Input-Driven Dynamic Program Debloating for Code-Reuse Attack Mitigation

Xiaoke Wang[†]
xkernel@whu.edu.cn
Wuhan University
Wuhan, China

Tao Hui[†]
taohui@whu.edu.cn
Wuhan University
Wuhan, China

Lei Zhao[*] [†]
leizhao@whu.edu.cn
Wuhan University
Wuhan, China

Yueqiang Cheng
yueqiang.cheng@nio.io
NIO
Mountain View, USA

ABSTRACT

Modern software is bloated, especially for libraries. The unnecessary code not only brings severe vulnerabilities, but also assists attackers to construct exploits. To mitigate the damage of bloated libraries, researchers have proposed several debloating techniques to remove or restrict the invocation of unused code in a library. However, existing approaches either statically keep code for all expected inputs, which leave unused code for each concrete input, or rely on runtime context to dynamically determine the necessary code, which could be manipulated by attackers.

In this paper, we propose Picup, a practical approach that dynamically customizes libraries for each input. Based on the observation that the behavior of a program mainly depends on the given input, we design Picup to predict the necessary library functions *immediately* after we get the input, which erases the unused code before attackers can affect the decision-making data. To achieve an effective prediction, we adopt a convolutional neural network (CNN) with attention mechanism to extract key bytes from the input and map them to library functions. We evaluate Picup on real-world benchmarks and popular applications. The results show that we can predict the necessary library functions with 97.56% accuracy, and reduce the code size by 87.55% on average with low overheads. These results indicate that Picup is a practical solution for secure and effective library debloating.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

software security, code debloating, attack mitigation

ACM Reference Format:
Xiaoke Wang, Tao Hui, Lei Zhao, and Yueqiang Cheng. 2023. Input-Driven Dynamic Program Debloating for Code-Reuse Attack Mitigation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616274

[*]Lei Zhao is the corresponding author.
[†]Full information of the affiliation: Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan, China

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

1 INTRODUCTION

Modern software is bloated, especially for libraries. For facilitating software development, developers typically import many features from libraries to synthesize new applications easily. However, such *one-size-fits-all* strategy integrates excessive functionalities into the code space of the program, where only a small set of functions are needed. For example, a previous study [50] shows that only 10% of the library functions are used in the Ubuntu Desktop environment.

Bloated libraries bring a detrimental impact on software security. First, the extraneous code may involve vulnerabilities that enable attackers to compromise the system. For example, the x32 ABI on Linux is rarely used by real-world programs, but contains a severe bug that grants attackers extra privileges [24]. Second, unnecessary code provides fertile ground for code reuse attacks, such as return-to-libc (ret2libc) [29] and return-oriented programming (ROP) [18, 19, 21, 54–56]. For example, the GNU C library (glibc) is linked to almost all applications, but it contains many gadgets that attackers can stitch to construct various malicious exploits [36, 37].

To mitigate the security damage of bloated libraries, researchers have proposed several *debloating* techniques [13, 44, 47, 50, 70], which remove or restrict the invocation of unused code. Based on when the *debloating* takes effect, these techniques can be classified into two categories: *offline* debloating and *online* debloating.

Offline debloating [13, 44, 50, 70] aims to trim libraries before an application runs. For example, `Piece-Wise` analyzes call dependencies during library compilation, and only enables necessary code when an application is loaded [50]. `Nibbler` achieves a similar goal for binaries [13]. To ensure the normal operations of an application, they have to retain all functions that the application may invoke. That is, all remaining code is loaded into the program code space even if some of them are unused for a concrete execution on a specific input. However, a concrete execution on a given input often traverses a small amount of all program paths and only invokes a small set of the remaining functions. As a result, although offline debloating techniques can remove many unnecessary codes, the library is still bloated for a concrete execution.

On the other hand, online debloating removes unnecessary library functions at runtime, aiming to achieve on-demand loading. For example, `BlankIt` [47] utilizes a decision-tree based predictor with the function calling context to predict and selectively load library functions at each call. Therefore, context-based debloating can further reduce the attack surface. However, such per-context debloating techniques are vulnerable to *context-corruption* attacks. Specifically, `BlankIt` [47] relies on three factors to predict the required functions: the call site location, the arguments, and the reverse dominance frontier (RDF) of arguments, all of which can be manipulated through memory corruptions (to modify arguments) or slight control-flow manipulation (to choose a proper RDF and call site). Therefore, attackers can fabricate the context for any library function and thus revive ROP or ret2libc attacks.

In this paper, we propose Picup, an online debloating approach that dynamically customizes libraries for each input. Picup aims to balance code reduction and enforcement reliability. Our insight is that the library demands by an application vary on different inputs, while the user-supplied input is the original trampoline for attackers. By predicting the library demands of the received input, we can restrict the suspicious program behavior for the lifetime of each input, which avoids only considering the entire lifetime of a program and blocks the attacker from manipulations on the debloating result. Thus, Picup not only reduces more code than offline techniques but also mitigates the potential threats of online techniques from being nullified by context-corruption attacks.

To achieve a practical per-input library debloating, Picup should satisfy two design requirements: robustness and functionality.

Robustness means that the decision-making process and result cannot be affected by attackers. To achieve this property, we hook input-receiving system calls to capture the input before it reaches the program. Picup predicts necessary library functions at such locations so that attackers have no chance to affect the decision-making. After prediction and debloating, Picup guarantees the same code size until getting the next input or reaching the end. Even if attackers manipulate program states at runtime, they cannot increase the attack surface by invalidating debloating results.

For the functionality requirement, Picup is designed to predict minimal-but-adequate library functions for supporting program normal operations as well as reducing the attack surface to the minimum. Considering among any-length and diverse input bytes, only a few of them contribute to determining the library demands, we adopt a convolutional neural network (CNN) with attention mechanism to identify sensitive bytes from the input with variable length and map the extracted bytes to library functions. Since Picup works for per-input rather than all library API call sites with different contexts, it does not frequently perform predictions during normal internal operations. Besides, we only need to build one model for the whole application, instead of many instrumented predictors for each API function. Thus, this solution is efficient and portable to handle most programs with any input format.

We implement a prototype of Picup, which first runs the program with given inputs to collect mappings from inputs to used library functions. Then, it applies CNN on the mapped data to construct an input-based prediction model. At runtime, Picup hooks each input-receiving system call and predicts the necessary library functions for each input. To demonstrate the effectiveness of Picup, we evaluate the prototype on the SPEC CPU 2006 benchmark and several real-world applications. The experimental results show that Picup can predict library functions with an average accuracy of 97.56%, and reduce the exposed code surface of libraries by 87.55%, thereby mitigating the risk of ROP gadgets and vulnerable functions. In addition, our protection only introduces 1.32% runtime overhead to SPEC CPU 2006 benchmarks and has an acceptable performance on real-world applications.

In summary, we make the following contributions:

- We propose per-input debloating, a practical approach that dynamically reduces the library attack surface for each input. Our method balances the code reduction and the enforcement reliability to achieve better security.
- We design and implement a system that captures user input and predicts library functions. To provide an accurate prediction, we adopt the neural network method to model and predict the library demand of each input.
- We evaluate Picup on SPEC CPU 2006 and popular applications. Results show that Picup can predict the library functions with 97.56% accuracy, and reduce the code size by 87.55% on average with low overheads.

The source code of Picup is available at: https://github.com/b1nsecWh/Picup.

2 MOTIVATION

2.1 Library Debloating

To illustrate existing debloating approaches and demonstrate their differences and limitations, we borrow a code fragment from previous studies [32, 47], shown in Figure 1. The code snippet contains two `if` conditional statements (s_3 and s_9). Given different inputs, this program will execute along with different paths. To be more specific, if the input indicates an administrator, the program will execute along with $\langle s_3, s_6, s_8, s_9, s_{12}\rangle$. Otherwise, if the input indicates a normal user, the program will execute along with $\langle s_3, s_4, s_8, s_9, s_{10}\rangle$. Moreover, there is a classical stack-based buffer overflow vulnerability in s_8, which could be exploited by attackers to change the execution path. Specifically, an attacker can construct an illegal input to overwrite `str` and `user` at s_8, resulting in an unintended attack at the `if` condition in s_9. Once the attack occurs, a malicious non-privileged user can perform any sensitive operations by invoking the library function `system` with arbitrary arguments.

