

# PACMAN: Securing Dependencies Continuously via Package-Oriented Debloating

Anonymous Author(s)

## ABSTRACT

Real-world software is usually built on top of other software provided as *packages* that are managed by *package managers*. Package managers facilitate code reusability and programmer productivity but incur significant software bloat by installing excessive dependent packages. This “dependency hell” increases potential security issues and hampers rapid response to newly discovered vulnerabilities. We propose a package-oriented debloating framework, PACMAN, for adaptive and security-aware management of an application’s dependent packages. PACMAN improves upon existing debloating techniques by providing a configurable fallback mechanism via post-deployment policies. It also elides the need to completely specify the application’s usage scenarios and does not require runtime support. Moreover, PACMAN enables to rapidly mitigate newly discovered vulnerabilities with minimal impact on the application’s functionality. We evaluate PACMAN on 10 popular and diverse Linux applications comprising 575K-39M SLOC each. Compared to a state-of-the-art approach, piecewise debloating, PACMAN debloats 66% of the packages per application on average, reducing the attack surface by removing 46% of CVEs and 69% (versus 66%) of gadgets, with significantly less runtime overhead and without the need to install a custom loader.

## 1 INTRODUCTION

The essence of software debloating is the removal of code artifacts that are not needed for certain use-cases of an application. In recent years, it has emerged as a promising approach to hardening security by removing excess code [13, 14, 24, 33, 35, 37, 39, 42], which can be done at various granularities of code artifacts such as basic blocks [39], functions [13], or groups of functions [37].

Software debloating has not gained widespread use despite its security benefits. We argue that a debloating technique must simultaneously satisfy the following criteria to be practical: i) it must provide a configurable fallback mechanism, ii) it must not require the complete specification of *all* usage scenarios, and iii) it must not require runtime support. As shown in Table 1, however, none of the existing techniques meet all of these desired criteria.

Since debloating techniques cannot be perfect<sup>1</sup>, they should have a *fallback* mechanism to handle the execution of the application when debloated code is required. Moreover, the mechanism should be configurable; as we argue in Section 3.2, users may have different requirements on fallback mechanisms for different applications. A recent technique, BLANKIt [33] provides a fallback mechanism, but

it is not configurable. Moreover, it needs runtime support for dynamic binary instrumentation using Pin [30], which introduces high performance overhead [54] and various compatibility issues [19], hindering its use in a real-world deployment.

Most existing debloating techniques do not provide any fallback mechanism. Instead, they require a specification of expected application usage scenarios (usually in the form of test cases) so that the application can be trimmed to *include only code needed for the specified usage scenarios*. However, it is hard to anticipate all possible usage scenarios of an application. Piecewise debloating [37] and tree shaking for JavaScript [53] avoid this requirement by using a static dependency graph. However, the dependency graph may be unsound for real-world applications [26, 40]. Consequently, the trimmed applications may fail to execute as a result of aggressively debloating the necessary code.

Furthermore, *none* of the existing techniques provide support for automated rapid security response by neutralizing newly-discovered but unpatched vulnerabilities. This is a growing problem as shown by recent studies that there is a considerable delay between the public disclosure of vulnerabilities and the issue of patches [25, 29, 31].

In this paper, we propose PACMAN, a package-level debloating framework that overcomes the above limitations. Existing techniques debloat a single target program at a time whereas most modern applications are built atop tens or even hundreds of *software packages*. Installing and updating such applications on end-users’ machines is automated by *package managers* (e.g., APT for Debian Linux, Homebrew for MacOS, NPM for JavaScript, PIP for Python, etc.). Package managers play an important role in managing dependencies and conflicts between packages. For example, Chromium version 57.0 for Linux directly depends on 39 other packages. The APT package manager resolves all the indirectly dependent packages and eventually installs 298 packages.

Package-level software bloat leads to numerous security problems besides space and performance issues. First, end-users are exposed to many potentially vulnerable packages installed under the hood. Second, malicious actors can cause widespread damage through popular dependent packages [55, 56]. Third, since each package is typically developed by a different vendor, it is challenging to keep track of and rapidly handle new vulnerabilities, especially when multiple packages are involved. For example, a recent vulnerability in VLC, a widely used media player, turned out to be a problem in a third-party package that is only used when video files in a certain format (.mkv) are played. The bug report was finally resolved a month after its public disclosure [51]. Other complications caused by package-level bloat include a larger software footprint in the system, inefficient dependency installation, and more constraints on inter-package dependencies that results in frequent issues with resolving dependency conflicts, especially at the time of updating them.

<sup>1</sup>Perfect debloating corresponds to dead-code elimination. The debloating techniques we are concerned with may alter the application’s functionality by removing live code, i.e., code that is needed by an execution.

**Table 1: Comparison of PACMAN to other debloating techniques.** For each of the features, we indicate whether the technique fully supports (✓), partially supports (○), or does not support (✗) the feature.

System	Granularity	Configurable Fallback Mechanism	Complete Testcases Not Needed	No Runtime Support	Supports Security Response
BINTRIMMER [39]	Basic Blocks	✗	✓	✓	✗
BLANKIt [33]	Function groups	○	✓	✗	✗
CHISEL [24]	Instructions	✗	✗	✓	✗
OCCAM [32]	Basic Blocks	✗	✗	✓	✗
PIECE-WISE [37]	Function groups	✗	✓	✗	✗
RAZOR [35]	Basic Blocks	✗	✗	✓	✗
<b>PACMAN</b>	Packages	✓	✓	✓	✓

We argue that debloating at the package level enables a generic debloating solution that is applicable to a wide range of applications. The resulting technique can be easily integrated into existing package managers, enabling it to be transparent and flexible to users. Moreover, as we show in this work, it enables us to develop a practical system for providing an automated rapid response to newly-discovered and unpatched vulnerabilities—a growing problem that is ignored by all existing debloating techniques.

First, PACMAN removes all statically unreachable packages. These are packages that are included in the application but absent in its static call graph. Our implementation uses SVF [45], a static value-flow analyzer, to construct such a call graph. On average, this removes 58% of the packages per application, confirming another recent study showing that most applications bear unnecessary dependencies [50]. This enables PACMAN to be usable even in the absence of test cases.

Second, if an application has a set of common *usage scenarios*—i.e., application use-cases, PACMAN uses a tracer to monitor the application and collect packages that are exercised in those use-cases. Packages that are statically reachable but not exercised are removed. This removes another 8% of the packages per application on average, and 65% of the packages in the case of Firefox, one of our larger benchmarks on which SVF times out. Unlike existing approaches, the availability of usage scenarios is an optimization rather than a requirement. In the absence of common usage scenarios, PACMAN can debloat all the packages and load on demand (based on a user-configurable policy) those packages that are needed during the application use. PACMAN achieves this by using *shadow packages* in the place of debloated packages. Shadow packages have the same interface structure as the original but only contain a small piece of code that performs management tasks. When a user requires a certain package that was not initially provided, the shadow package handles the request based on a flexible system configuration, which may either permit various modes of on-demand installation or discard the request.

Shadow packages offer PACMAN flexibility in the choice of static analyses and usage scenarios. For instance, a fast but conservative static analysis that preserves unnecessary packages can be supplemented by usage scenarios to remove them. Likewise, PACMAN permits aggressive static analyses that may remove necessary packages. Indeed, SVF [45] offers a broad range of static analyses with different costs, scalability, precision, and soundness issues. As explained in Section 3.1.1, PACMAN can effectively leverage different analyses in SVF based on application characteristics.

Furthermore, shadow packages allow PACMAN to automate the *secure dependency lifecycle*: Whenever a new vulnerability is reported, PACMAN efficiently replaces the offending package with its shadow version, and restores it later once the patch is available. By doing so, PACMAN can rapidly handle newly discovered security issues that usually require substantial delays for patches (e.g., 1.5 months on average, according to a recent study [25]).

Finally, unlike existing approaches, PACMAN does not require custom runtime support. It relies on the existing linker and loader. PACMAN thus provides all the desired features of an effective and practical debloating technique that also supports security response. The PACMAN framework is anonymously open-sourced and the benchmarks are publicly available to foster reproducibility and further advances in the field<sup>2</sup>. In summary, this paper makes the following contributions:

- We propose a package-oriented debloating framework, PACMAN, which provides adaptive and security-aware package management. PACMAN also provides a configurable fallback mechanism via post-deployment policies.
- We introduce shadow packages that allow us to automate the secure dependency lifecycle. They also afford PACMAN flexibility in the choice of static analyses and usage scenarios for determining the reachable packages.
- We evaluate PACMAN on ten popular and diverse Debian applications available through apt public repository, including Chrome and Firefox, comprising 575K-39M SLOC over 19–479 packages. Our evaluation shows that, on average, we can debloat 66% of the packages per application, which reduces the attack surface by removing 46% of CVEs and 69% of gadgets.
- We design and implement *autorespond*, a monitoring tool that automatically identifies packages containing newly discovered vulnerabilities and allows to disable their execution until a patch becomes available. Our evaluation shows that the approach taken by *autorespond* is effective at mitigating vulnerabilities with minimal impact on application functionality.

