SLASH: A Technique for Static Configuration-Logic Identification

Abstract—**Researchers have recently devised tools to debloat software and detect configuration errors. Several of these tools are based on the observation that programs are composed of an *initialization phase* followed by a *main-computation phase*. Users of these tools are required to *manually* annotate the BOUNDARY that separates these phases, a task that can be time-consuming and error-prone. (Typically, the user has to read and understand the source code or trace executions with a debugger.) Because errors can impair the tools' accuracy and functionality, the manual-annotation requirement hinders the ability to apply the tools on a large scale.**

In this paper, we present a field study of 24 widely-used C/C++ programs, identifying common BOUNDARY properties in 96% of them. We then introduce SLASH, an automated tool that locates the BOUNDARY based on the identified properties. SLASH successfully identifies the BOUNDARY in 87.5% of the studied programs within 8.5 minutes, using up to 4.4 GB memory. In an independent test, carried out after SLASH was developed, SLASH identified the BOUNDARY in 91% of a dataset of 21 popular C/C++ GitHub repositories. Finally, we demonstrate SLASH's potential to streamline the BOUNDARY-identification process of software-debloating and error-detection tools.

Index Terms—**Configuration, taint analysis, boundary.**

I. Introduction

Software configurability has emerged as a significant focus in contemporary research [1], [2], [3], [4], [5], [6], [7], [8], [9]. Concurrently, several initiatives proposed to elevate configurability as a first-class programming element [10] and aimed to forge consensus and promote best practices [11], [12]. One best practice is to organize programs to operate in two phases: (i) a phase for *initialization*, where configuration logic checks parameters and initializes corresponding values to control the program's activities, and (ii) a *main-computation phase* that performs actions in accordance with the specified configuration). One would hope that this structure is reflected in the code—i.e., there is a BOUNDARY between the configuration logic and the main computation. A number of recent papers [13], [14], [15], [16] also describe the advantages that this separation provides for the sake of configuration traceability [11], forensic analysis [16], optimizing programs [13], [14], and detecting configuration errors [15].

PCHECK [15] automatically generates a checker that detects configuration errors early to minimize damage from failures. It adds a call to the checker at the end of the initialization phase, as illustrated in Figure 1, which depicts the BOUNDARY location in the program `Squid`, a widely used open-source Web proxy server that supports 237 configurations. However, a user of PCHECK needs to *manually* annotate the `Squid` source code with the BOUNDARY location.

The situation is similar for program-debloating tools. Temporal-specialization [14] disables system APIs after the completion of the initialization phase, but also requires the tool user to annotate the BOUNDARY. LMCAS [13] specializes programs by executing them up to the BOUNDARY to capture the program's state according to the supplied inputs, where again the LMCAS user must annotate the location of the BOUNDARY.

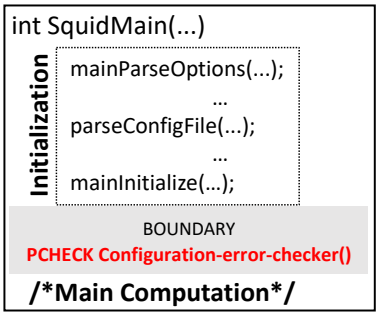

Figure 1: Location to invoke the configuration-error checker generated by PCHECK [15] in Squid—i.e., the BOUNDARY at the end of the initialization phase. PCHECK users need to annotate this location *manually*.

In the absence of a method to assist developers in identifying the BOUNDARY, the *manual* annotation is a time-consuming and error-prone task: the user has to read and understand the source code or trace executions with a debugger. Because BOUNDARY-identification errors can impair the tools' accuracy and functionality, the manual-annotation requirement hinders the ability to apply the tools on a large scale.

We thus first conduct a manual field study to comprehend how the BOUNDARY is implemented and to discern its distinguishing characteristics. Our corpus contains 24 widely used C/C++ programs (Table I) that employ configuration files or command-line parameters. Our study identifies various categories of BOUNDARIES: single-element BOUNDARIES, multi-element BOUNDARIES, or "blended" (i.e., no BOUNDARY). The study further indicates that 23 (96%) of the programs possess a single-element BOUNDARY. Accordingly, we developed SLASH, a tool to identify a BOUNDARY automatically. **SLASH focuses on the common case of identifying a BOUNDARY in applications that contain one or more single-element BOUNDARIES.** SLASH analyzes LLVM IR, and targets programs written in an imperative programming language, such as C or C++.

Our work makes the following contributions:

1) We conducted a manual field study to determine (i) to what extent real-world programs contain a BOUNDARY that separates configuration logic from the main-

computation logic, and (ii) for programs that do contain a BOUNDARY, what structural patterns can be used to identify the end of the initialization/configuration phase (§III).
2) We present an algorithm that either identifies a BOUNDARY that separates the initialization and main-computation phases, or reports that it was unable to do so (§V).
3) We implemented the BOUNDARY-identification algorithm in a tool, called SLASH, and evaluated it on (a) the 24 programs used in the manual field study, and (b) 21 popular C/C++ Github repositories not part of the manual field study (§VI). The implementation and artifacts of SLASH [1] is open-source.
4) We demonstrate that the BOUNDARIES that SLASH identifies are suitable substitutes for the ones identified manually (by the respective developers) for a software-debloating tool [13], two software security tools [14], [17], and a configuration-error-checking tool [15].

II. Background

This section provides background on some concepts and patterns that we relied on in our manual field study.

A. Program Phases

An example of a BOUNDARY is shown in Figure 2, which represents a scaled-down version of the UNIX word-count utility `wc`: `wc` reads a line from `stdin`, counts the number of lines and/or characters in the input stream, and prints the results on `stdout`. This program has two phases, the code for which is found in disjoint regions:

- The *initialization phase* starts at the entry point of `main` (line 1), and ends at line 16.
- The *main-computation phase* starts at line 18 and continues to the end of `main` (line 28).

When the configuration logic in the initialization phase is executed, a parameter expressed in some *external format*—here `argv[1]` as a C string—is translated to an *internal format* and assigned to one or more program variables that host run-time configuration data. These variables are known as *configuration-hosting variables* [18]. In Figure 2, after the configuration logic executes, the configuration-hosting variables `count_char` and `count_line` each hold internal-format values of `0` or 1. The main-computation logic then performs the primary processing function of the program, with its actions controlled by the values of `count_char` and `count_line`.

The two regions are tied together through the values of the configuration-hosting variables: when the main-computation phase executes, the values of `count_char` and `count_line` control which portions of the main logic execute. In `wc`, for instance, `count_char` controls whether lines 20 and 24 execute, and `count_line` controls whether lines 21 and 26 execute. In `wc`, there is a BOUNDARY at line 17 that physically separates the configuration logic from the main-computation phase.