Offline debloating approaches [13, 50, 70] trim libraries while supporting the program on all legitimate inputs. In this example, the program will execute along with two paths based on different inputs, indicated as a red solid line and a black solid line in Figure 1 (b). Therefore, offline debloating approaches prohibit invocations of any library functions except the seven APIs used in the code segment. However, a dynamic execution on a specific input only invokes APIs along the red path or the black path. Therefore, the remaining APIs are still bloated for a concrete execution. The bloated APIs enlarge the attack surface, and can assist attackers to obtain extra privileges, as demonstrated in the aforementioned attack.

Online debloating approaches aim to remove library functions dynamically. A recent work, BlankIt [47], restricts API invocations such that each library function can only be called at corresponding call sites within certain contexts. To achieve this, BlankIt designs a context-based model to predict necessary code and then embeds it into every API call site. That is, based on the execution context, BlankIt predicts and debloats code at each API call, to ensure that only part of the library functions is available.

However, the context-based prediction mechanism itself has limitations. Specifically, BlankIt needs to build the decision-tree based model for all API functions, which is hardly scalable and can bring an extremely high overhead when there are lots of API calls in one execution. What is worse, such a prediction mechanism can

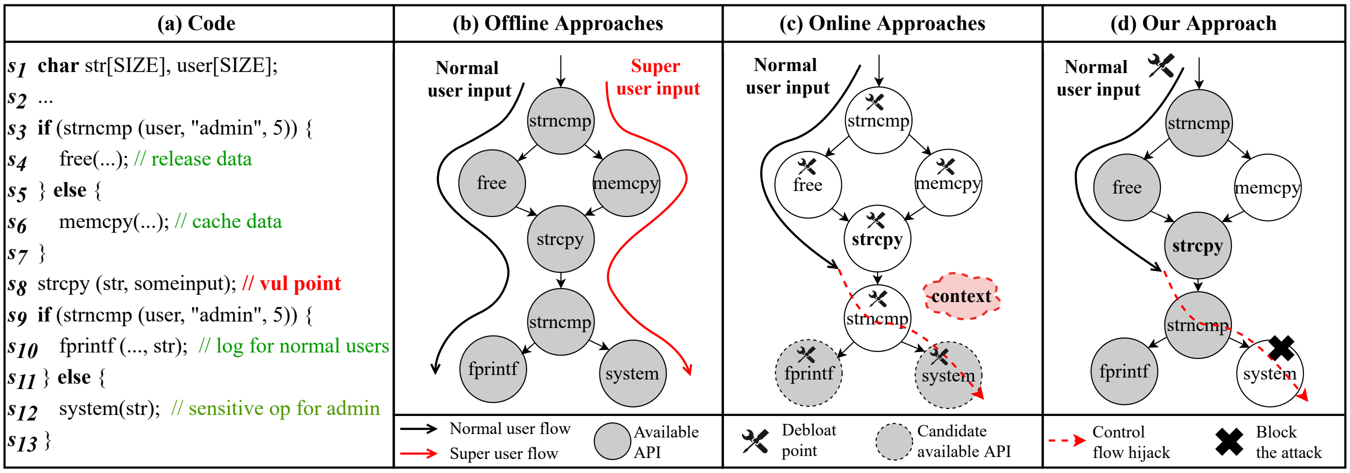

Figure 1: Debloating an example code with different solutions. The example code in (a) has a stack-based buffer overflow vulnerability at line s_8, enabling various attacks. Offline debloating approaches in (b) allow all imported APIs, and therefore, the `system` can be used for the attack; online approaches in (c), though they only activate one API at a time, may still allow `system` to be used by attacks due to attacker-controllable contexts. Our approach in (d) only provides APIs along with the execution path of one input, so even if the control flow is hijacked, the APIs on the execution path of other inputs are still not available.

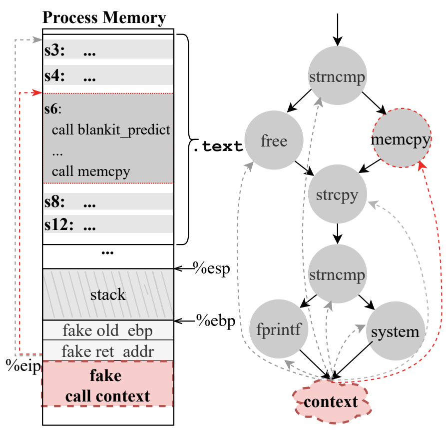

Figure 2: Attack on BlankIt via faked context. An attacker constructs fake calling contexts in the stack, which will mislead the prediction model in BlankIt and then allow the attacker to call any functions in section `.text`.

be bypassed by fabricating runtime context. First, because BlankIt only predicts at each call site, it is insensitive to control flow hijacking that occurs outside the call site. As shown in Figure 1 (c), the approach is also vulnerable to the aforementioned attack to call the `system` function. Moreover, as shown in Figure 2, the attacker can construct the calling context for any function by overrunning the stack frame. Through hijacking the return address, the attacker can jump to the target site and mislead the decision tree model to make a mistaken decision, and then copy unintended code back (e.g. `memcpy`). With appropriate context-corruption, the attacker is available to invoke any functions in the code section `.text`. Consequently, despite the excellent code reduction rate, the context-based approaches cannot always guarantee its enforcement.

In summary, offline debloating approaches still contain unneeded features for each concrete execution. By contrast, online debloating approaches ensure minimal library size at execution, but they are vulnerable to corruption attacks because of the over-reliance on context. In this case, we aim to balance code reduction and reliable enforcement to achieve better security.

2.2 Per-Input Online Debloating

Given an execution path, only functions on the path are invoked so that the other functions beyond the path are unnecessary. What’s more, we observe that the execution path of a program mainly depends on each received input. For the example in Figure 1, the two execution paths are determined by inputs that indicate admin users or not. If we can predict the APIs required on the coming execution path based on the input, then the functions (e.g. `memcpy` and `system`) will be blocked for normal users. Meanwhile, as attackers cannot interrupt the results until sending the next input, the aforementioned attack is not feasible anymore, even if attackers can corrupt the stack variable `user` via current input, calling `system` will not be allowed and finally trigger an execution exception, i.e., segmentation fault with invalid permission.

Following the above observation, we propose *per-input debloating*, which is to dynamically debloat unused code for each input. By predicting the dynamic API demand just for a specific input at runtime, and thus its code reduction rate is higher than existing offline debloating approaches. In addition, *per-input debloating* can guarantee the enforcement reliability, because attackers cannot affect the debloating results nor call unexpected APIs during the dynamic execution, even if attackers can manipulate program context.

2.3 Threat Model

We assume that the underlying hardware and operating system are trustworthy, and thus the prediction model and debloating operations with the necessary data can be protected. That is, the details of the prediction model can maintain agnostic to users and cannot be steered by them to influence the prediction. We do not restrict the attacker's knowledge of the memory layout. The attacker can read/write data and code sections of a process, and the attacker can hijack control flow by exploiting vulnerabilities such as buffer overflow and use-after-free. Note that other memory protection mechanisms, such as StackGuard [23], ASLR [59], DEP [58], and CFI [12], do not conflict with our approach.

3 PICUP DESIGN AND IMPLEMENTATION

In general, Picup works in two phases: the preparation and runtime phase. In the preparation phase, we collect the program execution traces on various inputs and then use them to train the prediction model. Meanwhile, we obtain the dependencies between functions to assist in identifying required functions in runtime. During the runtime phase, we capture the program input and utilize the trained model to predict the unnecessary library APIs. Finally, Picup enables required library functions during the dynamic execution and restricts the invocation to unused library functions.

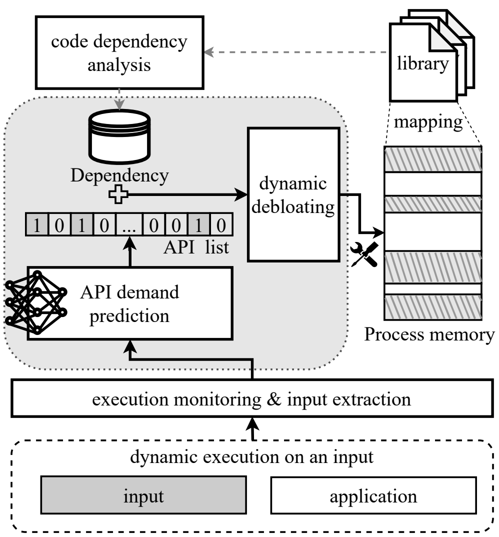

Figure 3: The overview of Picup.

Figure 3 shows the overview of Picup, which consists of four modules: execution monitoring and input extraction, input-driven prediction, code dependency analysis, and dynamic library debloating. The first module, execution monitoring and input extraction, aims to monitor the dynamic execution of programs and captures the input before the program receives it, which enables Picup to perform the subsequent prediction and library debloating.

