# VulShield: Protecting Vulnerable Code Before Deploying Patches

Yuan Li
Zhongguancun Laboratory,
Tsinghua University
lydorazoe@gmail.com

Chao Zhang†
Tsinghua University, JCSS,
Zhongguancun Laboratory
chaoz@tsinghua.edu.cn

Jinhao Zhu
UC Berkeley
jinhao.zhu@berkeley.edu

Penghui Li
Zhongguancun Laboratory
lipenghui315@gmail.com

Chenyang Li
Peking University
lcy2000@stu.pku.edu.cn

Songtao Yang
Zhongguancun Laboratory
onionyst@gmail.com

Wende Tan
Tsinghua University
twd2.me@gmail.com

***Abstract*—Despite the high frequency of vulnerabilities exposed in software, patching these vulnerabilities remains slow and challenging, which leaves a potential attack window. To mitigate this threat, researchers seek temporary solutions to prevent vulnerabilities from being exploited or triggered before they are officially patched. However, prior approaches have limited protection scope, often require code modification of the target vulnerable programs, and rely on recent system features. These limitations significantly reduce their usability and practicality.**

**In this work, we introduce VulShield, an automated temporary protection system that addresses these limitations. VulShield leverages sanitizer reports, and automatically generates security policies that describe the vulnerability triggering conditions. The policies are then enforced through a Linux kernel module that can efficiently detect and prevent vulnerability from being triggered or exploited at runtime. By carefully designing the kernel module, VulShield is capable of protecting both vulnerable kernels, and user-space programs running on them. It does not rely on recent system features like eBPF and Linux security modules. VulShield is also pluggable and non-invasive as it does not need to modify the code of target vulnerable software. We evaluated VulShield's capability in a comprehensive set of vulnerabilities in 9 different types and found that VulShield mitigated all cases in an automated and effective manner. For Nginx, the latency introduced per request does not exceed 0.001 ms, while the peak performance overhead observed in UnixBench is 1.047%.**

## I. Introduction

Security vulnerabilities are a significant threat to software systems. Every year, automated vulnerability detection methods like fuzzing [7], [8], [43] find a large number of memory corruption vulnerabilities in real-world software systems like Linux kernels and PHP interpreters. For example, in 2023, over 28,000 new vulnerabilities were identified and assigned CVE numbers, highlighting the ongoing and increasing challenges in maintaining software security [1]. Attackers can exploit these vulnerabilities to compromise the software systems, potentially leading to severe consequences such as data breaches, financial loss, etc. However, a significant number of vulnerable software remains deployed and running in the wild even after vulnerabilities have been discovered and reported [6]. This creates an attack window for malicious attackers to exploit them.

While hot-patching schemes [12], [16], [26], [31], [40], [57] have been proposed to reduce the repair time, studies show that about 25% of vulnerabilities remain unresolved even 30 days after disclosure [34]. This delay is mainly due to the need for developers to analyze vulnerable programs, identify root causes, and create patches, a time-consuming process. Alternatively, implementing a 'quick and dirty patch' can result in unintended program behavior [52]. Therefore, there is an urgent need to develop solutions to mitigate the threats in unpatched programs.

To protect vulnerable software *before* vulnerability patches are available, researchers have been seeking *temporary* protection mechanisms. Temporary protection solutions aim to mitigate the risks associated with vulnerabilities while permanent (official) patches are being developed. These solutions provide a stopgap measure to secure systems against potential exploits during the interim period and have shown great promise in the past [15], [27], [29], [55]. Prior solutions often monitor the execution of vulnerable programs and take a series of actions to remediate software vulnerabilities when they are triggered at runtime. However, existing temporary protection mechanisms fall short in three aspects:

- **C1: Limited protection scope.** They support a very limited range of vulnerabilities and programs. For instance, PET [55] can support five classes of vulnerabilities in Linux kernels, leaving many other prevalent vulnerabilities and user-space programs unprotected.
- **C2: Demand for code modification of vulnerable programs.** Most of these approaches [27], [29], [52], [55]

†Chao Zhang is the corresponding author. He is also affiliated with Joint Research Center for System Security, Tsinghua University (INSC) - Science City (Guangzhou) Digital Technology Group Co., Ltd.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240298
www.ndss-symposium.org

require inserting and compiling helper code or patches into the target vulnerable programs to achieve temporary protection. This makes it challenging to automatically scale these solutions across different software versions due to their inherent differences.

- **C3: Reliance of advanced features.** Some prior solutions [3], [56] rely on advanced features introduced recently, such as the combination of eBPF and Linux Security Modules (LSM) introduced after Linux 5.7 [22]. This reliance on advanced features makes them hardly compatible with legacy software, which remains widely used in practice [46], [47], [59].

Therefore, there is an urgent need to develop more versatile, non-invasive, and backward-compatible temporary protection mechanisms that can address a broader spectrum of vulnerabilities without requiring significant modifications to existing software. In this work, we present VulShield, a novel, fully automated approach that addresses the aforementioned limitations and offers temporary mitigation for vulnerabilities before patch releases. VulShield takes as input a sanitizer report, generates a policy that describes the vulnerability triggering conditions, and then takes a series of mitigation actions.

To resolve **C1**, VulShield leverages constraint expressions to define the triggering conditions, which could cover a wider range of vulnerability classes. This is based on the observation that security patches mostly implement security measures through the use of constraint expressions in conditional statements, as outlined in various studies [16], [39], [57], [60]. Additionally, we develop a standalone kernel module that probes the executions of both the kernel and user-space programs running on it, providing protection for both of them. This also avoids customized modification of the target vulnerable program (**C2**) and does not require any advanced system features, making VulShield completely backward compatible (**C3**).

It is worth noting that VulShield can protect already running programs on the fly without the need for recompiling or reloading. This is particularly crucial for programs with high service availability requirements, such as web servers and kernels. In summary, VulShield overcomes the limitations of previous work by providing proactive, scalable, and non-intrusive protection without hardware or other advanced dependencies. It mitigates a wide range of vulnerabilities before they are exploited without the need for additional instrumentation or manual preprocessing. VulShield's design ensures compatibility across different applications and kernel versions, offering effective and practical security for both running kernels and applications on the fly.

VulShield generates source-level policies based on source-level vulnerability information derived from sanitizer reports. The policies are then mapped to the binary level using the DWARF [19] debug information of the target binary, so that VulShield can generate the policy for vulnerable binaries without sanitizer implementation. Then VulShield enforces them using Policy Enforcer to provide protection. The policy is flexible with modularization, expressive, and extensible, well adapting to different vulnerability mitigation scenarios. The Policy Enforcer consists of three components tailored to address various vulnerabilities. 1) The *perception* component collects various program runtime information for subsequent decision and execution stages. It collects key data at specific code addresses, including memory access parameters to monitor for buffer overflows. 2) The *decision* component performs security checks at particular locations following the policy specification, and commands the subsequent stage to execute mitigation measures accordingly. Based on the information gathered from the perception component, it can make a variety of logical judgments and a comprehensive decision on the triggering condition of vulnerabilities. 3) The *execution* component takes the mitigation actions specified in the policy. For instance, this component may terminate the target user-space process when a vulnerability is triggered. Note that the mitigation actions are predefined and have limited capabilities, and thus would not introduce unexpected security risks.

We conducted a comprehensive assessment of VulShield in various aspects. We applied VulShield to a wide variety of vulnerabilities in real-world software, including user-space programs, servers, and kernels. The results showed that VulShield could effectively generate policies to protect the software from nine types of vulnerabilities. Furthermore, we performed an analysis on a large patch dataset from SecretPatch [53], showing that VulShield has high usability for temporary protection across a range of vulnerability types in principle. To evaluate the performance overhead of VulShield policies under high-frequency triggers, we used Nginx for stress testing. The results indicate that, on average, each request introduces up to 0.001 ms of latency due to the use of the policies. We further used the UnixBench benchmark suite to measure the impact of VulShield on overall system performance and tested the performance cost of combining different policies. The results show that the total performance overhead introduced by VulShield is no more than 1.047%. We also quantified the time required to generate VulShield policies. The results show that policies for kernel vulnerabilities can be generated within minutes, allowing VulShield to provide fast and effective temporary protection. We will open-source VulShield.*

In summary, we make the following contributions:

- We propose to leverage constraint expressions to describe vulnerability-triggering conditions to cover a wider range of vulnerabilities.
- We design and develop VulShield, an automated temporary vulnerability mitigation framework that can prevent vulnerabilities from being exploited or triggered. It can provide protection on the fly, without code modification or restart of the target program.
- Our evaluation demonstrated VulShield could protect more types of vulnerabilities in both kernels and user-space programs.

*https://github.com/vul337/VulShield

- VulShield exhibits negligible runtime overhead to the target programs.

## II. Background and Motivation

### A. Program Tracing

Program tracing is a method used in software development and debugging to monitor the execution of a program. It involves tracking and recording the sequence of instructions or operations performed by the program, often in real-time. There are several tools widely used for both user-space programs and OS kernels.

**Uprobe/Kprobe.** Linux Uprobe [32] and Kprobe [33] are lightweight tracing tools integrated into the Linux kernel, facilitating the tracking of function execution status for debugging and performance analysis. Uprobe (User-space Probes) enables developers to trace user-space functions specified by them without needing to recompile the binary, allowing dynamic insertion of probes into running processes and triggering user-defined handlers to collect data or perform actions. Kprobe (Kernel Probes), on the other hand, targets the kernel space by allowing the insertion of probe points into kernel functions, gathering runtime information with minimal system performance impact. Kprobes can be dynamically inserted and removed without the need for recompilation or rebooting, and are supported across multiple hardware architectures.

**eBPF.** The Berkeley Packet Filter (BPF) [44] is a tool in Linux used to capture and filter network packets, providing a powerful mechanism for network traffic analysis. The extended Berkeley Packet Filter (eBPF) [21] further enhances BPF with tracing capabilities similar to dtrace [14], enabling the execution of sandboxed programs within the kernel. eBPF allows developers and analysts to run specific functions without modifying the kernel, facilitating in-depth performance monitoring, security enforcement, and network troubleshooting. This extended functionality makes eBPF a versatile tool for real-time analysis and dynamic tracing of both user-space and kernel-space activities However, eBPF has significant privileges within the system, which can pose safety and security risks if not properly managed [41]. The ability to execute code at such a low level requires stringent validation and security measures to prevent potential misuse or vulnerabilities [38].

