HallMonitor: A Framework for Identifying Network Policy Violations in Software

Daniel Olszewski
University of Florida
dolszewski@ufl.edu

Weidong Zhu
University of Florida
weidong.zhu@ufl.edu

Sandeep Sathyanarayana
University of Florida
ssathyanarayana@ufl.edu

Kevin Butler
University of Florida
butler@ufl.edu

Patrick Traynor
University of Florida
traynor@ufl.edu

2022 IEEE Conference on Communications and Network Security (CNS) | 978-1-6654-6255-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/CNS56114.2022.9947243

***Abstract*—Debloating helps to remove unused and potentially vulnerable code from software. While such techniques are becoming more mature and practical, they focus on the features that are unwanted by users, and not on a wealth of functionality that is disallowed by administrative policy. For instance, while an administrator may use a firewall to block certain types of traffic, hosts readily interact with such traffic when the firewall is bypassed (e.g., via an encrypted tunnel). In this paper, we present HALLMONITOR, a tool that helps trim software functionality based on the violations of network policy. HALLMONITOR translates violations expressed in firewall and intrusion detection system rules (specifically, iptables and Snort) into parameters for detecting the implementation of such functions in source code spread amongst clients. We demonstrate the power of this approach first by removing echo functionality from ICMP, showing the ability to remove functions that enable higher-level attacks (e.g., network mapping). We then use network filtering rules to tag 14 out of 16 CVEs in Curl and Nginx (based on ground-truth from patches) to show the efficacy of our approach. In doing so, we demonstrate that network policy can be used to guide the removal not simply of code that users may not want, but instead of features that they are not allowed to use.**

I. INTRODUCTION

Commodity software comes packed full of features to best serve a wide set of potential clients, especially with the prominence of work-from-home scenarios. The difficulty with supporting such expansive functionality is not only that it requires additional storage and memory, but that it potentially increases the attack surface of mobile and dynamic networks. These unwanted features have commonly become known as software bloat.

The research community has recently invested significant effort in attempting to debloat software. Debloating looks for features such as dead code [1], [2], [3] or unused functions [4], [5], [6], [7], potentially reducing both the memory footprint and the number of exploitable vulnerabilities in client programs. These techniques have been driven primarily by users, whether through their explicit (and expert) configuration of debloating tools or via measurement of their actions. As such, these efforts generally do not consider the removal of functions that may violate policies within the larger administrative domain. As a trivial example, a domain administrator may enact a network filtering rule to prevent ICMP mapping of their network. However, hosts within that network would still respond to ICMP echo requests sent via a compromised host or unmonitored network tunnel. As such, an administrator may wish to additionally prevent network violations like this (and any other they already describe as filtering rules) at the hosts themselves.

In this paper, we present HALLMONITOR, a tool for removing the administrative domain policy violations from software based on network filtering rules. Specifically, we translate iptables and Snort configurations, which are both widely used by network administrators, into parameters through which HALLMONITOR searches software codebases for such violations. We then use two metrics: *semantic intersection*, which considers the summation of parameters in the candidate function to determine the likelihood of violation, and *inter-analysis* which estimate the ease of removal by combining paths through violations and affected variables throughout the paths. After walking through ICMP echo as an example, we then demonstrate the effectiveness of our approach in both Curl and Nginx, using code changes in CVEs as indicators of ground-truth.

We note that the problem we tackle is fundamentally different from that of debloating. Whereas debloating focuses on removing features a user does not use or need, *our efforts look to tag and remove functions they are not allowed to use.* As such, we make the following contributions:

- **Identify violations of network rules in source code:** We develop a method to translate between network filtering rules (i.e., iptables and Snort) and functions in source code that violate those rules/make the endpoint vulnerable. This approach makes policy portable, which is important as users are increasingly operating outside of traditional work environments (e.g., work from home).
- **Develop metrics for measuring the viability of function removal:** After identifying the candidate violations, we perform analysis on the resulting call graph and the control flow graph to evaluate the effect of the violated functions. Therefore, we can demonstrate the removal viability and the correctness of the violations.
- **Demonstrate performance against real-world violations:** We use 16 CVEs for both Curl and Nginx as ground truth (each containing software patch code) for determining how well HALLMONITOR locates violations in mature codebases. In each case, HALLMONITOR finds 7 of 8 vulnerabilities, demonstrating that our approach rapidly and accurately finds violations using tools systems administrators already possess.

It is critical for readers to recognize that network-based mitiga-

tions alone have limited impact. Specifically, while these techniques are indeed valuable, firewalls and other middleboxes are often circumvented in practice (e.g., via encrypted tunnels), allowing for malicious payloads to reach vulnerable hosts. Moreover, we note that the changing nature of work, which is increasingly mobile and conducted outside of traditional offices, means that an increasing number of networked assets will operate beyond the protection of such network appliances. HALLMONITOR works as a complementary defense for network administrators to provide an in-depth defense in the wake of these challenges. Fundamentally, we seek to solve problems raised by Ioannidis et al. [8], who remark on the challenges of enforcing policy at endpoints given the ease with which unauthorized entry points to the network can be created. While their approach relies on enforcing firewall policies on the host through complicated IPsec mechanisms, our approach allows administrators to remove potentially-malicious functionality directly from endpoint code.

The remainder of our paper is organized as follows: Section II provides a background; Section III outlines our definitions and design of HALLMONITOR; Section IV shows a case study of HALLMONITOR performing on CVEs; Section V discusses the results and their significance; Section VI contains plans for future work and the limitations of HALLMONITOR; Section VII discusses the related work; Section VIII concludes our paper.