Given a captured input, Picup will predict the library demand on the execution path for it. To do so, we design a neural network model to make predictions driven by inputs. As the demand of a program on libraries is API functions, we make the output of the prediction model as a list that indicates which APIs are needed.

Additionally, considering an invoked API function typically calls other functions and such sub-functions are also required during execution, we make a dependencies analysis to identify the sub-functions for every exported function in the preparation phase.

After identifing the required library functions, Picup will use the component of dynamic debloating to remove unnecessary functions. To achieve this, we restrict the permission to unnecessary functions by assigning corresponding memory pages as non-executable.

In the following content of this section, the design details of the above four modules will be presented.

3.1 Execution Monitoring and Input Extraction

To debloat library functions for each input, Picup needs to obtain the input and mapping library addresses about the program execution. Besides, Picup should be automatically triggered to perform operations such as model prediction and dynamic debloating. Therefore, we monitor each execution of the program and focus on extracting the received input.

There are several design requirements in execution monitoring and input extraction. First, the execution monitoring should introduce as little impact as possible on the normal running of the program. Besides, the input extraction should be reliable and generic for types of interfaces for receiving inputs. For example, a program can receive inputs from the command line (e.g., *stdin*), from the file system, and the network interface (e.g., *socket*). Therefore, Picup requires a generic approach to identify types of inputs. In addition, the input extraction should be reliable to ensure that our protection cannot be bypassed even if attackers can manipulate the state of the program. For example, when the attack shown in Section 2.1 happens, we should ensure that attackers do not have any chance to corrupt our protection mechanism.

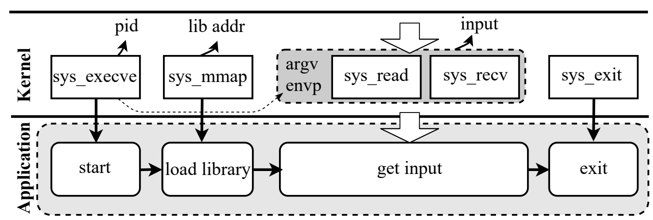

Figure 4: Execution monitoring and input extraction. By hooking syscalls, Picup is awakened when the target program starts execution and sleeps on exit. The address can be obtained when the library is mapped into memory, and the input can be captured before it reaches the program.

In this study, we design a lightweight and generic approach for execution monitoring and input extraction by hooking *syscalls*. As shown in Figure 4, Picup works at the kernel mode, thus the hooking will not be directly affected by any operation from user space. Meanwhile, this approach avoids introducing additional context switching between user mode and kernel mode. To be more specific, Picup monitors the running state of dynamic execution by hooking *sys_execve*, *sys_exit*. By hooking *sys_execve*, Picup is automatically triggered to start working every time the target program starts running. By hooking *sys_exit*, Picup is able to identify the exit of the execution and releases resources in time.

Besides, Picup leverages *sys_open(at)* and *sys_mmap* to identify libraries that are mapped to the process as well as their memory layouts. Picup will use these memory layouts to debloat unnecessary library functions via memory access control.

To capture program inputs via types of interfaces, Picup hooks system calls related to I/O. In detail, Picup hooks *sys_execve* to obtain the inputs (e.g. `argv`, `envp`) provided before the program runs. During execution, Picup captures the inputs (e.g., `stdin`, `file`) from character devices and block devices by hooking *sys_read* and related system calls including *sys_readv*, *sys_pread*, *sys_preadv*. Similarly, Picup captures the inputs from the network (e.g. `socket`) by hooking *sys_recv*, *sys_recvfrom*, *sys_recvmsg*, *sys_recvmmsg*.

Hooking system calls is generic to identify received inputs. Whenever a program receives an input, Picup captures the input and further debloats library functions before the program processes it. More importantly, we can ensure the robustness of Picup. As we mentioned before, the hooking works at the kernel level so that an attacker does not have any chance to modify the hooked input even if the program in user mode is hijacked.

The module of execution monitoring and input extraction is implemented by using a loadable kernel module with the help of Kprobes [42], which enables us to dynamically break into any kernel system call and collect debugging information non-disruptively.

3.2 Input-Driven Prediction

Picup trains the prediction model in the preparation phase. In the runtime phase, our input-driven prediction model predicts required library functions for the execution on a given input.

The main challenge in input-driven prediction is various inputs make different contributions to predict program execution demand. In general, there are both *control* bytes and *data* bytes in the input, which are different for predicting program execution. Specifically, the *control* bytes determine the program behaviors, which contribute more to the prediction results, while the *data* bytes are merely used to hold the input content. Moreover, the *control* bytes in different inputs are specific to input formats, and thus vary a lot from input to input. Without an in-depth understanding of input formats, it is difficult to distinguish *bytes* for various inputs.

In recent years, artificial intelligence has been developing at a rapid pace. The advances in deep learning provide us with a chance to build a prediction model for program inputs. To address the above challenge of extracting *control* bytes without a deep understanding of the input format, we propose to leverage CNN for predicting the library functions demand. The convolutional neural network (CNN) [43, 45, 67] performs outstandingly in artificial intelligence tasks such as image recognition, especially for feature extraction. By leveraging CNN, our model can automatically learn *control* bytes that determine program behaviors.

However, as demonstrated in previous techniques [51], *control* bytes often represent a small fraction of all the bytes in program input. With this impact, our CNN-based prediction model may decrease the accuracy for inputs with large sizes. To improve the performance of our prediction model, we further design to leverage the attention mechanism [60, 63] as feature refinement to identify the contribution of each byte extracted by CNN. In detail, the attention layer works after the CNN extracted the feature bytes. It generates a weight map to indicate the importance of each extracted byte and then makes the bytes with high weights play an active role in the model decision. In this way, we can enhance our model by making it pay more attention to *control* bytes.

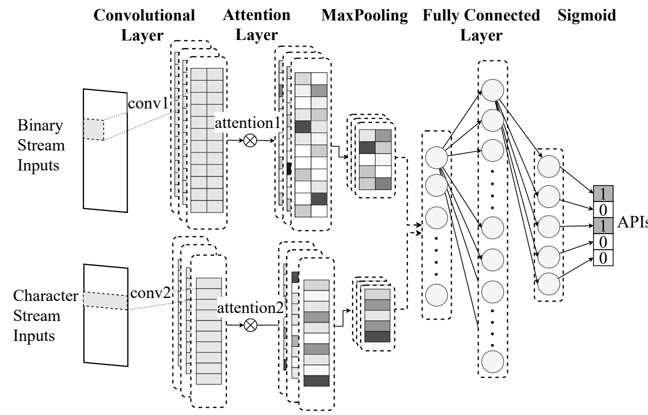

Figure 5: The prediction model in Picup. The features are first extracted in the convolutional layer, and then refined in the attention layer. In the fully connected layer, the model predicts whether the API will be invoked or not.

To deal with various types of program inputs, we divide inputs into two categories according to their forms. One is the binary stream (e.g., images, ELF file), and the other is the character stream (e.g., argv). The model applies different types of convolutional kernels to extract *control* bytes in these two types of inputs. Specifically, the binary stream inputs are processed by the kernel same to LeNet-5 [43], and the character stream inputs are processed by the kernel same to TextCNN [68] after each word is encoded into vector.

The last layer of the model contains neurons with the same number as the APIs in Global Offset Table (GOT) of the target program. The value of each neuron is normalized into a 0-1 range, denoting the probability of requiring the corresponding API. After processing with a threshold (0.5 by default), the model will output an API list, where 1 means the API is required and 0 is not.

In the implementation, we build the prediction model based on the torch framework [46] with Convolutional Block Attention Module [63]. For binary stream input, we directly convert each byte to a grayscale pixel. For character stream input, to encode all words even if the word has not appeared before, we apply fastText [20] to generate a distributed representation of each word.

3.3 Code Dependency Analysis for Libraries

As an API usually not only executes its own code, but also calls other functions in the library, an API list is insufficient to identify all required code for a certain execution. To address this problem, we perform a code dependency analysis for libraries in the preparation phase to record the functions each API depends on.