### B. Massive Vulnerabilities and Delayed Patching

With the success of automated vulnerability detection approaches, many new vulnerabilities are discovered on a daily basis. However, patching these vulnerabilities has become a significant problem. Many applications remain unpatched, creating a substantial attack window during which vulnerabilities can be exploited. Frei et al. [23] found that a substantial number of vulnerabilities are known to insiders before public disclosure, with over 20% being known at least 20 days in advance. Moreover, 21.2% of all CVEs were not promptly fixed after their disclosure, and over 25% remained unresolved even after 30 days [34]. Notably, about 30% of Windows vulnerabilities were not patched at the time of disclosure, with some remaining unpatched for over 180 days [23]. This introduces the risks of both 0-day and 1-day vulnerability exploitation. A fundamental reason for the prevalent use of unpatched software is that official vulnerability patches are not developed or released in a timely manner. Patching vulnerabilities is primarily a time-consuming process that demands manual analysis of the vulnerable application to develop a patch. The situation is even worse in today's software supply chain. For instance, a vulnerability found in the upstream Linux kernel might also influence downstream Android kernels from OEM vendors such as Samsung. These downstream kernels often undergo heavy modifications or customizations for their own products, further complicating and delaying the patching process.

> Due to the prolonged patch release time and the large number of routinely discovered vulnerabilities, there is an urgent need to design *automated temporary solutions* to prevent unpatched vulnerabilities from being triggered or exploited before patch releases. Such solutions could provide protection during the attack window from vulnerability discovery to patch applications, safeguarding the software and its users.

### C. Existing Temporary Solutions

Existing works often monitor the execution of vulnerable programs and take a series of actions to remediate software vulnerabilities when they are triggered at runtime. We summarize the most relevant works in Table I. Some works [27], [29] use the Security Workarounds for Rapid Response (SWRR) mechanism to insert a return instruction with an error-handling code at the entry of the vulnerable function based on the inherent error-handling mechanism of the program. Some others [3], [55] utilize eBPF [21] to provide temporary protection for vulnerabilities. Additionally, InstaGuard [15] relies on hardware features (breakpoint and watchpoint) to monitor the target process. As introduced in Section I, existing works present significant limitations that hinder their usability and practicality.

**C1: Limited protection scope.** As we mentioned earlier, PET [55] can support five classes of vulnerabilities in Linux kernels, including integer underflow and overflow, out-of-bound access, use-after-free, uninitialized access, and data race, it cannot support prevalent vulnerabilities class such as null pointer dereference and other undefined behavior bugs like divide-by-zero. It does not support errors in user-space programs. The policies of eBPFGuard [3] are LSM hook scoped, but lacks the capability to provide instruction-level vulnerability detection and exploitation prevention. InstaGuard [15] can only provide temporary protection for some kernel system programs and is limited by the number of detection points due to the constraints of hardware breakpoints; SWRR-based solutions [27], [29] are unable to protect applications without well-designed error-handling codes.

Bowknot [52] only undoes the effects of kernel bugs after they are triggered and cannot proactively protects against vulnerabilities.

**TABLE I:** Comparison with existing works. ✓: Yes; ✘: No.

| Approach | Category | Patch Independence | Protection Scope | No Code Mod. | No Adv. Features | Automation |
|---|---|---|---|---|---|---|
| VULMET [57] | Hot Patch | ✘ | Same as Official Patch | ✓ | ✓ | ✓ |
| KARMA [16] | Hot Patch | ✘ | Same as Official Patch | ✓ | ✓ | ✓ |
| RapidPatch [26] | Hot Patch | ✘ | Same as Official Patch | ✘ | eBPF | ✘ |
| SWRR-based tools [27], [29] | Temporary Fix | ✓ | Error-Code-Returning Functions | ✘ | ✓ | ✓ |
| InstaGuard [15] | Temporary Fix | ✓ | 7 Vul. Types in Kernel System Programs | ✘ | Hardware breakpoints/watchpoints | ✓ |
| eBPFGuard [3] | Temporary Fix | ✓ | LSM Hooks Scope | ✓ | eBPF + LSM | ✘ |
| Bowknot [52] | Temporary Fix | ✓ | Undoes the Effects of Kernel Bugs Post-trigger | ✘ | ✓ | ✘ |
| PET [55] | Temporary Fix | ✓ | 5 Vul. Types in Linux Kernels | ✘ | eBPF + bpf_send_signal | ✘ |
| VulShield | Temporary fix | ✓ | 9 Vul. Types in Linux Kernels and User-Space Programs | ✓ | ✓ | ✓ |

**C2: Demand for code modification** Most of prior approaches [27], [29], [52], [55] require modification of the target vulnerable programs, through methods like recompilation and reloading. PET, in particular, requires modifications to the kernel as it relies on a set of customized BPF helper functions. InstaGuard [15] requires a library to be loaded in each monitored process. SWRR-based tools [27], [29] use configuration and also need to insert code to the vulnerable functions. This need for modification adds complexity and can introduce new potential points of failure. To maintain the kernel's functionality when bugs are triggered, Bowknot adds extra code as a workaround to the kernel.

**C3: Reliance of advanced features.** Prior solutions [3], [15], [56] rely on advanced system and hardware features such as the combination of eBPF and LSM that are not always available. InstaGuard [15] requires hardware features (breakpoint and watchpoint) to monitor the target process, which is not always feasible as well.

**Difference against hot patching.** Hot patching is a relevant technique that can temporarily fix vulnerabilities. However, its main goal is to ensure continuous software operation without rebooting the software. It often involves dynamically injecting or modifying code in the running program, e.g., loading new code into memory and redirecting function calls to the updated code. However, hot patching assumes the availability of the official patches [16], [26], [57] and provides protection based on the released patches. *This work distinguishes itself from hot patching solutions by neither assuming the availability of a patch nor requiring the presence of one.* On the other hand, poorly designed hot patches, if not applied carefully, may introduce new security issues such as new execution paths or code permissions, posing a risk of misuse [30], disrupt system functionality, or cause unpredictable behavior [52].

### *D. A Motivating Example*

CVE-2021-42008 is an out-of-bounds access vulnerability within the Linux kernel that has been present in the Linux kernel for 16 years, affecting multiple versions prior to 5.13.13. The vulnerability would trigger a slab out-of-bounds writing in the `decode_data` function in lines 6, 7, and 8, as shown in Listing 1. Developing a patch to fix the vulnerability needs to consider all three locations, where inexperienced developers might fail to cover all and a temporary protection tool is still helpful. Existing temporary protection solutions could not sufficiently handle this vulnerability. SWRR-based approaches only allow the `decode_data` function to return immediately, thus disabling its normal operational functionality and affecting related kernel features. InstaGuard is incapable of providing temporary protection for the kernel. eBPFGuard can only perform checks at the LSM hooks level and fails to detect the vulnerability effectively. PET requires features (i.e., `bpf_send_signal`) introduced in newer kernel versions (e.g., version 5.3). It thus could not protect the vulnerable code before kernel version 5.3 due to the lack of features. This is often infeasible in practice. In addition, PET performs detection at the appropriate vulnerability trigger point based on the sanitizer report and lacks data flow analysis. Therefore, the temporary protection provided by PET for the vulnerability is likely to cover only one of the vulnerability trigger points.

```
1  static void decode_data(struct sixpack *sp,
       unsigned char inbyte)
2  {
3    unsigned char *buf;
4  ....
5    buf = sp->raw_buf;
6    sp->cooked_buf[sp->rx_count_cooked++] = buf[0] |
       ((buf[1] << 2) & 0xc0);
7    sp->cooked_buf[sp->rx_count_cooked++] = (buf[1]
       & 0x0f) | ((buf[2] << 2) & 0xf0);
8    sp->cooked_buf[sp->rx_count_cooked++] = (buf[2]
       & 0x03) | (inbyte << 2);
9    sp->rx_count = 0;
10 }
```

**Listing 1:** The vulnerable code of CVE-2021-42008.

## III. Insights

In this work, we address the above-mentioned limitations and design VulShield. In this section, we present the key insights.

**Constraint expressions for describing vulnerability conditions.** We reviewed research on security patch characterization and summarized two key features: 1) patches are often small [53], and 2) they mostly use constraint expressions in conditional statements [16], [39], [57], [60]. These expressions encode logical constraints about program variables. For example, to fix a buffer overflow, a patch would enforce an access offset within the buffer's size, e.g., 0 <= offset < size (more details in Appendix A). Approximately 70% of patches for memory corruption vulnerabilities rely on logical expressions, using variables, operands, and logical operators for vulnerability detection and resolution [60]. On the other hand, vulnerability sanitizers (e.g., ASan [45] and UBSan [18]) have demonstrated the ability to detect vulnerabilities of

common vulnerability types using their detection logic. This in return also suggests that we are able to similarly capture and detect a vulnerability at runtime for temporary protection.

**A kernel module using basic probes.** To ensure VulShield's wide applicability, it only uses basic features present in both recent and legacy software. VulShield utilizes basic program tracing features for OS kernels and user-space programs to monitor program states. Combined with predefined constraint expressions, VulShield can detect vulnerabilities at runtime. We implement this through a standalone kernel module that can be pre-installed in the system. This module can be flexibly enabled on the fly, providing protection for both kernel and user-space programs, including those already running.

## IV. Methodology

### A. Overview

To effectively mitigate unpatched vulnerabilities, we propose VulShield, which implements pre-installed functional components with information-gathering functions, generic detection logic, and typical actions, to shield multiple vulnerability exploits by constructing the inputs to the functional components. We use VulShield policies to construct these inputs. Specifically, VulShield policies are used to describe where and what information needs to be collected how to use that information to perform a security check for multiple vulnerabilities, and how to shield the vulnerability exploitation. By decomposing the VulShield policy into individual entries for each functional component and flexibly combining and setting the entries, the VulShield policy has sufficient expressiveness for multiple vulnerability types.

Figure 1 shows the overall system framework of VulShield. The policy generator is used to automatically generate policies. Furthermore, the policy enforcer, which resides in the kernel space of the target device, enforces the policies and blocks security vulnerabilities. To ensure the security of the communication, there is a userland process that receives the policies and forwards them to the policy enforcer. Specifically, VulShield analyzes the related source code files (or LLVM Bitcode File) and error reports generated by sanitizers. This process involves conducting both vulnerability and data flow analyses to determine the vulnerability type and choose the existing templates to generate a source-level mitigation policy. Additionally, if using the expressions-based policy templates, VulShield evaluates and generates the appropriate detection points and corresponding expressions (such as the function entry point or return points). The policy generated at this point is at the source code level and the target machine may be in production environments without sanitizers. So, VulShield utilizes such source-level policies combined with debug information from the target program to generate the binary-level policy of the target binary, including the specific binary address and register information. If the target binary lacks DWARF debug information, developers can either pre-compile binaries with identical configurations or install the relevant kernel-debug package.