## 2 ILLUSTRATIVE OVERVIEW

The overall architecture of PACMAN is depicted in Figure 1. It takes as input the metadata (all dependent packages) of the application to be debloated. Initially, PACMAN determines a set of required packages of the application based on a reachability analyzer. PACMAN installs only the required packages and uses dummy shadow

<sup>2</sup>Code and Data: <https://rb.gy/0ffzjp>, Docker Image: <https://rb.gy/nzcuc9>

packages for all other dependent packages. These shadow packages do not provide any application functionality but maintain the *application binary interface* (ABI) to satisfy system load-time requirements (Section 3.2.1). Every shadow package contains a simple stub that enables PACMAN to handle executions that require the corresponding package.

Shadow packages play two important roles: fallback mechanism and security response. If the execution of an application flows into a shadow package, then depending on a configurable policy, PACMAN seamlessly installs either the corresponding original package or a safer sanitized [41] version which uses runtime monitoring to enhance security. Also, a package containing an unpatched vulnerability can be rapidly disabled on demand, i.e., replaced with a dummy shadow package, and can only be enabled once the patch for the corresponding vulnerability is available. We also provide *autorespond*, a tool that automatically identifies newly discovered but unpatched vulnerabilities and corresponding packages. This information can be used in combination with PACMAN to provide a *secure dependency lifecycle*.

We next elucidate each component of PACMAN using vlc media player [52] as our running example.

## 2.1 Dependency Graph

PACMAN maintains the information of all dependencies between packages (❶ in Figure 1). Direct dependencies of each package are available from the specified application metadata. From this information, PACMAN computes all indirect dependencies.

In our example, vlc 3.0.2 for Debian has 479 dependent packages overall, reachable from its 10 direct dependencies by apt. Among them, 324 packages are identified as reachable from vlc by SVF, a sophisticated static analyzer [45]. By doing so, PACMAN also reduces the attack surface of vlc by eliminating 122 of the known CVEs.

## 2.2 Usage Scenario Database

PACMAN captures application features by observing which dependent packages the application uses under each usage scenario (❷ in Figure 1). We construct a *usage scenario database* for each application from the following sources: (a) common use cases, (b) questions from Stack-Overflow, (c) application tutorials, and (d) developer-provided test suite. In the case of vlc, we collected 299 media files to cover all the media formats supported by vlc. Among the 155 dependent packages with shared libraries identified by the static analyzer, only 134 dependent packages with shared libraries are exercised to play all the collected media files. These numbers confirm prior work [50] that applications installed by existing package managers such as apt are acutely *over-privileged* and we further analyze the reasons for it in Section 4.

The collected usage scenarios are used to eliminate dynamically unreachable packages. Suppose the user wants to install vlc. PACMAN installs only 134 packages and uses shadow packages for all the other dynamically unreachable packages. By doing so, PACMAN can further prevent all potential vulnerabilities those packages, such as another 42 of the known CVEs in the case of vlc.

As we show in Section 4, for most users, the default installation provided by PACMAN works without requiring the execution of any shadow packages. However, cases where an input requires a

shadow package execution are handled using our *shadow package database*, as explained next.

## 2.3 Shadow Package Database

For each package, PACMAN generates the shadow and sanitized versions along with the original version (❸ in Figure 1). The shadow package is a stripped-down version and does not provide original application functionalities. Also, it contains a stub that allows seamless execution of the application in cases where the code in the corresponding original package is required.

PACMAN installs shadow packages for all dependencies of an application except for the required packages (Section 2.2). Thus, in the beginning, the application loads original versions of required packages, and shadow versions for the other packages. In consequence, the application has significantly reduced attack surface at runtime and does not load more packages under usual usage scenarios. In the vlc example, the system initially installs 134 packages to support all the usage scenarios. For the remaining 187 packages, it instead installs the shadow version of the library.

Finally, it allows automating the secure dependency lifecycle. For instance, suppose vlc had been installed with full support for all usage scenarios. Whenever a new vulnerability is reported, the package installer simply substitutes some dependent packages with their shadow versions, which efficiently disables the vulnerability. Once the patched version is available, the corresponding package can be re-enabled.

## 2.4 Post-Deployment Policy

If an input requires the execution of a shadow package, our stub in the shadow package loads the sanitized version of the package and propagates execution to it (❹ in Figure 1). The sanitized version has the same functionalities as the original version but is instrumented with a runtime monitoring mechanism [41]. It thereby enhances security at the expense of runtime overhead. For example, if the sanitized version of libebml is used instead of the original libebml, the average runtime overhead over all the test cases is less than 200ms, and over the test cases that particularly use this package is less than 1s. This default behavior of installing sanitized packages can be changed by using various post-deployment policies (Section 3.2).

Suppose a new vulnerability such as CVE-2019-13615 [18] is discovered and *autorespond* identifies that the corresponding vulnerable package is libebml. In this case, *autorespond* informs our package installer about libebml, which will be disabled by replacing it with the corresponding shadow version and setting a flag in the shadow package database. If a user plays a (potential exploit) .mkv file, the shadow package forces the program execution to safely stop at the beginning of the shadow package (i.e., the first invoked function in libebml), or passes the execution to its sanitized package, depending on the policy.

## 3 DESIGN AND IMPLEMENTATION

In this section, we explain the design of various components of the PACMAN framework.

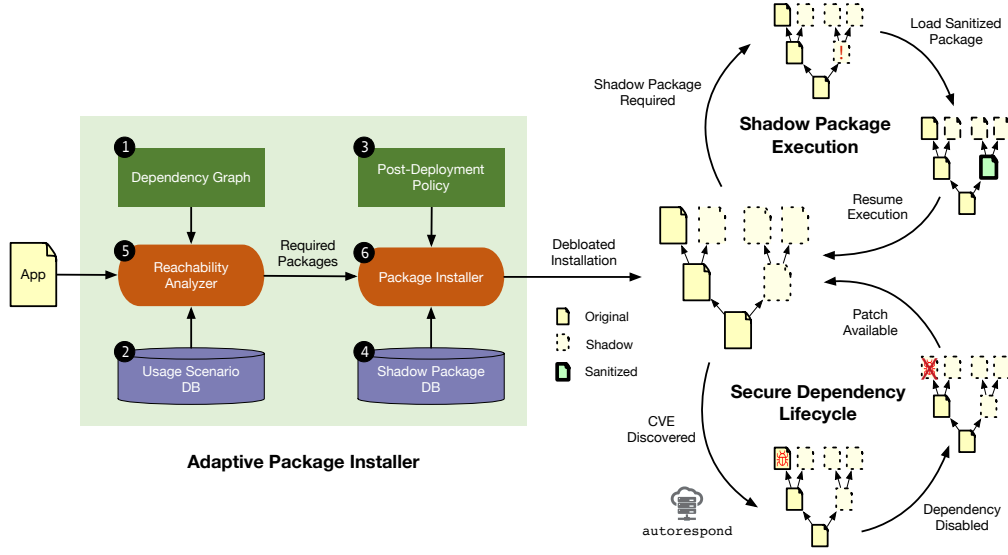


Figure 1: Overview of the PACMAN Framework.

### 3.1 Reachability Analyzer

The reachability analyzer’s goal is to determine the set of packages required by common usages of the given application (5 in Figure 1). Using static and dynamic reachability, we compute the transitive closure of all the packages that the application depends on.

**3.1.1 Static Reachability Analysis.** We construct a static analysis for reachability using SVF [44], a tool for interprocedural value-flow analysis of C and C++ programs. Our analysis constructs a function call-graph and then traverses it starting from the application’s main function to find all the reachable functions.

We address two challenges with static reachability. First, in order to create a complete call-graph, our analysis must handle function pointers and externally-defined code, i.e., code in shared libraries. To resolve function pointer targets, for smaller programs, e.g., `wget`, we use Andersen’s pointer analysis and for larger programs, e.g., `firefox`, we use less precise but more scalable type-based analysis. To handle shared libraries, we generate LLVM bit-code for all of an application’s dependent shared libraries and link it with the application, effectively simulating static linking which gives SVF a whole-program view. Secondly, we use dynamic tracing, as explained below, to capture flows into shared libraries that are explicitly loaded using runtime code-loading mechanisms [15].