The technique we use is similar to control flow analysis [14, 15], but focuses on inter-procedural control flow transfer. Specifically, we search for all the call instructions of each function and iteratively analyze their destinations. As a result, all functions that possibly be invoked by an API are regarded as dependencies. To avoid duplicated analysis, we maintain a list of analyzed functions with their dependencies. Note that this step is in the preparation phase, so it does not bring extra overhead to the running phase. Additionally, as the execution traces of inputs are also required to be collected via tracing tools [31, 40] in the preparation phase, we

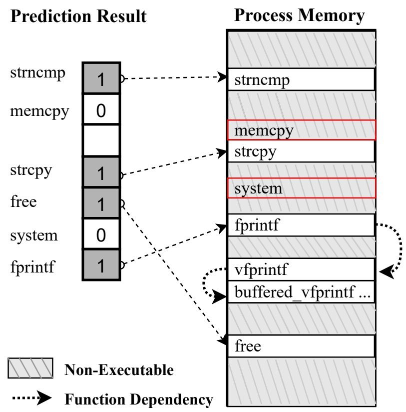

Figure 6: Library dynamic debloating. According to the prediction result, all library code in the process memory is set to non-executable except for these APIs that will be called and their dependencies.

can also evaluate the addresses recorded by the traces to validate and fix the dependencies in practice. Based on the dependencies of each function, Picup is able to identify other necessary library functions that will be invoked by the required API functions so that ensuring the required APIs work properly in the runtime phase.

In our prototype, because the analyzed inter-procedural control flow transfer based on binary code in practice is usually neither sound nor complete, we implement the code dependency analysis by leveraging both the symbolic analysis in angr [62] and the static analysis in BARF [33] to enable the analysis as we demonstrated above. To avoid missing valid dependencies by required code that may influence normal functionality, we conservatively combine the results of angr and BARF. That is, the final dependency includes all relationships recorded by angr and BARF.

3.4 Dynamic Debloating

For dynamic debloating, we need to know the layout of libraries mapped into the process memory, i.e., where are the libraries in memory, so that we can further locate the required code. To do this, as described in Section 3.1, Picup hooks *sys_open(at)* and *sys_mmap* during execution. As *sys_open(at)* is used to open a file and *sys_mmap* is used to map files into memory, hooking these syscalls enables us to identify each loaded library and get the base memory address of each library when the library is loading.

With the memory layout of libraries, Picup can identify the address of each required library code in the process memory based on the prediction result and the code dependencies for debloating. As the operating system uses pages to manage process memory, Picup chooses to change the executable permissions of the library code by pages. Compared to directly modifying the content in memory, this way introduces a relatively negligible load. Specifically, since there is usually a small portion of the code required by a specific input, Picup disables the executable permissions for all pages by default so that we do not need to operate on too many pages each time. Afterward, for each input and the corresponding prediction result, the memory pages occupied by the required code will be identified and set as executable before the process switches to user space, i.e. before the program handles input.

It should be noted that the component of dynamic debloating runs in kernel space, and works in cycle by determining the start, exit and input of the execution, so we can strictly maintain the permission during the execution of the process in user space until the next input or exit. Meanwhile, Picup monitors the system calls related to memory permission control, such as *sys_mprotect*, *sys_mmap*. As a result, any call that tries to recover the debloated code as executable via the memory permission control related system calls from the user space will be treated as illegal. With this setting, we can prevent the results of debloating from being affected by users to ensure reliable enforcement. That is, the restricted code cannot be invoked even if the calling context is manipulated.

Taking Figure 6 as an example, the APIs (`strncmp`, `strcpy`, `free`, `fprintf`) that are predicted will be called and their dependent functions (`buffered_vfprintf`, etc.) are retained as enabled. Except for required functions, all other code (include `system` and `memcpy`) is set to non-executable. If an attacker hijacks the control flow to `system` by affecting the code branch, the relevant code in memory is not executable. Instead, this will cause a fault that is caught by Picup and then be handled with a specific security policy.

In the prototype of Picup, we implement the module of dynamic debloating as a part of the loadable kernel module. With the kernel module, Picup is automatically triggered to debloat library code whenever the program gets an input.

4 EVALUATION

We implemented a prototype of Picup with 1,510 lines of C code and 1,140 lines of python code on the platform of Ubuntu 18.04 LTS. To demonstrate the effectiveness of our approach, we conduct comprehensive evaluations with four objectives.

- **Code reduction.** How much code can Picup reduce from the libraries?
- **Functionality guarantee.** How accurate is our model? Can the target program work properly under Picup?
- **Security.** How much can Picup improve the security of the target program?
- **Runtime overhead.** How much is the runtime overhead introduced by Picup?

4.1 Experiment Setup

Baseline Techniques. We select a state-of-the-art online debloating technique, BlankIt [47], as the baseline for comparison. Additionally, we also discuss the results of Piece-Wise [50], a representative offline debloating technique.

Datasets. We evaluate Picup on SPEC CPU 2006 [34] and several real-world applications. SPEC CPU 2006 is employed for evaluation in the previous study BlankIt [47]. To be more specific, we evaluate Picup on the same 17 C/C++ applications selected by BlankIt.

Besides SPEC CPU 2006, to show that Picup can debloat real-world applications with different running statuses by handling various types of input interfaces, we employ multiple real-world applications for evaluation. Specifically, the real-world applications include two web-server applications (*nginx* [7] and *lighttpd* [41]), two database applications (*memcached* [3] and *redis* [4]) for which the input interface is the network, and three GNU Binutils programs [17] (*readelf*, *objdump* and *nm*) for which the input interface is files and command lines. In terms of running status, these real-world applications include both long-running and standalone processes. All of the long-running applications are running under the default configuration files and options. For web-server applications, we deploy a copy of static HTML pages from Wikipedia[2] .

Test Cases. The SPEC CPU 2006 dataset includes three types of test cases for every application, denoted as "test", "train" and "ref", respectively. Following the configuration in BlankIt, we use the test cases of both "test" (small) and "train" (medium) as training samples to train the prediction model. Then, we use the test cases of "ref" (large) to test the model. Please note that some applications in SPEC CPU 2006 only consist of a small number of input cases, which may lead to the over-fitting problem in the prediction model. To alleviate the impact, we further leverage AFL [65], a representative coverage-guided fuzzing tool, to randomly generate more input cases for training. Specifically, we take the original input cases as initial seeds for fuzzing, and then retrieve the newly generated seeds from the working directory of fuzzing as part of the dataset since these new seeds represent inputs that trigger different program states and usually have diverse content.

For real-world applications, we collect the running logs from the open deployment environment for the web-server and database applications, and then extract inputs from the logs to construct the set of test cases. For GNU Binutils applications, we randomly collect both ELF and non-ELF files to construct the set of test cases. Finally, we collect more than 10000 inputs for each application, and then use ltrace [40] to record the APIs invoked by each input. These datasets are partitioned for training, and testing with a ratio of 9:1, which is a typical ratio in neural network jobs.

4.2 Code Reduction

To demonstrate the effectiveness of our approach in code surface reduction, we leverage the code reduction rate as a metric and the calculation is $reduction_{code} = \frac{\sum_n (r/T)}{n}$, where T refers to the number of total instructions in libraries, r refers to the number of instructions removed by debloating techniques, and n refers to the number of executions during testing.

To compare the performance of online and offline debloating techniques, we further calculate the reduction rate of the imported APIs. The calculation is $reduction_{API} = \frac{\sum_n (u/I)}{n}$, where I refers to the total number of imported APIs for the application; u refers to the number of import APIs that are removed in dynamic execution.

4.2.1 Code Reduction on SPEC CPU 2006. Table 1 shows the code reduction rates on SPEC CPU 2006. Note that the data of BlankIt is directly referenced from the literature [47].

Overall, Picup achieves 88.21% code reduction on average and we can observe that the reduction rate of Picup is less than that of BlankIt. The reason is that when calling an API, BlankIt only remains the current API function and its sub-functions based on execution contexts, while Picup remains all the API functions that are predicted to be used by the currently received input. However, the predictions made by BlankIt are overly dependent on the program context, which can be easily forged by attackers. As discussed in Section 2, an attacker can copy back any API code they want in the section `.text` through context-corruption. Thus, despite achieving a high code reduction rate, the context-based online debloating approaches cannot always guarantee its security enforcement.

Table 1: : Code reduction on SPEC CPU 2006.

Benchmark	BlankIt	Picup
401.bzip2	97.7%	89.12%
403.gcc	97.2%	83.57%
429.mcf	94.5%	85.83%
433.milc	98.0%	92.66%
444.namd	97.0%	90.54%
445.gobmk	95.7%	85.11%
450.soplex	97.4%	86.38%
453.povray	96.9%	87.31%
456.hmmer	97.9%	87.78%
458.sjeng	97.8%	86.65%
462.libquantum	97.9%	91.67%
464.h264ref	97.9%	88.93%
470.lbm	97.8%	92.44%
471.omnetpp	95.6%	88.46%
473.astar	96.6%	89.31%
482.sphinx3	97.6%	86.00%
483.xalancbmk	96.9%	87.85%
Average	**97.08%**	**88.21%**

Table 2: Code/API reduction on real-world applications.