II. BACKGROUND

Software debloating seeks to remove unnecessary code, whether it is dead code in the sense of unused libraries or if it is dead in a specified user configuration (e.g., debug for an application in production). The secondary focus of these approaches is on reducing the attack surface of an application. Most recent work focuses on removing code that is never used during runtime (e.g., third-party libraries unnecessarily imported [2], [1]), code that would only be used on the process starting (e.g., web servers binding to a socket [9]), and code that does not meet a user-defined set of required functionalities [4]. Other approaches use machine learning to interact with binaries and identify when code can be removed [7]. These methods for removal focus on allowed functionality within the application to motivate the removal of code. We base HALLMONITOR on *disallowed processes within the network* (e.g., a firewall blocking an ICMP packet) to motivate which code to remove. Figure 1 shows a high-level comparison of our process and debloating. Mapping network activity to a host's or client's applications is a challenging problem in and of itself, as there is no centralized place to pull packet governance in a network [10], further exacerbated by remote work. Thus, considering endpoints in the source code provides network administrators a complementary defense in the face of these challenges.

Many current debloating techniques require extensive knowledge of both the application used for debloating and the system that is debloated. This varies from requiring user input on each function's necessity within the source code to scripting the required functionality. As pointed out by Qian et al. [4], the problem of debloating based on user input is an open and difficult problem. Thus, optimization is based on heuristics.

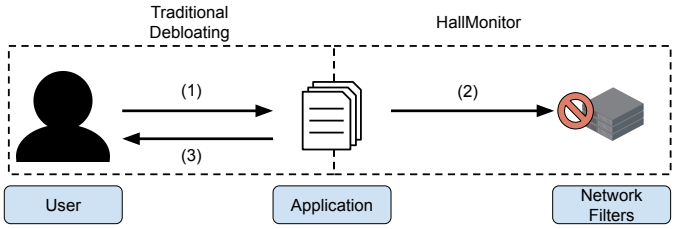

Fig. 1: Traditional Debloating techniques consider how a user interacts with the application. We consider how the application interacts with the network (i.e., what will the network block from the application). We can see that (1) the user gives some input to the application. (2) the application sends information through the network, but the network filters block it. (3) the application informs the user. HALLMONITOR takes what the network filter rules block to identify source code that would allow this to happen.

For identifying code, traditional debloating techniques focus on whitelisting functions and anything not whitelisted (i.e., functionality the user does not use) is considered for removal. *We focus on identifying code that violates network policies for removal* and thus, fundamentally change the problem. While recent work [5], [11], [4], [12] seeks to either automatically reduce code size or require easier user interaction, we propose HALLMONITOR to find policy violations in the source code. Primarily, HALLMONITOR outputs targeted and meaningful violation analysis by employing tools already in use (e.g., firewalls).

III. DESIGN AND IMPLEMENTATION

We seek to identify functions that would not be allowed by a network filter and thus are violations of policy that extend the attack surface. To find the lines of code, we take the network filter's rules and extract semantic meaning (i.e., create search terms from the rules that indicate associated functionality within the source code). We further identify the correctness of the found functions by analyzing the cover of the call graph and using the semantic intersection, I_{sem}. An overview of our process can be seen in Figure 2. We will refer to the network filter rules as R, generated search parameters as S, and the code we are analyzing as C. Thus, our goal is to:

1) Generate S from R.
2) Use S to find the set of violation functions, F, in C.
3) Build the call graph, G, and control flow graph CFG from C.
4) Determine the ease of removal for F by analyzing the number of paths in G and CFG.

We present our working definitions and the formal framework of HALLMONITOR in Section III-A and Section III-B. Next, we show how HALLMONITOR constructs the callgraph and analyzes the code in Section III-C, Section III-D, and Section III-E. Finally, we go through a detailed example of how HALLMONITOR operates on `icmp.c` in Section III-F.

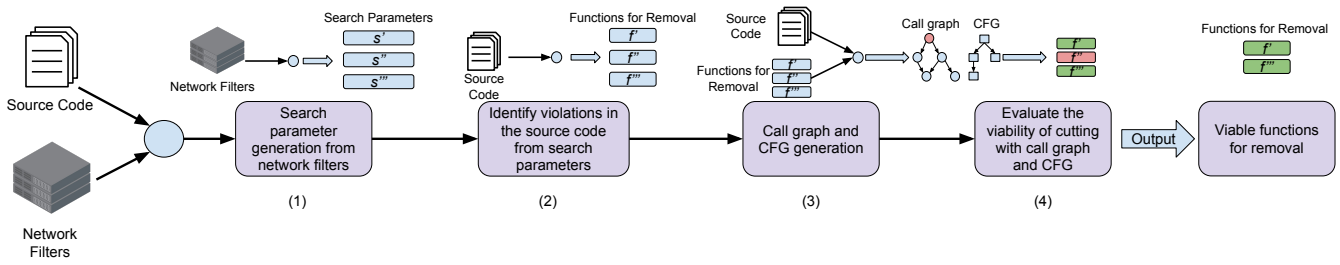

Fig. 2: The overview of our process. We take in network filters and the source code to analyze. (1) Generate search parameters from the network filters. (2) Identify violations in the source code using the search parameters. (3) Generate the call graph and control flow graph (CFG) with source code. (4) Analyze the viability of removal based on the call graph and CFG.

A. Working Defintions

In this section, we define all of the working definitions we use. **Network Filter Rules**: R is a set of rules. Each rule, r, blocks functionality within the network. Our primary motivation is that the disallowed functionality directly correlates to features in the application. Thus, R points to redundant features at the application level.

Arguments: Each r contains arguments. We define these individual elements as a, and they specify the behavior of r. Each a provides our framework for generating semantic meaning. They are syntactically denoted by an argument flag (e.g., "-p" in iptables is a protocol flag).