[1] https://github.com/secure-software-engineering/neck

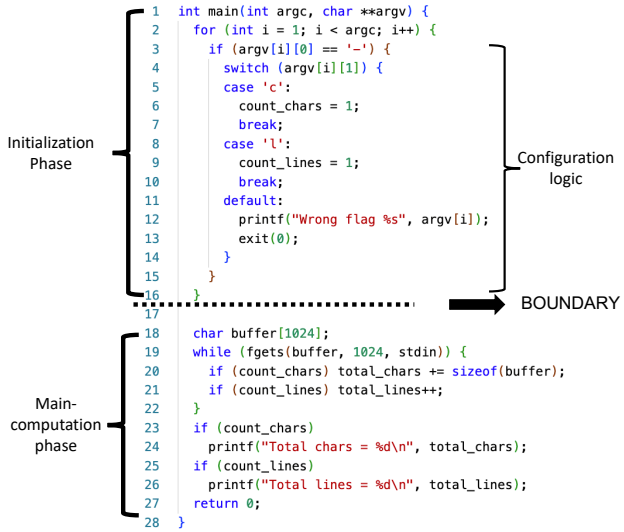

Figure 2: A scaled-down version of the `wc` utility. The BOUNDARY could be just before line 17, just before line 18, or just before line 19. LMCAS executes `wc` up to the BOUNDARY during the course of its analysis but its users are required to annotate the BOUNDARY location *manually*.

B. Configuration-Logic Phases

To inform the process of identifying a BOUNDARY, it is crucial to understand the configuration logic and patterns inside the initialization phase. We adopt the taxonomy of configuration design of Zhang et al. [4], which involves the following configuration phases:

1) **Parse and Assign:** run-time-configuration information is first parsed and translated. Translated values of configuration parameters are typically Booleans, integers, or strings. In a **command-line program**, the inputs are provided via command-line arguments. In C/C++ programs, command-line arguments are passed to `main()` via the parameters `argc` and `argv`: `argc` holds the number of command-line arguments, and `argv[]` is an array (of that length) of pointers; the elements of `argv[]` point to the different arguments passed to the program. The application then assigns values of `argv[]` elements to configuration-hosting variables according to a predefined argument-value mapping.

   Similar logic is used in **configuration-file programs**. They also permit command-line arguments, yet they receive further arguments using configuration files, whose location is typically provided via one command-line argument. The configuration file is frequently parsed using a system-call API. For example, *Nginx* uses the Linux system call `pread`, and *DNSProxy* uses the C library function `fgets`. The application then usually assigns configuration-hosting variables values according to a predefined keyword-value mapping.
2) **Check** and **Exception/Error Handling:** in general, these steps are intertwined with the parse-and-assign step. They validate the provided inputs based on certain constraints, identify incorrect configuration pa-

Table I: Results of the manual field study. Column 2 gives LOC (in thousands), based on readable LLVM IR. Column 3 indicates whether the program can receive additional configuration settings through a configuration file. Column 5 specifies whether the location selected based on our manual inspection lies within or outside `main`.

Program	kLOC	Config File	BOUNDARY Category	Inside main
End-User Programs				
curl-7.47.0	31.6	✓	Single	✗
date-8.32	56.9	✗	Single	✓
diff-2.8	37.2	✗	Single	✓
du-8.32	109.1	✗	Single	✓
echo-8.32	11.0	✗	Single	✓
gzip-1.2.4	26.7	✗	Single	✓
id-8.32	15.1	✗	Single	✓
kill-8.32	12.1	✗	Single	✓
objdump-2.33	1049.0	✗	Single	✓
psql-15	189.1	✗	Single	✓
readelf-2.33	413.7	✗	Single	✓
sort-8.32	55.9	✗	Single	✓
tcpdump-4.10.0	608.6	✗	Single	✓
uniq-8.32	14.8	✗	Single	✓
wc-8.32	17.8	✗	Single	✓
wget-1.17.1	165.5	✓	Single	✓
Server Programs				
bind-9.15.8	1755.1	✓	Single	✓
DNSProxy-1.17	3.4	✓	Single	✓
httpd-2.4.51	179.0	✓	Single	✓
knockd-0.5	10.9	✓	Single	✓
lighttpd-1.4.54	174.0	✓	Single	✗
mini-httpd1-1.19	16.4	✓	Single	✓
Nginx-1.19.0	589.2	✓	Single	✓
PostgreSQL-15	4626.3	✓	Multi	✗

rameters and—if present—provide user feedback and terminate the program.

Parse-assign-check steps are executed inside a loop until all configuration parameters are processed or an error arises. Once parameters pass checks, the main program can utilize these (translated) values to select functionalities. This processing completion denotes the transition from the initialization to the main-computation phase.

III. Understanding BOUNDARY Characteristics

This section describes our manual field study, conducted to determine (i) to what extent real-world programs contain a BOUNDARY that separates configuration logic from the main-computation logic, and (ii) for programs that do contain a BOUNDARY, what structural patterns exist that could be used to automate the process of identifying the BOUNDARY.

A. Methodology

Selection of subject programs. We manually inspected 24 widely-used [13], [14], [19], [20], [9] end-user and server C/C++ programs, listed in Table I. The configurations of these programs are provided either through command-line arguments or configuration files. The end-user programs include utilities (i.e., `sort`, `objdump`, `diff`, and `gzip`). The server programs include web servers (i.e., `Nginx`), DNS servers (i.e., `DNSProxy`), and database programs (i.e., `PostgreSQL`).

Manual-inspection procedure. We manually inspected the source code of the programs to see if we could identify a BOUNDARY location. The inspection was conducted by one person from our team, but a second opinion was obtained for challenging cases, such as Nginx, httpd, and PostgreSQL. The manual field study was performed as follows:

(a) We built the program and ran it with `-help` to display all runtime configuration parameters. For config-file programs, we also inspected the program's default configuration-file templates to identify the set of predefined keywords (e.g., `Nginx` uses the directive *gzip* to enable/diable compression).

(b) Next, we identified the entry-point function in the source code. Because the study considered C/C++ programs, we searched for a function named "`main`" that has two parameters named "`argc`" and "`argv`."

(c) We identified the locations in the source code where the configuration parameters are parsed, assigned, and checked (thereby identifying the configuration-hosting variables). A regular-expression search (based on the knowledge gained from step (a) was sufficient for identifying such locations. We observed that some programs parse the configuration parameters outside of `main`; for instance, the `main` function of a command-line program might invoke another function and pass `argv` as a parameter. For configuration-file programs, we performed a regular-expression search to find (i) method names in APIs for reading/parsing files, and (ii) keywords used in the configuration file.

To supplement source-code inspection, we ran the programs with a debugger (GDB) to track the use of `argv` and identify the location where the provided configuration parameters are parsed. Similarly, we ran the configuration-file programs with GDB by specifying the configuration-file-parsing API as breakpoints.

(d) We sought to identify a location for the BOUNDARY. We looked to see if the location just after the end of the loop containing the parse-assign-check logic was acceptable. In some cases, the BOUNDARY location was a bit further along in the program because the values of configuration-hosting variables are sometimes set or adjusted after the parse-assign-check loop when one configuration feature overrides another.

B. Results

Categories of BOUNDARIES. Our study identifies several types of BOUNDARIES within programs:

- **Single-element BOUNDARIES**: one or more sites exist that, individually, are each an acceptable BOUNDARY location (as in `wc` from Figure 2).
- **Multi-element BOUNDARIES**: No single-element BOUNDARY exists, but a collection of sites separate the program's configuration logic from the main-computation phase.
- **"Blended" BOUNDARIES**: the application's configuration logic is "blended" into the main-computation phase, yielding no clear BOUNDARY of the foregoing two types.

The manual-field study showed that 23 programs possess single-element BOUNDARIES. As illustrated in Figure 2, it is possible that there are multiple locations

Table II: Number of programs in each BOUNDARY category.