Application	Code	API
nginx-1.14.0	90.31%	95.22%
lighttpd-1.4.59	86.68%	85.88%
redis-5.0.5	87.59%	94.78%
memcached-1.6.9	88.70%	90.82%
objdump-2.30	83.43%	84.81%
readelf-2.30	85.25%	72.26%
nm-2.30	79.65%	76.55%
Average	**85.94%**	**85.76%**

In addition, based on our statistics, an average of 45.69% imported API functions of SPEC CPU 2006 programs are invoked by different inputs, which means that offline techniques like Piece-Wise [50] need to retain an average of 54.31% unnecessary imported API functions as well as their dependencies. Note that we did not directly make a fair comparison between Picup and Piece-Wise on the 17 SPEC CPU 2006 programs, because Piece-Wise customizes the compiler and loader to generate ELF files and perform linking, but its full implementation is not available. According to the original experiments of Piece-Wise, it evaluates 11 of our selected 17 SPEC CPU 2006 programs only with the compiled musl-libc (i.e., the other libraries are retained without change). In such a situation, its code reduction rate on musl-libc achieves 60% in worst-case and 86% in best-case, both of which are lower than the average reduction rate of Picup on all the shared libraries that a program depends on.

4.2.2 Code Reduction on Real-World Applications. Table 2 shows the code reduction rate on the seven real-world applications. To find out how many imported APIs of applications for Picup are restricted in execution, we also count the reduction rate of the imported API.

As shown in Table 2, the average code reduction rate is 85.94% and the average imported API (the API in GOT) reduction rate is

85.76%. To investigate which code is reduced, we took a manual analysis on `nginx`. Specifically, for the 90.31% code removal rate of `nginx`, all code of several imported libraries (e.g. *libz*, *libcrypto*, *libcrypt*, *libssl*) is even removed during execution based on our manual analysis, because these libraries are not often used by normal HTTP requests. For the 95.22% API removal rate of `nginx`, we also notice that some risk APIs (e.g. *execve*, *syscall*) that are rarely invoked by general input are successfully disabled by PICUP in execution. In Section 4.4, we will further analyze how does the reduced attack surface by our approach improves security.

In short, in addition to removing most code, our approach specifically restricts many risk APIs according to the learning of realistic situations by predictive models.

4.3 Functionality Guarantee

In this section, we evaluate whether PICUP can guarantee the normal functionalities of executions on different inputs. If PICUP accurately retains the required functions, then the program can undoubtedly run normally. Therefore, we use the prediction accuracy as a metric, which is defined as the percentage of APIs that are correctly predicted and the calculation is $Accuracy = \frac{\sum I_a}{I}$, where I_a refers to the API correctly predicted, and I refers to the number of imported APIs in a binary GOT.

Besides, there are two situations for inaccurate predictions: false positive and false negative. A false positive refers to an API that is useless but predicted to be required, and a false negative indicates an API that is required but predicted to be useless. As false negative can influence normal functionalities of the target program, we take False Negative Rate (FNR) to further show the negative impact of our prediction model on normal executions. FNR is calculated as $FNR = \frac{\sum I_{fn}}{I}$, where I_{fn} refers to the API with false negatives, and I refers to the number of all invoked APIs during executions.

It should be noted in our evaluation, once the prediction does not have false negatives, the program will work normally as the obtained code dependency has been already revalidated and fixed based on the pre-collected traces of the samples. However, it is hard to perform complete and sound code dependency analysis on binary in practice, especially for resolving indirect calls, and missing code dependency may also influence normal functionality. If such cases occur, we need to trace the execution of the input and confirm that the control transfer is natural, then we can fix the code dependency based on the trace to alleviate the negative impact. In this way, the code dependency will continuously be more complete.

4.3.1 Accuracy and FNR on SPEC CPU 2006. Table 3 shows the prediction accuracy and FNR for the 17 applications in SPEC CPU 2006. Note that Piece-Wise [50] has no accuracy data as it is an offline approach without prediction. For BlankIt, we find that it is hard to fairly reproduce its results due to its public implementation and datasets are incomplete, so we directly present the accuracy from the original paper [47]. While BlankIt does not report the FNR, so its FNR is missing in Table 3.

From Table 3, we can observe that the accuracy of PICUP (97.34%) is higher than that of BlankIt (94.35%). In particular, PICUP outperforms BlankIt in prediction for 12 out of the 17 applications, as underlined in Table 3. Moreover, PICUP also outperforms BlankIt even for the worst case. PICUP provides 91.98% accuracy on *433.milc*, whereas BlankIt merely achieves 60% accuracy on *462.libquantum*.

Table 3: Prediction accuracy and FNR on SPEC CPU 2006.

Benchmark	Accuracy		FNR
	BlankIt	PICUP	
401.bzip2	91%	100.00%	0.00%
403.gcc	99%	95.71%	1.07%
429.mcf	94%	94.44%	5.56%
433.milc	100%	91.98%	0.00%
444.namd	99%	100.00%	0.00%
445.gobmk	84%	93.08%	2.75%
450.soplex	92%	100.00%	0.00%
453.povray	97%	100.00%	0.00%
456.hmmer	98%	98.52%	0.00%
458.sjeng	97%	100.00%	0.00%
462.libquantum	60%	96.00%	0.00%
464.h264ref	100%	95.34%	0.00%
470.lbm	98%	94.44%	0.00%
471.omnetpp	99%	99.04%	0.00%
473.astar	100%	98.00%	0.00%
482.sphinx3	99%	100.00%	0.00%
483.xalancbmk	97%	98.35%	0.00%
Average	**94.35%**	**97.34%**	**0.55%**

Table 4: Accuracy and FNR on real-world applications.

Application	Accuracy	FNR
nginx-1.14.0	98.95%	0.59%
lighttpd-1.4.59	98.15%	0.98%
redis-5.0.5	96.28%	2.98%
memcached-1.6.9	96.08%	1.82%
objdump-2.30	99.36%	0.31%
readelf-2.30	99.11%	0.22%
nm-2.30	98.70%	0.78%
Average	**98.09%**	**1.10%**

Meanwhile, we can also observe that PICUP achieves 0.00% FNR for all applications except three applications, *403.gcc*, *429.mcf*, and *445.gobmk*. Specifically, the FNR for these applications are 1.07%, 5.56%, and 2.75%, respectively. Our manual analysis shows that the test cases of *429.mcf* are uninformative numbers with very large size. Therefore, it is difficult for the model to extract accurate features that determine program behaviors. The input of *403.gcc* contains some complex syntax information, which makes it difficult to predict. The input of *445.gobmk*, a smart-game-format file, consists of semantics-rich fields. It defines some special symbols to represent specific program states. Such semantics-rich fields are difficult to extract by our prediction model.

4.3.2 Accuracy and FNR on Real-World Applications. Table 4 shows the accuracy and FNR on seven real-world applications.

From Table 4, we can observe that the average prediction accuracy of PICUP is 98.09% and the FNR is 1.10% on seven real-world applications. Overall, these results indicate that our prediction model can accurately predict the *control* bytes in inputs that determine program behaviors. To find out what key bytes are extracted, we also made a further analysis on the prediction process. Take *nginx* as a brief example, we found that our model successfully identifies the word "gzip" contributes to the invocation for the API

Table 5: Reduction of CVE vulnerability functions in Glibc on SPEC CPU 2006. ✓ means the vulnerability function is eliminated by Picup, while ✗ means the function is not eliminated.

Glibc Vulnerability			Benchmark																	
CVE-ID	Glibc	Vul.	bzip	gcc	mcf	milc	namd	gobmk	soplex	povray	hmmer	sjeng	libquantum	h264ref	lbm	omnetpp	astar	sphinx3	xalancbmk	Total
2021-35942	≤2.33	wordexp	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	**17/17**
2021-3326	≤2.32	iconv	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	**16/17**
2020-27618	≤2.32	iconv	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	**16/17**
2020-29562	2.30-2.32	iconv	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	**16/17**
2020-1752	2.14-2.32	glob	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	**17/17**
2009-5155	<2.28	parse _reg_exp	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	**17/17**
2018-1000001	≤2.26	realpath	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	**16/17**
2018-11236	≤2.27	realpath	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	**16/17**
2018-11237	≤2.27	mempcpy	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	**15/17**
2018-6485	≤2.26	posix _memalign	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	**0/17**
	Total		**9/10**	**6/10**	**9/10**	**9/10**	**9/10**	**9/10**	**9/10**	**9/10**	**9/10**	**9/10**	**9/10**	**9/10**	**9/10**	**8/10**	**9/10**	**9/10**	**6/10**	-

deflateInit2_ [1], and this API actually will be invoked if the received input requests compression through gzip [30].