**3.1.2 Dynamic Reachability Tracing.** As we show in Section 4.1, for most of the applications, we can easily find common usage scenarios. Our implementation traces packages reached at runtime for each test case by tracing actual program execution flow into shared libraries. To capture this dynamic package usage, we instantiate our shadow packages with a special *tracing* post-deployment policy. Under this policy, invocations to a shadow library are trapped, recorded, and then forwarded to the respective real library so that the application can continue execution. As a result, our system is able to effectively monitor all program flows at runtime to gather a complete picture of application functionality usage.

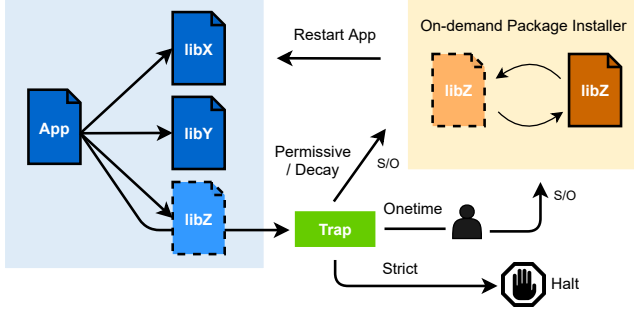
Unlike most of the existing debloating techniques [24, 32, 35], computing required artifacts is *not a strict requirement for PACMAN but rather an optimization* because of our configurable post-deployment policies. If more accurate static analysis results or usage scenarios are provided, the runtime overhead for the fallback mechanism can decrease. As we show in Section 4, PACMAN works even in the absence of required packages.

### 3.2 Package Installation

PACMAN installs the packages that are determined as reachable by the reachability analyzer, and based on the post-deployment policy (6 in Figure 1). For each of the remaining dependencies, PACMAN places a *shadow package* onto the system.

**3.2.1 Shadow Packages.** The goal of using shadow packages is to remove unwanted functionality but maintain the ABI between a package and its dependencies. Binaries linked against shared libraries reserve address space for references in shared libraries that the loader resolves at runtime. On POSIX-like systems, when an executable boots, the loader pulls all of its dependencies into process memory space for reference resolution and code execution. As such, any dependency debloating system must either rewrite an application to change its ABI by removing all references to shared removed dependencies, or provide some mechanism to satisfy the ABI without keeping the original libraries on the system. The former results in a smaller binary whose executions cannot flow into removed libraries. However, this process is rigid, and does not seamlessly allow for post-deployment adjustments; namely, adding and removing dependencies to adjust to changes in the security profiles of each package requires additional rewriting. PACMAN instead creates gutless ABI-compatible mock libraries, which we term *shadow libraries* that belong in part to larger *shadow packages*.

We create the shadow package from any given original package by removing all code inside each function body and replacing it with hooks to our secure runtime framework. The hooks in the shadow



**Figure 2: Post-Deployment Policies and Shadow Package.** S/O stands for sanitized or original package.

packages interact with the package installer to implement various post-deployment policies, as explained below. These shadow packages will be stored in our package database for our package installer to use when processing installation requests.

Shadow libraries built in this manner integrate cleanly into standard POSIX build environment; we do not require modifications to the linker and loader to build applications. This allows a user to drop any POSIX application into our framework. Furthermore, we address post-deployment executions into debloated packages with a series of user-specified *post-deployment policies* that trap and safely handle runtime faults, i.e., executing debloated packages.

**3.2.2 Post-Deployment Policies.** Shadow libraries allow customizing installations through the use of post-deployment policies that trap executions into removed functionality, called fault, and respond based on one of the following modes:

- **Strict mode:** Faults are treated as undesired behavior, meant to force an application to be used as defined by the usage scenarios. Shadow packages are configured to force the program execution to stop; therefore, any security vulnerabilities in uninstalled packages are prevented.
- **Onetime mode:** Faults are treated as requests for additional onetime application functionality. In this mode, PACMAN traps faults into shadow packages and prompts the user to allow the installation of the corresponding application package. If the user allows, then the package will be installed, and the application will be restarted. After the application exits, the package will be replaced with the shadow package. The user will be prompted every time the application requires a shadow package.
- **Decay mode:** In this mode, faults are also treated as requests for additional application functionality. PACMAN traps faults into shadow packages and issues a request to the runtime environment installer to install the original library (without prompting the user). PACMAN reboots the application once the original library has been installed. Furthermore, every package installed through shadow packages is monitored such that if the package is not used in the last  $N$  invocations of the application, it will be replaced with the corresponding shadow package.
- **Permissive mode:** This mode is similar to Decay mode. However, there is no decay and no on-demand installed packages will be reverted to shadow packages.

Furthermore, all the above modes can be configured such that the additional packages can be either the unmodified original package (O) or a *sanitized* version (S) with additional runtime checks (which we use as the default choice).

For each of our deployment policies, program executions that flow into removed libraries instead flow into our isolated post-deployment hooks which safely trap program execution and exit. We elucidate this process in Figure 2. The modes that trigger the installation of shadow packages can be configured to install either the original (O) or sanitized (S) version of the package; however, each fault may only install a library present in the original dependencies, and thus does not introduce additional security issues.

### 3.3 Secure Dependency Lifecycle

PACMAN can realize continuous secure dependency lifecycle by integrating with existing vulnerability databases or discovering systems such as the CVE database [17], GitHub security alerts [23], or OSS-Fuzz [6]. While the initial debloating improves security by preventing 0-day attacks in removed packages, new vulnerabilities in previously “safe” packages could continue to be discovered. In this case, systems that “protect then deploy” are unable to promptly respond to the newly discovered vulnerabilities [25]. These n-day attacks—where a known vulnerability and patch exists for a package but developers have not yet responded and applied a fix—require system administrators and/or developers to immediately respond, or disable the application until the patch is applied.

PACMAN bridges the gap between continuing to provide an application service with a known vulnerability and disabling the application until such a vulnerability is fixed. Since PACMAN does not require any modifications to an application, users can *post-deploy disable* a vulnerable package by removing it from the system and replacing it with the corresponding shadow package. Instead of completely disabling the application, we simply remove a dependency, and a subset of its functionality as a result. As discussed earlier, PACMAN traps executions into this disabled package and safely exits the application. When the vulnerable package is patched, users simply re-enable this functionality by installing the patched package and removing the shadow package from the system.

Since maintaining security post-deployment is a continuous process, we provide a suite of automation tools called *autorespond*, that maintains a *secure dependency environment*. It can continuously scan for open vulnerabilities in installed packages and automatically replace them with shadow packages until a patch is available.

We illustrate how *autorespond* works in Figure 3. *autorespond* implements three batch-running processes, *respond*, *scan*, and *install*, that maintain up-to-date vulnerability information about packages installed on the system and respond to open threats—we anticipate that Linux system administrators will use this software daily to address packages vulnerable to n-day attacks. In essence, *autorespond* automates the secure dependency lifecycle (see Section 3.3) on Linux-like systems.

**Vulnerability Collection.** To collect package-level vulnerability information, *respond* supports subscription to notification systems such as GitHub security alerts [23], which sends push notifications about the packages that contain newly found vulnerabilities. We also provide a crawler for CVE databases [4, 46]. The crawler parses



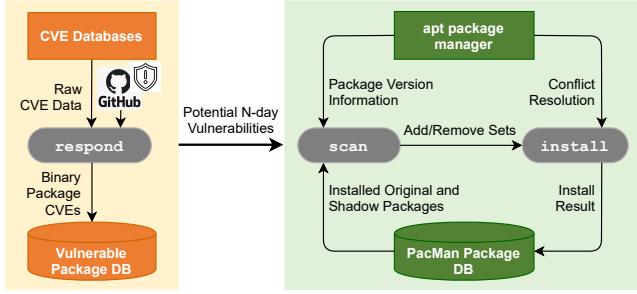


Figure 3: PacMAN autorespond.

a CVE entry’s description to identify the package containing the vulnerability. The CVE entries are associated with source packages, from which often multiple binary packages are built. We only consider the binary packages containing C shared libraries, which we refer to as *Visible Dependencies*.

Through this process, respond maintains an up-to-date vulnerability database at package-level granularity, a key component of package-oriented debloating.