For the threat model, we assume that target applications and kernels are benign but may contain exploitable vulnerabilities. As a common practice [55], we assume the attackers are under unprivileged user permissions since privileged users like root already bypass most mitigation. Besides, considering the possibility of attackers tampering with forged policies, we use a signature verification mechanism for the policies used. The developer signs the policies, and the policy enforcer verifies the policies, thus ensuring that the policy enforced by the module is not a malicious policy forged or tampered with by an attacker. After the policy has been generated, signed, and distributed to each target device, the userland process forwards it to the kernel-state policy enforcer. After verification, the policy enforcer extracts specific information from the validated policy and uses this information to enforce various policies. We also assume the authorized developers are trusted. During the transmission from the developer to the VulShield enforcer module, policies are signed by the authorized developers, ensuring that only authorized developers can use VulShield to generate a valid policy. Subsequently, prior to enforcing any policy, the enforcer validates both the authenticity and integrity of the policy. Consequently, we can reasonably assume that attackers are unable to exploit VulShield by installing malicious policies. Note that VulShield's automatic policy generation relies on bug reports from sanitizers. In real-world scenarios, developers typically perform vulnerability analysis and remediation based on these reports. Therefore, we assume that developers can obtain the appropriate sanitizer bug reports when they discover a vulnerability or are notified by third-party security experts.

### B. Design Requirements

For security reasons, the capabilities of the policy enforcer should be restricted to prevent abuse of the VulShield. Thus, VulShield should be prevented from executing potentially dangerous operations and meet the following requirements:

- **Restricted memory write.** The policies of VulShield should not possess the capability to write to memory. Any write operation could lead to unintended behaviors such as tampering with critical data.
- **Restricted code inclusion.** The policies of VulShield should not support dynamic code injection or parsing of executable code, in order to prevent abuse. This might open an attack surface to attackers.
- **Restricted execution altering.** In detecting anomalous behavior, VulShield should restrict the execution of the program rather than freely altering the code execution path.
- **Support the normal execution.** For kernels, stopping them at random can be much more costly, so it is also a challenge to be concerned about guaranteeing the proper execution of such cases. To ensure normal kernel operation, VulShield can use the following strategies: (1) check the function parameters at function entry and return the appropriate error-handling codes; (2) check error-handling codes returned by the function on return; (3) utilize the error-handling basic

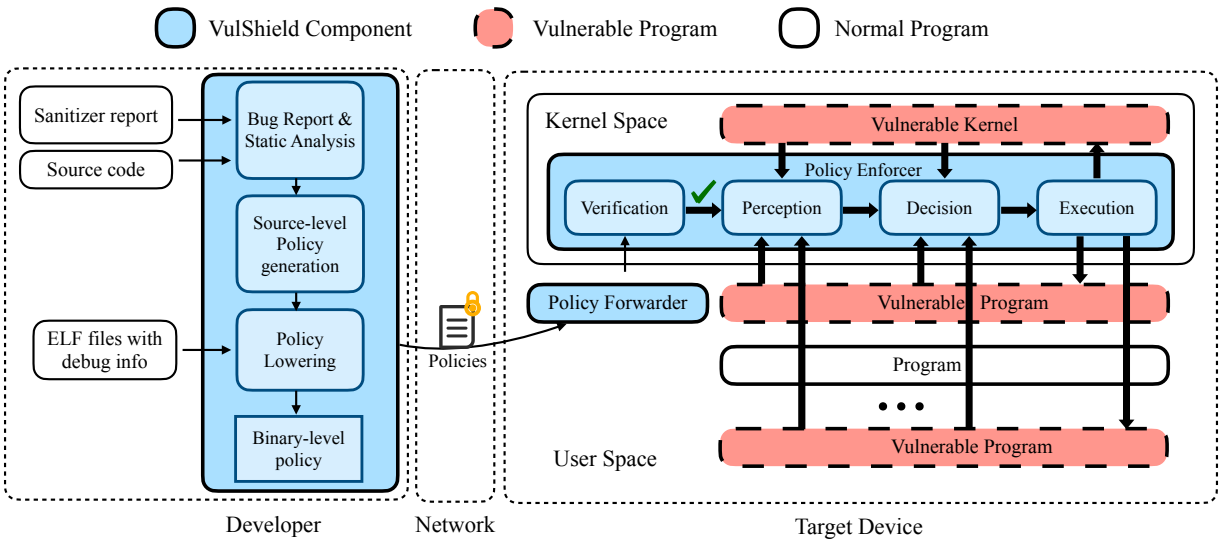


**Figure 1:** Overview of VulShield.

block in the vulnerable function; or (4) use mechanisms that do not alter the control flow and data flow.

- **Restricted information printing.** Despite being able to read memory, to prevent information leakage, VulShield should not print or output any information other than warnings.

In subsequent sections, we will describe in detail how VulShield can be used as an effective vulnerability shield when the above conditions are met.

### C. VulShield Policy

To effectively shield vulnerability exploits while conforming to the requirements listed in § IV-B, we present the VulShield policies that can collect sufficient runtime information to perform safety checks for various types of vulnerabilities. List. 2 demonstrates an example of VulShield policy with a detailed description of each component as follows.

- **Perception points.** Perception point is optional in the policy. There could be no, one, or multiple perception points depending on the vulnerabilities. `offset` specifies the address where the perception point is inserted. Each perception point must include at least one of `value` and `counter`. `value` contains two entries: `val` represents the register/memory value we need to collect at the perception point and `name` is a label representing this value, which we'll refer to later in the `Constraint` section. `counter` is specified when dealing with temporal vulnerabilities. `flag` is a two bits value that represents: no counter used (00), counter decremented (01), and counter incremented (10). `num` is the increment/decrement value each time the perception point is reached and `name` is the label of this counter.
- **Decision points and associated information.** The decision point is required for every policy. Similar to the perception point, the decision point contains the `offset`, and optional `value` entry(s). When the target program reaches the decision point, the constraint will be checked and the mitigation action will be enforced if the constraint satisfies.
- **Constraint expressions.** The constraint expressions section consists of variables, constants, logical judgments, arithmetic operators, and a small number of whitelisted functions (e.g., `strlen`). Note that if the mitigation action of the policy is a safe and efficient mitigation mechanism, the expression can be simply `true`, which means the mitigation action will always be executed.
- **Mitigation actions.** Mitigation actions are divided into two categories. The first category involves behavior chosen when an exception is found, such as hanging the program, interrupting it, issuing a warning, or returning an error-handling code. The other category is safe and efficient mitigation mechanisms, such as utilizing the QS (Quarantine and Sweeper) mitigation, specified in § IV-E3.

### D. Policy Generator

VulShield can parse bug reports generated from sanitizers such as ASan. Subsequently, it generates policies automatically. This capability empowers VulShield to rapidly furnish users with provisional mitigation.

To make it easier to understand the policy design of VulShield, this section describes how to generate the required security policy for the motivating example (CVE-2021-42008). To simulate the situation when a vulnerability is initially discovered, we compiled Linux with AddressSanitizer (KASan) and utilized a Proof of Concept (PoC) to capture the call stack when the vulnerability is detected. Initially, VulShield is capable of parsing the bug report produced by ASan to identify the nature of the vulnerability, additional vulnerable information—encompassing pertinent variables—and the original line of code where the vulnerability resides. Subsequently, VulShield engages in a meticulous static analysis of the vulnerable function's source code, to identify the perception point, and decision point at the source level and formulate the corresponding mitigation measures. Then VulShield needs

to translate the source-level policy to adapt to the target `vmlinux`. Thereafter, VulShield employs the debugging information from the `vmlinux` to map the relevant source lines—such as the decision point—to specific fragments of binary code. Ultimately, VulShield applies the Capstone to disassemble these particular binary code fragments, thereby pinpointing the exact binary address associated with the decision point.

To make it easier to address common vulnerability types and further reduce the burden on developers, we automate the policy generation process by adapting each policy to the specific vulnerability characteristics. For instance, in the case of a buffer overflow vulnerability, we examine whether the access instruction employs an out-of-bounds pointer. A detailed overview of policy designs for distinct vulnerability types can be found in Table II.

*1) Source-level Policy Generation:* VulShield is designed to automatically analyze bug reports from sanitizers in order to generate security policies. For user-space programs, error reports from user-space sanitizers typically encompass detailed source code information, including the specific line where a crash occurs. VulShield can efficiently leverage this information to conduct an in-depth analysis of the pertinent source code details. In contrast, for kernel-space programs, error reports from kernel sanitizers often include only binary-level information. To address this, VulShield employs the `faddr2line` or `eu-addr2line` tools to analyze and extract the corresponding source code information. Following this initial analysis, VulShield proceeds to statically analyze the relevant source code files (or LLVM bitcode files) alongside the error reports generated by the sanitizers.

Based on these analyses, VulShield identifies the vulnerability type and gathers information about related variables. Then, VulShield evaluates whether a source-code-level mitigation policy can be generated using existing templates. Additionally, if the mitigation policy involves expression-based detection, and developers need to ensure the normal operation of the target program—such as in cases of buffer overflow in the kernel—VulShield will employ SVF (Static Value-Flow Analysis Framework) [49] to perform analysis on the vulnerable function containing the targeted vulnerability. This analysis generates the expressions used to detect vulnerabilities and evaluates whether the expressions can be effectively placed at the function entry or return points. Taking motivation example (i.e., CVE-2021-42008) as an example, UBSan reports an error of "array-index-out-of-bounds". VulShield first confirms that the index is the value of the `sp` sub-object according to the `use-def` chain, and confirms the offset of the sub-object in the `sp` structure. In addition, VulShield utilizes SVF to perform data flow analysis on the index and analyze the changes in its value in the function. Taking this function as an example, the index (i.e., `sp->rx_count_cooked`) is incremented twice before the last use as the index in the function. Therefore, if the vulnerability is detected at the beginning of the function, it is necessary to compare `sp->rx_count_cooked+2` with the maximum value of the array instead of `sp->rx_count_cooked`. In instances where the expression instrumental in detecting the vulnerability cannot be efficaciously positioned at the function's entry or return point, VulShield proceeds to analyze the exit basic block in the function (such as the `FunExitICFGNode` in SVF). It analyzes to ascertain the optimal detection point for the expression and identifies the optimal function exit basic block dominated by this detection point. This includes but is not limited to, basic blocks about the function's internal exception-handling mechanisms (e.g., `goto error`).