Furthermore, we can also find that the prediction accuracy of two cached database applications is relatively lower than that of other applications. Based on our manual analysis, we find that some API calls depend on the environment of the program, such as whether the query data exists in memory. However, it is difficult for our model to obtain the environmental information from the input alone. This reason leads to less accuracy of the model in these applications. Conservatively predicting the required functions by turning down the threshold for neuronal activation is a possible way to mitigate the negative impact brought by false negatives on the normal functionalities of the target program, but it may also bring more false positives so that the code reduction rate will be reduced. In fact, it is almost impossible to eliminate all inaccurate predictions on diverse real-world programs even if considering every potential factor. In Section 6, we present discussions on the possible ways to handle false predictions.

4.4 Security

To find out how much our approach improves the security, we evaluated Picup from the following three perspectives.

4.4.1 Reduction of ROP Gadgets. To show how much of the program's attack surface is reduced, we firstly count the ROP gadgets that are removed after receiving each input. ROP gadgets are pieces of code that can be run in a certain order to carry out attacks by using return-oriented programming. The reduced gadgets will restrict an attacker's capabilities in practice, so that the difficulty and effort required for attacks will also increase even if there are still other security risks in the remaining code.

To measure the reduction of ROP gadgets, we leverage angrop[9], an ROP gadget finder and chain builder. For each program in the SPEC CPU 2006, we recorded the code that is removed by Picup under test case input and then calculated the removed gadgets rate according to the proportion of removed gadgets in all gadgets of each library. Table 6 shows the ROP gadgets reduction rate in libraries on SPEC CPU 2006. In total, an average of 77.24% of ROP gadgets were removed by Picup when the program is running. Specifically, Picup removes an average of 70.51% ROP gadgets from libc-2.27 and the reduction rate reaches 94.43% for libm-2.27.

Table 6: Reduction of ROP gadgets on SPEC CPU 2006.

Benchmark	libc-2.27.so	libm-2.27.so	libgcc_s.so.1	libstdc++.so.6.0.25	Avg.
bzip2	78.12%	-	-	-	78.12%
gcc	68.34%	-	-	-	68.34%
mcf	72.39%	-	-	-	72.39%
milc	78.15%	99.01%	-	-	88.58%
namd	71.00%	93.17%	76.20%	92.53%	83.22%
gobmk	72.33%	97.32%	-	-	84.83%
soplex	68.16%	93.76%	50.71%	60.73%	68.34%
povray	60.95%	90.09%	40.82%	91.95%	70.95%
hmmer	67.46%	87.16%	-	-	77.31%
sjeng	73.98%	-	-	-	73.98%
libquantum	77.66%	94.77%	-	-	86.21%
h264ref	66.91%	96.95%	-	-	81.93%
lbm	75.78%	99.57%	-	-	87.67%
omnetpp	64.21%	97.17%	40.82%	68.46%	67.67%
astar	68.24%	93.76%	46.45%	92.53%	75.24%
sphinx3	67.67%	88.40%	-	-	78.03%
xalancbmk	67.32%	96.47%	46.45%	70.56%	70.20%
Avg.	70.51%	94.43%	50.24%	79.46%	77.24%

4.4.2 Reduction of Glibc Vulnerability. Another security benefit of Picup is the reduction of vulnerable code in the library. That is, the library code containing the vulnerability is removed by Picup for specific input during one execution. It helps the program to avoid related attacks. To demonstrate this ability, we collected a total of 10 CVEs on glibc (GNU C Library) published in recent years. We prepared vulnerable libraries linked by the SPEC CPU 2006 benchmark program, and checked whether vulnerable functions were effectively removed by the debloating process.

Table 5 shows the evaluation result, including the 10 CVEs vulnerability functions on the 17 SPEC CPU 2006 benchmark programs. In 9 out of CVEs, the vulnerability functions were removed with an effect of no less than 15/17. In particular, three [26–28] of them were removed in all program runs. The worst result was the *posix_memalign* function in CVE-2018-6485 [25], which was retained by Picup for being a possible dependency for the program. From the perspective of the programs, 14 of the 17 programs also achieved a 90% reduction rate for the glibc vulnerability functions.

4.4.3 Case Study: Real-World Exploit Defense in Nginx. In order to further study the effectiveness of Picup under real exploits, we

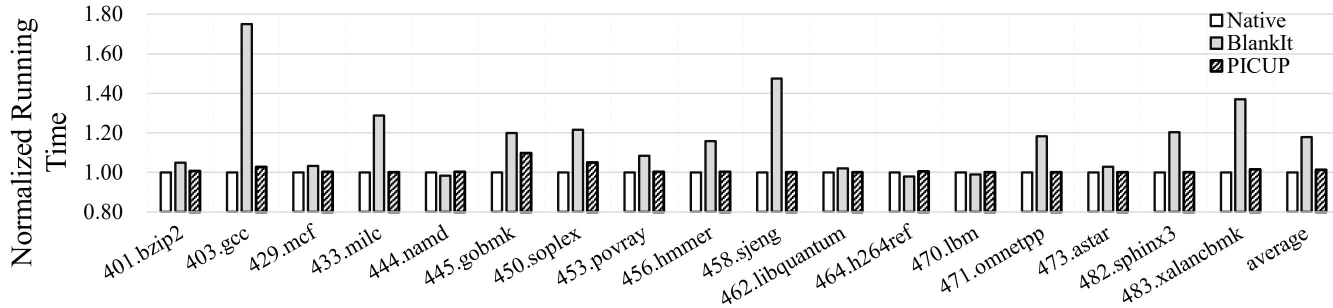

Figure 7: Normalized running time on SPEC CPU 2006.

Table 7: Details about the exploits in nginx defended by Picup

No.	Exploit Source	Type	Blocked Key API(s)
I	exploit-db	ret2system	system
II	github	ret2shellcode	mprotect
III	BROP	BROP	strcmp,usleep,dup2,execve

```
GET / HTTP/1.1
Host: 127.0.0.1
Transfer-Encoding: Chunked
Connection: Keep-Alive

[chunk size]
[chunk content]
```

(a) A normal input

```
GET / HTTP/1.1
Host: 127.0.0.1
Transfer-Encoding: Chunked
Connection: Keep-Alive

[large chunk size]
[payload]
```

(b) An exploit input

Figure 8: A normal input and an exploit input. The difference lies in the size and content of the chunk.

chose nginx [7], the most popular web server in the world [11], as the object of discussion.

CVE-2013-2028 [10] is a high-risk stack overflow vulnerability in the versions 1.39 to 1.40 of nginx. The remote attackers are allowed to crash or attack nginx via a chunked Transfer-Encoding request with a large chunk size, which triggers an integer signedness error and a stack-based buffer overflow.

We gathered three exploits from exploit-db [6], github [8] and BROP [5, 37] that can successfully attack nginx under CVE-2013-2028. The exploit I from exploit-db calls the `system` API in libc by reading the address in the GOT and adding an offset. The exploit II from github first makes the buffer executable by calling the `mprotect` API through the ROP chain, and then executes the shellcode written to the buffer. The exploit III from BROP is more complex. Briefly, the attack steps include: ① find a stop gadget and PLT; ② find the BROP gadget to control the first two arguments to calls; ③ find `strcmp` in the PLT to control the first three arguments to calls; ④ find `write` in the PLT to dump the entire binary to find more gadgets; ⑤ build a shellcode and exploit the server. Table 7 shows the details of these exploits.

Although these exploits differ in their approaches, the requests they send to nginx are actually similar in format (See Figure 8(b)). More specifically, in order to ensure that the vulnerability is successfully triggered, the header of these requests must be "Transfer-Encoding chunked", the same as the normal requests in Figure 8(a). These exploit requests differ from normal requests only in the oversized chunk size and the constructed payload. Therefore, when receiving these exploit requests, Picup will treat them as normal requests and provide the same APIs as normal requests.

In general, all these attacks can be successfully defended by Picup. In detail, exploit I fails since the `system` is not executable, and exploit II fails to run the shellcode when `mprotect` does not work. For exploit III, BROP fails in step ③ when trying to control the first three arguments since `strcmp` cannot be executed. Besides, at the last step it also cannot redirect the socket to standard input and output because `dup2` is disabled, cannot write payload because `usleep` is disabled, and cannot execute the shell because `execve` is disabled. Even if there are other ways to perform attacks, we believe the restricted APIs will at least increase their difficulty.