**Package Analysis.** scan collects information from three key sources, namely, PACMAN-installed packages, apt package manager, and the vulnerable package database, to determine which packages to remove from and add to the system. Since PACMAN-built packages are opaque to system administrators and apt, the package installer tracks the set of original and shadow packages on the system and keeps it in a *PACMAN package database*.

PACMAN relies on the apt package manager for installed package version information which is useful for aligning packages on the system with information from the vulnerable package database. Using these two sources of information, scan builds two package sets: a remove set, for original packages installed on the system that have open vulnerabilities, and an add set, for shadow packages that have since received patches addressing their CVEs.

**Installation.** install processes the remove and add sets from scan: original packages in the remove set are replaced by shadow packages, and shadow packages in the add set are replaced by original packages. At first glance, this process might seem straightforward; however, patched packages are likely new versions of previous packages and may be in conflict with the system. In these cases, PACMAN forwards installation requests to apt for conflict resolution, and then manually issues the installations of shadow packages from its own package database.

## 4 EVALUATION

In order to assess the effectiveness of PACMAN as a security-oriented debloating technique, we aim to evaluate each component of the framework as follows:

**RQ1. Effectiveness of Static Reachability** How effective is PACMAN at debloating real-world applications in the absence of test cases?

**RQ2. Effectiveness of Dynamic Reachability** How effective is PACMAN at debloating real-world applications when their common usage scenarios are available?

**RQ3. Effectiveness of Fallback Mechanism** How effective is PACMAN’s fallback mechanism in handling cases that are missed by the static and dynamic reachability analyses?

**RQ4. Effectiveness of autorespond** How effective is PACMAN at providing rapid security response to unpatched vulnerabilities?

### 4.1 Evaluation Methodology

**4.1.1 Benchmark Suite.** We evaluate PACMAN on 10 widely-used Linux applications shown in Table 2. They were broadly selected from two domains: (1) Linux command-line tools, e.g., wget, that usually have a small number of features by following the standard Linux development philosophy “do one thing well”; and (2) popular graphical user applications, e.g., firefox, that are widely used by both casual and expert users. These applications have bountiful feature sets with a large number of dependent packages, and thereby have ample room for security improvements through package-oriented debloating. All the experiments, except when compared with Piece-Wise, are conducted on a Debian system with package dependency information provided by the apt package manager.

**4.1.2 Usage Scenarios.** Table 2 shows the total number of test cases for each benchmark. To reflect various use cases, we selected these test cases from multiple sources such as Online tutorials, Stack-Overflow, and, CommandlineFu. For browsers, we collected the top 500 websites from Alexa [9], which we show to be adequate in Section 4.3. For other applications, we perform a comprehensive search of various online sources to collect relevant use cases. We provide more details on our use cases collection methodology in Appendix A.1.

**4.1.3 Vulnerability Data.** Our package vulnerability database contains information about each package’s known vulnerabilities, i.e., CVEs, that are collected from CVE databases [4, 46].

**4.1.4 Attack Surface.** We consider an application’s attack surface to be the number of CVEs and code reuse gadgets in its dependent packages. We use the vulnerability data (Section 4.1.3) to obtain information regarding CVEs. We use GadgetSetAnalyzer [5] to determine the code reuse gadgets.<sup>3</sup>

### 4.2 Effectiveness of Static Reachability

We measure the effectiveness of PACMAN at removing statically unreachable packages in terms of the number of removed dependencies as well as the number of removed CVEs and gadgets. We then compare packages debloated by PACMAN with packages debloated by the state-of-the-art debloating technique, Piece-Wise [37].

We report the results of our static reachability analysis in Table 3. All applications contain a significant number of statically unreachable packages. On average, PACMAN removes 58% of packages across all of the applications. This number is significantly less than the number of actually installed packages by the baseline. For instance, in git, only 29 out of the 56 visible packages are even *potentially* reachable. Interestingly, most of these statically unreachable packages are part of an application’s indirect dependencies, i.e., packages that developers must include to use one of its direct dependencies.

<sup>3</sup>As shown by recent work [16], GadgetSetAnalyzer provides a more accurate measure of the number of gadgets compared to commonly used tools such as ROPGadget [8].

**Table 2: Benchmark Statistics.** KLOC reports the number of lines of source code for each application and its dependencies. Direct Deps is the number of direct dependencies, and All Deps is the total number including direct and transitive dependencies installed by apt. Visible Deps are packages that PACMAN can disable, i.e., packages with shared libraries (see Section 3.3).

Benchmark (Debian)	Version	KLOC	Direct Deps	All Deps (A)	Visible Deps (V)	V/A	Test Cases	Description
bc-1.07.1	1.07.1	903	4	19	14	73.7%	257	GNU arbitrary precision calculator
gawk-4.2.1	4.2.1	1,252	5	20	15	75.0%	329	Pattern scanner and processor
wget-1.20.1	1.20.1	4,589	9	65	39	60.0%	461	Retrieving files from the web
curl-7.64.0	7.64.0	5,794	3	77	50	65.0%	661	Tool for transferring data
git-2.20.1	2.20.1	575	13	92	56	60.9%	740	Revision control system
xpdf-3.04	3.04	11,600	13	141	92	65.2%	100	Portable PDF reader
firefox-68.2	68.2	25,819	35	260	187	72.0%	500	Web browser
chromium-57	57.0	38,998	39	298	152	51.0%	500	Web browser
gimp-2.10.8	2.10.8	26,653	50	376	250	66.5%	121	Advanced image manipulation tool
vlc-3.0.2	3.0.2	24,935	10	479	321	67.0%	299	Multimedia player and streamer

**Table 3: Static package-level debloating and attack surface reduction by PACMAN.** The “Present in apt installation” column shows statistics of the visible installed packages and existing number of CVEs and gadgets for each application by apt. The “Statically Unreachable” column reports the amount of reduction by PACMAN for statically unreachable packages. TO entry indicates static analysis timeout.

Benchmark (Debian)	Present in apt installation			Reduced by PACMAN (Static)			
	Deps (V)	CVEs (C)	Gadgets (G)	Deps (% of V)	Indirect Deps	CVEs Removed (% of C)	Gadgets Removed (% of G)
bc-1.07.1	14	30	21,522	10 (71%)	9	13 (43%)	12,863 (60%)
gawk-4.2.1	15	29	26,520	12 (80%)	10	16 (55%)	22,388 (84%)
wget-1.20.1	39	50	143,355	22 (56%)	22	26 (52%)	104,978 (73%)
curl-7.64.0	50	68	168,433	23 (46%)	23	22 (32%)	69,024 (41%)
git-2.20.1	56	75	164,823	27 (48%)	27	25 (34%)	122,830 (75%)
xpdf-3.04	92	154	263,879	53 (58%)	53	51 (31%)	196,095 (74%)
firefox-68.2	187	182	717,451	TO	TO	TO	TO
chromium-57	152	513	338,005	75 (49%)	75	121 (46%)	181,334 (54%)
gimp-2.10.8	250	289	901,662	155 (62%)	149	85 (33%)	696,648 (77%)
vlc-3.0.2	321	374	1,361,996	155 (48%)	150	122 (33%)	699,309 (50%)
<b>Average</b>				58%		40%	66%

For example, all of the 27 statically unreachable git packages are indirect dependencies. The results suggest that most indirect dependencies truly are software bloat. For example, git directly depends on libcurl3-gnutls—a rich client-side transfer library—which in turn depends on librtmp1—a high-quality media streaming protocol. Even though git requires libcurl3-gnutls for SSL connection, it does not use the streaming protocol provided by librtmp1. While one might blame a developer for the vulnerabilities and code bloat within their application, these results suggest that much of that responsibility lies beyond the developer’s direct control.

Even long-maintained small utilities are not immune to dependency bloat. For instance, bc—a streamlined Linux command-line tool with 29 years of development—contains 10 statically unreachable dependencies, one of which is libncurses6, a direct dependency. Curiously, we suspected this might be a bug and reported it to the bc maintenance team, who confirmed that it was indeed unnecessary to provide bc’s functionality and exists as part of an outdated automated build setup, rather than any usage of bc itself.

Next, we compare PACMAN against the state-of-the-art baseline, Piece-Wise [37]. We compile all the dependencies of the packages with the Piece-Wise compiler in the default mode and measure

the number of removed unique gadgets and break them down into ROP, JOP, and COP. At the time of writing this paper, the working version of Piece-Wise was only available on Ubuntu. So, unlike other experiments in this section that are conducted on a Debian machine, we deploy PACMAN on an Ubuntu machine with compatible versions of the benchmark applications so that we can compare the same set of application dependencies. Furthermore, to ensure the correct deployment and measurement of Piece-Wise, we verify that the function reduction percentage reported for curl in Table 4 is similar to that reported in Table 5 in [37].