The description of source-level policy as illustrated in List. 3: 1) The source file's path, the name of the decision point, and the specific line number. 2) A precise definition of the constraints, comprising variable names, constants, operators, logical evaluations, and a limited set of whitelisted functions calls for constraint formulation. 3) Pertinent details concerning impacted variables, constants, and counters are to be supplied. For variables, it is essential to offer the source details of their derivation. Furthermore, specific information about the variable at that perception point is necessary; for instance, whether it serves as a function call parameter at

```
---
Perception point:
  - Offset: 0xdead
    Value:
      - {Val: reg, Name: val1}
    Counter:
      - {Flag: 10, Num: 1, Name: counter1}
  - Offset: 0xbeef
    Value:
      - {Val: offset+reg, Name: val2}
    Counter:
      - {Flag: 01, Num: 1, Name: counter2}
Decision point:
  - Func: func1
    Offset: 0xaaaa
    Value:
      - {Val: [reg+offset], Name: val3}
      - {Val: offset+[reg], Name: val4}
Constraint: (val1+val2)*val3==func1(val4)
Mitigation actions: hang/kill/warning/...
---
```

**Listing 2:** The components of a policy.

```
---
VulType: array-index-out-of-bounds
Variables:
  - {Name: rx_count_cooked, Type: sub-object,
    Index:10, ObjType: S.struct.sixpack, Value: {
    Name: sp, Type: pointer, Value: 0th parameter}
    }
Constraint: sp->rx_count_cooked + 2>=400
Decision point:
  - {FuncName: decode_data, Type: func_entry,
    Source: drivers/net/hamradio/6pack.c:832}
Mitigation action: return
---
```

**Listing 3:** The source-level policy of CVE-2021-42008.

that specific source location. Concerning constants, precise numeric data is required. In the case of counters, information concerning their addition or subtraction within the source and the counter's index (representing its corresponding value) are mandatory. For example, the counter index recording heap allocations and releases corresponds to the heap variable's base address. 4) The developer's selection of the mitigation action.

*2) Policy Lowering for Binary-level Policy :* In real-world application scenarios, the vulnerable application and kernel are without sanitizers. To adapt to the target vulnerable application or kernel, VulShield needs to translate the source-level policies to binary level accordingly. Therefore, the policy that is read and enforced by the policy enforcer on the target device should be binary-level information, as shown in List. 4.

The generator parses the debug information of the binary and translates the vulnerability details from the source level to the binary level. First, it maps the source code location to the binary addresses. However, because the source code location information may not have a one-to-one mapping to binary addresses, one line of source code may correspond to multiple sections of binary code. As a result, second, the generator analyzes the target binary code fragment and further verifies the binary address of the decision point based on the provided vulnerability information. For inline functions, the generator performs a thorough analysis of the call stack information to identify the function containing the decision point. It then refines its search for the code fragment containing the decision point based on the source information of the inline function's call point in the call stack. In some cases, certain functions may be inlined multiple times, resulting in multiple candidate code fragments for the decision point.

### *E. Policy Enforcer*

The module begins by receiving and verifying the signature of the forwarded policy. Once verified, the module enforces the designated policy, collecting required data from applications or the kernel. This data is then analyzed to detect any potential attacks. If a potential attack is identified, the policy enforcer activates a suitable mitigation action to prevent the vulnerability from being triggered. The enforcer consists of three core components: perception, decision, and execution, each of which is detailed in the following subsections.

*1) Perception:* For certain vulnerabilities, only the runtime information at the decision point is necessary to ascertain the possibility of an attack, such as null pointer dereference

```
---
Decision point:
  - Func: decode_data
    offset: 0x0
    value:
      - {val: [rdi+0x1cc], name: count}
Constraint: count+ 2>=400
Mitigation action: return
---
```

**Listing 4:** The binary-level policy of CVE-2021-42008.

or division by zero. However, more intricate vulnerabilities like heap buffer overflows, use after free, and race conditions necessitate the amalgamation of data from multiple runtime execution points to reach a valid determination. This requires VulShield to gather process runtime information before the decision points.

**Value.** For vulnerabilities such as heap overflows, VulShield only needs to collect the value of registers or the value stored at specific memory addresses when the program is executed at a specific address.

**Counter.** For some vulnerabilities, such as double free and race conditions, VulShield needs to collect state information. For example, it needs to know whether the memory accessed was freed for UAF (Use After Free). Therefore, for this type of vulnerability, we use a counter scheme. For example, VulShield assigns a counter to the target buffer for a UAF vulnerability. When memory is allocated, the counter is incremented by 1. When memory is freed, the counter is decremented by 1. And VulShield will check the target memory counter at the memory access point to see if the counter is 0, thus determining if a UAF vulnerability is triggered at that point.

*2) Decision:* The observation has shown that constraints can be used to represent the detection logic. Such constraints typically consist of variables, constants, operators, logical judgments, and a small number of whitelisted function calls (such as `strlen`). Buffer overflows, out-of-bounds accesses, division by zero, NULL pointer dereference, integer overflows, and partial logic vulnerabilities can all be represented by such constraints. Therefore, most of the vulnerability detection in this paper is performed by the mitigation action of such constraints. Suppose a buffer overflow of a user-state program generates a policy based on the vulnerability type. In this case, we will insert the decision point before the vulnerability trigger point, check the relationship between the value of the target register and the target memory object, and interrupt or hang the program execution or issue a warning if the boundary is crossed. Some mitigation actions achieve mitigation through trick implementations, such as utilizing the QS (Quarantine and Sweeper) mitigation. In many cases, such mitigation actions do not need constraints to detect anomalous behavior. Therefore, for policies that use such mitigation actions, if they do not require conditional execution using constraints, their constraint results are true by default.

*3) Execution:* Note that the perception component focuses solely on reading data, while the decision component returns a boolean variable as a detection result for potential attacks. These components work together to identify potential security vulnerabilities. In the execution component, VulShield employs the mitigation action, which comes in two types: actions for constraining program execution and actions for mitigation.

**Program execution restriction.** For common vulnerabilities like buffer overflows and division by zero, maintaining regular program functioning without relying on error-handling codes necessitates modifying program control and data flow, which could inadvertently introduce security risks. VulShield

**TABLE II:** Security policy design for common vulnerabilities. The Common Weakness Enumeration (CWE) is a classification list of weaknesses and vulnerability types. The design details of Quarantine and Sweeper (QS) and Delayed Execution until Completion (DEC) are in Section IV-E3.

| Shield Mechanism | Vulnerability Type | | CWE No. | Shield policy | |
|---|---|---|---|---|---|
| | | | | vulnerable source | Shield Template |
| Expression-based Detection | null pointer dereference | | CWE-476 | *p | $p \notin [0, PageSize)$ |
| | divide-by-zero | | CWE-369 | $a \div b$ | $b \neq 0$ |
| | buffer overflow | buffer... | CWE-121<br>CWE-122<br>CWE-124<br>CWE-787 | *p | p+sizeof(p)<base(p)+size(p)<br>&& p>base(p) |
| | | array... | CWE-125<br>CWE-129<br>CWE-119<br>CWE-787 | *(base(p)+offset) | 0<=offset<size(p) |
| | | memcpy/strcpy/... | CWE-119<br>CWE-120 | memcpy(p,q,s) | p>=base(p) &&<br>p+s<=base(p)+size(p) |
| | integrity overflow | | CWE-190 | a op b | min=<a op b <=max |
| | use-after-free | | CWE-416 | malloc(p);free(p);use(p) | account(malloc(p))==account(free(p))+1 |
| | double free | | CWE-415 | free(p);free(p) | account(malloc(p))==account(free(p)) |
| | other vulnerability | | ..... | assert(C) | C |
| Other Mitigation | use-after-free | | CWE-416 | Quarantine+Sweeper | |
| | double free | | CWE-415 | Quarantine+Sweeper | |
| | race condition | | CWE-362 | Action Delay until Completion | |

addresses this by limiting program execution to avoid vulnerability triggering. For generic programs, the options VulShield provides to developers are to kill the process and to hang the process. In the case of kernels, they tend to have a robust error-handling mechanism and need to ensure the proper operation of the program. Therefore, for such cases, VulShield can set the decision point to the function entry or return point and use its error-handling mechanism to ensure the program's normal operation. Furthermore, VulShield is also endowed with the capability to goto the error-handling basic block within the vulnerable function, thereby ensuring the maintenance of the program's normal operation. Developers can also extend VulShield to choose the same mitigation action for programs with a robust error-handling mechanism (such as Nginx) as the kernel. Unlike Talos [27], VulShield will only return error-handling codes when it detects a potential attack and will not affect the functionality of the target function. Also, for both cases, VulShield provides the option of issuing warnings, from which developers can choose.

**Other mitigation actions.** For temporal memory vulnerabilities or race conditions, a combination of expression constraints and counter schemes is effective for detection. However, high-frequency functions must consider performance overhead. Therefore, VulShield also provides optional mitigation actions that do not alter program control or data flow, delivering practical mitigation with reasonable overhead.

- *Quarantine and Sweeper.* According to the analysis and summary of PUMM [58], an attacker can only exploit the UAF vulnerability if the target memory region is reallocated. Therefore, delaying the deallocation of the target memory region is an effective mitigation. ASan [45] proposes the quarantine mechanism, a list array to manage and delay the release of memory objects about to be deallocated. Quarantine and Sweeper (QS), inspired by MarkUs [10] and MineSweeper [20], uses a quarantine zone with a sweeper to effectively address temporal memory vulnerabilities. When an object is deallocated, it is first placed in the quarantine zone rather than being released immediately, ensuring that the object is not released prematurely. The sweeper is responsible for a memory sweep to determine whether there are pointers still pointing to each object in the quarantine zone, especially when the number of objects awaiting processing reaches a predetermined threshold. If no pointer still points to an object, then this object is deallocated. Notably, a linear scan of the memory is performed by an independent thread, which increases efficiency. When the number of objects to be processed exceeds a predefined threshold, the sweeper runs at a fixed frequency. When the number of objects exceeds twice the threshold, the sweep frequency is doubled. Both the threshold and frequency are configurable, allowing users to tailor the system to their specific needs. In the default configuration, the threshold is set to a quarter of the maximum quarantine size and the frequency is set to every 10 seconds. This adaptive approach reduces the risk of system performance degradation due to excessive accumulation of unprocessed objects. When the number of objects queued for processing reaches the quarantine's maximum capacity, and these objects continue to be referenced by dangling pointers, VulShield implements a policy to randomly release some of these objects. The policy is to reduce the risk of distributed denial of service (DDoS) attacks and Heap Feng Shui. At the same time, the system generates an alert warning of potential DDoS attacks or Heap Feng Shui [54]. Upon receipt of this alert, it is imperative that maintenance personnel perform a manual effort to assess and address the specific circumstances. In addition, each execution of deallocation, especially `kfree`, initiates a check to confirm whether the target object is already in quarantine. If the object is found in quarantine, this triggers an alert for a potential double-free vulnerability.
- *Delayed Execution until Completion.* A race condition

vulnerability is triggered when multiple program flows interact incorrectly. Attackers wishing to exploit such concurrency bugs often need to manipulate execution sequences through the exploit carefully. Completion is a mechanism embedded within the Linux kernel that facilitates inter-thread communication, enabling one thread to notify another of the completion of a specified event. VulShield adopts the idea of Completion to mitigate race conditions. This is achieved by enforcing the sequential execution of events, wherein an event that is scheduled to occur subsequently is only initiated following the successful completion of a preceding event. If the preceding event does not finish, the subsequent execution is deferred. To prevent potential deadlocks, VulShield introduces a timeout. Specifically, if the timeout is reached and the expected event is not completed, VulShield activates a predefined mitigation action. This predefined action is responsible for implementing user-specified actions, such as emitting a warning or terminating the target process, thus ensuring the system's robustness and responsiveness.