Search Parameters: S_r is a set of strings. Each $s \in S_r$ is generated from each r in R. S contains the semantic meaning of R (i.e., an r that blocks SSH generates semantic meaning that can be referenced in the source code). This semantic meaning is derived from arguments, a in r.

Sensitivity: Since there are multiple search parameters per rule, not all search parameters in a given rule will trigger during analysis. Thus, we specify a sensitivity parameter, $\sigma \in \mathcal{Z}$, that dictates how many times a function must trigger the search parameters. If a function triggers more than σ search parameters, we add this funciton to F. A higher σ will result in less output whereas a lower σ will result in more output.

Violations: F is the set of functions in the application, C, we are analyzing that is not allowed by the network. Thus, allowing a violation of network policy. Each f in F is a function and is contained in the call graph of C.

Call Graph: The call graph, G, is the graph of functions that denote call order and function dependency.

Control Flow Graph: The control flow graph, CFG, represents the paths that a program could be traversed during the execution.

Entry: The function that is the root of the call graph, G, is the Entry. We define e as the Entry into the call graph.

Parent Set: The Parent Set of f is the set of functions that directly call f.

Child: The function that f directly calls is the Child of f.

parseArgument(): We pull the semantic information, s', from each argument, a, in each rule, r. We filter s' into the complete semantic information s. For example, if s' is "HTTP", then to get all of the semantic information out, we would filter s' into s =["hypertext", "transfer", "protocol"].

lookup(): R is built over ports and IP addresses. To get semantic meaning from them, we perform a look-up of which protocol is built over the port or what the hostname for the IP address is.

Algorithm 1 Identify Lines of Code as Violations.

```
1: procedure IdentifyViolations(C,R,σ)
2:     S = []
3:     F = []
4:     // Parse R
5:     for each rule r in R do
6:         for each flag f in r do
7:             s = f.parseArgument()
8:             if s is Port or Address then
9:                 s = lookup(s)
10:            end if
11:            S[r].append(s)
12:        end for
13:    end for
14:    // Depth-search C
15:    for each file f in C do
16:        for each line of code c in f do
17:            for r in R do
18:                if c contains more than σ parameters from S[r] then
19:                    F.append(c)
20:                end if
21:            end for
22:        end for
23:    end for
24:    return F
25: end procedure
```

B. Identify Violations

Algorithm 1 outlines the process for generating the search parameters and identifying violations within the code. First, we take in the set of network filter rules, R, and begin parsing them into the associated semantic search parameters. For this, we convert the arguments, a, in each rule, r, to search parameters. Then, if a is an IP address port, we convert a into the name of the port or the hostname using the IANA defined port names [13] (e.g., "-p 80" becomes "http"). Next, we perform a depth-first search through each directory and file to analyze the code. We consider each line of code separately, except when a function or condition spans multiple lines. In that case, each line is concatenated and then analyzed. HallMonitor then iterates through each rule against that

Algorithm 2 Call graph generation algorithm.

```
1: procedure CALLGRAPHGEN(F)
2:     CallGraph = []
3:     for each Func in F do
4:         CallGraph = CallGraph.append(Func)
5:         for each Element in CallGraph do
6:             ParentSet = [functions that are calling Element]
7:             CallGraph += ParentSet
8:         end for
9:         for each Element in CallGraph do
10:            ChildSet = [functions that are called by Element]
11:            for Each Child in ChildSet do
12:                ParentsOfChild = [functions that are calling Child]
13:                if ParentsOfChild in CallGraph then
14:                    CallGraph = CallGraph.append(Child)
15:                end if
16:            end for
17:        end for
18:    end for
19:    Return CallGraph
20: end procedure
```

line of code, comparing each of the search parameters in each r against the line of code. If there are more than σ instances, we then consider this line of code as a candidate for violation.

C. Call Graph Generation

Functions from set F will be deemed as the entry points for building the call graph. However, traditional strategies [14], [9], [12] for building the call graph are started from the beginning (i.e., entry function) of the workflow, whereas the entry functions in set F could locate in the middle of the call graph, meaning the entries could also be referenced by other functions. Thus, we need to start the traversal by searching both directions (i.e., called and referenced functions) starting from functions in F. However, this traversal approach will lead to two issues: (1) the functions that are called by F could be used by functions useful for other modules and we cannot include them in the call graph; (2) the functions calling F could call other functions, which need to be included into the call graph if they are not called by other modules.

Algorithm 2 shows the process to generate the call graph based on the entry functions (F). We use $CallGraph$ to record the functions in our generated call graph and it is initialized with an empty set. Thus, for entry functions, we first insert them to the $CallGraph$. Then, we need to find out all the functions ($ParentSet$) that call the entry function. Since the functions in F are only served for the target functionality we want to remove, the functions within $ParentSet$ could be seen as the ad-hoc functions for the target functionality. Therefore, we append the $ParentSet$ to the end of $CallGraph$ and continue the traversal of $CallGraph$ until no functions call the elements within the $CallGraph$. Then, we start the traversal again to the $CallGraph$ to find out all the functions ($Child$) that are called by the $CallGraph$. However, some $Child$ functions are shared and called by other functions. Thus, we cannot include those functions into $CallGraph$. Therefore, if $Child$ is only called by the functions within $CallGraph$, it will be appended to the $CallGraph$. In the end, $CallGraph$ and the functions within it are only correlated with F. Thus, we will perform viability analysis with the generated $CallGraph$.