We observe that, in the absence of test cases, for the Ubuntu version of the applications (Table 4), PACMAN reduces 61% of gadgets, which is 5% less than that of Piece-Wise. Although Piece-Wise is effective at debloating functions, the finer-grained unit of debloating results in more runtime overhead. We discuss the overhead more in Section 4.6.

In summary, using static reachability to remove unreachable packages, on average, PACMAN removes 58% of Debian application dependencies, 40% of total application bugs, and 66% of the gadgets. It also removes 61% of the gadgets for Ubuntu applications. Although PACMAN removes fewer gadgets statically compared to

**Table 4: Comparison between the attack surface reduction by PACMAN and Piece-Wise without any test cases.**

Benchmark (Ubuntu)	Reduced by PACMAN (Static)				Reduced by Piece-Wise			
	Uniq	ROP	JOP	COP	Uniq	ROP	JOP	COP
bc-1.06.95	63.4%	69.3%	46.1%	28.1%	75.0%	73.7%	77.8%	77.4%
gawk-4.1.3	36.1%	38.8%	28.0%	16.0%	42.6%	41.4%	47.4%	43.1%
wget-1.17.1	44.5%	50.0%	28.0%	16.1%	56.0%	56.4%	54.8%	56.0%
curl-7.47.0	44.6%	51.3%	26.0%	11.0%	55.7%	57.2%	51.6%	49.0%
git-2.7.4	78.4%	87.1%	55.0%	35.3%	—*	—	—	—
xpdf-3.04	68.5%	75.5%	50.6%	39.0%	74.8%	76.0%	72.2%	71.2%
firefox-84.0.2	79.6%	83.3%	67.0%	58.3%	—*	—	—	—
chromium-87.0	57.6%	64.5%	33.5%	23.6%	73.1%	75.8%	63.0%	62.3%
gimp-2.8.16	71.0%	75.2%	55.7%	46.2%	79.5%	80.2%	76.2%	76.5%
vlc-2.2.2	72.4%	79.0%	52.3%	40.4%	75.0%	74.7%	76.2%	75.0%
<b>Average</b>	61%	67%	44%	31%	66%	67%	65%	64%

\* In some cases, the piecewise prototype does not produce correct results. We omit these cases from the table.

Piece-Wise, it imposes less runtime overhead. Unlike other debloating tools that solely rely on static analysis, PACMAN is not susceptible to soundness issues with SVF, as any packages that are mistakenly classified as unreachable can still be loaded using one of our post-deployment policies which we evaluate later in Section 4.4.

### 4.3 Effectiveness of Dynamic Reachability

For each application, PACMAN can debloat it further by removing dynamically unreachable packages for a given set of use cases. We measure how many more CVEs and gadgets can be removed via dynamic reachability *after* statically unreachable packages are trimmed. We also empirically show the adequacy of the test cases we collected for this experiment.

First, we execute all the collected test cases as described in Section 4.1.2 using dynamic tracing to determine which packages are dynamically executed. Then, we deploy PACMAN in *strict* mode and an initial installation with the union of all the dynamically reached packages. We report the results of our experiment in Table ?? . Even with a large number of statically unreachable packages, applications still contain dynamically unreachable packages and benefit from their removal. The average number of dynamically unreachable packages across all benchmark Debian applications is 66%, which is an additional 8% compared to statically unreachable packages as demonstrated in Table 3. For example, in *git*, of the 29 potentially reachable packages, PACMAN discovers that another 4 are dynamically unreachable. In contrast to statically unreachable packages, these packages implement features that *some* users may need, but may be removed for most users without adversely affecting the functionality. For instance, two of *git*’s dynamically unreachable packages, *libgssapi-krb5-2* and *libkrb5support0*, provide Kerberos authentication support, a network authentication protocol that most *git* users do not need.

PACMAN reduces the attack surface further than static debloating, resulting in an overall reduction of 46% of total application bugs and 69% of gadgets. On closer inspection of the CVEs, we further notice that 30% of these are of high severity [34]. This positively qualifies our result and shows that PACMAN is effective at reducing applications’ attack surface.

One concern that might arise is whether the test suite we collected in Section 4.1.2 is adequate to make general claims about the

**Table 5: Attack surface reduction by PACMAN and Piece-Wise.**

Benchmark (Ubuntu)	Reduced by PACMAN				Reduced by Piece-Wise			
	Uniq	ROP	JOP	COP	Uniq	ROP	JOP	COP
bc-1.06.95	76.6%	80.1%	78.2%	74.8%	75.0%	73.7%	77.8%	77.4%
gawk-4.1.3	48.9%	58.6%	53.6%	41.5%	42.6%	41.4%	47.4%	43.1%
wget-1.17.1	59.8%	60.0%	55.1%	53.7%	56.0%	56.4%	54.8%	56.0%
curl-7.47.0	61.5%	58.8%	57.2%	48.8%	55.7%	57.2%	51.6%	49.0%
git-2.7.4	71.5%	79.8%	55.4%	35.4%	—*	—	—	—
xpdf-3.04	76.5%	79.1%	70.9%	72.0%	74.8%	76.0%	72.2%	71.2%
firefox-84.0.2	79.9%	83.7%	64.1%	48.7%	—*	—	—	—
chromium-87	76.8%	75.5%	63.8%	67.7%	73.1%	75.8%	63.0%	62.3%
gimp-2.8.16	78.1%	80.5%	77.2%	76.9%	79.5%	80.2%	76.2%	76.5%
vlc-2.2.2	76.9%	79.4%	71.3%	71.4%	75.0%	74.7%	76.2%	75.0%
<b>Average</b>	71%	74%	65%	59%	66%	67%	65%	64%

\* In some cases, the piecewise prototype does not produce correct results. We omit these cases from the table.

effectiveness of PACMAN at reducing the attack surface of applications. So, we evaluate our test suite in two different ways: (1) on unseen inputs generated by AFL, a state-of-the-art fuzzer, and (2) on unseen inputs provided by real users through a user-study for a PACMAN-debloomed application, *firefox*.

Again, we deploy the applications in the *strict* mode. Hence, any execution that requires trimmed packages leads to the termination of the application. Since we are claiming that the test cases are diverse enough and cover the majority of the functionalities, we want to ensure we do not encounter such terminations. For command-line applications, we run AFL on PACMAN-debloomed applications with our set of test cases as the seed inputs. The fuzzer does not find any failure-inducing inputs in these applications after 24 hours. We also deploy a PACMAN-debloomed *firefox* based on our test cases consisting of the top 500 Alexa websites, and ask 8 users to use the application as they would normally use a web browser. We do not impose any requirements or restrictions on how they can use the application. We observe that no runtime failures occur since all the dependencies that are required for these users’ typical uses are covered, even though they report performing a variety of tasks not exercised in the training set: browsing password-protected websites, playing live radio / TV / online games, downloading files, training and running deep neural net models in the browser (Keras.js), changing preferences, and many more. Moreover, they do not report any unexpected behavior in this version of *firefox*.



These two experiments show that our test cases are well-balanced and extensively exercise each application’s features so that PACMAN does not simply harden the application by installing too few dependencies. Note that, unlike some of the existing techniques [24, 35], PACMAN does not require use cases. A user may even start with an empty installation and install the dependencies on-demand. Nonetheless, PACMAN can take advantage of any anticipated use cases to reduce the number of on-demand installations by starting with a reasonable minimal installation.

Finally, we discuss how PACMAN compares to Razor [35], a post-deployment tool that aims at debloating binaries guided by users’ test cases. Unfortunately, Razor does not support shared libraries, so it does not apply directly to the complete set of application dependencies. One workaround is to statically build the application with all the dependencies before running Razor on it. However, it is not a desirable solution for real-world deployment. The most important advantage of shared libraries is having only one copy of the library loaded in memory even if more processes depend on it. For static libraries, every process has its own copy of the code, which leads to significant memory bloat. We further tried to debloat the dynamically-linked binaries of the applications in our benchmark. Unfortunately, the debloated binaries had several segmentation faults and we could not evaluate them.