4.5 Runtime Overhead

In this section, we evaluate the runtime overhead of Picup on SPEC CPU 2006 and real-world applications. Besides, we also compare the runtime overhead of Picup with that of BlankIt [47].

4.5.1 Runtime Overhead on SPEC CPU 2006. First, we measure the running time of each benchmark in SPEC under Picup and compare it with the original running time. All the applications are tested three times, and the average normalized running time is shown in Figure 7. Please note that the data in Figure 7 about BlankIt is directly cited from the literature [47].

From Figure 7, we can draw several observations. First, our approach introduces less than 0.5% runtime overhead for 11 out of the 17 applications. In particular, the runtime overhead is only increased by 0.08% and 0.04% for 458.sjeng and 471.omnetpp, respectively. Second, Picup introduces lower overhead than BlankIt for 14 out of all the 17 applications. The overhead of our approach is only 1.32%, whereas the overhead of BlankIt is 18%. Third, we can observe that Picup outperforms BlankIt even for the worst case. The worst overhead of Picup is 9.68% for *445.gobmk*, while the worst case of BlankIt is 76% for *403.gcc*.

The main reason why Picup performs better than BlankIt is that BlankIt debloats for every API call during execution while Picup performs debloating only once when an execution receives an input. For instance, an execution of *458.sjeng* invokes API functions 24,766 times. With such a large number of API calls, BlankIt will perform debloat 24,766 times too, and thus introduces 150% overhead. As a comparison, Picup only introduce 0.08% overhead.

4.5.2 Runtime Overhead on Real-World Applications. Second, as shown in Table 8, we measure the incremental time brought by Picup and BlankIt on real-world applications by calculating the processing time between two inputs and minus the raw time. Due to the lack of partial code of BlankIt for training the prediction

Table 8: Runtime overhead on real-world applications.

Application	API Calls Rate	Picup (ms)	BlankIt (ms)
nginx	109.20	39.85	3.77
lighttpd	155.83	40.47	4.78
redis	200.05	17.52	3.04
memcached	1014.30	14.68	6.70
nm	27676.49	102.25	2294.07
readelf	84298.25	101.84	10482.16
objdump	1284344.10	101.32	117610.61

model, we skip the lightweight decision tree based predictions and directly estimate the overhead of BlankIt by applying the simplified rules as follows. First, when an API is called, we just copy the API and all its sub-functions (i.e., dependencies) back. Second, we clear all copied-back functions before another API is called.

Since Picup works on every input while BlankIt works on every API call, the key overhead difference between them is related to the ratio of input to API call, although the time spent by Picup on each input and BlankIt on each call may vary. For revealing the above factor, we measure the amount of API calls that the program makes after each input, which we call as *API Calls Rate*. As shown in Table 8, for low *API Calls Rate* programs, BlankIt performs better than Picup, but Picup also has an acceptable results. With the increasing of *API Calls Rate*, the overhead of BlankIt can be extremely high. Taking objdump as an example, it calls the API on average 1284344.10 times after each input and BlankIt generates 117610.61 ms of extra time, which is 1160x longer than Picup. By contrast, the overhead of Picup does not depend on *API Calls Rate* but can be influenced by the input itself, so Picup has a similar overhead on the applications that receive the same type of input.

5 RELATED WORK

Software debloating is a scheme to reduce the attack surface by eliminating unused code. Among types of debloating techniques, some of them aim to debloat the code in the program space rather than libraries. Trimmer [57] identifies unnecessary functions by user-defined configurations and then statically eliminates the code. DamGate [22] is a framework for dynamic feature customization. It uses static and dynamic analysis to rewrite binaries. Chisel [35] employs a reinforcement learning approach based on user-provided test cases to debloat software. Razor [48] uses binary rewriting to produce the program that only supports necessary functionalities. It does this by collecting the execution code of the software running on a given input and then uses heuristics to infer the non-execution code associated with the given input.

There are also several approaches in library debloating. PieceWise [50] introduces a specialized compiler to accomplish program debloating. It uses static analysis and training-based techniques to compute function-level dependencies and then removes unneeded functions at load time. Nibbler [13] performs similar library specialization at the binary level. It creates an application-level FCG by extracting the function call graph (FCG) of the binary and all imported libraries, and removes any untouchable code. In embedded systems, Ziegler et al. [70] do this by both static analysis and dynamic tracing, and a recent work μTrimmer [66] explores the offline library debloating for binaries on MIPS architecture. These offline library debloating techniques directly remove unused code from libraries and are robust during the dynamic execution. However, they retain code for all inputs and cannot remove more code for each concrete execution.

The current online technique, BlankIt [47], is a context-sensitive approach for debloating. It leverages a decision tree to predict sub-functions that will be used by the calling API based on call site, arguments and reverse dominance frontier of arguments, and then provides only those sub-functions. Although BlankIt removes more code, it is vulnerable if an attacker forges proper contexts.

There are also debloating techniques for specific applications. Kasr [69] aims to remove unused code from OS kernels. Several studies [52, 53] aim to slim down the containers. Other studies [38, 39, 61] aim to debloat Java programs, the Java Virtual Machine (JVM), web applications [16], Bluetooth stack [64], and browsers [49].

6 DISCUSSION

Error Handling. The prediction model may make false predictions, which can further result in exceptions that the required code is prohibited by Picup. Therefore, it is significant to handle errors and distinguish them from attacks. As in previous techniques [47], Picup can employ similar error handling mechanisms, such as a virtual machine with check pointing, memory forensics and memory safety check. For example, when an application runs fault due to a page permission error, we move the entire process to a secure monitored environment to continue running. If the process runs without risk operations, it is assumed that a prediction error has occurred and the corresponding page permissions will be restored. Conversely, an attack is considered to have occurred.

Malicious Input. Attackers may construct adversarial samples to maliciously misguide our prediction model and further invoke unused code. We acknowledge that this situation cannot be thoroughly avoided, but we argue that there are potential approaches that make such attacks difficult. First, we can employ some negative samples and label them as all-zero lists, which indicates that no code in libraries is permitted, to train the prediction model for robustness. Second, with the threat model of our approach, the details of the prediction model are agnostic to users. That is, the prediction model is a black box to attackers, which increases the difficulty for attackers to implement attacks.

7 CONCLUSION

In this study, we aim to balance the code reduction and the enforcement reliability of debloating and propose Picup, a per-input debloating approach that dynamically reduces the library attack surface for each input. We evaluate Picup on real-world benchmarks and popular applications. The experimental results show that Picup is a practical solution for secure and effective library debloating, which can predict the necessary library functions with 97.56% accuracy, and reduce the code size by 87.55% on average with low overheads.

ACKNOWLEDGMENTS

We want to thank all anonymous reviewers for their valuable comments and suggestions. This work is partly supported by National Natural Science Foundation of China under Grant No.62172305.

REFERENCES

[1] 2004. zlib 1.2.11 Manual. https://www.zlib.net/manual.html. Accessed July 09, 2021.

[2] 2007. Wikimedia Downloads. https://dumps.wikimedia.org/. Accessed Oct 10, 2021.

[3] 2009. Memcached. https://memcached.org/. Accessed July 08, 2021.

[4] 2010. Redis. https://redis.io/. Accessed July 08, 2021.

[5] 2013. Blind Return Oriented Programming (BROP). http://www.scs.stanford.edu/brop/. Accessed Oct 10, 2021.

[6] 2013. Nginx 1.3.9 < 1.4.0 - Chuncked Encoding Stack Buffer Overflow (Metasploit). https://www.exploit-db.com/exploits/25775. Accessed Oct 10, 2021.

[7] 2015. Nginx. https://nginx.org/. Accessed July 08, 2021.

[8] 2020. CVE-2013-2028 Exploit. https://github.com/m4drat/CVE-2013-2028-Exploit. Accessed Oct 10, 2021.

[9] 2021. angrop. https://github.com/angr/angrop. Accessed Oct 10, 2021.

[10] 2021. CVE-2013-2028 Detail. https://nvd.nist.gov/vuln/detail/CVE-2013-2028. Accessed Oct 10, 2021.

[11] 2021. September 2021 Web Server Survey. https://news.netcraft.com/archives/category/web-server-survey/. Accessed Oct 10, 2021.

[12] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (Alexandria, VA, USA) *(CCS '05)*. Association for Computing Machinery, New York, NY, USA, 340–353. https://doi.org/10.1145/1102120.1102165

[13] Ioannis Agadakos, 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* (San Juan, Puerto Rico, USA) *(ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 70–83. https://doi.org/10.1145/3359789.3359823

[14] Frances E. Allen. 1970. Control Flow Analysis. *SIGPLAN Not.* 5, 7 (July 1970).

