of previously undetected defects. This is particularly notable given the maturity of the systems analyzed and the active efforts by both developers and security researchers to identify such issues. Our findings also revealed challenges in consistently identifying what constitutes a defect and determining responsibility for remediation, pointing to pervasive inconsistency between developer intent and downstream usage. This issue affects the correctness and safety of code in ways that are as-of-yet not well-explored. Furthermore, we found that behavioral code smells were powerful for identifying code deep in dependencies which required more testing, and found a multitude of defects using these techniques.

Our work highlights that these defects persist and require further investigation; current testing methodologies may be insufficient to detect a broad class of latent defects. Based on our findings, we infer that there is a defect in our testing status quo: behavioral code smells and other heuristics to infer the presence of defects must be researched further, lest we leave latent defects dormant.

Acknowledgements

This work was supported in part by the European Research Council (ERC) under the consolidator grant RS^3 (101045669) and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy (EXC 2092 CASA – 390781972). We thank the developers of aya-rs, the underlying software enabling our work with eBPF. In addition, we thank the developers of PCRE2, dotconf, Docker (Moby), Kubernetes, the Open Container Initiative, Golang, Firefox, Chromium, Rust, and GDB for the time and effort taken to remediate the reported issues. We would especially like to thank Philip Hazel and William Hubbs, the maintainers of PCRE2 and dotconf, respectively, who addressed our reports diligently and in a timely manner at significant personal cost. Without the support of free and open-source works and their maintainers, our research would simply not be possible. Finally, we (authors 1–4), thank the Saarbrücken Graduate School of Computer Science for their support in our doctoral programs.

Ethical Considerations

This work presents a tool that facilitates finding weaknesses in programs, which is inherently dual-use. Our work affects several stakeholders, including software developers, end users, and the broader software community. Our goal is to uncover latent bugs in Linux applications in order to improve security while minimizing potential damage. Our work was guided by the ethical principles outlined in the Menlo Report. The principle of beneficence motivated us to design the study in a way that maximizes societal and technical benefits by uncovering latent defects that could lead to failures or security vulnerabilities. Respect for persons was ensured by not involving any people directly in the study. The broader software and security community benefits from improved defect detection and system reliability. Finally, respect for law and public interest was maintained by conducting all experiments in accordance with applicable laws and organizational policies, without unauthorized access to or exploitation of software.

One potential harm is the concrete risk that the disclosure of defects could be exploited by attackers. This risk was mitigated by conducting all experiments on controlled, non-production systems and following coordinated disclosure practices for all newly identified defects. More specifically, we contacted the developers of the software projects for which OS-SANITIZER identified a defect and worked with them to fix the identified flaws. Only the flaws which have been entirely remediated or labeled as "won't fix" by the developers have been shared in this paper with technical details. In addition, we have privately shared our tool with security testing groups most affected by our tool to ensure a wide initial vulnerability search. Despite these mitigating measures, a certain residual risk remains, in particular the possibility that detailed technical descriptions could be misused by malicious actors. After weighing these risks against the benefits of improved software security, reproducibility, and community knowledge, we have decided to continue our research and publish the results, while taking precautions to mitigate any negative impact.

Open Science

To encourage future research and ensure the reproducibility of our results, we have made a complete artifact of this work available. This artifact includes the implementation of our prototype, the experimental setup, all evaluation targets, anonymized evaluation data, and the analysis scripts. The launcher implemented for OS-SANITIZER consists of about 2.5k lines of Rust code. It is typically executed as a systemd unit, running in the background continuously. The passes described in this work consist of around 2.8k lines of Rust code. The research artifact can be found on GitHub at https://github.com/os-sanitizer/os-sanitizer and Zenodo at https://doi.org/10.5281/zenodo.17979528.

By making these resources publicly available, we aim to lower the barriers to reproducibility, facilitate comparison with future techniques, and encourage further research into dynamic defect inference with eBPF. We intend to keep OS-SANITIZER available as a "living artifact" on GitHub, updating the software wherever possible to ensure it remains functional, while also preserving a static snapshot of the artifact's state at the time of acceptance on Zenodo. Researchers who seek to replicate and reproduce our work are welcomed to reach out to us if they need assistance, especially when these artifacts require maintenance. Additionally, researchers who intend to *use* our artifact are highly encouraged to request or contribute bug fixes or features that improve this work.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1), 2004.

[3] Kim Bäckström. Industrial surveys on software testing practices: A literature review. *University of Helsinki*, 2022.

[4] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. JIT-Picking: Differential fuzzing of JavaScript engines. In

ACM Conference on Computer and Communications Security (CCS), 2022.

[5] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. Accelerating Linux security with eBPF iptables. In *ACM SIGCOMM Conference on Posters and Demos*, 2018.