## V. System Implementation

The Policy Generator leverages sanitizer error reports to extract vulnerability-related information, then utilizes `pyelftools` for parsing and analyzing target ELF files and DWARF debug information. It maps the source code locations to binary code segments based on information from the `.debug_line` section. Additionally, `Capstone` is employed to pinpoint the specific binary addresses of decision points. The Policy Enforcer is implemented as a kernel module that utilizes KProbe and Uprobe to gather runtime information, enforce security checks, and implement developer-specified mitigation actions. VulShield applies policies using KProbe and Uprobe, which require the security policy to include relative offsets of decision points. It analyzes the target ELF file using `pyelftools` to extract program headers and thereby determine the relative offset of the target address. When calculating the internal offsets of structures, discrepancies arise between the actual and theoretical offsets due to memory alignment. Consequently, VulShield utilizes the '-fdump-record-layouts-simple' flag to dump layout information during source code analysis, facilitating subsequent analysis of the actual offsets within structures.

## VI. Evaluation

This section evaluates the VulShield prototype in terms of security and performance and answers the following questions:

- **RQ1:** Does VulShield introduce any security risk?
- **RQ2:** Is the policy of VulShield sufficiently expressive, and is the enforcer flexible enough to effectively mitigate a wide range of vulnerability types?
- **RQ3:** What is the overhead introduced by VulShield?
- **RQ4:**How long does VulShield take to automatically generate a security policy?

### A. Security Risk Analysis

**Fake policy.** Suppose an attacker tries to exploit VulShield's mitigation on user processes or kernels. To prevent such attacks, VulShield's enforcer verifies the signature of each policy before applying it. The threat model assumes that only policies signed by authorized developers can be successfully verified, effectively thwarting unauthorized attempts.

**Incorrect policy.** VulShield streamlines the policy generation process by automatically extracting and translating vulnerability conditions from sanitizer reports into precise expression constraints. So the correctness and effectiveness of these policies are inherently dependent on the sanitizer's false positive rate. Fortunately, modern sanitizers like ASan have zero false positive rates [35]. While VulShield can restrict program execution, it does not introduce new code or execution paths. This non-intrusive approach preserves the original functionality and stability of the system. Even in the case of a false positive, VulShield does not introduce any new security risks. The impact of such a scenario is comparable to the mitigations like Talos, which are generally acceptable.

**Conflicting policy.** VulShield configures unique probes for each policy, ensuring that each policy operates independently without interfering with others. During the execution of a program protected by VulShield, multiple security policies are applied sequentially. If a bug's trigger requires an index integer that exceeds a value that causes another overflow, VulShield will intercept and address the first potential vulnerability. It then proceeds to check for subsequent vulnerabilities. This means that even if the application has multiple layered vulnerabilities that are interconnected, VulShield systematically evaluates and enforces policies for each one.

### B. Capability Analysis

**Environments and test cases.** Our security tests are run in a virtual machine, and the kernel tests are performed using QEMU. Since VulShield's automatic policy generation relies on bug reports from sanitizers, the only criterion for selecting vulnerabilities for testing is the availability of a public proof-of-concept (PoC) that can be effectively detected by a sanitizer. Since the Magma test suite is all real-world vulnerabilities with public PoCs, VulShield uses Magma's public PoC to filter CVEs that can be detected by ASan or UBSan with effective bug reports. In addition, we have made efforts to reproduce more vulnerabilities and obtain bug reports to further evaluate the effectiveness of VulShield. In the end, we successfully reproduced 32 vulnerabilities with valid bug reports, covering a variety of vulnerabilities, including those in user-space programs, service applications, and the kernel. To collect more vulnerability information, KASan uses multi-shot mode. During initial testing of CVE-2013-2028, ASan did not report call stack information. We have improved ASan to output call stack information.

**Results.** The results of our evaluations are detailed in Table III, which demonstrates the effectiveness of VulShield in addressing a comprehensive range of vulnerability types. These results confirm that VulShield, using sanitizer bug

**TABLE III:** Security test results with 32 CVEs and 9 vulnerability types.

| Target Binary | | CVE No. | Vulnerability Type | Policy Type | Is it effective? |
|---|---|---|---|---|---|
| Magma | libpng | CVE-2013-6954 | NULL Pointer Dereference | Expression-based | ✓ |
| | libtiff | CVE-2016-10270 | Out-of-bounds Access | Expression-based | ✓ |
| | | CVE-2019-7663 | Invalid Address dereference | Expression-based | ✓ |
| | libxml | CVE-2017-9047 | Buffer Overflow | Expression-based | ✓ |
| | | CVE-2016-1836 | Use After Free | Expression-based | ✓ |
| | openssl | CVE-2016-2108 | Buffer Underflow | Expression-based | ✓ |
| | | CVE-2016-6309 | Use After Free | Expression-based | ✓ |
| | | CVE-2017-3735 | Out-of-bounds Access | Expression-based | ✓ |
| | php | CVE-2018-14883 | Out-of-bounds Access | Expression-based | ✓ |
| | | CVE-2019-11034 | Out-of-bounds Access | Expression-based | ✓ |
| | poppler | CVE-2019-14494 | Divide By Zero | Expression-based | ✓ |
| | | CVE-2019-9959 | Integer Overflow | Expression-based | ✓ |
| | | CVE-2019-10873 | NULL Pointer Dereference | Expression-based | ✓ |
| | | CVE-2019-10872 | Out-of-bounds Access | Expression-based | ✓ |
| | | CVE-2019-7310 | Out-of-bounds Access | Expression-based | ✓ |
| | | CVE-2018-13988 | NULL Pointer Dereference | Expression-based | ✓ |
| | | CVE-2018-10768 | NULL Pointer Dereference | Expression-based | ✓ |
| | | CVE-2017-14617 | Divide By Zero | Expression-based | ✓ |
| | sqlite3 | CVE-2015-3414 | NULL Pointer Dereference | Expression-based | ✓ |
| libIEC61850 | | CVE-2018-18957 | Buffer Overflow | Expression-based | ✓ |
| LibreDWG | | CVE-2020-21813 | Buffer Overflow | Expression-based | ✓ |
| Nginx | | CVE-2013-2028 | Integer Overflow | Expression-based | ✓ |
| gpac | | CVE-2020-19488 | NULL Pointer Dereference | Expression-based | ✓ |
| libxml2 | | CVE-2021-3518 | Use After Free | Expression-based | ✓ |
| Linux kernel | | CVE-2017-18344 | Out-of-bounds Access | Expression-based | ✓ |
| | | CVE-2021-42008 | Out-of-bounds Access | Expression-based | ✓ |
| | | CVE-2022-2588 | Use After Free | QS | ✓ |
| | | CVE-2021-3492 | UAF+Double Free+Mising Release | QS | ✛ |
| | | 6dc9ae7 [50] | Use After Free | QS | ✓ |
| | | ca4463b [36] | Race condition+UAF | QS | ✓ |
| | | a834b99 [51] | Data race | DEC | ✓ |

✓: effectively detect and prevent the triggering of reported vulnerabilities. ✛: only support partial the reported vulnerabilities

reports, can reasonably formulate robust policies, obviating the need for extensive vulnerability analysis by developers. However, it was observed that one automatically generated policy cannot cover all vulnerabilities reported by sanitizer. For CVE-2021-3492, the ASan bug report included vulnerability information related to double free and use-after-free, while the Kmemleak only reported an allocation point where the object had no associated free, requiring manual effort by developers to accurately identify the needed release point. In addition, for CVE-2019-9959, ASan reported an "allocation-size-too-big" error, while UBSan reported no issues. As a result, the generated policy effectively detected the issue in the ASan report and killed the target process. CVE-2013-2028 also has only ASan bug reports and no UBSan bug reports, but VulShield can provide robust protection through analyzing the ASan bug reports. ca4463b [36] is a use-after-free vulnerability triggered by race conditions. According to the KASan bug report, the use of the QS execution mechanism can prevent use-after-free from occurring, providing effective mitigation. While Table III only evaluates 9 vulnerability types, the effectiveness of VulShield is not limited to these specific types. Vulnerabilities that can be effectively detected using constraint expressions, such as logical vulnerabilities, can also be mitigated using VulShield. This demonstrates the flexibility and adaptability of VulShield in addressing a wider range of vulnerability scenarios than those explicitly evaluated.

### C. Usability Analysis

We further assess the usability of VulShield to understand how many security patches can potentially be mitigated using VulShield. We analyzed the patches of 1,458 vulnerabilities provided by SecretPatch [53]. We are not able to apply VulShield to all vulnerabilities/patches in the dataset because it is not affordable to configure their compiling or running environments, necessary for deploying VulShield. Therefore, we choose to manually analyze the patches and check if they can be supported by VulShield. To this end, we examine the code patterns corresponding to the capability for expression in VulShield. For example, we manually identify the patches that use `if` statements for conditional assessments, followed by either returning, invoking exception handling functions, or goto the blocks within the same function for exit. We also check patches that log information before performing jumps or calls as the output of log information is ancillary to the processes of vulnerability detection and handling. We then correlate these patches to the different categories based on the vulnerability types. The results demonstrated that VulShield has a high usability. In particular, VulShield supported 825 (56.6%) out of 1,458 vulnerabilities in the dataset. This includes 690 vulnerabilities corresponding to the vulnerability types listed in Table II and 135 vulnerabilities in the other 26 vulnerability types, delineated in Table IV. *It is worth highlighting that all the cases in the 26 vulnerability types could not be supported*

**TABLE IV:** Result of security patch analysis.

| Vulnerability Type | CVE No. | Vulnerability Type | CVE No. |
|---|---|---|---|
| CWE-20 | 32 | CWE-287 | 2 |
| NVD-CWE-Other | 21 | CWE-617 | 2 |
| CWE-264 | 13 | CWE-255 | 1 |
| CWE-189 | 12 | CWE-320 | 1 |
| CWE-200 | 9 | CWE-404 | 1 |
| CWE-399 | 8 | CWE-682 | 1 |
| CWE-834 | 6 | CWE-704 | 1 |
| NVD-CWE-noinfo | 5 | CWE-74 | 1 |
| CWE-269 | 3 | CWE-770 | 1 |
| CWE-284 | 3 | CWE-824 | 1 |
| CWE-388 | 3 | CWE-835 | 1 |
| CWE-400 | 3 | CWE-89 | 1 |
| CWE-22 | 2 | CWE-94 | 1 |

**TABLE V:** The performance overhead of Nginx is evaluated based on the parameters `n` (total number of requests) and `c` (number of concurrent requests).

| Nginx | w/o VulShield | | w/ VulShield | |
|---|---|---|---|---|
| | Time per request | | Time per request | |
| | a concurrent group | a single request | a concurrent group | a single request |
| n=100000, c=10 | 0.130 ms | 0.013 ms | 0.140 ms | 0.014 ms |
| n=100000, c=100 | 1.313 ms | 0.013 ms | 1.411 ms | 0.014 ms |
| n=100000, c=1000 | 15.075 ms | 0.015 ms | 15.827 ms | 0.016 ms |
| n=100000, c=10000 | 136.712 ms | 0.014 ms | 136.238 ms | 0.014 ms |

*by the previous work* [15], [55]. Note that we made our manual analysis conservative and the actual supported vulnerabilities could be even higher than the presented numbers. These results not only underscore the scalability potential of VulShield, but also provide empirical evidence in support of the resolution of **RQ2**, highlighting VulShield's exceptional ability to protect against vulnerabilities.