Dataset	Single-elem.	Multi-elem.	Blended
Manual field study	23	1	0
Previously unseen	18	1	3

where a single-element BOUNDARY could be located. (The term "multiple single-element BOUNDARIES" should not be confused with "multi-element BOUNDARY"). E.g., in Figure 2 the BOUNDARY could be just before line 17, just before line 18, or just before line 19. Column 4 of Table I indicates which of the programs had multiple single-element BOUNDARIES. The remaining program, `PostgreSQL` (a database server), has a multi-element BOUNDARY. It is a Swiss-Army-knife program, consisting of several stand-alone programs, each with its own BOUNDARY or BOUNDARIES.

The corpus of programs used in the manual field study (Table I) had no instances of "blended" BOUNDARIES. However, our evaluation dataset of previously unseen programs (Table IV) contains three programs with "blended" BOUNDARIES. Table II shows the numbers of programs in each BOUNDARY category.

Behaviour of configuration-hosting variables. In 20 of the 24 programs, the values of the configuration-hosting variables are not changed after they are given values within the parse-assign-check loop. In three cases (`tcpdump`, `Gzip`, and `readelf`), we observed that feature interactions—in one case, involving a compile-time configuration feature (`#ifdef`)—can cause the program to change the value of a configuration-hosting variable after the parse-assign-check loop.

FIELD STUDY RESULTS: The study revealed that:
- 23 of the 24 programs have a single-element BOUNDARY that divides the program into its configuration logic and its main-computation logic.
- The assignments to configuration-hosting variables typically all occur inside a parse-assign-check loop that processes the command line or configuration file.
- Because of interactions among configuration features, the values of configuration-hosting variables are sometimes adjusted after the parse-assign-check loop.

C. Findings: Properties of the BOUNDARY

Our study revealed a set of properties that were common across the 23 programs in which we were able to identify a single-element BOUNDARY. We use these properties in §IV and §V to address the problem of *automatically* identifying a suitable single-element BOUNDARY location.

The main property that we can infer from Figure 2 is that the BOUNDARY divides a program's control-flow graph into two disjoint subgraphs. This property implies that the BOUNDARY is a so-called *articulation point* in the program (i.e., a vertex cut of the control-flow graph, of size 1). In addition, we observed that the BOUNDARY should be:

- **Reachable** from the entry point of the program.
- **Executed** exactly once, which eliminates the possibility that the BOUNDARY is inside a loop or a conditional statement.

To validate our hypothesis, we instrumented the programs by adding a print statement at the location where the BOUNDARY is identified (e.g., line 17 in Figure 2) and then executed the program several times with different sets of configuration parameters. If the print statement is executed, the run indicates the BOUNDARY is reachable from the entry; if the print statement is printed only once, the run indicates that the BOUNDARY is executed exactly once (for the given configuration).

In the debugging performed in step (c) of the manual inspection steps in § III-A, we dynamically traced the uses of `argv` and calls to configuration-file parsing APIs as a way to pinpoint the locations where configurations are parsed, assigned, and checked. This use of dynamic tracing of `argv` during manual analysis suggests that a technique to identify BOUNDARIES automatically will need to perform dataflow analysis to track uses of `argv` statically.

FINDINGS: Properties relevant to identifying automatically a program's BOUNDARY include:
1) Configuration-hosting variables are data dependent or control dependent on `argv`.
2) The BOUNDARY should be located after at least one loop.
3) The BOUNDARY represents an articulation point in the program's control-flow graph.
4) The BOUNDARY should be reachable from the entry point and executed only once.

IV. THE BOUNDARY-IDENTIFICATION PROBLEM

This section provides an abstract overview of the BOUNDARY concept, presenting the process of BOUNDARY identification as a form of staging transformation (§IV-A). It then defines the BOUNDARY-identification task (§IV-B and §IV-C).

A. An Idealized View

At an abstract level, automated BOUNDARY identification can be thought of as a kind of *staging transformation* [21] that isolates (or "stages") the processing of a program's configuration parameters. Staging transformations were originally proposed to separate a program's computation into stages for optimization purposes. In our context, given a program $P(x)$ with body $f(x)$, where x represents some configuration parameter, we wish to consider P as having the form shown below in the second line:

$$\begin{aligned} P(x) &= f(x) \\ \rightarrow \quad P(x) &= \textbf{let } t = \mathit{translate}(x) \textbf{ in } g(t) \end{aligned} \tag{1}$$

Here, $\mathit{translate}(\cdot)$ converts from the external configuration-specification format to the internal format, and t is a configuration-hosting variable. Thus,
- "**let** $t = \mathit{translate}(x)$ **in** ..." represents the configuration logic of P.

- "$g(t)$" represents the main logic, which performs the primary processing function of P, based on the value of configuration-hosting variable t.

The BOUNDARY-identification challenge is to find the code that constitutes *translate* within function definition f.

This abstract characterization of BOUNDARY identification permits giving a "rational reconstruction" of some previous work. For instance, both PCHECK [15] and Zhang et al. [4] propose methodologies to ensure that the value of a configuration parameter is checked against an appropriate constraint $\varphi(\cdot)$ on the parameter before it is used. Thus, if one has program $P(x)$ in the form shown on the second line of Eqn. (1), the essence of both PCHECK and the Zhang et al. paper is to transform P as follows:

$$\begin{array}{rcl} & P(x) & = \textbf{ let } t = \mathit{translate}(x) \textbf{ in } g(t) \\ \rightarrow & P(x) & = \textbf{ let } t = \mathit{translate}(x) \\ & & \quad \textbf{in if } \varphi(t) \textbf{ then } g(t) \textbf{ else abort} \end{array} \tag{2}$$

Furthermore, several issues discussed in [11], [12], [1], [3] can be characterized as "it is advantageous to separate the configuration logic from the program's main-computation logic for the sake of facilitating configuration tracking and analysis," which at an abstract level amounts to:

$$\begin{array}{l} \textit{Given program } P \textit{ in the form} \\ \quad P(x) = \textbf{let } t = \mathit{translate}(x) \textbf{ in } g(t) \\ \textit{Analyze the usage of } t \textit{ in } g(t) \end{array} \tag{3}$$

For instance, LOTRACK's usage analysis aims to identify code fragments corresponding to load-time configurations [1], [3] in Android and Java programs.

Finally, LMCAS [13] relies on manual techniques to identify a program's BOUNDARY, and then performs partial evaluation [22] with respect to the values of configuration-hosting variables. Abstractly, LMCAS operates similarly to what was discussed above, except that P is now a two-argument program, $P(x, y)$.

$$\begin{array}{rcl} & P(x,y) & = \textbf{ let } t = \mathit{translate}(x) \textbf{ in } g(t,y) \\ \rightarrow & P_x(y) & = g_t(y) \end{array} \tag{4}$$

Program $P_x(y)$ is a version of $P(x, y)$ specialized with respect to a specific value of x. The body of $P_x(y)$ is obtained by finding and evaluating $t = \mathit{translate}(x)$, and then running a partial evaluator on g with static input t to create $g_t(y)$, which is a version of $g(t, y)$ specialized on the value of t.

B. Terminology and Notation

Because the BOUNDARY constitutes an articulation point in the program, we formulate the BOUNDARY-identification task as a vertex-cut graph-partitioning problem.

Definition 1: Let $G = (V, E, v_{en}, v_{ex})$ denotes the Interprocedural Control-Flow Graph (ICFG) of a program P. Vertices v_{en} and v_{ex} represent the entry vertex corresponding to the `main` and end of the program, respectively.