D. Viability Assessment

Based on the analysis of network filter rules and source code, we could get the function (*viol_function*) where the violation happened and the function (*trigger_function*) that is calling *viol_function*. The goal of our work is to remove a feature according to the violation and thus we need to demonstrate the viability to achieve such a target. For this reason, we evaluate the viability of cutting by analyzing the semantic context from the source code. Since the removal of a specific function, which could be used by other functions, can lead to the failure of compilation and malfunction of other features, we thus need to know the scope of the affected functions when we want to mitigate based on the *viol_function*. Furthermore, if we cannot remove a feature by simply trimming some functions, we must perform fine-grained analysis (e.g., manual analysis by an engineer) to the source code and we need to discern the workload for analyzing the source code to indicate the viability of removal. Thus, we propose *inter-analysis* as a strategy to identify the feasibility of removal.

Inter-analysis evaluates the correlation between the *viol_function* and other functions in the source code to demonstrate the affected scope when we remove *viol_function*. We exploit the following metrics to demonstrate the affected scope: path number and source code coverage ratio. Within the call graph, the path refers to the function dependency throughout the call graph. Therefore, we assume several paths intersect with a function and refer to that function as the cross-point. We set the cross-point to *viol_function* and search the $CallGraph$ to figure out the path number (i.e., $TotalPaths$). More paths mean a larger range and thus indicate the difficulty will be higher in this circumstance. Removing a function can only lead to an error in the functions that are calling or indirectly calling the removed function. We define those functions that are directly or indirectly calling the deleted function as *upper-functions*. Thus, we calculate the ratio (i.e., *UpperRatio*) of the line number of *upper-functions* to the line number of the entire source code to illustrate the difficulty of trimming by which the higher code coverage ratio of *upper-functions* means higher difficulty and lower viability for trimming. In addition, if the path number and the *upper-functions* code coverage ratio are too high, we cannot remove the feature by simply deleting the *viol_function* and thus we need to have a fine-grained analysis.

E. Semantic Intersection

We propose another metric for analyzing the correctness of HALLMONITOR. The semantic intersection of the function, I_{sem}, is defined for each rule and function. It considers the total number of times within the function definition that the search parameters are triggered for the rule normalized over the number of search parameters. We can see a visual depiction of I_{sem} in Figure 3. Formally, B represents the set of strings in the code block of the function definition, S_r is the set of search parameters, and

$$I_{sem} = \frac{|S_r \cap B|}{|S_r|}. \tag{1}$$

It is important to note that this metric can be greater than one as a search parameter $s \in S_r$ can trigger more than one time

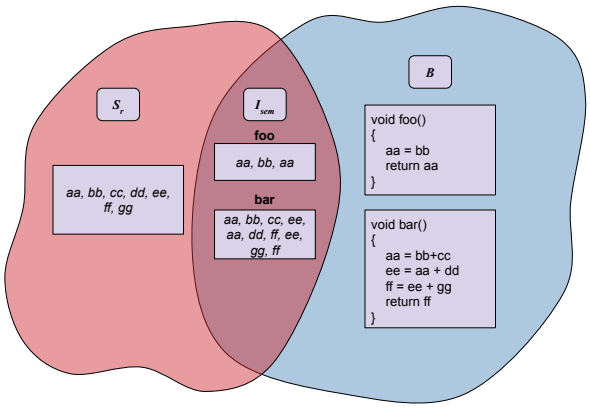

Fig. 3: We show I_{sem} as the intersection between the search parameters, S_r, and the code block B. For example, in the code block of foo. The search parameters *aa* and *bb* appear for a total of three times. In the block of code bar, the parameters appear 10 times. Therefore, I_{sem} is taken to be that intersection divided over the number of search parameters (7) in this case. For foo, I_{sem} = 3/7 = 0.42. For bar, I_{sem} = 10/7 = 1.42. Thus, we would be interested in function bar and not function foo.

within the code block. We can think of I_{sem} as the coverage of HALLMONITOR over a violation. If I_{sem} is greater than one, it shows that HALLMONITOR found more instances of the search parameters in the code block than there are search parameters. This indicates a high likelihood of violation.

F. ICMP Example

We consider a trivial example of ICMP in the Linux kernel. Network administrators will block incoming and outgoing ICMP messages, as ICMP can allow an attacker to map the internal network. Iptables suffice to block ICMP echo and echo replies. Consider the following iptables rule.

```
iptables -A FORWARD -p icmp --icmp-type
    0 -j DROP
```

This rule blocks all incoming and outgoing ICMP echo replies. Any source code functionality filtered by this iptables rule is obsolete. The function, `icmp_reply` provides the functionality of replying to ICMP echo and is a part of the attack surface. Thus, we motivate our work by identifying violations of network policies that the source code allows for. These violations extend the attack vector and should be removed.

Network Filter Parsing: For this example, we use iptables. We can see how HALLMONITOR parses the previous iptables rule in Figure 4. We pull out each argument from the rule and generate a group of search terms from each argument. The table name `output` becomes a search term. We add each word in `icmp`, internet control message protocol, as well as `icmp` itself. For the type, we perform a lookup according to IANA standards [15] and then clean the parameters (e.g., extend acronyms and remove special characters). The final list of search terms is output, icmp, internet, control, message, protocol, echo, and reply.

Source Code Analysis: We now analyze the `icmp.c` file from the Linux kernel with all of the generated search parameters.

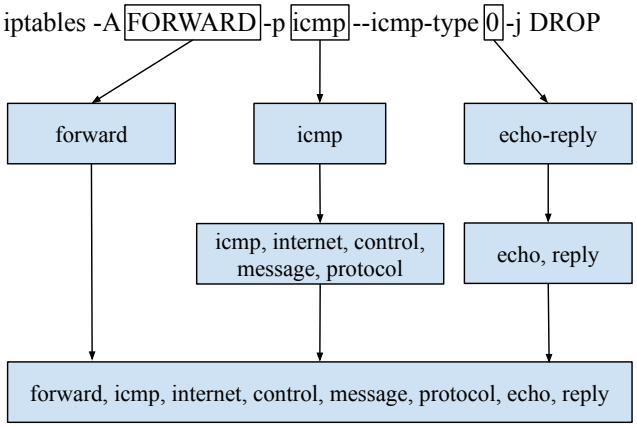

Fig. 4: An example of parsing the iptable rule. HALLMONITOR pulls out flags from the iptables rule. It then generates search parameters from the flags by removing special characters and fully evaluating acronyms.