[6] James Bucek, Klaus-Dieter Lange, and Jóakim V. Kistowski. SPEC CPU2017: Next-Generation Compute Benchmark. In *ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2018.

[7] George Burgess, IV. The anatomy of clang fortify. https://android.googlesource.com/platform/bionic/+/refs/heads/main/docs/clang_fortify_anatomy.md, 2022.

[8] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software*, 2019.

[9] Juan Cespedes. ltrace - a library call tracer. https://ltrace.org, 2013.

[10] Oliver Chang. Rename execSan to SystemSan. https://github.com/google/oss-fuzz/pull/8369, 2022.

[11] Feng Chen, Traian Florin Serbanuta, and Grigore Rosu. jPredictor: A predictive runtime analysis tool for Java. In *International Conference on Software Engineering (ICSE)*, 2008.

[12] Cilium. BPF architecture. https://docs.cilium.io/en/latest/bpf/architecture/, 2024.

[13] Contributors from Apple, Google, Microsoft, and Mozilla. Announcing Speedometer 3.0. https://browserbench.org/announcements/speedometer3, 2024.

[14] Jonathan Corbet. Restricting path name lookup with openat2(). https://lwn.net/Articles/796868/, August 2019.

[15] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5), 2009.

[16] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[17] Daan De Meyer, Davide Calvaca, and Andrii Nakryiko. Changes/fno-omit-frame-pointer - Fedora Project Wiki. https://fedoraproject.org/wiki/Changes/fno-omit-frame-pointer, 2023.

[18] Daan De Meyer, Davide Calvaca, and Andrii Nakryiko. Format security FAQ - Fedora Project Wiki. https://fedoraproject.org/wiki/Changes/fno-omit-frame-pointer, 2023.

[19] Luca Deri, Samuele Sabella, Simone Mainardi, Pierpaolo Degano, and Roberto Zunino. Combining system visibility and security using eBPF. In *Italian Conference on Cybersecurity (ITASEC)*, 2019.

[20] Docker. Docker: Accelerated container application development. https://www.docker.com/.

[21] eBPF Foundation. eBPF: Introduction, tutorials and community resources. https://ebpf.io, 2024.

[22] William Findlay, Anil Somayaji, and David Barrera. bpfbox: Simple precise process confinement with eBPF. In *ACM SIGSAC Cloud Computing Security Workshop*, 2020.

[23] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. Fuzzing binaries for memory safety errors with QASan. In *IEEE Secure Development Conference (SecDev)*, 2020.

[24] Francesca Arcelli Fontana, Jens Dietrich, Bartosz Walter, Aiko Yamashita, and Marco Zanoni. Antipattern and code smell false positives: Preliminary conceptualization and classification. In *IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

[25] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. The Addison-Wesley Object Technology Series. Addison-Wesley, Boston, 28. printing edition, 2013.

[26] Free Software Foundation. BPFBackEnd - GCC Wiki. https://gcc.gnu.org/wiki/BPFBackEnd, 2024.

[27] Free Software Foundation. gcc/gcc/system.h - gcc-mirror/gcc. https://github.com/gcc-mirror/gcc/blob/55947b32c38a40777aedbd105bd94b43a42c2a10/gcc/system.h#L66-L77, 2024.

[28] Free Software Foundation. GDB: The GNU Project Debugger. https://www.sourceware.org/gdb/, 2024.

[29] Free(b)soft. speechd. https://github.com/brailcom/speechd, 2024.

[30] freedesktop.org. systemd-journald.service. https://www.freedesktop.org/software/systemd/man/latest/systemd-journald.service.html.

[31] freedesktop.org. NetworkManager. https://networkmanager.pages.freedesktop.org/, 2025.

[32] Poorna Gaddehosur, Dave Thaler. Making eBPF work on Windows. https://opensource.microsoft.com/blog/2021/05/10/making-ebpf-work-on-windows/, May 2021.

[33] GitHub. CodeQL documentation: Executing a command with a relative path. https://codeql.github.com/codeql-query-help/java/java-relative-path-command/, 2024.

[34] GitHub. CodeQL documentation: Use of dangerous function. https://codeql.github.com/codeql-query-help/cpp/cpp-dangerous-function-overflow/, 2024.

[35] GitHub. CodeQL documentation: Potentially overflowing call to snprintf. https://codeql.github.com/codeql-query-help/cpp/cpp-overflowing-snprintf/, 2025.

[36] Google. The Go programming language. https://go.dev/.

[37] Google. Google Chrome - the fast & secure web browser built to be yours. https://www.google.com/chrome/, 2024.

[38] Google. go/src/os/removeall_at.go at b750841906c84e894dfa3ee43e0f65d94f989b01. https://github.com/golang/go/blob/b750841906c84e894dfa3ee43e0f65d94f989b01/src/os/removeall_at.go, 2024.