Without loss of generality, we assume that each vertex in G is reachable from v_{en} along a path in which each procedure-return edge is matched with its closest preceding unmatched procedure-call edge (a so-called "interprocedurally valid path" [23, §7-3]). A vertex v in the control-flow graph G_Q of some procedure Q is said to be an *articulation point* if removing v, and all control-flow edges into and out of v, partitions G_Q into two non-empty subgraphs.

C. Problem Definition

In the abstract view of the BOUNDARY-identification problem discussed in §IV-A, a command-line or configuration-file program $P(x) = f(x)$ has the form shown on the second line of Eqn. (1). Eqn. (1) is stated in an abstract form, as if we were considering a program in a functional programming language. However, we need to translate this idea to something that is suitable for an imperative programming language, such as C/C++. In such a case, configuration parameter x will be `argv`.

Our goal is to identify *translate*(x), whose end is considered to be the BOUNDARY (but in an imperative program), which leaves us with two questions:

1) What is a suitable "choke point" in the program, analogous to the hand-off from "$t = \mathit{translate}(x)$" to $g(t)$ in Eqn. (1)?
2) What does "the program has finished *translate*(x)" mean?

With respect to question (1), a natural approach is in terms of the articulation points of the program's control-flow graph G: the candidate choke points are the articulation points of G (denoted by V_{AP}). In general, G can have many articulation points. We need some other conditions to specify which member of V_{AP} we want: the BOUNDARY separates the vertices of G into the configuration logic (denoted by V_c)—which is analogous to *translate*(x)—and vertices belong to the main-computation logic (denoted by V_m)—which is analogous to $g(t)$.

With respect to question (2), discovering the end of *translate*(x) entails identifying the *configuration-hosting variables* [18] (which are analogous to variable t in Eqn. (1)). These variables are either (a) assigned configuration values directly, or (b) control dependent on branch expressions involving configuration quantities. As supported by the findings from our manual study (§III), the assignments to configuration-hosting variables (i) typically all occur inside a parse-assign-check loop that processes the command line or configuration file, but (ii) some additional assignments to them may occur after the parse-assign-check loop because of interactions among configuration features. Configuration-hosting variables are typically live variables in the main logic; moreover, they are used without their values being modified in the main-computation phase [1], [24]. We formalize this concept as follows:

CONFIGURATION-HOSTING VARIABLES (C_{host}).
1) Let v_x denote the CFG vertex that models the passing of configuration parameter x to `main`.
2) Let $V_{host} = \{v_0, v_1, \ldots, v_n\}$ denote the set of vertices that represent assignments to configuration-hosting variables: $v_i \in V_{host}$ if
   a) v_i is flow dependent on v_x, denoted by $v_x \longrightarrow_f v_i$, or
   b) v_i is control dependent on a vertex w_x that uses x.
3) The set of configuration-hosting variables C_{host} is the set of variables that are assigned to at some member of V_{host}.

For instance, in the scaled-down word-count program in Figure 2, the global variables `count_chars` and `count_lines` are assigned values at lines 6 and 9, respectively; these assignments are control-dependent on vertices that use `argv` (i.e., the branch-conditions on lines 3 and 4, which play the role of w_x in item (2b)). Thus, by item (2b), V_{host} consists of the assignments at lines 6 and 9, and by item (3), C_{host} is {`count_chars`, `count_lines`}.

With the concept of C_{host} in hand, we can now state the BOUNDARY-identification problem.

PROBLEM DEFINITION: Find an articulation point B of CFG G that is reachable from v_{en}, and
1) is located after a loop,
2) post-dominates every assignment to a member of C_{host}, and
3) for each $c \in C_{host}$, all paths from B to v_{ex} are free of definitions to c.

Return the closest B to the entry as the BOUNDARY.

The control-flow vertex for line 16 of Figure 2 is an articulation point that meets the conditions of the problem definition:
- it is located after the end of the for-loop on lines 2-16
- it post-dominates every assignment to `count_chars` and `count_lines`, and
- all paths from that point to the end of the program are free of definitions to `count_chars` and `count_lines`.

Finally, the articulation point at line 16 is the closest to the entry point in terms of distance along control-flow edges.

V. AUTOMATIC BOUNDARY IDENTIFICATION

This section presents our algorithm to solve the BOUNDARY-identification problem defined in §IV-C. The algorithm is given as Alg. 1. The discussion of Alg. 1 is structured in two parts, which correspond to lines 5–34 and 35–55, respectively.

- **Identification of BOUNDARY Candidates (§V-A).** This phase identifies a set of BOUNDARY candidates, which are a subset of the set of articulation points of the control-flow graph.
- **BOUNDARY Identification (§V-B).** This phase eliminates all BOUNDARY candidates that do not satisfy the three properties of a BOUNDARY given in the problem definition in §IV-C.

Algorithm 1: Single-element BOUNDARY-identification algorithm.

```
   Input: Program P, SrcProcedure
   Output: BOUNDARY
1  Entry point Entry = getEntryPointBasicBlock (P)
2  G = computeICFG (P)
3  ConfigHostVars V_host = {}
4  BoundaryCandidates BC = {}
5  /* Identification of BOUNDARY Candidates (Section V-A)      */
6  T = computeTaintAnalysis(G, SrcProcedure);
7  foreach node ∈ G do
8      foreach Res ∈ T.getTaintResultsAt(node) do
9          foreach Op ∈ node.operands() do
10             if Op = Res ∧ isAssignmentInst(node) then
11                 V_host ∪= node.getBasicBlock()
12             if Op = Res ∧ isControlDependent(node, Op) then
13                 V_host ∪= getAssignmentsIn(node.getBasicBlock())
14 C_host = identifyVariablesAssignedToIn(V_host)
15 /* Add a loop's successors, if C ∈ V_host is within a loop    */
16 SCCs = computeStronglyConnectedComponents(G)
17 foreach C ∈ V_host do
18     if isInLoopStructure(C, SCCs) then
19         BC ∪= getLoopExitBlocks(C)
20     else BC ∪= C
21 AP = computeArticulationPoints(G)
22 BC = BC ∩ AP
23 foreach C ∈ BC do
24     if isReachableFromEntry(C) then
25         /* Find proxies in main for BOUNDARY candidates in
              other procedures                                  */
26         if C ∉ main then
27             foreach CallSite ∈ main do
28                 /* direct and transitive reachability      */
29                 if isReachable(CallSite, C) then
30                     BC = BC \ {C}
31                     BC ∪= CallSite.getBasicBlock()
32     else BC = BC \ {C}
33
34 BC = BC ∩ AP          /* Proxies must be articulation points */
35 /* BOUNDARY Identification (Section V-B)                     */
36 DG = computePostDominators(G)
37 foreach C ∈ BC do
38     /* C1: Check that BOUNDARY is located after a loop      */
39     if !followsLoop(C) then
40         BC = BC \ {C}
41     /* C2: Check whether C post-dominates every assignment d
           to a variable in C_host                             */
42     PostDominatesFlag = true
43     foreach assignment d to a variable in C_host do
44         if !postDominates(DG, C, d.getBasicBlock()) then
45             PostDominatesFlag = false
46             BC = BC \ {C}
47     /* C3: Check for definition-free paths from C to v_ex using
           constant propagation                                */
48     if PostDominatesFlag then
49         foreach var ∈ C_host do
50             if !isConstant(G, var, C, v_ex) then
51                 BC = BC \ {C}
52 BOUNDARY = closestReachableFromEntry(BC)
53 if BOUNDARY = ∅ then
54     return null
55 else return BOUNDARY
56
```

A. BOUNDARY *Candidates (Lines 5–34)*

This phase performs taint analysis and control-flow analysis to identify a set of BOUNDARY candidates. It performs taint analysis to identify the set of configuration-hosting variables C_{host} and the set of assignments V_{host} defined in §IV-C (lines 6–14). Some adjustments are

made when the latter assignments are either inside a loop (lines 16–20), or in a procedure other than `main` (lines 23–33). In particular, for each candidate not in `main`, a proxy location is considered just after the appropriate call in `main` that would reach the candidate.