HALLMONITOR performs a line-by-line comparison of the file match to each search parameter. A line of code is considered for further evaluation once it triggers more than or equal to σ. Consider $\sigma = 2$ as our example[1]. For instance, in the function `icmp_echo`, we find lines 1, 5, 9, and 14, as they all contain at least two of the search parameters, echo, icmp, or reply. The four lines of code we found are just a subset of the total lines HALLMONITOR finds within `icmp.c`. The lines of code where the search parameters are in a function (1 and 14) show an area within the call graph that represents violations, and we further evaluate their priority by analyzing the call graph.

Call Graph Generation: Based on the result from the source code analysis, we could start the generation of call graph from function *icmp_reply* with the method in Section III-C. Then, we perform inter-analysis with *icmp_reply* as the *viol_function* and *icmp_echo* as the *trigger_function*.

Viability Assessment: Using the strategies within Section III-D, we calculate the path number and upper-function ratio as 2 and 7.3%, respectively. Thus, we know the viability for cutting, in this case, is *High* since only a small scope of functions in the source code are affected by the *viol_function*. Moreover, within *icmp_echo*, only a very small portion of statements are correlated to the variables used by *icmp_reply*. Therefore, it is also possible to trim the *viol_function* by only analyzing the *trigger_function*.

IV. EXPERIMENTAL RESULTS

We propose the following research questions to guide our experiments:

RQ1 *Security*: Can we use network policies associated with known vulnerabilities to find network violations in source code?

RQ2 *Viability*: How difficult are the violations we identify to remove from the source code?

[1] σ is an integer and directly specifies how many search parameters must be triggered to be considered a candidate violation

```
1 static bool icmp_echo(struct sk_buff *
      skb)
2 {
3   struct net *net;
4   net = dev_net(skb_dst(skb)->dev);
5   if (!net->ipv4.sysctl_icmp_
      echo_ignore_all)
6   {
7     struct icmp_bxm icmp_param;
8     icmp_param.data.icmph = *icmp_hdr(
      skb);
9     icmp_param.data.icmph.type =
      ICMPECHO_REPLY;
10    icmp_param.skb = skb;
11    icmp_param.offset = 0;
12    icmp_param.data_len = skb->len;
13    icmp_param.head_len = sizeof(struct
      icmphdr);
14    icmp_reply(&icmp_param, skb);
15    }
16  return true;
17 }
```

Fig. 5: caption=Excerpt from `icmp.c` to demonstrate how HALLMONITOR analyzes code. We bold our generated search parameters found in this code section.

RQ3 *Scope*: Does HALLMONITOR reduce the analysis an engineer would have to perform to manage the violations?

To evaluate our results, we propose metrics to evaluate the difficulty, complexity, and correctness of our approach. We then demonstrate HALLMONITOR and the associated metrics in an empirical case study.

Metrics: We will evaluate the violations according to the metrics *inter_analysis* and I_{sem}. Finally, we consider another metric, the total number of functions identified over the total lines of code within the source code.

With F as the set of functions that violate policies and C as the total set of code, we let $U = \frac{|F|}{|C|}$, that HALLMONITOR identifies. For the purpose of scope, we want this metric to be small as this shows that HALLMONITOR is not just outputting the majority of the program as violations. Thus, limiting the amount of work an end-user will need to do. As $|F|$ is dependent on σ, the sensitivity parameter, we present our results with the context of varying σ.

Empirical Case Study: We use a case study to show the correctness of HALLMONITOR on larger, mature codebases that provide measures of ground-truth against which HALLMONITOR can be evaluated. Specifically, the exploitation of vulnerabilities represents a violation of network policy, and many CVEs contain lines of code corresponding to vulnerabilities. As such, we consider identifying patched code for CVEs (similar to Ghavamnia et al. [9]) as an unbiased measure for the correctness of our approach. *We do not use CVEs or patch notes in* HALLMONITOR. We compare the output of HALLMONITOR on a codebase to CVEs for the given codebase.

Nginx [16] and Curl [17] are two networking tools that are customizable and present many attack vectors. We look at eight CVEs for Nginx and Curl. We use patch release notes and associated code modification for both Nginx and Curl. The primary goal is to show that by using search parameters generated from Snort and iptables, we can point to direct functions before patching what the patch modified. We use the set of Snort Community rules [18] and a list of 12 iptables rules that target common functionality. Moreover, after we get the entry functions from previous approaches, we generate a call graph for further analysis. Since Nginx and Curl are written with C language, we use LLVM to generate Intermediate Representation (IR) code (i.e., bitcode) and then link them with link-time optimization (LTO) tool to get an overall bitcode. After that, we generate an overall call graph with LLVM provided tool (i.e., `opt`) and then perform the methodology in Section III-C to generate the call graph based on the *viol_function* (i.e., entry function). Additionally, we generate CFG of the *trigger_function* from the bitcode. Therefore, we could deploy *inter-analysis* and *intra-analysis* with generated call graph and CFG.