[15] Frances E. Allen. 1970. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization* (Urbana-Champaign, Illinois). Association for Computing Machinery, New York, NY, USA, 1–19. https://doi.org/10.1145/800028.808479

[16] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is More: Quantifying the Security Benefits of Debloating Web Applications. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1697–1714. https://www.usenix.org/conference/usenixsecurity19/presentation/azad

[17] G BINUTILS. 2007. GNU Binutils. https://www.gnu.org/software/binutils/. Accessed July 08, 2021.

[18] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *2014 IEEE Symposium on Security and Privacy*. 227–242. https://doi.org/10.1109/SP.2014.22

[19] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (Hong Kong, China) *(ASIACCS '11)*. Association for Computing Machinery, New York, NY, USA, 30–40. https://doi.org/10.1145/1966913.1966919

[20] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics* 5 (06 2017), 135–146. https://doi.org/10.1162/tacl_a_00051 arXiv:https://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl_a_00051/1567442/tacl_a_00051.pdf

[21] Erik Bosman and Herbert Bos. 2014. Framing Signals - A Return to Portable Shellcode. In *2014 IEEE Symposium on Security and Privacy*. 243–258. https://doi.org/10.1109/SP.2014.23

[22] Yurong Chen, Tian Lan, and Guru Venkataramani. 2017. DamGate: Dynamic Adaptive Multi-Feature Gating in Program Binaries. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation* (Dallas, Texas, USA) *(FEAST '17)*. Association for Computing Machinery, New York, NY, USA, 23–29. https://doi.org/10.1145/3141235.3141243

[23] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks.. In *USENIX security symposium*, Vol. 98. San Antonio, TX, 63–78.

[24] CVE. 2013. Privilege Escalation Vulnerability in Linux x32 Configuration. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0038. Visited on July 07, 2021.

[25] CVE. 2018. CVE-2018-6485. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6485. Visited on Oct 10, 2021.

[26] CVE. 2019. CVE-2009-5155. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-5155. Visited on Oct 10, 2021.

[27] CVE. 2019. CVE-2020-1752. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-1752. Visited on Oct 10, 2021.

[28] CVE. 2021. CVE-2021-35942. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-35942. Visited on Oct 10, 2021.

[29] Solar Designer. 1997. Getting around non-executable stack (and fix). *http://ouah.bsdjeunz.org/solarretlibc.html* (1997).

[30] Peter Deutsch et al. 1996. GZIP file format specification version 4.3. (1996).

[31] Srikar Dronamraju. 2021. Uprobe-tracer: Uprobe-based Event Tracing. https://docs.kernel.org/trace/uprobetracer.html.

[32] H.H. Feng, J.T. Giffin, Yong Huang, S. Jha, Wenke Lee, and B.P. Miller. 2004. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*. 194–208. https://doi.org/10.1109/SECPRI.2004.1301324

[33] Christian Heitman and Iván Arce. 2014. BARF: a multiplatform open source binary analysis and reverse engineering framework. In *XX Congreso Argentino de Ciencias de la Computación (Buenos Aires, 2014)*.

[34] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[35] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 380–394. https://doi.org/10.1145/3243734.3243838

[36] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*. 969–986. https://doi.org/10.1109/SP.2016.62

[37] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1868–1882. https://doi.org/10.1145/3243734.3243739

[38] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 12–21. https://doi.org/10.1109/COMPSAC.2016.146

[39] Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. 2016. Feature-Based Software Customization: Preliminary Analysis, Formalization, and Methods. In *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*. 122–131. https://doi.org/10.1109/HASE.2016.27

[40] Cespedes Juan and Machata Petr. 2021. Ltrace man page. *URL https://man7.org/linux/man-pages/man1/ltrace.1.html* 1 (2021).

[41] Jan Kneschke. 2003. Lighttpd. https://www.lighttpd.net/. Accessed July 08, 2021.

[42] R. Krishnakumar. 2005. Kernel Korner: Kprobes-a Kernel Debugger. *Linux J.* 2005, 133 (May 2005), 11.

[43] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791

[44] Collin Mulliner and Matthias Neugschwandtner. 2015. Breaking payloads with runtime code stripping and image freezing. *Black Hat USA* (2015).

[45] Keiron O'Shea and Ryan Nash. 2015. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458* (2015).

[46] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[47] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. 2020. BlankIt Library Debloating: Getting What You Want Instead of Cutting What You Don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 164–180. https://doi.org/10.1145/3385412.3386017

[48] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *28 USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1733–1750. https://www.usenix.org/conference/usenixsecurity19/presentation/qian

[49] 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* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 461–476. https://doi.org/10.1145/3372297.3417866

[50] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 869–886. https://www.usenix.org/conference/usenixsecurity18/presentation/quach

[51] Mohit Rajpal, William Blum, and Rishabh Singh. 2017. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596* (2017).

[52] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: Automatically Debloating Containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 476–486. https://doi.org/10.1145/3106237.3106271

[53] Vaibhav Rastogi, Chaitra Niddodi, Sibin Mohan, and Somesh Jha. 2017. New Directions for Container Debloating. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation* (Dallas, Texas, USA) *(FEAST '17)*. Association for Computing Machinery, New York, NY, USA, 51–56. https://doi.org/10.1145/3141235.3141241

[54] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.* 15, 1, Article 2 (March 2012), 34 pages. https://doi.org/10.1145/2133375.2133377

[55] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *2015 IEEE Symposium on Security and Privacy*. 745–762. https://doi.org/10.1109/SP.2015.51

[56] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '07)*. Association for Computing Machinery, New York, NY, USA, 552–561. https://doi.org/10.1145/1315245.1315313

[57] 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* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 329–339. https://doi.org/10.1145/3238147.3238160

[58] Vincent Abella Starr Andersen. 2004. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, Data Execution Prevention. http://technet.microsoft.com/en-us/library/bb457155.aspx.

[59] PaX Team. 2003. Address Space Layout Randomization (ASLR). https://pax.grsecurity.net/docs/aslr.txt.

[60] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).

[61] Gregor Wagner, Andreas Gal, and Michael Franz. 2011. "Slimming" a Java virtual machine by way of cold code removal and optimistic partial program loading. *Science of Computer Programming* 76, 11 (2011), 1037–1053. https://doi.org/10.1016/j.scico.2010.04.008 Special Issue on Principles and Practice of Programming in Java (PPPJ 2008).

[62] F. Wang and Y. Shoshitaishvili. 2017. Angr - The Next Generation of Binary Analysis. In *2017 IEEE Cybersecurity Development (SecDev)*. 8–9. https://doi.org/10.1109/SecDev.2017.14

[63] Sanghyun Woo, Jongchan Park, Joon-Young Lee, and In So Kweon. 2018. CBAM: Convolutional Block Attention Module. In *Proceedings of the European Conference on Computer Vision (ECCV)*.

[64] Jianliang Wu, Ruoyu Wu, Daniele Antonioli, Mathias Payer, Nils Ole Tippenhauer, Dongyan Xu, Dave (Jing) Tian, and Antonio Bianchi. 2021. LIGHTBLUE: Automatic Profile-Aware Debloating of Bluetooth Stacks. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 339–356. https://www.usenix.org/conference/usenixsecurity21/presentation/wu-jianliang

[65] M. Zalewski. 2017. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[66] Haotian Zhang, Mengfei Ren, Yu Lei, and Jiang Ming. 2022. One Size Does Not Fit All: Security Hardening of MIPS Embedded Systems via Static Binary Debloating for Shared Libraries. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 255–270. https://doi.org/10.1145/3503222.3507768

[67] Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-Level Convolutional Networks for Text Classification. In *Proceedings of the 28 International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 649–657.

[68] Ye Zhang and Byron Wallace. 2015. A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. *arXiv preprint arXiv:1510.03820* (2015).

[69] Zhi Zhang, Yueqiang Cheng, Surya Nepal, Dongxi Liu, Qingni Shen, and Fethi Rabhi. 2018. KASR: A reliable and practical approach to attack surface reduction of commodity OS kernels. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 691–710. https://doi.org/10.1007/978-3-030-00470-5_32

[70] Andreas Ziegler, Julian Geus, Bernhard Heinloth, Timo Hönig, and Daniel Lohmann. 2019. Honey, I Shrunk the ELFs: Lightweight Binary Tailoring of Shared Libraries. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 102 (Oct. 2019), 23 pages. https://doi.org/10.1145/3358222

Received 2023-02-02; accepted 2023-07-27