The algorithm performs control-flow analysis to identify the set of articulation points in G (line 21). The outcome of this phase is the intersection between the set of articulation points and the (adjusted) set of vertices that represent assignments to configuration-hosting variables (first performed at line 22 to reduce the cost of the proxy-finding loop, then at line 34).

B. BOUNDARY *Identification (Lines 35–55)*

This phase eliminates all BOUNDARY candidates that do not satisfy the three conditions of a BOUNDARY given in the problem definition in §IV-C, namely, each $C \in BC$ must (C1) be located after a loop, (C2) post-dominate every assignment to a member of C_{host}, and (C3) for each $var \in C_{host}$, all paths from B to v_{ex} are free of definitions to var. The algorithm removes from BC any vertex C that fails to satisfy all three conditions; see lines 39–40, 42–46, and 48–51, respectively.

To verify condition (C1), the algorithm checks whether candidate $C \in BC$ is located after a loop (lines 39-40). To verify condition (C3), the algorithm can use standard techniques—e.g., IFDS-based [25], inter-procedural constant propagation for each configuration-hosting variable var, starting at candidate $C \in BC$. If the analysis reports that var is not constant at exit point v_{ex}, then var might be (re)defined on some path from C to v_{ex}, and hence condition (C3) is violated.

This phase returns `null` if all BOUNDARY candidates are eliminated (lines 53–54); otherwise, it returns the candidate that is closest to v_{en}, the entry point of procedure `main` (lines 52 and 55). The distance metric used to calculate the closest BOUNDARY candidate is the shortest path in terms of control-flow edges.

C. Discussion

Limitations. Alg. 1 gives an idealized algorithm that works on an ICFG, yet SLASH operates on the ICFG only partially. Specifically, for (a) finding post-dominators, and (b) finding articulation points our implementation works on individual CFGs in a procedure-by-procedure manner. Listing 1 sketches this variant:

```
AP, DTs := ∅;
foreach procedure P {
        DTs := DTs ∪ computePostDominators(CFG(P))
        AP := AP ∪ computeArticulationPoints(CFG(P))
            }
```

Listing 1: Variant of *computePostDominators* and *computeArticulationPoints* used in our implementation of Algorithm 1

Consequently, when a BOUNDARY candidate is located in some procedure `p` other than `main`, the candidate is relocated to the CFG of `main` by finding a proxy location for the candidate. We use the basic block of `main` that contains the call site that calls `p` (directly or transitively).

A second limitation of SLASH is that it targets only single-element BOUNDARIES. When run on a multi-element-BOUNDARY program, it could either return the empty set (a correct answer with respect to the question of whether a single-element BOUNDARY exists) or some singleton set (which is a false positive). (In the latter case, our experience is that SLASH returns one of the elements of the multi-element BOUNDARY, and the rest of the elements are other BOUNDARY candidates.) When run on a program in the "blended"-BOUNDARY case, SLASH returns the empty set (correct with respect to the question of whether a single-element BOUNDARY exists). In this case, it never returns a singleton set because the intertwining of the configuration logic and the main-computation logic causes the properties required of a single-element BOUNDARY to be violated.

Time Complexity. The overall worst-case running time of the algorithm is bounded by $O(|E|\cdot|D|^3+|N|^2+|E|)$, where N and E are the sets of nodes and edges, respectively, of the program's ICFG, and D is the domain(s) used in the data-flow problems (taint analysis and constant propagation) [25].

VI. EVALUATION

Our experiments were designed to answer three questions:

- **RQ1:** Can SLASH identify the BOUNDARY location correctly for command-line and configuration-file programs? §VI-A
- **RQ2:** How expensive is SLASH in terms of running time and memory usage? §VI-B
- **RQ3:** Can SLASH alleviate the manual efforts required to use existing debloating tools §VI-C

The evaluations were carried out on an Ubuntu 16.04 PC with an Intel i7-5600U CPU @ 2.6GHz and 16 GB RAM.

A. *RQ1: Accuracy of* SLASH

The accuracy of SLASH is defined as its ability to determine whether the subject program contains an acceptable single-element BOUNDARY. A correct answer means that SLASH identified one of the suitable BOUNDARY locations, or correctly indicated that the program lacks any suitable location.

We conducted our evaluation using two sets of programs: (i) the programs considered in the manual field study (§III, Table I), and (ii) the 22 programs listed in Table IV, which were neither involved in the manual field study nor examined to determine BOUNDARY properties. The latter set was introduced to provide an unbiased test of accuracy results.

1) Accuracy based on Manual-Field-Study Dataset: For each program in the evaluation dataset, we measured the accuracy of SLASH in identifying the correct BOUNDARY location using the following methodology:

Table III: Results of SLASH's evaluation. Columns 3 & 4 represent the number of pointers and allocation sites, respectively. Column 5 indicates the outcome of SLASH's analysis: True Positive (TP), False Positive (FP), and False Negative (FN) are denoted by the symbols ☑, ☒, □, respectively. Column 6 specifies SLASH's average running time in seconds, and column 7 indicates the maximum amount of memory usage in MB (both over 10 runs).

Program	kLOC	#ptr	#alloc	Accuracy	Analysis Time	Memory
End-user Programs						
curl-7.47.0	31.6	10228	785	□	119.6	1526
date-8.32	56.9	7613	979	☑	26.9	698
diff-2.8	37.2	5331	842	☑	2.9	213
du-8.32	109.1	19168	2756	☑	3.5	299
echo-8.32	11.0	1687	305	☑	0.5	72
gzip-1.2.4	26.7	2952	420	☑	0.6	112
id-8.32	15.1	2368	418	☑	1.8	135
kill-8.32	12.1	1819	348	☑	1.7	130
objdump-2.33	1049.0	209651	21984	☑	37.7	1261
psql-15	189.1	37921	3798	☑	7.4	305
readelf-2.33	413.7	76842	8325	☑	4.5	400
sort-8.32	55.9	9902	1549	☑	4.0	254
tcpdump-4.10.0	608.6	152999	11945	☑	24.1	1431
uniq-8.32	14.8	2340	442	☑	1.9	136
wc-8.32	17.8	3000	509	☑	1.8	138
wget-1.17.1	165.5	30069	4036	☑	2.7	276
Server Programs						
bind-9.15.8	1755.1	326408	41391	☑	228.8	4292
DNSProxy-1.17	3.4	583	79	☑	0.1	53
httpd-2.4.51	179.0	75032	6927	☑[a]	2.2	278
knockd-0.5	10.9	2062	177	☑	0.1	57
lighttpd-1.4.54	174.0	34932	3527	☑	1.7	195
mini-httpd1-1.19	16.4	2935	323	☑	1.9	147
Nginx-1.19.0	589.2	116710	9307	☑	32.7	1232
PostgreSQL-15	4626.3	880507	126401	☒	575.6	8313.7

[a] SLASH did not succeed "out of the box," but was successful when provided with suitable stubs for two library functions.