	CVE	Rules	σ	I_{sem}	#Path	UpperRatio
Nginx	2018-16845	Snort	3	27.10	12	0.40%
	2017-7529	Both	2	1.65	1	0.02%
	2016-4450	Snort	2	8.00	3840	1.87%
	2014-3556	Snort	2	1.50	2	0.10%
	2013-4547	iptables	2	0.85	6	0.34%
	2012-1180	-	-	-	-	-
	2009-3555	Snort	3	3.80	1	0.39%
	2009-2629	Snort	3	12.10	4	0.40%
Curl	2021-22890	Both	3	1.38	6	0.19%
	2020-8177	Snort	3	2.47	2	0.55%
	2018-1000007	iptables	3	2.70	192	0.90%
	2016-8624	Snort	3	24.40	36	2.28%
	2016-7141	Snort	2	0.31	6	0.67%
	2016-5419	Snort	3	2.63	36	2.29%
	2016-4802	-	-	-	-	-
	2016-0755	Snort	3	2.61	36	2.56%

TABLE I: This table shows the results for the empirical study from the network filters. We show the different CVEs for which network filters were able to identify associated patch code, and the sensitivity, σ, required to find them. We are able to identify seven out of eight CVEs for both Nginx and Curl. The next column shows the metrics (I_{sem}, $\#Path$, and $UpperRatio$), for each of the identified vulnerabilities. The metrics give an indication for how possible the removal will be.

Case Study Results: Of the eight CVEs for both Nginx and Curl, we can identify code modified by the associated patch for seven out of the eight. The results for each CVE can be seen in Table I. Only two of the CVEs associated I_{sem} are lower than 1.0, with the rest over one and three over 10.0. Indicating that semantically, many of the associated functions contained numerous instances of the associated search parameters. It is important to distinguish I_{sem} from the sensitivity, σ. I_{sem} is a metric that regards the total number of search parameters found throughout the function definition, whereas σ only considers line-by-line what the semantic entropy is. The CVEs associated with Nginx were split on sensitivity. Three CVEs were found at a higher sensitivity of three, whereas four were found at a sensitivity of two. For Curl, we can see that of the found CVEs only one required a low sensitivity of 2. The rest we can find at three. To compare these sensitivities, we refer to Figure 6. Iptables output an average of 1.5%, 0.5%, and 0% violations for σ equals two, three, and four, respectively. Snort outputs

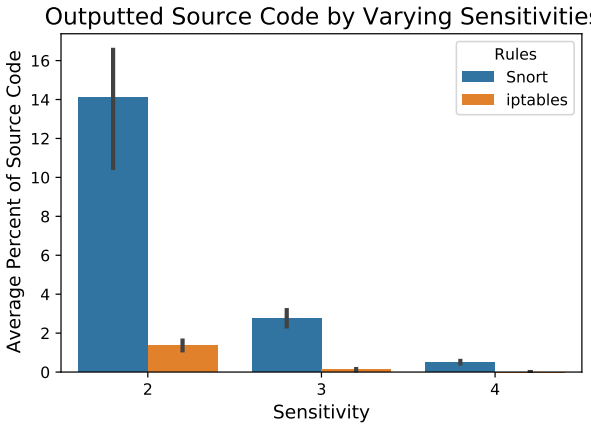

Fig. 6: This Figure shows the percent of code HALLMONITOR outputs out of the total size of the source code with the different levels of sensitivity. The split is between the two network filters we used, iptables and Snort. Notice that iptables outputs significantly less at $\sigma = 2$ than Snort rules do. This is due to the available semantic information. For Snort, σ at 3 results in the most meaningful output as it has the smallest violation identification while being able to find 9 of the 16 CVEs. For iptables, σ at 2 contains the most information.

an average of 15.1%, 3.1%, 0.5% for σ equals two, three, and four, respectively. Snort outputs more due to the available semantic information from the arguments, a, in each Snort rule.

For Nginx, the average percent violations identified out of the total lines of code, U, for sensitivities of two, three, and four are 14.9%, 2.18%, and 0.42%, respectively. For Curl, the sensitivities of two, three, and four resulted in a violation identification of 17.2%, 3.5%, and 0.6%. Overall, a sensitivity of four is too small to trigger any meaningful code related to the patch. At a sensitivity of three, there are enough triggered lines of code to find many of the CVEs. After we analyze the CVEs with our iptables-based and Snort-based approaches, we can use the *viol_function* for viability assessment. We can see the associated viability metrics in Table I. The largest I_{sem} was 27.10 with the largest found being 0.31. For the $TotalPaths$, we have a significantly large path number for CVE-2016-4450 at 3,840, indicating it is unlikely to have a simple removal for the identified code.

V. DISCUSSION

a) Security: The primary goal of HALLMONITOR is to identify violations in source code. We consider how well HALLMONITOR can identify violations by comparing patch code associated with CVEs for the known systems with the output of HALLMONITOR. Unlike other debloating techniques which seek to remove as much code as possible, we seek to identify targeted lines of code that remove disallowed functionality from the network. Performance measurements such as overall binary reduction and ROP removal do not hold the same meaning for our approach. Similarly, Razor [4] and Chisel [7] compare against CVEs related to Linux kernel commands (e.g., gzip, grep, mkdir, mv). None of these CVEs touch network functionality and thus are out of the scope of our analysis.

Overall, we found the patch code for seven out of the eight CVEs in both Nginx and Curl. Thus, we find violations associated with known vulnerabilities (**RQ1**). The CVEs we consider relate to read and write, TLS/SSL, buffer overflow, and parsing errors. For example, CVE-2009-3555 is a renegotiation vulnerability in SSL protocol. The violation in question is `ngx_http_ssl_verify_callback`. The Snort rule that triggered this violation creates an alert when a packet destined for the webserver has an established connection and attempts to overflow a challenge length. HALLMONITOR finds five out of the six functions modified by the patch as violations. Further reflecting the coverage of HALLMONITOR is that for this CVE, I_{sem} is 3.8. Thus, showing that the search parameters that found the function has high coverage over the associated code. This indicates that the identified violations in the source code are closely related to vulnerabilities.