[39] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. Fuzzilli: Fuzzing for JavaScript JIT compiler vulnerabilities. In *Symposium on Network and Distributed System Security (NDSS)*, 2023.

[40] Philip Hazel. PCRE - perl compatible regular expressions. https://www.pcre.org/.

[41] Philip Hazel. [pcre-dev] PCRE 8.20-RC1 is available - please test. https://lists.exim.org/lurker/message/20110912.132523.ffc4dab3.en.html, 2011.

[42] Richard Henderson. glibc/libio/iogets.c at glibc-2.41 - bminor/glibc. https://github.com/bminor/glibc/blob/glibc-2.41/libio/iogets.c.

[43] Zoltán Herczeg. Extending the PCRE library with static backtracking based just-in-time compilation support. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2014.

[44] Sabine Houy and Alexandre Bartel. Twenty years later: Evaluating the adoption of control flow integrity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 34(4):103:1–103:30, April 2025.

[45] Jeff Huang. UFO: Predictive concurrency use-after-free detection. In *International Conference on Software Engineering (ICSE)*, May 2018.

[46] William Hubbs. dotconf. https://github.com/williamh/dotconf, 2024.

[47] Jinghao Jia, YiFei Zhu, Dan Williams, Andrea Arcangeli, Claudio Canella, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, and Tianyin Xu. Programmable system call security with eBPF. *arXiv preprint arXiv:2302.10366*, 2023.

[48] Ralf Jung. cargo-careful: Execute Rust code carefully, with extra checking along the way. https://github.com/RalfJung/cargo-careful, 2025.

[49] Gary G. Koch and Stuart A. Gansky. Statistical considerations for multiplicity in confirmatory protocols. *Drug Information Journal*, 30(2):523–534, April 1996.

[50] Paul Kranenburg, Branko Lankester, and Rick Sladkey. strace - the linux syscall tracer. https://strace.io, 2025.

[51] Swati Kumar and Jitender Kumar Chhabra. Two level dynamic approach for feature envy detection. In *International Conference on Computer and Communication Technology (ICCCT)*, 2014.

[52] Joshua Levin and Theophilus A. Benson. ViperProbe: Rethinking microservice observability with eBPF. In *IEEE International Conference on Cloud Networking (CloudNet)*, 2020.

[53] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement study on Linux container security: Attacks and countermeasures. In *Annual Computer Security Applications Conference (ACSAC)*, 2018.

[54] The Linux man-pages project. *Linux man pages online*, 6.17 edition, 2025. https://man7.org/linux/man-pages/index.html.

[55] LLVM. AddressSanitizer. https://clang.llvm.org/docs/AddressSanitizer.html, 2025.

[56] LLVM. Control flow integrity. https://clang.llvm.org/docs/ControlFlowIntegrity.html, 2025.

[57] LLVM. MemorySanitizer. https://clang.llvm.org/docs/MemorySanitizer.html, 2025.

[58] LLVM. ThreadSanitizer. https://clang.llvm.org/docs/ThreadSanitizer.html, 2025.

[59] LLVM. UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2025.

[60] Leonardo Mariani. A fault taxonomy for component-based software. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 2003.

[61] Aaron Merey. Introducing debuginfod, the elfutils debuginfo server. https://developers.redhat.com/blog/2019/10/14/introducing-debuginfod-the-elfutils-debuginfo-server, 2019.

[62] Mesa 3D. The Mesa 3D Graphics Library. https://mesa3d.org/, 2024.

[63] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. Creating complex network services with eBPF: Experience and lessons learned. In *IEEE International Conference on High Performance Switching and Routing*, 2018.

[64] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu. A framework for eBPF-based network functions in an era of microservices. *IEEE Transactions on Network and Service Management*, 2021.

[65] Francesco Minna and Fabio Massacci. SoK: Run-time security for cloud microservices. Are we there yet? *Computers & Security*, 2023.

[66] MITRE. CWE-134: Use of Externally-Controlled Format String (4.14). https://cwe.mitre.org/data/definitions/134.html.

[67] MITRE. CWE - Common Weakness Enumeration. https://cwe.mitre.org/index.html, 2024.

[68] Mohamed Husain Noor Mohamed, Xiaoguang Wang, and Binoy Ravindran. Understanding the security of Linux eBPF subsystem. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*, 2023.

[69] Mozilla. Firefox for Desktop. https://www.mozilla.org/en-US/firefox/new/.

[70] Damien Neil. os: safer file open functions. https://github.com/golang/go/issues/67002, 2024.

[71] Damien Neil. path/filepath: Walk/WalkDir susceptible to symlink race. https://github.com/golang/go/issues/70007, 2024.

[72] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[73] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2010.