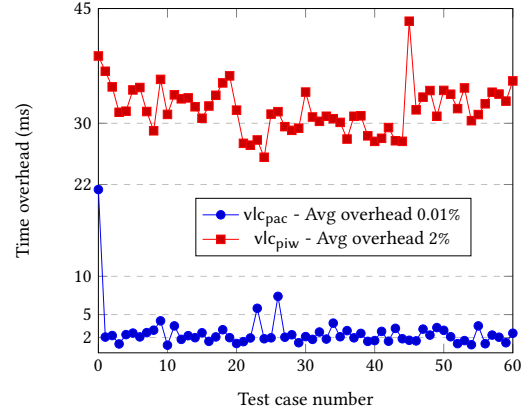
In summary, PACMAN effectively debloats dynamically unreachable packages which we demonstrate with an adequately diverse set of use cases. While both Razor and PACMAN are capable of dynamic debloating, PACMAN is also designed to work not only with the directly dependent shared libraries, but also the indirect ones.

#### 4.4 Effectiveness of Fallback Mechanism

Due to incompleteness, static and dynamic unreachability analyses may fail to predict that some packages are required. Therefore, a fallback mechanism is required to deal with these cases. PACMAN implements a fallback mechanism by introducing post-deployment policies, as described in Section 3.2.2. These policies allow the user to decide the behavior of the application in face of a runtime fault. We measure the performance overhead of *permissive* and compare it with the runtime overhead that Piece-Wise incurs. Then, we compare PACMAN with the only state-of-the-art debloating approach that implements a fallback mechanism, BlankIt [33].

First, we deploy *vlc* without any dynamic or static information, i.e., we consider all the dependent packages unused and trim all of them by replacing them with shadow packages. We choose *vlc* because it is the largest application in our benchmark. Then, we execute the trimmed *vlc* (*vlc<sub>pac</sub>*) with the complete set of use cases, and the *permissive* post-deployment policy, i.e., any required package will be loaded on-demand during the execution. Figure 4 demonstrates the performance overhead encountered by *vlc<sub>pac</sub>* compared to the original untrimmed application when executed with various test cases. *vlc<sub>pac</sub>* encounters around two milliseconds delay on each test case. This delay constitutes only 0.01% of the running time on average over all our test cases of *vlc*.

In general, for each test case, the delay encountered by *vlc<sub>pac</sub>* is proportional to the number of debloated packages required to execute the test case. This can be observed by a few spikes on the *vlc<sub>pac</sub>* line, which indicates that the corresponding test case required



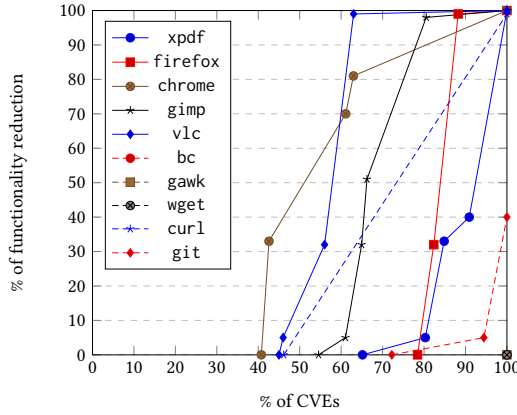
**Figure 4: Time overhead in milli-seconds of debloated *vlc* by PACMAN (*vlc<sub>pac</sub>*) and Piece-Wise (*vlc<sub>piwi</sub>*) when compared with original *vlc* for various test cases.**

a debloated package to be loaded. It is interesting to see that the first test case itself loads most of the required packages. Once *vlc<sub>pac</sub>* reaches a stable state, i.e., all the necessary packages are loaded, we encounter only a small delay that is due to the initialization of the data structures required by post-deployment policies. This shows an interesting use-case of self-customization: Using a *permissive* post-deployment policy, a user can customize an application according to her needs. Once satisfied, the deployment policy can be changed to *strict* mode that only incurs an insignificant overhead.

To demonstrate that such overhead is still less than the overhead imposed by Piece-Wise, we run the same set of test cases with a version of *vlc* that is trimmed by Piece-Wise (*vlc<sub>piwi</sub>*) and measure the overhead again which is demonstrated in Figure 4. We can see that *vlc<sub>piwi</sub>* has a relatively larger overhead of roughly 32 milliseconds in every invocation of the application. This overhead constitutes 2% of the running time on average over all the test cases. This overhead is due to the need to walk through the dependency graph even if the corresponding library is not used during runtime. In other words, Piece-Wise fails to keep many of the dynamically unused functions out of memory. Besides, Piece-Wise is susceptible to soundness issues in SVF since it does not provide any fallback mechanisms. For instance, a bug like issue 70 [26] in SVF could result in crashes in the corresponding trimmed application as reachable code could be potentially trimmed because of the bug. Furthermore, using a post-deployment policy for function-level debloating has a significant performance penalty.

Next, we briefly discuss a state-of-the-art debloating technique, BlankIt, that implements a fallback mechanism. Unfortunately, at the time of writing this paper, BlankIt was not compatible with our benchmark applications. Nevertheless, we discuss the sources of overhead in it and argue that PACMAN incurs lower overhead.

In BlankIt, once a runtime failure occurs, it enters an *audit* mode that checks for memory safety. This mode is implemented via Memcheck’s SGCheck extension. This extension reads all of the type and variable information in the executable and shared dependencies. Then, the stack and global array overrun checks are done while compressing the in-memory representation of DWARF data



**Figure 5: CDF of functionality reduction for various applications because of disabling vulnerable packages for high and critical severity CVEs by autorespond.**

to make these memory-intensive operations feasible. These operations require several range checks per memory access. These are all computationally expensive operations that might incur an overhead of up to 20x [33]. According to SGCheck’s manual [47], for a real-world application, such as OpenOffice, it can take up to one minute to perform the check. Besides, BlankIt uses Pin [30] to dynamically instrument call sites which is an additional runtime overhead of up to 1.7x [33]. In contrast, PACMAN only incurs an insignificant overhead to load the data structures that are required for post-deployment policies; and 2-5x overhead of AddressSanitizer if the user prefers the sanitized package instead of the unsanitized one.

In summary, our results show that PACMAN-debloat applications are robust even in the absence or failure of static and dynamic information. This also implies that PACMAN has a reasonable choice of the debloating unit, i.e., package-level, that enables us to provide an effective and practical fallback mechanism.

#### 4.5 Effectiveness of autorespond

In this section, we evaluate the effectiveness of autorespond to rapidly respond to unpatched vulnerabilities. As explained in Section 3.3, autorespond handles unpatched vulnerabilities by disabling the packages that contain them. However, disabling the required packages affects the application’s usability. To evaluate this, we measure the reduction in functionality when packages containing vulnerabilities are disabled.

We gather all the *high and critical severity* [34] CVEs along with the corresponding package information. For each of these packages, we instruct PACMAN to replace the package with our shadow package, and measure the number of scenarios that terminate the execution after flowing into a shadow package. Note that here we are inspecting a case where terminating the execution without flowing into the vulnerable package is in fact the expected behavior. We repeat this process for each CVE of each application.

Figure 5 shows the CDF of the percentage of use scenarios affected against the percentage of the CVEs for each application. The graph shows that for a given  $(x, y)$  co-ordinate on an application line, disabling packages for  $x\%$  of CVEs affects  $y\%$  or less of the application’s functionality (i.e., collected use scenarios). For command-line utilities, except for curl and git, *none* of the CVEs

affect the functionality of the corresponding applications. Even in the case of git, 95% of the CVEs affect less than 5% of the functionality. In the case of graphical user applications, at least 40% of the CVEs do not affect the corresponding application’s functionality. In the case of firefox, 78% of the CVEs do not affect any of its functionality. Some CVEs affect a fraction of the application’s functionality. An example of such vulnerability is CVE-2017-5130 in libxml2 that affects some versions of the chromium browser. However, there are certain CVEs in common libraries, such as CVE-2018-14550 in libpng, where disabling the corresponding packages will affect 100% of the application’s functionality. Although the functionality reduction is high, we argue that this would be only for a short duration as vulnerabilities in commonly used libraries are usually fixed quickly because of their impact [29].

These results show that disabling only the affected packages can be an effective automatic response to handling vulnerabilities without reducing much of an application’s functionality. Furthermore, unlike other response techniques [12, 25], the approach is fully-automated, application-independent, and applicable to all vulnerability categories. Therefore, the approach taken by autorespond is effective at responding to vulnerabilities with minimal impact on application functionality.