HALLMONITOR cannot identify any violations related to CVE-2012-1180. This CVE is a memory disclosure with a specially crafted backend for Nginx. Since HALLMONITOR does not perform deep packet inspection, we are unable to identify vulnerabilities related to buffer overflow or errors incurred by processing malformed data. For Curl, we are unable to identify CVE-2016-4802, which allows an attacker to execute arbitrary code when Curl is initialized with telnet or SSPI. The violations associated with the patch cannot be detected by HALLMONITOR because the patch modifies initialization functions and library loading. Thus, while HALLMONITOR can target many network-related violations pertaining to vulnerabilities, it is unable to identify vulnerabilities that take advantage of dynamically linked library loading or initialization of the project.

b) Viability: To show the viability of removal, we consider the metrics of *inter_analysis* for the CVEs we identified violations for. These metrics consider the number of paths going through the violation. The *inter_analysis* relates how removing the violation will impact the surrounding functions. We see that eight out of the fourteen violations identified for the CVEs have less than 10 paths through them. Thus, they are candidates for quick and easy removal. Of the remaining six, four have less than 50 paths through them, one has 192, and the last has 3,840 paths through it. These violations are less viable than the others. Overall, eight out of the fourteen violations HALLMONITOR identifies are easily removed (**RQ2**), with the remaining six requiring more complex removal.

c) Scope: Unlike traditional software debloating systems, we do not approach the problem as a total removal. We are concerned about *targeted* and *meaningful* output and consider the total code outputted as a good measure of the efficiency and usefulness of HALLMONITOR. By lowering σ, we can raise and lower our false-positive rate similar to lowering the probability threshold in an Intrusion Detection System. The problem is that by lowering σ we increase the total output. If HALLMONITOR outputs 85% of the source code as a violation, this does not indicate or instruct an end-user in any meaningful way. It is of the utmost importance that HALLMONITOR outputs a small percentage of the source

code as violations. Thus, with a small number of violations, the end-user can further analyze the importance of the violations.

Iptables output significantly fewer candidate violations than the Snort rules. This is primarily due to the fact that the Snort rules contain more semantic information (e.g., authors put an alert comment on each rule). The largest amount of output for iptables and Snort rules is at $\sigma = 2$, which outputs 1.5% and 15.1%, respectively. We notice for Nginx that four of the eight CVEs can only be found at this level (two of which are found by iptables), while only one CVE is found for Curl. Increasing σ to three, HALLMONITOR outputs only 0.5% of the source code for iptables and 3.1% for Snort rules. Similarly, six out of the remaining seven CVEs we identify for Curl are found at $\sigma = 3$, while only three for Nginx. Thus, we can limit the output for an end-user verifying the ease of use as well as the correctness of HALLMONITOR (**RQ3**).

VI. LIMITATIONS AND FUTURE WORK

HALLMONITOR identifies possible functions to trim from the callgraph based on allowed network functionality. It does not perform deep packet inspection (DPI). This results in missing CVEs related to packet contents and packet parsing. We are also unable to identify CVEs related to a buffer overflow. Primarily, many patches involving an overflow of a variable are difficult to catch and relate to buffer size rather than actual network functionality. The search parameters can arbitrarily trigger a line of code. Our biggest limitation is semantic differences between function names and generated search parameters. In other words, concatenated words used for function or variable names are not triggered by the complete generated word. Other times, the search parameters erroneously trigger lines of code simply because the line of code contained the characters of the search parameters without the same semantic meaning (e.g., "app" flagging a line of code containing "overlapped").

We see this work as a promising start for relying on network-based software debloating. Future work includes combining search parameter generation by connecting it to the CVE information that is present in many Snort rules. We could use the associated patch code with the CVE to generate new search parameters for the source code analysis. Another avenue to consider is the semantic differences between the search parameters and the code. Semantic analysis of source code has been used to detect plagiarism of code [19], identify topics in source code [20], and fix code [21]. Using similar approaches, we can guide the analysis by using semantic differences. In future work, we consider modifying the search algorithm to always iterate over network search parameters. Then for the line of code to be flagged, it would need to hit specific search parameters directly related to the code (e.g., "http" will always be searched in the line of code, but it will not be flagged unless it also hits "connect_to" generated by the network filter).

VII. RELATED WORK

a) Software Debloating: Many approaches have been suggested in removing unused code. Jiang et al. [2] propose JRed, a Java Runtime Environment specialization framework. By analyzing a constructed callgraph and using points-to analysis, they can trim unused classes and methods from a Java program. Ghavamnia et al. [9] propose an elegant temporal system call specialization approach for reducing the attack surface to servers. They do so by statically analyzing source code and identifying functions not used after server-initialization and thus reducing attack surfaces of web servers. For developing a complete usage profile, Azad et al. showed a reduction in code size and CVEs for web servers by developing a complete usage case [6]. They did this by crawling the webserver, using tutorials from the internet to get usage, and simulating the user interaction with random clicks. As software systems grow in complexity, the end-user may not use all of the features associated with a program. This provides a great opportunity for debloating code and reducing the attack surface. Another approach by Mulliner et al. [1] removed Dynamic Loaded Libraries from the Control Flow Graph. CHISEL [7], CARVE [5], TRIMMER [12], TOSS [22], and RAZOR [4] use varying CFI policies derived using machine learning. They analyze user interaction, tutorials, and configuration to do so.