[74] Open Container Initiative. opencontainers/runc. https://github.com/opencontainers/runc, 2024.

[75] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development*, 5(1):7, December 2017.

[76] Jihyeok Park, Hongki Lee, and Sukyoung Ryu. A survey of parametric static analysis. *ACM Computing Surveys*, 54(7), July 2021.

[77] Mathias Payer and Thomas R. Gross. Protecting applications against TOCTTOU races by user-space caching of file metadata. In *ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, March 2012.

[78] Dimitri Prestat, Naouel Moha, Roger Villemaire, and Florent Avellaneda. Dynamics: A tool-based method for the specification and dynamic detection of android behavioral code smells. *IEEE Transactions on Software Engineering*, 50(4):765–784, 2024.

[79] Razvan Raducu, Ricardo J. Rodriguez, and Pedro Alvarez. Defense and Attack Techniques Against File-Based TOCTOU Vulnerabilities: A Systematic Review. *IEEE Access*, 10:21742–21758, 2022.

[80] Red Hat. What is SELinux? https://www.redhat.com/en/topics/linux/what-is-selinux, 2019.

[81] Rust Rustfmt WG. Rustfmt. https://rust-lang.github.io/rustfmt/, 2025.

[82] Rust Team. debug_assert in std. https://doc.rust-lang.org/std/macro.debug_assert.html, 2025.

[83] Rust Team. Miri: An interpreter for rust's mid-level intermediate representation. https://github.com/rust-lang/miri, 2025.

[84] Rust Team. Rust Programming Language. https://www.rust-lang.org/, 2025.

[85] SELinux Project. ObjectClassesPerms - SELinux Wiki. https://web.archive.org/web/20250108164533/https://selinuxproject.org/page/ObjectClassesPerms, 2013.

[86] Roland Shoemaker. os: RemoveAll susceptible to symlink race. https://github.com/golang/go/issues/52745, 2022.

[87] David Soldani, Petrit Nahi, Hami Bour, Saber Jafarizadeh, Mohammed F. Soliman, Leonardo Di Giovanna, Francesco Monaco, Giuseppe Ognibene, and Fulvio Risso. eBPF: A new approach to cloud-native observability, networking and security for current (5g) and future mobile networks (6g and beyond). *IEEE Access*, 2023.

[88] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. Exploiting and protecting dynamic code generation. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[89] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for security. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[90] Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, Nan Guan, and Yu Jiang. Finding correctness bugs in eBPF verifier with structured and sanitized program. In *European Conference on Computer Systems (EuroSys)*, 2024.

[91] Sysdig, Inc. Falco. https://falco.org, 2025.

[92] The Aya Contributors. Getting started - aya. https://aya-rs.dev/book/, May 2025.

[93] The Kubernetes Authors. Kubernetes: Production-grade container orchestration. https://kubernetes.io/.

[94] The Linux Kernel Archives. BPF maps - The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/bpf/maps.html.

[95] The Linux Kernel Archives. Kernel Address Sanitizer (KASAN) - The Linux Kernel documentation. https://www.kernel.org/doc/html/v6.9/dev-tools/kasan.html.

[96] The Linux Kernel Archives. Linux security modules - The Linux Kernel documentation. https://www.kernel.org/doc/html/v6.9/userspace-api/lsm.html, July 2023.

[97] Dave Jing Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Peter C. Johnson, and Kevin R. B. Butler. LBM: A security framework for peripherals within the Linux kernel. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[98] Viet-Hoang Tran and Olivier Bonaventure. Making the Linux TCP stack more extensible with eBPF. In *Technical Conference On Linux Networking (NetDev)*, 2019.

[99] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. Toss a fault to your Witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[100] Ben Tucker. ENV33-C. Do not call system(). https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87152177, 2023.

[101] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications. *ACM Computing Surveys*, 2021.

[102] Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger. STING: Finding name resolution vulnerabilities in programs. In *USENIX Security Symposium*, 2012.

[103] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Verifying the verifier: eBPF range analysis verification. In *International Confernce on Computer-Aided Verification (CAV)*, 2023.

[104] Jinpeng Wei and Calton Pu. TOCTTOU vulnerabilities in UNIX-style file systems: An anatomical study. In *USENIX Conference on File and Storage Technology (FAST)*, 2005.

[105] Tianjun Weng, Wanqi Yang, Guangba Yu, Pengfei Chen, Jieqi Cui, and Chuanfu Zhang. Kmon: An in-kernel transparent monitoring system for microservice systems with eBPF. In *IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence)*, 2021.

[106] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. JITfuzz: Coverage-guided fuzzing for JVM just-in-time compilers. In *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023.

[107] Mathieu Xhonneux, Fabien Duchene, and Olivier Bonaventure. Leveraging eBPF for programmable network functions with IPv6 segment routing. In *ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2018.