#### 4.6 Real-world Case Studies: Chromium Browser and VLC

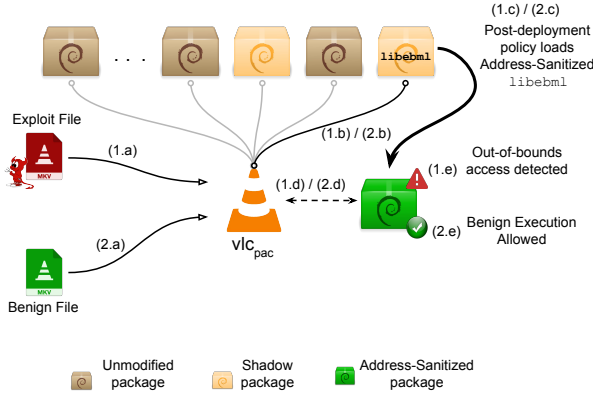
In this section, we present a real-world deployment of VLC media player (Figure 6). We also provide a similar case study of the most popular web browser [10], chromium in Appendix A.2. During normal use (i.e., when debloated packages are not needed), VLC (and Chromium) execute similar to the original applications with no additional overhead. We show that using PACMAN will prevent vulnerabilities without any loss of functionality.

**VLC.** As shown in Table 3, PACMAN debloated vlc by removing 59% of packages, including libebml, which has the CVE-2019-13615, out-of-bounds access vulnerability (As shown by Listing 1 in Appendix). Consider the case when an attacker tries to exploit CVE by tricking the user into executing a specially crafted .mkv file that requires the libebml package. However, libebml is debloated, but because of our post-deployment policy, i.e., permissive with sanitized packages, we load the sanitized version of libebml and continue execution. The sanitized version of the libebml prevents the vulnerability, a memory out-of-bounds write, as all memory accesses in a sanitized package are checked for in-bounds access. This is shown in Figure 6 by (1.a) - (1.e). Thus all attempts to exploit this vulnerability is prevented by vlc<sub>pac</sub>. Furthermore, all benign files requiring libebml execute correctly ((2.a) - (2.e) in Figure 6) with a small overhead of less than 1s caused by the address sanitization. Nonetheless, this shows that PACMAN, with its post-deployment policies, provides an effective way to prevent security vulnerabilities with minimal performance overhead.

### 5 LIMITATIONS

Despite being an effective debloating technique, PACMAN has several limitations.

**Use Scenarios.** We surveyed various sources to collect *use scenarios* as discussed in Section 4.1 in order to ensure the quality of



**Figure 6: PACMAN debloated vlc ( $vlc_{pac}$ ) prevents execution of the exploit .mkv file for CVE-2019-13615 (1.a-1.e) but allows the benign file (2.a-2.e).**

the results reported in Section 4.3. They mainly consist of well-known usages of applications, commands, and files available online. However, we do not claim to capture *every* functionality that users may exercise in an application. The static debloating is independent of any test cases, but in dynamic debloating, depending on the use cases of interest, the final set of dependencies installed on the system varies. In real-world uses of PACMAN for application customization, usage scenarios of individual users can be captured using the provided tracing tool. We did not capture individual users' activities for our evaluation due to privacy concerns. However, as shown in Section 4, users can self-customize their applications by using the permissive post-deployment policy.

**Source Code.** Our current implementation is based on LLVM which requires the source code of packages to create corresponding shadow packages. Consequently, unlike other tools [37, 39], the current version of PACMAN cannot be directly used on binary-only packages. However, with recent binary rewriting techniques such as RetroWrite [21], we expect to successfully port PACMAN to binary-only packages.

**Misconfiguration.** Although post-deployment policies provide flexibility in handling debloated packages, there is a risk of misconfiguration that can render the debloatation ineffective. For instance, always ON permissive mode to use original packages will cause the shadow packages to always load the original package, thus not providing any benefits of debloatation. However, with safe defaults and documentation, such misconfigurations can be avoided [20].

## 6 RELATED WORK

**Software debloating.** A large body of research has proposed techniques to debloat software in order to decrease size and improve security [13, 14, 22, 24, 27, 28, 33, 35–39, 42, 43, 53]. Most of these techniques debloat at a granularity that is finer than package-level, e.g., statement- or function-level. Debloating at package-level granularity can cause PACMAN to exclude more desired usage scenarios compared to those techniques. Conversely, it enables spot removal of newly discovered vulnerabilities without manual effort or runtime overhead, whereas existing techniques require re-analyzing the original application or incur runtime overhead.

Even higher-level approaches to debloating have been proposed, such as configuration-oriented debloating [28, 42], which aims to specialize an application based on static configuration constants and directives, and container debloating [38], which reduces the image size of application containers such as those provided by Docker. In contrast, PACMAN targets individual applications in a given configuration, offering benefits complementary to those approaches. Some techniques focus on more specialized debloating tasks such as debloating the Chromium browser [36] whereas PACMAN targets a wide variety of applications.

**Package managers.** Most of the research literature on package managers focuses on dependency and conflict resolution when installing a new package. Apt-pbo [48] addresses the dependency management problem using pseudo-boolean optimization. Opium [49] combines SAT solvers, pseudo-boolean solvers, and ILP solvers to find an optimal set of dependencies. These techniques can find the minimal set of dependencies that a package requires for *installation* with respect to statically determined dependencies. Instead, our approach aims to install the minimal set of dependent packages that are enough to *execute* (possibly a subset of) usage scenarios. Also, they are not designed to support a security-aware package installation.

**Rapid response to vulnerabilities.** Huang et al. propose Talos [25] that provides a security workaround for rapid response. It uses existing error-handling code within an application to prevent vulnerable code from executing. Our approach is complementary to Talos as we provide a more efficient and robust response to newly discovered vulnerabilities at the package level while their approach can disable vulnerable pieces of code at a finer-grained level.

**Security-aware dependency management.** Recently, on the GitHub marketplace, there are two general trends for security-aware dependency management: (1) apps that help developers keep dependencies up to date, and (2) apps that detect vulnerabilities in dependencies. Depfu [1] is an application in the first category which creates automatic pull requests to update dependencies in order to turn this task into a continuous process. Snyk [2] belongs to the second category which helps developers track security vulnerabilities in dependencies. If a direct or transitive dependency is vulnerable, it provides an automated update to fix the vulnerability as a dependency update; if one does not exist, it provides proprietary patches. Our system focuses mostly on the second category and provides a mechanism to disable dependencies.

## 7 CONCLUSION

We presented a package-oriented debloating framework, PACMAN, for adaptive and security-aware management of an application's dependent packages. PACMAN enables package-level removal of security vulnerabilities in a manner that minimizes disruption to the application's desired usage scenarios. Our experiments on a suite of 10 widely used Linux applications demonstrate that PACMAN can effectively debloat applications, and provide rapid response to newly discovered vulnerabilities from already installed packages.

## REFERENCES

- [1] 2019. Depfu. <https://depfu.com/>.