### *D. Performance Analysis*

In order to answer **RQ3**, we test the performance of VulShield on an Inspiron 3910 Compact Desktop running the Linux 5.15.0 kernel with the Ubuntu 20.04.1 release.

*1) Stress Test:* To test the performance of VulShield under high-frequency triggering, we used CVE-2013-2028 in Nginx 1.4.0 to showcase the overhead introduced by VulShield for applications in real-world scenarios.

**Test inputs and configuration.** In the performance evaluation, we compiled Nginx with the default options. The vulnerability is located in `ngx_unix_recv` function and we use the Apache `ab` tool to send requests to reach it. Note that to avoid the impact of network latency, the requests were sent from the same machine as the Nginx server. Since we need to evaluate the performance overhead of VulShield when Nginx is functioning under normal conditions without any active attacks, we carefully prepared the request data so that it would only execute the vulnerable code but would not trigger the vulnerability. In our evaluation, we sent a total of 100000 requests with different concurrency to assess the impacts of VulShield under realistic scenarios of concurrent request handling. Specifically, we included four groups in the concurrency of 10, 100, 1000, and 10000, respectively. To enhance experimental accuracy, we conducted each test case 10 times, and the average was taken as the final result. This aimed to minimize the impact of any potential fluctuations or anomalies in the results.

**Results.** The results of the Nginx tests are shown in Table V. As the level of concurrency increases, the difference in request times between systems with and without VulShield decreases. It reveals that VulShield is not adversely affected by program concurrency and that the performance impact of VulShield decreases progressively. Furthermore, by examining the average time cost per request, it is observed that the latency introduced by VulShield per request is at most 0.001ms. Note that in real-world situations with the network latency between the client and the server, a request would take a much longer time, e.g., 1500 ms [2], thus we believe the below 0.001 ms overhead introduced by VulShield is negligible. At a concurrency level of 10,000, the latency introduced by VulShield per request is negligible. Considering these results, it is clear that the performance overhead caused by VulShield is minimal. Even when dealing with high-risk vulnerabilities in the service program, the overhead caused by VulShield is negligible and has no significant impact on the normal execution of the program.

**Concurrency.** We simulated up to 10,000 concurrent requests to mimic a high-load, multi-threaded environment. The results demonstrated that VulShield effectively supports the concurrent application of policies across multiple threads, maintaining stability and performance even under extreme conditions. This confirms that VulShield can reliably enforce concurrent policies simultaneously in real-world scenarios.

*2) System-Wide Performance Impact:* We use UnixBench [25] to test the performance impact of VulShield for the kernel operation. In this experiment, the policy of CVE-2017-18344 represents the use of expression-based detection, the policy of CVE-2022-2588 represents the use of the QS (Quarantine and Sweeper), and the policy of a834b99 [51] represents the use of the DEC (Delayed Execution until Completion). In addition, this solution simultaneously measures the performance overhead under multiple policy combinations. This experiment uses UnixBench's default options for testing, i.e., directly using the `./Run -c 1` command for testing. By default, UnixBench runs several rounds of experiments and averages the results. We use the configuration of 20 CPUs in the system and 1 parallel copy of tests. Additionally, we utilize the `numactl` command to bind each execution run to the same CPU core.

**Results.** The results of the performance tests are presented in Table VI. The table lists the scores for each sub-item under each scenario and performance overhead. Based on the scores of each case, it can be judged that the performance overhead is negligible when no security policy is implemented or when a policy based on expression detection is implemented. When implementing a single policy, QS has a greater performance

**TABLE VI:** Performance testing based on UnixBench.

| Test Items | 0 policy | | 1 policy | | | 2 policies | | | 3 policy |
|---|---|---|---|---|---|---|---|---|---|
| | w/o VulShield | w/ VulShield | Expression-based | QS | DEC | Expression-based + QS | Expression-based + DEC | QS + DEC | Expression-based + QS + DEC |
| Dhrystone 2 using register variables | 6502.0 | 6436.1 | 6411.4 | 6419.2 | 6416.2 | 6413.2 | 6364.9 | 6311.3 | 6419.9 |
| Double-Precision Whetstone | 2028.8 | 2029.6 | 2028.8 | 2029.2 | 2030.0 | 2029.0 | 2030.3 | 2028.8 | 2030.2 |
| Execl Throughput | 2262.6 | 2270.1 | 2272.2 | 2261.6 | 2267.8 | 2257.4 | 2263.9 | 2263.5 | 2266.3 |
| File Copy 1024 bufsize | 7101.5 | 7097.5 | 7016.9 | 7086.5 | 7060.4 | 7063.7 | 7054.6 | 7111.6 | 6995.6 |
| File Copy 256 bufsize | 4687.1 | 4717.2 | 4723.8 | 4709.9 | 4729.6 | 4722.2 | 4728.0 | 4715.8 | 4718.3 |
| File Copy 4096 bufsize | 12954.1 | 12467.8 | 11974.8 | 11915.7 | 12513.3 | 12031.6 | 11824.6 | 12317.2 | 11646.3 |
| Pipe Throughput | 3356.5 | 3354.4 | 3369.3 | 3379.8 | 3361.5 | 3381.5 | 3366.6 | 3365.0 | 3397.7 |
| Context Switching | 1277.8 | 1293.6 | 1293.0 | 1297.2 | 1290.0 | 1290.0 | 1290.7 | 1293.4 | 1284.7 |
| Process Creation | 2612.1 | 2665.5 | 2632.6 | 2642.2 | 2642.7 | 2584.2 | 2542.9 | 2597.7 | 2612.6 |
| Shell Scripts (1 concurrent) | 4338.5 | 4344.3 | 4348.1 | 4340.8 | 4334.4 | 4332.4 | 4333.8 | 4327.6 | 4337.6 |
| Shell Scripts (8 concurrent) | 4054.0 | 4058.4 | 4058.5 | 4052.8 | 4053.8 | 4045.4 | 4044.5 | 4017.0 | 4043.3 |
| SysCall Overhead | 2478.1 | 2478.4 | 2476.9 | 2478.4 | 2479.6 | 2477.2 | 2478.6 | 2476.7 | 2471.6 |
| System Benchmarks Index Score | 3685.7 | 3684.3 | 3665.4 | 3667.3 | 3679.7 | 3659.6 | 3647.1 | 3662.6 | 3651.2 |
| Overhead | - | 0.038% | 0.551% | 0.499% | 0.163% | 0.708% | 1.047% | 0.627% | 0.936% |

**TABLE VII:** The latency of critical operations.

| Policy Type | Action | Time |
|---|---|---|
| Expression-based Detection | Expression Judgment | 1138 ns |
| | Counter ++ | 130 ns |
| | Counter – | 104 ns |
| Quarantine& Sweeping | Put into quarantine | 704 ns |
| | Sweep (Total 13919700 KB) | 2265ms |
| Delayed Execution until Completion | Init Completion | 435 ns |
| | Set Completion | 428 ns |
| | Timeout For Completion | User-defined (default=1000us) |

**TABLE VIII:** Policy Generation Time.

| Example | Step | Time |
|---|---|---|
| CVE-2021-42008 | Bug report Analysis | 0.046s |
| | Static Analysis | 1.143s |
| | Policy Lowering | 8m41.869s |
| 6dc9ae7 [50] | Bug report Analysis | 0.068s |
| | Policy Lowering | 6m53.238s |

overhead than DEC. The performance overhead of composite policies is not significantly different from that of individual policies. Among the various sub-items, `Dhrystone` and `File Copy` have the largest performance impact. The overall average performance overhead is extremely low, with the largest performance overhead being 1.047%. The test results illustrate the lightweight, efficient, and flexible multi-policy combination of VulShield.

*3) Microbenchmark:* In order to more accurately determine the performance overhead caused by VulShield, this work sets up a microbenchmark test. Specifically, the measured target is executed 10 times, and the median of all measurements is selected as the test result. To obtain accurate measurement results, this work refers to the measurement method using TSC [9] and uses `tsc_khz` to convert the number of clock cycles into time units (such as nanoseconds and milliseconds) to obtain more intuitive results.

**Results.** As shown in Table VII, the mitigation action is remarkably lightweight, which enhances its applicability in real-time systems. The expression-based detection mechanism includes three actions: counter addition and subtraction, and expression-based detection. The operations related to counter addition and subtraction are particularly efficient, requiring only about 100 nanoseconds. In contrast, expression-based detection, which requires the manipulation of multiple variables, takes longer, about 1138 nanoseconds. Despite the relative increase, this duration remains within acceptable limits for timely processing. VulShield uses the Quarantine and Sweeping (QS) mechanism to address temporal memory safety violations, with two main actions. The first action verifies the presence of memory objects in quarantine to detect duplicate frees before placing them in quarantine. Then the sweeping action scans the entire memory space, which, according to the statistics, is 13,919,700 KB in total. The sweep process is initiated only when the amount of space to be released reaches a pre-defined threshold and is executed in a separate thread to minimize disruption. Although the execution time for this action is 2265 milliseconds, the performance overhead is considered acceptable in the context of the overall system operation. In addition, this work uses the Delayed Execution until Completion (DEC) mechanism to mitigate the race condition. The Init Completion and Set Completion actions effectively act as lock and unlock operations, respectively. The timeout threshold for the Wait For Completion function is a user-defined parameter, which in this paper is set to 1000 microseconds by default. This flexibility allows for customized settings that optimize the balance between security measures and system performance.