1) Manually annotate the target program's source code with the (single-element) BOUNDARY location that is closest to the entry point of `main` in terms of distance along control-flow-graph edges, and generate the LLVM IR bitcode. This information serves as ground truth.
2) Pass an un-annotated LLVM IR bitcode of the same target program to SLASH, which annotates one basic block of the bitcode as the BOUNDARY.
3) Check whether the basic block identified by SLASH matches the ground truth. If the check passes, then SLASH is successful in identifying the BOUNDARY.

The result shows that the accuracy rate of SLASH is 91.67%, with 95.65% for recall and precision. SLASH fails to report an accurate BOUNDARY location for `curl`, `httpd`, and `PostgreSQL` because of the following reasons:

- *A Swiss-army-knife program*: as mentioned in §III, `PostgreSQL` requires a multi-element BOUNDARY. While this category is not within the purview of the SLASH, we still ran SLASH on `PostgreSQL`. SLASH was able to correctly identify the entire set of elements of the multi-element BOUNDARY as BOUNDARY candidates, but returned the one nearest to the program entry point as per line 52 of Alg. 1.
- *Definitions of two argument-parsing functions are unavailable*: `httpd` uses `libapr (Apache Portable Runtime)` (specifically `apr_getopt_init` and `apr_getopt`) to parse command-line arguments. Only the declarations of these functions exist in the LLVM bitcode; their definitions are not available, which prevents SLASH from performing taint analysis—and thus from identifying the correct BOUNDARY location. However, when provided with suitable stubs—i.e., taint-analysis summaries that describe the dependencies of outputs on inputs in `apr_getopt_init` and `apr_getopt`—SLASH is able to identify the BOUNDARY location correctly.
- *Configuration logic and main-computation phase in the **same** procedure called from `main`*: SLASH always places the BOUNDARY inside `main`, just after the call site that contains the code identified as the configuration logic. However, in `curl` both the configuration logic and the main-computation logic reside in the **same** callee of `main`. SLASH is unable to identify the BOUNDARY correctly because no location in `main` separates the configuration logic from the main-computation logic.

Table IV: Results of SLASH's evaluation based on the previously unseen programs. Popularity is based on the number of stars. Column 5 reports the BOUNDARY type based on the manual inspection of the source code. Column 6 indicates the outcome of SLASH's analysis: True Positive (TP), False Positive (FP), and False Negative (FN) are denoted by the symbols ☑, ☒, □, respectively.

Repo (Program)	Category	Popularity	kLOC	BOUNDARY Type	Accuracy
AFLplusplus(afl-fuzz)	Software Testing	3.7k	164.9	Single	☑
blurhash(blurhash_encoder)	Image Processing	13.8k	58.6	Blended	☑
Caffe(caffe)	Machine Learning	33.5k	17.2	Blended	☑
fish-shell(fish)	Utility	21.8k	696.3	Single	☑
coreutils(chown)	Utility	3.5k	3.37	Single	☑
coreutils(rm)	Utility	3.5k	3.4	Single	☑
GoAccess(goaccess)	Web	16.4k	189.7	Single	☑
hashcat(hashcat)	Utility	17.4k	978	Single	☑
jq(jq)	Utility	25.3k	300.1	Single	☑
masscan(masscan)	network	21.3k	223	Single	☒ □
memcached(memcached)	Data	12.6k	73.1	Single	☑
n^3(nnn)	Data	16.4k	46.4	Single	☑
Redis(redis)	Data	60k	1437.4	Single	☑
rethinkdb(rethinkdb)	Data	26.2k	839.1	Multi	☒
skynet(skynet)	Games	12k	629.7	Blended	☑
Tesseract(tesseract)	Image Processing	51.9k	2204.6	Single	☑
the_silver_searcher(ag)	Utility	25k	28.2	Single	☑
tmux(tmux)	Utility	29.5k	593.6	Single	☑
trojan(trojan)	Network	17.8k	346.6	Single	☑
twemproxy(nutcracker)	Network	11.8k	152.8	Single	☑
wrk(wrk)	Web	34.5k	1203.3	Single	☑
zstd(zstd)	Data Compression	21.1k	352.3	Single	☑

2) Accuracy based on Previously Unseen Programs: This dataset contains programs that were not employed to deduce the BOUNDARY's traits (§III).

Selection of Subject Programs. This dataset was acquired by cloning starred C/C++ projects from GitHub with 3*k* stars or more, which yielded 100 repositories. We then excluded repositories that (a) include other programming languages (such as Python, JavaScript, etc.), (b) incorporate GUI functionality, (c) did not contain an entry point (i.e., firmware), or (d) did not build successfully. This process yielded the 22 repositories listed in Table IV. Of these repositories, 18 possess a single-element BOUNDARY, one has a multi-element BOUNDARY, and two lack any BOUNDARY (see Table II). These findings corroborate the outcomes of the field study (§III).

For each program in the unseen dataset, we assessed the accuracy of SLASH using the following approach:

1) Generate the LLVM IR bitcode (without any instru-

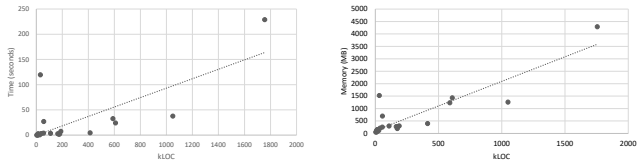

Figure 3: Lines of code versus time (left) and memory (right).

mentation) and analyze it using SLASH.

2) Manually examine the source code to compare with SLASH's outcome. SLASH's result is considered correct if (i) the BOUNDARY identified by SLASH aligns with the manually identified location, or (ii) SLASH does not detect any BOUNDARY and the manual inspection confirms the absence of an acceptable single-element BOUNDARY (i.e., the program has multi-element BOUNDARIES or NO BOUNDARY).

Table IV presents the BOUNDARY-identification results for the unseen programs. The result shows that the accuracy rate of slash is 86.96%, with 95.24% and 90.91% for recall and precision, respectively.

The three cases for which SLASH reported an inaccurate result were as follows:

- *rethinkdb*: SLASH returns a single location in `main`, rather than a set of locations that constitute a multi-element BOUNDARY.
- *masscan*: Although the identified BOUNDARY satisfies all BOUNDARY properties, the code before this BOUNDARY does not actually parse the configurations. Instead, it configures the program to report debug information in case it crashes. A single variable, `is_backtrace`, is initialized inside a loop that parses `argv`, whereas the rest of the configuration-hosting variables are parsed inside another loop inside the function `masscan_command_line`, which is called from `main`. SLASH does mark the latter location as a candidate BOUNDARY initially, but it is then eliminated because it is not the closest to the entry of `main` (line 52 of Alg. 1).
- *Redis*: Despite running on a server with 192GB of memory, SLASH's data-flow analysis phase exhausted memory.

B. RQ2: Performance of SLASH

We measured the analysis time and memory usage for each program in Table III (averaged over 10 runs) using the UNIX `time` tool, which provides the total analysis time and the peak memory usage that a process uses. As shown in Fig. 3, analysis times and memory consumption scale roughly linearly with lines of code. (In both plots, the outliers are `curl` and `bind` on the high side and `readelf` and `objdump` on the low side.)