- [2] 2019. Snyk. <https://snyk.io/>.
- [3] 2020. CommandlineFu. <https://www.commandlinefu.com/>.
- [4] 2020. CVE Details. <https://www.cvedetails.com/>.
- [5] 2020. GadgetSetAnalyzer. <https://github.com/michaelbrownuc/GadgetSetAnalyzer>.
- [6] 2020. OSS-Fuzz. <https://github.com/google/oss-fuzz>.
- [7] 2020. OSS-Fuzz Issues. <https://bugs.chromium.org/p/oss-fuzz/issues>.
- [8] 2020. ROPGadget.
- [9] 2020. The top 500 sites on the web. <https://www.alexa.com/>.
- [10] 2021. Browser Market Share Worldwide. <https://gs.statcounter.com/browser-market-share>.
- [11] 2021. News for Package libxml2. <https://tracker.debian.org/news/865443/>.
- [12] 0patch. 2020. Vulnerability Micropatching. <https://0patch.com/>.
- [13] Ioannis Agadakis, Di Jin, David Williams King, Vasileios P Kemerlis, and Georgios Portokalidis. 2019. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*.
- [14] Babak Amin Azad, Pierre Laperdix, and Nick Nikiforakis. 2019. Less is More: Quantifying the Security Benefits of Debloating Web Applications. In *28th USENIX Security Symposium*.
- [15] David M Beazley, Brian D Ward, and Ian R Cooke. 2001. The inside story on shared libraries and dynamic loading. *Computing in Science & Engineering* 3, 5 (2001), 90–97.
- [16] Michael D Brown and Santosh Pande. 2019. Is less really more? towards better metrics for measuring security improvements realized through software debloating. (2019).
- [17] MITRE Corporation. 1999. The Common Vulnerabilities and Exposures. <https://cve.mitre.org>.
- [18] MITRE Corporation. 2019. CVE-2019-13615. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-13615>.
- [19] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. 2019. SoK: Using dynamic binary instrumentation for security (and how you may get caught red handed). In *Proceedings of the ACM Asia Conference on Computer and Communications Security*.
- [20] Constanze Dietrich, Katharina Krombholz, Kevin Borgolte, and Tobias Fiebig. 2018. Investigating system operators' perspective on security misconfigurations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1272–1289.
- [21] Sushant Dinesh, Nathan Burrow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.
- [22] Masoud Ghaffarinia and Kevin W Hamlen. 2019. Binary Control-Flow Trimming. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
- [23] GitHub. 2020. Managing security vulnerabilities. <https://docs.github.com/en/github/managing-security-vulnerabilities>.
- [24] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
- [25] Zhen Huang, Mariana D'Angelo, Dhaval Miyani, and David Lie. 2016. Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.
- [26] SVF Issues. 2019. Unsound points-to results. <https://github.com/SVF-tools/SVF/issues/70>.
- [27] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. Jred: Program customization and bloatware mitigation based on static analysis. In *Proceedings of the IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*.
- [28] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-Driven Software Debloating. In *Proceedings of the 12th European Workshop on Systems Security (EuroSec)*.
- [29] Frank Li and Vern Paxson. 2017. A large-scale empirical study of security patches. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
- [30] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. (2005).
- [31] A Machiry, N Redini, E Camellini, C Kruegel, and G Vigna. 2020. SPIDER: Enabling Fast Patch Propagation in Related Software Repositories. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.
- [32] Gregory Malecha, Ashish Gehani, and Natarajan Shankar. 2015. Automated software winnowing. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*.
- [33] Girish Mururu, Chris Porter, Prithayan Barua, and Santosh Pande. 2020. BlankIt library debloating: getting what you want instead of cutting what you don't. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*.
- [34] NIST. 2020. Vulnerability Metrics. <https://nvd.nist.gov/vuln-metrics/cvss>.
- [35] Chenxiong Qian, Hong Hu, Mansour Alharthi, Simon Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *Proceedings of the 28th USENIX Security Symposium*.
- [36] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. 2020. Slimium: Debloating the Chromium Browser with Feature Subsetting. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 461–476.
- [37] Anh Quach, Aravind Prakash, and Lok Kwong Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *Proceedings of the 27th USENIX Security Symposium*.
- [38] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: Automatically Debloating Containers. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE)*.
- [39] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. 2019. BinTrimmer: Towards Static Binary Debloating Through Abstract Interpretation. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.
- [40] Gregor Richards, Sylvain Lebesne, Brian Burg, and Jan Vitek. 2010. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1–12.
- [41] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.
- [42] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*.
- [43] Benjamin Shteinfeld. [n.d.]. LibFilter: Debloating Dynamically-Linked Libraries through Binary Recompilation. ([n.d.]).
- [44] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*.
- [45] SVF-Tools. 2020. Static Value-Flow Analysis. <http://svf-tools.github.io/SVF/>.
- [46] Debian Security Team. 2020. Security Bug Tracker. <https://security-tracker.debian.org/>.
- [47] Valgrind Team. 2021. SGCheck: an experimental stack and global array overrun detector. <https://www.valgrind.org/docs/manual/sg-manual.html/>.
- [48] Paulo Trezentos, Inês Lynce, and Arlindo L. Oliveira. 2010. Apt-pbo: solving the software dependency problem using pseudo-boolean optimization. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.
- [49] Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. 2007. Opium: Optimal package install/uninstall manager. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*.
- [50] Ubuntu. 2019. We reduced our Docker images by 60% with --no-install-recommends. <https://ubuntu.com/blog/we-reduced-our-docker-images-by-60-with-no-install-recommends>.
- [51] VideoLAN. 2019. VLC Bug Report 22474. <https://trac.videolan.org/vlc/ticket/22474>.
- [52] VideoLAN. 2020. VLC Media Player. <https://www.videolan.org/vlc/index.html>.
- [53] Jeremy Wagner. 2020. Reduce JavaScript Payloads with Tree Shaking. <https://developers.google.com/web/fundamentals/performance/optimizing-javascript/tree-shaking>.
- [54] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symposium*.
- [55] Zdnet. 2019. Malicious Python Packages. <https://www.zdnet.com/article/two-malicious-python-libraries-removed-from-pypi/>.
- [56] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *Proceedings of the 28th USENIX Security Symposium*.

## A APPENDICES

### A.1 Usage Scenarios

We selected these test cases from multiple sources to reflect a variety of use cases. Particularly, for command-line applications, we draw use cases from:

- (1) Online tutorials. We google the term "X tutorial" where X is the application name, and collect all commands from web pages that appear on the first two pages of the search result.
- (2) StackOverflow. We use StackOverflow to search each application's name, sort the results by the number of upvotes, and

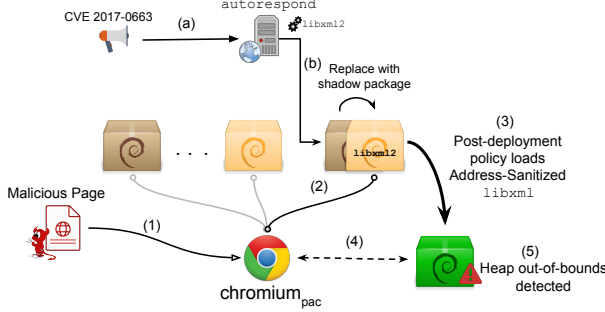


```

// ReadIndex could be more than the size of PossibleIdNSize array
★ if (DataStream.read(&PossibleIdNSize[ReadIndex++], 1) == 0) {
    return NULL;
}
ReadSize++;
} while (!bFound && MaxDataSize > ReadSize);

```

**Listing 1: CVE-2019-13615: Out-of-bounds access in libebml that is prevented by PACMAN by using Address Sanitizer enabled libebml.**



**Figure 7: autorespond disables (a-b) libxml2 from CVE-2017-0663 announcement and PACMAN debloated chromium prevents execution of a malicious webpage trying to exploit the CVE.**

crawl the first three pages to collect all the commands from the accepted answers.

- (3) CommandlineFu. We search the name of the command in CommandlineFu [3] and collect all results with at least one upvote.
- (4) Test Suite. We collect commands included by the developers in the application’s regression test-suite.

For firefox and chromium, we collect the top 500 websites from Alexa [9]. We then open, browse, scroll, and randomly interact with different elements of the page. In Section 4.3, we demonstrate the adequacy of these test cases.

Finally, for gimp, vlc, and xpdf, we survey their manuals to find the media formats they support. For each format  $X$ , we collected all files of the format on the first three pages of Google search, querying “sample  $X$  files”, where  $X$  is a format depending on the application. The use cases comprise of opening these file formats as well as basic interactions with the graphical user interface. In particular, for gimp, we make new documents, save files, use the brush, and apply image transformations; for vlc, we play and pause video and audio files in the supported formats, add and remove subtitles, change sound level, and stream video from a Youtube link; for xpdf, we open, scroll, zoom, and print documents.

## A.2 Chromium Case study

**Chromium.** As shown in Table 3, PACMAN debloats chromium by removing 59% of the dependencies. However, libxml2 is not debloated because most web pages need it. Specifically, 350 out of top 500 websites on Alexa need it. Consider the deployment of PACMAN-debloats chromium, i.e., chromium<sub>pac</sub> (Figure 7) with

default post-deployment policy (permissive with sanitization) along with autorespond to provide a *secure dependency lifecycle*. When a shadow package is needed, we use a sanitized version (i.e., package compiled with address sanitizer) of the corresponding package.

Consider CVE-2017-0663, a severe heap buffer over-flow vulnerability found in libxml2 on May 17, 2017 on OSS-Fuzz [7]. Once the announcement is made, autorespond disables libxml2 by replacing it with the shadow package on the user’s machine until an official fix is available, as shown by (a) and (b) in Figure 7. It took two months to fix the CVE [11]. Before the fix is released, if the user inadvertently (or lured by an attacker) visits a malicious web page that tries to exploit CVE-2017-0663 and requires libxml2, our post-deployment policy, i.e., permissive with sanitized packages loads the sanitized version of libxml2. However, in the sanitized package all memory access are checked to be in-bounds thus preventing CVE-2017-0663, as shown by (1) - (5) in Figure 7. Furthermore, all the benign web pages that require libxml2 execute properly as they do not try to exploit any vulnerability, with a small overhead of less than 950ms caused by the address sanitization.