### *E. Generation Time Analysis.*

We further showcase the time demanded for generating policies. In general, VulShield could automatically generate policies effectively within several minutes. To validate it, we selected CVE-2021-42008 to represent the generation of policies based on expression and 6dc9ae7 to represent the generation of policies using other mitigation. We then use the Linux `time` command to measure the policy generation time. The test results are shown in Table VIII. 6dc9ae7 does not require analysis of expressions, so there is no static analysis

phase. The time to analyze the sanitizer reports is less than 0.1 ms, and since the static analysis is only in the target function, it is completed within a few seconds. In addition, since both vulnerabilities are kernel-related, VulShield use `pyelftools` parses the kernel's ELF and debug information, which adds some time cost. Thus, the translation from source-level policy to binary-level policy is the most time-consuming, taking several minutes. Moreover, the policy for CVE-2021-42008 incurs greater time demand due to the need to analyze and obtain the actual offset of the target object within the structure during the policy lowering. Because the kernel contains more information and is considered more complicated, parsing its ELF and debug information takes longer than what is typically required for most user-space programs. As a result, VulShield generates policies for user-space programs more quickly. These experimental results indicate that the time cost for VulShield to automatically generate policies is manageable, which has significant benefits in temporary protection.

## VII. Discussion

**Scalability of VulShield.** As analyzed in Section VI-C, the underlying framework of VulShield is capable of supporting a broader range of vulnerabilities. However, the detection capabilities of existing sanitizers limit the types of vulnerabilities for which VulShield can automatically generate policies. Developers who wish to extend VulShield's support for additional vulnerabilities can manually compose source-level policies, thereby achieving a semi-automated policy generation process. Alternatively, VulShield could be adapted to support more sanitizers, thereby enhancing its ability to automatically generate policies. In summary, the potential of VulShield for temporary protection is positive and promising.

**Policy generation limitations.** When the sanitizer report is incomplete, such as one lacking crucial details like call stack information, VulShield cannot generate a policy. Sanitizer reports provide detailed insights into detected vulnerabilities, including the exact locations in the source code where issues occur and the conditions under which they are triggered. This information is necessary for VulShield to generate a policy.

**Future work.** VulShield provides temporary protection against vulnerability exploitation based on sanitizer reports. The solution might not completely stop all possible ways of triggering the vulnerability if not covered in the sanitizer reports. However, the goal of VulShield is not to completely fix a vulnerability— official patch releases should do that—but to automatically and quickly provide vulnerability remediation. In the future, we plan to integrate VulShield with automated vulnerability root cause analysis tools [13] to generate more comprehensive temporary protection policies.

## VIII. Related work

**Live kernel patching.** Several live patching tools have been proposed by industry engineers for the Linux kernel. For instance, Ksplice [11] and kpatch [5] apply patches at the instruction or function level, while kGraft [4] maintains both vulnerable and patched versions of target functions and dynamically selects which version to execute. However, using these tools requires developers to determine the appropriate hot patch, which is unnecessary in the case of VulShield.

**Patch migration.** Some work has focused on adapting official patches to different kernel or application versions. For example, KARMA [16] and VULMET [57] generate adaptive source or binary patches for different versions. patchDroid [40] uses a dynamic code injector to apply binary patches in memory. RapidPatch [26] uses eBPF for device-specific patches. VulShield does not focus on the same problem; it can mitigate vulnerabilities before official patches are released.

**Automatic detection and repair.** Some research efforts attempt to automatically detect and repair certain vulnerabilities. First-Aid [24] is designed to detect memory bugs in applications and can generate and apply runtime patches for such memory bugs. DIRA [48] focuses primarily on detecting and repairing control flow hijacking attacks. It removes the associated control flow hijacking vulnerabilities from the program by recompiling it. These studies are limited to detecting and fixing specific vulnerabilities and are not intended to temporarily mitigate real-world vulnerabilities. First-Aid primarily verifies the consistency effect of generated patches and generates diagnostic reports to accelerate patch development by developers. DIRA, with a 25% performance overhead in its repair process, is better suited for identifying vulnerabilities in development rather than providing temporary protection. Thus, these works are more focused on streamlining the development of official patches, which is outside the scope of VulShield.

**Input filtering.** Another line of work tries to filter or alter the input that can trigger the vulnerability. Bouncer [17] implements input filters by combining static analysis and dynamic symbolic execution to filter malicious inputs; VSEF [42] is a similar idea to prevent malicious attacks by filtering out specific inputs. SOAP [37] is an automatic input rectification system designed specifically for overflow vulnerabilities. These studies mainly target vulnerabilities activated by function inputs. However, certain complex vulnerabilities, such as Use-After-Free (UAF) and data races, cannot be prevented by merely filtering function inputs. Additionally, changing function inputs could lead to unforeseen side effects. Therefore, VulShield avoids altering function inputs, concentrating on detecting and blocking the exploitation of vulnerabilities.

**Temporary protection.** InstaGuard [15] determines whether an attack exists by combining the data collected by the watchpoints and kills the process or logs the attack if detected. VulShield is similar to InstaGuard in that it supports temporary protection of vulnerabilities by providing patching policies rather than patching code directly. However, InstaGuard requires the target application to load the libraries in advance and therefore increases the attack surface besides failing to protect applications already running. Besides, InstaGuard's detection points are limited in the number of hardware breakpoints, resulting in limited policies that InstaGuard can deploy. VulShield does not rely on hardware breakpoints or watchpoints, allowing it to scale efficiently and monitor numerous processes and threads. Talos [28] and RVM [29] avoid the execution

of vulnerable functions by returning with an error code, regardless of whether this vulnerability is triggered or not. Unlike these two works, VulShield only uses the error handling mechanism when a potential attack is detected. Bowknot [52] undoes the effects of kernel bugs after they're triggered. Unlike Bowknot, VulShield proactively protects against vulnerabilities in both userspace programs and the kernel before exploitation. Bowknot also requires additional code instrumentation and manual preprocessing that VulShield does not. PET [55] uses kernel sanitizer reports to identify the conditions that trigger vulnerabilities of the targeted kernel, applying eBPF during runtime to monitor and react if these conditions occur. Compared to PET, VulShield offers broader coverage, addressing additional vulnerability types, including those in user space, which PET cannot cover. Moreover, PET's dependence on eBPF's newer version limits its applicability. PET also requires custom BPF helper functions that require modification of the kernel source code, which prevents its deployment on kernels already in use. VulShield operates without kernel modifications and can be applied across different kernel versions.

## IX. Conclusion

VulShield provides a fast and efficient temporary solution to mitigate vulnerability exploitation. VulShield utilizes policies for effective mitigation. It comprises a policy generation engine that rapidly generates effective mitigation policies and a runtime mitigation component that efficiently detects and prevents vulnerability exploitation during runtime processes. Our experiments show that VulShield can effectively mitigate at least 9 different types of vulnerabilities. Specifically, for Nginx, each request introduces up to 0.001 ms of latency, and the maximum performance overhead of UnixBench is 1.047%. These results show that VulShield can effectively shield against various vulnerabilities without interfering with the normal execution of applications and kernels.

## Acknowledgement

We would like to thank our shepherd and anonymous reviewers for their insightful comments and feedback. This work was supported, in part, by the National Natural Science Foundation of China (U24A20337), National Key R&D Program of China (2021YFB2701000), the Joint Research Center for System Security, Tsinghua University (Institute for Network Sciences and Cyberspace) - Science City (Guangzhou) Digital Technology Group Co., Ltd., and HUAWEI Technologies.

## References

[1] "2023 cve data review," https://jerrygamblin.com/2024/01/03/2023-cve-data-review/.

[2] "How long does an http request take?" https://decadecity.net/blog/2012/09/15/how-long-does-an-http-request-take.

[3] "Introducing ebpfguard: A library for inline mitigation of threats using lsm hooks," https://www.deepfence.io/blog/ebpfguard-a-library-for-inline-mitigation-of-threats/.

[4] "kgraft: Live patching of the linux kernel." https://documentation.suse.com/sles/12-SP5/html/SLES-kgraft/index.html.

[5] "kpatch: Dynamic kernel patching," https://www.redhat.com/zh/blog/introducing-kpatch-dynamic-kernel-patching/.

[6] "Open reports in syzbot," https://syzkaller.appspot.com/upstream/open.

[7] "American fuzzy lop," https://github.com/google/AFL, 2014.

[8] "syzkaller - kernel fuzzer," https://github.com/google/syzkaller, 2016.

[9] (2022) Measuring time. [Online]. Available: https://cseweb.ucsd.edu/classes/wi22/cse221-a/timing.html

[10] S. Ainsworth and T. M. Jones, "Markus: Drop-in use-after-free prevention for low-level languages," *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 578–591, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:212425000

[11] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *Proceedings of the 4th ACM European conference on Computer systems*, 2009, pp. 187–198.

[12] M. T. Azim, I. Neamtiu, and L. M. Marvel, "Towards self-healing smartphone software via automated patching," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 623–628.

[13] T. Blazytko, M. Schlögel, C. Aschermann, A. Abbasi, J. Frank, S. Wörner, and T. Holz, "{AURORA}: Statistical crash analysis for automated root cause explanation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 235–252.

[14] B. Cantrill, M. W. Shapiro, A. H. Leventhal *et al.*, "Dynamic instrumentation of production systems." in *USENIX Annual Technical Conference, General Track*, 2004, pp. 15–28.

[15] Y. Chen, Y. Li, L. Lu, Y.-H. Lin, H. Vijayakumar, Z. Wang, and X. Ou, "Instaguard: Instantly deployable hot-patches for vulnerable system programs on android," in *2018 Network and Distributed System Security Symposium (NDSS'18)*, 2018.

[16] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei, "Adaptive android kernel live patching," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1253–1270.

[17] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado, "Bouncer: Securing software by blocking bad input," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 117–130.

[18] L. Developers, "Undefined behavior sanitizer," 2017.

[19] DWARF Committee. (2023) Dwarf debugging information format. [Online]. Available: https://dwarfstd.org/

[20] M. Erdős, S. Ainsworth, and T. M. Jones, "Minesweeper: a "clean sweep" for drop-in use-after-free prevention," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 212–225.

[21] M. Fleming, "A thorough introduction to ebpf," *Linux Weekly News*, vol. 3, 2017.

[22] Frederick Lawler. Live-patching security vulnerabilities inside the linux kernel with ebpf linux security module. [Online]. Available: https://blog.cloudflare.com/live-patch-security-vulnerabilities-with-ebpf-lsm/

[23] S. Frei, M. May, U. Fiedler, and B. Plattner, "Large-scale vulnerability analysis," in *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, 2006, pp. 131–138.

[24] Q. Gao, W. Zhang, Y. Tang, and F. Qin, "First-aid: surviving and preventing memory management bugs during production runs," in *Proceedings of the 4th ACM European Conference on Computer systems*, 2009, pp. 159–172.