SLASH's taint analysis is influenced by a program's characteristics, such as the number of pointer variables (both # of pointer variables declared and # of statements that use a pointer variable), stores/loads, indirect call sites, etc. [20], [26], which affect the number of data-flow facts that need to be propagated through the target program and are not strictly linked to the number of lines of code. Table III provides information about the number of pointers and allocation sites in each program. We calculated the Pearson Correlation Coefficient to establish the strength of the relationship between kLOC, number of pointers, allocation sites, analysis time, and memory usage. The Correlation Coefficient in general was over 0.95, which indicates a positive linear relationship between these factors.

C. *RQ3*: Effectiveness of SLASH

We discuss three case studies to demonstrate the benefits of leveraging SLASH in state-of-the-art software-debloating tools [13], software security [14], [17], and configuration-error detection tools [15]. The case studies explain how the integration of SLASH with these tools alleviates the manual effort required by a developer, thus making the tools easier to use. These tasks involve combing through source code to track `argc` and `argv` usage, and potentially debugging complex programs to ensure the BOUNDARY is reachable from the program's entry point.

1) Software Debloating Tools: LMCAS [13] is a debloating tool that applies partial evaluation to specialize a program to a particular run-time configuration. Currently, the LMCAS user is required to annotate the program to specify the BOUNDARY.

We incorporated SLASH into the LMCAS pipeline: a program's LLVM IR bitcode is annotated by SLASH with the program's BOUNDARY location, and then passed to LMCAS. We evaluated our extension of LMCAS on the programs (obtained from the LMCAS dataset) listed in Table V. Our aim was to understand the efficiency gained by integrating SLASH into LMCAS. Improved efficiency means (1) reducing or eliminating the burden on the LMCAS user of identifying a program's BOUNDARY location, and (2) making sure that automatic BOUNDARY identification does not affect LMCAS's ability to create a correctly working debloated program. Correct functionality can be validated by running the debloated programs with the supplied test inputs, omitting the flags specifying the features for which they have been debloated. We matched the output of the debloated program with that of the original program, which was supplied the appropriate feature flags and the same inputs. If the output is the same, the debloated program is considered to have preserved the functionality. We also check that the SLASH-annotated programs do not crash the LMCAS debloating pipeline.

Table V reports the results of this experiment. SLASH reduces the analysis time of the BOUNDARY-identification step in LMCAS from minutes to a few seconds. It also eliminates human error due to manual analysis. Finally, SLASH + LMCAS preserves the functionality of debloated programs (and SLASH does not break the debloating pipeline of LMCAS).

Table V: Effectiveness of SLASH in facilitating BOUNDARY-identification for LMCAS.

Program	BOUNDARY ident. time		Accuracy	Functionality Preserved
	LMCAS (minutes)[a]	SLASH (seconds)		
chown	5 - 10	0.6	✓	✓
date	5 - 10	1.8	✓	✓
gzip	5 - 10	0.3	✓	✓
rm	5 - 10	0.7	✓	✓
sort	5 - 10	1.4	✓	✓
uniq	5 - 10	0.4	✓	✓

[a]Reported by the LMCAS authors [13].

Table VI: Effectiveness of SLASH in facilitating transition-point identification for the `temporal-specialization` and `C2C`.

Program	Accuracy	Master func. in config. logic	Worker func. in main logic
Bind	✓	✓	✓
Memcached	✓	✓	✓
Nginx	✓	✓	✓
Apache[a]	✓	✓	✓
Lighttpd	✓	✓	✓
Redis	✓	✓	✓
Vsftpd[b]	✓	✓	✓

[a]Not in `C2C` dataset. [b] Not in `temporal-specialization` dataset

2) Software Security Tools: `temporal-specialization` and `C2C` [14], [17] reduce the attack surface of programs by disabling unneeded system calls. The system calls to disable are determined by splitting programs into phases of initialization and processing: any system call never used in the processing phase is to be disabled once the (manually identified) "transition point" between the phases is reached. A transition point is the same concept as a BOUNDARY, so SLASH can be applied to the problem of transition-point identification. The functions for the initialization and serving phases are called the *master* and *worker*, respectively. Our evaluation tested whether the function calls representing each phase are correctly separated. We performed the following steps: (i) for each program in Table VI (obtained from the `temporal-specialization` and `C2C` datasets), we ran SLASH to identify the BOUNDARY; (ii) as ground truth, we used the master and worker functions employed in the `temporal-specialization` and `C2C` evaluation: If the configuration logic identified by SLASH includes a call to the master function, and the main-computation logic identified by SLASH contains a call to the worker function, we considered SLASH to have identified an appropriate transition point. As shown in Table VI, SLASH identified an appropriate transition point in each example.

3) PCHECK: PCHECK is an analysis tool that aims to detect configuration errors. It generates configuration-checking code to be invoked after program initialization. PCHECK requires users to identify the BOUNDARY manually.

We could not integrate SLASH with PCHECK because its implementation is unavailable. PCHECK's dataset includes three Java and three C/C++ programs, but the only BOUNDARIES defined in the paper are for `Squid` (C++) and `HDFS` (Java). Thus, we focused our attention on Squid (786k LOC). SLASH successfully identified `Squid`'s BOUNDARY in 41 seconds.

VII. THREATS TO VALIDITY

We outline threats to the validity of our approach, along with the applied mitigations:

- **Scope of the study** (Internal & external validity). We investigated programs whose configurations are provided through command-line input or configuration files. Some of our findings may not generalize to other kinds of software, such as event-driven programs (e.g., Android programs). For the field study, we selected a diverse set of widely used, mature programs. However, to avoid bias, we evaluated SLASH using popular programs from GitHub that we had *not* used to identify BOUNDARY properties.
- **Robustness of BOUNDARY properties** (External validity). The properties used by SLASH to infer BOUNDARY locations were inferred from the 24 programs in Table I. Moreover, the properties used by SLASH do not depend on heuristics like function/variable names and data types like int/string. SLASH also does not exploit special idioms that are used by some programmers for parsing programs' configurations. For instance, we observed that the GNU Coreutils programs use a particular idiom (i.e., the invocation of the function `getopt_long` inside a while-loop) for parsing command-line parameters. Instead, we decided to have SLASH rely on high-level structural properties that are driven by program-configuration semantics. The evaluation of 21 unseen programs validates the identified BOUNDARY properties in common programs and confirms SLASH's effectiveness in boundary identification.
- **Incorrect propagation of data flows** (External Validity). Our taint analysis, discussed in §V-A, is sound under practical assumptions, such as system and `llibC` calls behaving as expected. It does not account for `lsetjmp` and `llongjmp` usage or dynamically loaded code via `dlopen/dlsym`. If these assumptions are violated, the analysis becomes unsound.

Finally, there is the question of the soundness SLASH's results when run on the three different kinds of BOUNDARY cases.

- **Single-element**: `masscan` shows that SLASH is fallible, and can return an incorrect answer when a single-element BOUNDARY exists (see §VI-A2). The `masscan` result constitutes both a false positive and a false negative.
- **Multi-element**: in our limited experience, SLASH identifies the locations of a multi-element BOUNDARY as (individual) candidates, but because SLASH returns just a singleton location in `main`, the answer returned is a false positive.
- **Blended**: SLASH returns `null`, because the "blended" case involves violations of single-element-BOUNDARY properties.