b) Automatic Patching: The research community seeks to apply automatically generated patches. Keromytis introduces work that would allow a central authority to automatically update its deployed systems by monitoring exploits discovered via a honeypot [23]. Further work explores the feasibility and usefulness of automatic patches against network worms [24], [25]. Patching software vulnerabilities remains largely a manual task, but recent research suggests that identification of such vulnerabilities remains an open and promising area [26]. Many methods include using patches from other software to inform new systems. Systems such as GenProg and Par use genetic algorithms to generate patches from previously created patches [27], [28]. Qi et al. [29] build on these ideas by measuring the effectiveness of random algorithms. SimFix [30] compares against a set of defined patches with differences in source code to build a candidate patch. Tools like Genesis [31] and Prophet [32], use these previous patches to identify fixes for current vulnerabilities, while Sidiroglou et al. [33], [34] uses donor software applications to mitigate bugs in code. Another focus is on mitigating the downtime of servers, such as KSplice [35]. Other tools [36], [37], [38] mitigate atomicity and concurrency bugs. Binary and byte-based patching focus on mitigating known software vulnerabilities without source code by using evolutionary algorithms and computation [39], [40]. ClearView [41] observes the execution of the code's binaries and learns invariants of the program to patch errors in deployed code.

VIII. CONCLUSION

We present HALLMONITOR, a system that infers source code violations of network policies. Current debloating techniques focus on what the user does not use. We use network policies to identify violations in the source code (i.e., what the user is not allowed to do), and then perform analysis on the targeted associated functionality. We present two metrics that show the viability and correctness of HALLMONITOR. Then, relying on mature codebases, we show HALLMONITOR can identify code associated with 14 out of 16 CVEs for Nginx and Curl. By leveraging this novel perspective, we can efficiently find source code violations.

IX. Acknowledgements

We thank the reviewers for their helpful feedback. This work was supported by Office of Naval Research N68335-19-C-0633 and National Science Foundation CNS-1562485. The findings and conclusions in this paper are only of the authors and do not necessarily represent the opinions of ONR and NSF.

References

[1] C. Mulliner and M. Neugschwandtner, "Breaking payloads with runtime code stripping and image freezing," *Black Hat USA*, 2015.

[2] Y. Jiang, D. Wu, and P. Liu, "Jred: Program customization and bloatware mitigation based on static analysis," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, 2016.

[3] Y. Jiang, Q. Bao, S. Wang, X. Liu, and D. Wu, "Reddroid: Android application redundancy customization based on static analysis," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018.

[4] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, "Razor: A framework for post-deployment software debloating," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[5] M. D. Brown and S. Pande, "Carve: Practical security-focused software debloating using simple feature set mappings," in *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, 2019.

[6] B. A. Azad, P. Laperdrix, and N. Nikiforakis, "Less is more: Quantifying the security benefits of debloating web applications," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[7] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[8] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith, "Implementing a distributed firewall," in *Proceedings of the 7th ACM conference on Computer and communications security*, 2000.

[9] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[10] M. R. McNiece, R. Li, and B. Reaves, "Characterizing the security of endogenous and exogenous desktop application network flows," in *International Conference on Passive and Active Network Measurement*. Springer, 2021.

[11] M. Ghaffarinia and K. W. Hamlen, "Binary control-flow trimming," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[12] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, "Trimmer: application specialization for code debloating," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[13] "IANA: Service Name and Transport Protocol Port Number Registry." [Online]. Available: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

[14] J. Wu, R. Wu, D. Antonioli, M. Payer, N. O. Tippenhauer, D. Xu, D. J. Tian, and A. Bianchi, "LIGHTBLUE : Automatic profile-aware debloating of bluetooth stacks," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021.

[15] "IANA: Internet control message protocol (ICMP) parameters," 1981. [Online]. Available: https://www.iana.org/assignments/icmp-parameters/icmp-parameters.xhtml#icmp-parameters-codes-0

[16] "Nginx," https://www.nginx.com/.

[17] "Curl," https://curl.se/docs/security.html.

[18] "Snort community rules." [Online]. Available: https://www.snort.org/downloads/#rule-downloads

[19] G. Cosma and M. Joy, "An approach to source-code plagiarism detection and investigation using latent semantic analysis," *IEEE transactions on computers*, 2011.

[20] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Information and software technology*, 2007.

[21] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013.

[22] Y. Chen, S. Sun, T. Lan, and G. Venkataramani, "Toss: Tailoring on server systems through binary feature customization," in *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*, 2018.

[23] A. D. Keromytis, ""patch on demand′′ saves even more time?[network security]," *Computer*, vol. 37, no. 8, 2004.

[24] M. Vojnović and A. Ganesh, "On the effectiveness of automatic patching," in *Proceedings of the 2005 ACM workshop on Rapid malcode*, 2005.

[25] S. Sidiroglou and A. D. Keromytis, "Countering network worms through automatic patch generation," *IEEE Security & Privacy*, 2005.

[26] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *Proceedings of the 40th international conference on software engineering*, 2018.

[27] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, 2011.

[28] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013.

[29] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[30] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018.

[31] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[32] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[33] S. Sidiroglou-Douskos, E. Lahtinen, A. Eden, F. Long, and M. Rinard, "Codecarboncopy," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[34] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

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

[36] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011.

[37] G. Jin, W. Zhang, and D. Deng, "Automated concurrency-bug fixing," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[38] P. Liu and C. Zhang, "Axis: Automatically fixing atomicity violations through solving control constraints," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012.

[39] E. Schulte, S. Forrest, and W. Weimer, "Automated program repair through the evolution of assembly code," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, 2010.

[40] Z. Deng, X. Zhang, and D. Xu, "Bistro: Binary component extraction and embedding for software security applications," in *European Symposium on Research in Computer Security*. Springer, 2013.

[41] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed software," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009.