[25] Google. byte-unixbench. [Online]. Available: https://code.google.com/archive/p/byte-unixbench/

[26] Y. He, Z. Zou, K. Sun, Z. Liu, K. Xu, Q. Wang, C. Shen, Z. Wang, and Q. Li, "{RapidPatch}: Firmware hotpatching for {Real-Time} embedded devices," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2225–2242.

[27] Z. Huang, M. DAngelo, D. Miyani, and D. Lie, "Talos: Neutralizing vulnerabilities with security workarounds for rapid response," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 618–635.

[28] ——, "Talos: Neutralizing vulnerabilities with security workarounds for rapid response," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 618–635.

[29] Z. Huang and G. Tan, "Rapid vulnerability mitigation with security workarounds," in *Proceedings of the Workshop on Binary Analysis Research (BAR'19)*, 2019.

[30] Isaac Casanova. (2017) Apple rolls back roll-out.io. [Online]. Available: https://medium.com/@isaacacasanova/apple-rolls-back-rollout-io-6a9a6cd9702f

[31] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, "Detecting past and present intrusions through vulnerability-specific predicates," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 91–104, 2005.

[32] J. Keniston and S. Dronamraju, "Uprobes: User-space probes," *Linux Foundation Collaboration Summit*, 2010.
[33] J. Keniston, P. S. Panchamukhi, and M. Hiramatsu, "Kernel probes (kprobes)," *Documentation provided with the Linux kernel sources (v2. 6.29)*, 2016.
[34] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2201–2215.
[35] Y. Li, W. Tan, Z. Lv, S. Yang, M. Payer, Y. Liu, and C. Zhang, "Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1901–1915.
[36] linux kernel. (2020) Vt_disallocate freeing in-use virtual console. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ca4463bf8438b403596edd0ec961ca0d4fbe0220
[37] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard, "Automatic input rectification," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 80–90.
[38] H. Lu, S. Wang, Y. Wu, W. He, and F. Zhang, "Moat: Towards safe bpf kernel extension," *arXiv preprint arXiv:2301.13421*, 2023.
[39] S. A. Mokhov, M.-A. Laverdiere, and D. Benredjem, "Taxonomy of linux kernel vulnerability solutions," in *Innovative Techniques in Instruction Technology, E-learning, E-assessment, and Education*. Springer, 2008, pp. 485–493.
[40] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda, "Patchdroid: Scalable third-party security patches for android devices," in *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013, pp. 259–268.
[41] National Vulnerability Database. (2021) Cve-2021-3490. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-3490
[42] J. Newsome, D. Brumley, D. Song, J. Chamcham, and X. Kovah, "Vulnerability-specific execution filtering for exploit prevention on commodity software." in *NDSS*, 2006.
[43] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "parmesan: Sanitizer-guided greybox fuzzing," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.
[44] J. Schulist, D. Borkmann, and A. Starovoitov, "Linux socket filtering aka berkeley packet filter (bpf)," *Documentation/networking/filter. txt*, 2018.
[45] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, 2012, pp. 309–318.
[46] R. Shariffdeen, X. Gao, G. J. Duck, S. H. Tan, J. Lawall, and A. Roychoudhury, "Automated patch backporting in linux (experience paper)," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 633–645.
[47] Y. Shi, Y. Zhang, T. Luo, X. Mao, Y. Cao, Z. Wang, Y. Zhao, Z. Huang, and M. Yang, "Backporting security patches of web applications: A prototype design and implementation on injection vulnerability patches," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1993–2010.
[48] A. Smirnov and T.-c. Chiueh, "Dira: Automatic detection, identification and repair of control-hijacking attacks." in *NDSS*. Citeseer, 2005.
[49] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*. ACM, 2016, pp. 265–266.
[50] syzbot. (2023) Kasan: use-after-free read in madvise_update_vma. [Online]. Available: https://syzkaller.appspot.com/bug?id=6dc9ae7ccfcf2d2e5237d4e68f1c9f63e866d0ef
[51] ——. (2023) Kcsan: data-race in netlink_getname / netlink_insert (4). [Online]. Available: https://syzkaller.appspot.com/bug?id=a834b993b63ed43938194af3accb08c0a5042877
[52] S. M. S. Talebi, Z. Yao, A. A. Sani, Z. Qian, and D. Austin, "Undo workarounds for kernel bugs," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2381–2398.
[53] X. Wang, K. Sun, A. Batcheller, and S. Jajodia, "Detecting" 0-day" vulnerability: An empirical study of secret security patch in oss," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 485–492.
[54] Y. Wang, C. Zhang, Z. Zhao, B. Zhang, X. Gong, and W. Zou, "{MAZE}: Towards automated heap feng shui," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1647–1664.
[55] Z. Wang, Y. Chen, and Q. Zeng, "{PET}: Prevent discovered errors from being triggered in the linux kernel," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4193–4210.
[56] ——, "{PET}: Prevent discovered errors from being triggered in the linux kernel," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4193–4210.
[57] Z. Xu, Y. Zhang, L. Zheng, L. Xia, C. Bao, Z. Wang, and Y. Liu, "Automatic hot patch generation for android kernels," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 2397–2414.
[58] C. Yagemann, S. P. Chung, B. Saltaformaggio, and W. Lee, "PUMM: Preventing Use-After-Free using execution unit partitioning," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 823–840. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/yagemann
[59] S. Yang, Y. Xiao, Z. Xu, C. Sun, C. Ji, and Y. Zhang, "Enhancing oss patch backporting with semantics," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2366–2380.
[60] L. Zhao, Y. Zhu, J. Ming, Y. Zhang, H. Zhang, and H. Yin, "Patchscope: Memory object centric patch diffing," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 149–165. [Online]. Available: https://doi.org/10.1145/3372297.3423342

## APPENDIX

Wang et al. [53] characterized the security patches in open-source software (OSS). They revealed that security patches tend to modify a smaller amount of code compared to non-security patches. Additionally, security patches were observed to have a higher likelihood of modifying operators and operands.

Mokhov [39] identified that approximately 25% of Linux kernel security patches addressed vulnerabilities by ensuring the validity of function input parameters. Moreover, over 18% of security patches involve checking the error status of certain functions, i.e., the return value. These observations align with similar findings from VULMET [57] and KARMA [16], which propose that most Linux kernel security vulnerabilities stem from malicious inputs, and protection can be achieved through filtering these inputs. Furthermore, PatchScope [60] found that 43.5% of official patches for such vulnerabilities utilize security checks similar to dynamic vulnerability detection mechanisms. Additionally, 25.1% of official patches modify existing security checks to address the vulnerabilities. These security checks can often be expressed using expressions, resulting in approximately 70% of official patches for such vulnerabilities being expressible in this manner.

The extensive analysis of security patches outlined above indicates the central role played by expressions in delineating security patches for the majority of vulnerabilities.

**TABLE A.9:** Security patch analysis details.

| Vulnerability Type | CVE No. | Vulnerability Type | CVE No. |
|---|---|---|---|
| CWE-20 | CVE-2011-2518, CVE-2012-2136, CVE-2012-6647, CVE-2012-6696, CVE-2013-1763, CVE-2013-1819, CVE-2013-4254, CVE-2013-4587, CVE-2013-6380, CVE-2013-7015, CVE-2014-0038, CVE-2014-1874, CVE-2014-4611, CVE-2014-8323, CVE-2014-8324, CVE-2014-9584, CVE-2015-3288, CVE-2015-5685, CVE-2015-7509, CVE-2016-4809, CVE-2016-5351, CVE-2016-6515, CVE-2016-9390, CVE-2017-1000252, CVE-2017-14169, CVE-2017-14230, CVE-2017-15868, CVE-2017-5226, CVE-2017-6837, CVE-2018-10087, CVE-2018-14361, CVE-2018-8050 | CWE-287 | CVE-2016-7144, CVE-2016-7145 |
| NVD-CWE-Other | CVE-2012-1013, CVE-2013-2130, CVE-2013-4265, CVE-2014-3631, CVE-2014-9491, CVE-2015-4692, CVE-2015-7566, CVE-2015-8543, CVE-2015-8630, CVE-2015-8812, CVE-2016-2186, CVE-2016-2187, CVE-2016-2188, CVE-2016-3136, CVE-2016-3137, CVE-2016-3138, CVE-2016-3140, CVE-2016-3689, CVE-2016-4817, CVE-2016-4951, CVE-2017-7273 | CWE-617 | CVE-2017-9499, CVE-2018-15822 |
| CWE-264 | CVE-2011-0006, CVE-2011-0989, CVE-2011-1477, CVE-2011-2495, CVE-2011-4080, CVE-2012-2319, CVE-2013-0268, CVE-2013-1774, CVE-2013-6383, CVE-2014-9922, CVE-2015-2686, CVE-2015-9004, CVE-2016-10318 | CWE-255 | CVE-2011-4966 |
| CWE-189 | CVE-2011-2496, CVE-2012-2383, CVE-2012-2384, CVE-2012-2673, CVE-2012-2674, CVE-2012-2675, CVE-2013-6367, CVE-2014-0791, CVE-2014-3587, CVE-2015-4167, CVE-2016-2070, CVE-2016-3135 | CWE-320 | CVE-2016-10011 |
| CWE-200 | CVE-2011-2707, CVE-2013-1928, CVE-2013-1944, CVE-2014-2038, CVE-2015-8569, CVE-2015-8575, CVE-2016-0823, CVE-2017-14954, CVE-2018-16658 | CWE-404 | CVE-2017-7472 |
| CWE-399 | CVE-2011-2479, CVE-2011-4326, CVE-2012-1583, CVE-2012-6697, CVE-2013-2015, CVE-2013-5634, CVE-2013-7021, CVE-2014-9420 | CWE-682 | CVE-2017-8326 |
| CWE-834 | CVE-2017-14054, CVE-2017-14055, CVE-2017-14056, CVE-2017-14059, CVE-2017-14170, CVE-2017-14171 | CWE-704 | CVE-2018-12453 |
| NVD-CWE-noinfo | CVE-2011-1182, CVE-2014-3480, CVE-2016-9842, CVE-2017-10662, CVE-2017-7184 | CWE-74 | CVE-2016-3695 |
| CWE-269 | CVE-2014-3153, CVE-2014-3534, CVE-2014-4943 | CWE-770 | CVE-2018-16645 |
| CWE-284 | CVE-2015-8845, CVE-2016-3713, CVE-2016-6198 | CWE-824 | CVE-2018-14356 |
| CWE-388 | CVE-2017-5577, CVE-2017-7616, CVE-2017-8072 | CWE-835 | CVE-2018-1999012 |
| CWE-400 | CVE-2011-2906, CVE-2012-6638, CVE-2017-14223 | CWE-89 | CVE-2013-7262 |
| CWE-22 | CVE-2011-3602, CVE-2018-14355 | CWE-94 | CVE-2017-8284 |