VIII. Related Work

Multi-cut for Program Decomposition. Multi-cut algorithms [27] have been used in several program-optimization methods [28], [29], [30]. Ma et al. [30] presented a vertex-cut framework on LLVM IR graphs to partition coarse-grained dataflow graphs into parallel clusters to improve performance of applications in multi-core systems. In our work, only a degenerate form of min-cut is used: the algorithm identifies the set of articulation points, each member of which constitutes a cut-set of size 1. However, SLASH's static taint analysis is an improvement on the data-dependency analysis used in [30], which relies on dynamically generated traces.

Tracing Program Configurations. LOTRACK [1], [3] applies taint analysis for identifying all code that is influenced by load-time configurations in Android and Java programs. In Android applications, the identification of a BOUNDARY appears to be less of a problem: because Android apps, essentially plugins with a specific lifecycle in the Android framework, usually have their configuration logic completed (i.e., typically inside `onCreate` and before `onStart`) by the time of executing their main activity. Hence, this program point can thus serve as the BOUNDARY. This observation does not hold for regular Java programs, then we foresee SLASH can be leveraged to solve this BOUNDARY identification challenge in this context. Finally, LOTRACK relies on the assumption that configuration APIs are known; however, identifying such APIs can be cumbersome. SLASH does not require configuration APIs, the taint analysis of `argv` is sufficient to identify configuration-hosting variables including APIs that read configuration files.

IX. Conclusions and Future Work

This paper presents an algorithm and tool, called SLASH, to statically identify programs' configuration logic. Our evaluation on widely used C/C++ command-line and configuration-file programs confirmed the existence of a BOUNDARY and found that SLASH *automatically* identified a suitable BOUNDARY for 87.5% of the programs. Finally, we demonstrated an application of SLASH to reduce the manual-annotation burden in software-debloating and error-detection tools.

Additionally, we envision that SLASH can be used as a linting tool to alert developers that they have intertwined a program's configuration logic with its main-computation logic. Thus, SLASH supports ongoing initiatives [10] to promote configurability as a first-class programming concept.

In future work, we would like to examine the existence of BOUNDARIES in GUI programs and event-driven programs. Furthermore, multi-cut algorithms could allow SLASH to handle Swiss-Army-knife cases.

References

[1] M. Lillack, C. Kästner, and E. Bodden, "Tracking load-time configuration options," *IEEE Transactions on Software Engineering*, vol. 44, no. 12, pp. 1269–1291, 2018.

[2] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter, "Using symbolic evaluation to understand behavior in configurable software systems," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 445–454.

[3] M. Lillack, C. Kästner, and E. Bodden, "Tracking load-time configuration options," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 445–456. [Online]. Available: https://doi.org/10.1145/2642937.2643001

[4] Y. Zhang, H. He, O. Legunsen, S. Li, W. Dong, and T. Xu, "An evolutionary study of configuration design and implementation in cloud systems," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 188–200.

[5] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 307–319. [Online]. Available: https://doi.org/10.1145/2786805.2786852

[6] A. Rabkin and R. Katz, "Static extraction of program configuration options," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 131–140.

[7] S. Zhang and M. D. Ernst, "Which configuration option should i change?" in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 152–163.

[8] A. Rabkin and R. Katz, "Static extraction of program configuration options," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 131–140.

[9] S. Zhou, X. Liu, S. Li, W. Dong, X. Liao, and Y. Xiong, "Confmapper: Automated variable finding for configuration items in source code," in *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2016, pp. 228–235.

[10] P. Gazzillo and M. B. Cohen, "Bringing together configuration research: Towards a common ground," in *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 259–269. [Online]. Available: https://doi.org/10.1145/3563835.3568737

[11] J. Meinicke, C.-P. Wong, B. Vasilescu, and C. Kästner, "Exploring differences and commonalities between feature flags and configuration options," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 233–242. [Online]. Available: https://doi.org/10.1145/3377813.3381366

[12] M. Sayagh, N. Kerzazi, B. Adams, and F. Petrillo, "Software configuration engineering in practice interviews, survey, and systematic literature review," *IEEE Transactions on Software Engineering*, vol. 46, no. 6, pp. 646–673, 2020.

[13] M. Alhanahnah, R. Jain, V. Rastogi, S. Jha, and T. Reps, "Lightweight, multi-stage, compiler-assisted application specialization," in *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, 2022, pp. 251–269.

[14] 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)*. USENIX Association, Aug. 2020, pp. 1749–1766.

[15] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early detection of configuration errors to reduce failure damage," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 619–634.

[16] M. A. Inam, W. U. Hassan, A. Ahad, A. Bates, R. Tahir, T. Xu, and F. Zaffar, "Forensic Analysis of Configuration-based Attacks," in *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS'22)*, Feb. 2022.

[17] S. Ghavamnia, T. Palit, and M. Polychronakis, "C2c: Fine-grained configuration-driven system call filtering," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for

Computing Machinery, 2022, p. 1243–1257. [Online]. Available: https://doi.org/10.1145/3548606.3559366

[18] A. A. Ahmad, A. R. Noor, H. Sharif, U. Hameed, S. Asif, M. Anwar, A. Gehani, J. H. Siddiqui, and F. M. Zaffar, "Trimmer: An automated system for configuration-based software debloating," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[19] P. D. Schubert, B. Hermann, and E. Bodden, "Phasar: An interprocedural static analysis framework for c/c++," in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar and L. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 393–410.

[20] P. D. Schubert, R. Leer, B. Hermann, and E. Bodden, "Know your analysis: How instrumentation aids understanding static analysis," in *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ser. SOAP 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 8–13. [Online]. Available: https://doi.org/10.1145/3315568.3329965

[21] U. Jørring and W. L. Scherlis, "Compilers and staging transformations," in *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '86. New York, NY, USA: Association for Computing Machinery, 1986, p. 86–96. [Online]. Available: https://doi.org/10.1145/512644.512652

[22] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*. USA: Prentice-Hall, Inc., 1993.

[23] M. Sharir and A. Pnueli, "Two approaches to interprocedural data flow analysis," in *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[24] F. Angerer, A. Grimmer, H. Prähofer, and P. Grünbacher, "Configuration-aware change impact analysis," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. IEEE Press, 2015, p. 385–395.

[25] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 49–61. [Online]. Available: http://doi.acm.org/10.1145/199448.199462

[26] P. D. Schubert, R. Leer, B. Hermann, and E. Bodden, "Into the woods: Experiences from building a dataflow analysis framework for c/c++," in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2021, pp. 18–23.

[27] J. Chuzhoy and S. Khanna, "Polynomial flow-cut gaps and hardness of directed cut problems," *J. ACM*, vol. 56, no. 2, apr 2009.

[28] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar, "Min-cut program decomposition for thread-level speculation," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, ser. PLDI '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 59–70. [Online]. Available: https://doi.org/10.1145/996841.996851

[29] K. Ootsu, T. Abe, T. Yokota, and T. Baba, "Loop performance improvement for min-cut program decomposition method," in *2010 First International Conference on Networking and Computing*, 2010, pp. 78–87.

[30] G. Ma, Y. Xiao, T. L. Willke, N. K. Ahmed, S. Nazarian, and P. Bogdan, "A vertex cut based framework for load balancing and parallelism optimization in multi-core systems," *CoRR*, vol. abs/2010.04414, 2020. [Online]. Available: https://arxiv.org/abs/2010.04414