2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00067

ConfTainter: Static Taint Analysis For Configuration Options

Teng Wang[†‡], Haochen He[†‡], Xiaodong Liu[†*], Shanshan Li[†*], Zhouyang Jia[†], Yu Jiang[§], Qing Liao[¶], Wang Li[†]
[†]National University of Defense Technology, Changsha, China
[§]Tsinghua University, Beijing, China
[¶]Harbin Institute of Technology, Shenzhen, China
Email:{wangteng13, hehaochen13, liuxiaodong, shanshanli, jiazhouyang} @nudt.edu.cn,
jy1989@mail.tsinghua.edu.cn, liaoqing@hit.edu.cn, liwang2015@nudt.edu.cn

***Abstract*—The prevalence and severity of software configuration-induced issues have driven the design and development of a number of detection and diagnosis techniques. Many of these techniques need to perform static taint analysis on configuration-related variables to analyze the data flow, control flow, and execution paths given by configuration options. However, existing taint analysis or static slicer tools are not suitable for configuration analysis due to the complex effects of configuration on program behaviors.**

In this *experience paper*, we conducted an empirical study on the propagation policy of configuration options. We concluded four rules of how configurations affect program behaviors, among which implicit data-flow and control-flow propagation are often ignored by existing tools. We report our experience designing and implementing a taint analysis infrastructure for configurations, CONFTAINTER. It can support various kinds of configuration analysis, e.g., explicit or implicit analysis for data or control flow. Based on the infrastructure, researchers and developers can easily implement analysis techniques for different configuration-related targets, e.g., misconfiguration detection. We evaluated the effectiveness of CONFTAINTER on 5 popular open-source systems. The result shows that the accuracy rate of data- and control-flow analysis is 96.1% and 97.7%, and the recall rate is 94.2% and 95.5%, respectively. We also apply CONFTAINTER to two types of configuration-related tasks: misconfiguration detection and configuration-related bug detection. The result shows that CONFTAINTER is highly applicable for configuration-related tasks with a few lines of code.

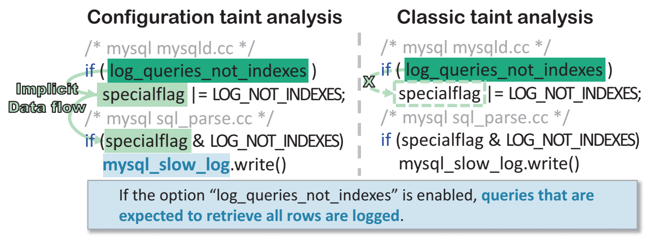
***Index Terms*—static taint analysis, configuration, data flow, control flow**

I. INTRODUCTION

Modern software systems introduce an increasing number of configuration options to provide flexibility and customizability for users [1]–[3]. At the same time, it also brings more configuration-induced failures and errors, such as configuration-related bugs and misconfigurations. For example, in one of Google's main production services, administrators find misconfigurations are the second largest reason for service failures [4]. At Facebook, 16% of service-level incidents are induced by configuration errors [5] and are considered a key reliability challenge at Facebook scale [6].

The prevalence and severity of software configuration-induced issues have driven the design and development of a number of detection and diagnosis techniques [7]–[13]. Many studies use static analysis methods, which analyze the data flow, control flow, and execution paths given by configuration options, to detect or diagnose configuration-induced issues. For example, SPEX [7] infers configuration constraints and detects misconfigurations by tracking the data and control flow of configuration options, and identifying summarized code patterns. ConfDiagnoser [8] diagnoses configuration errors by using dependence analysis on configuration options and identifying the execution paths it affects.

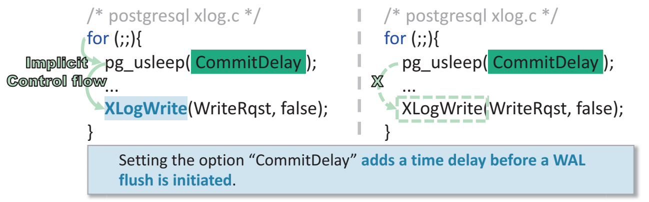
These detection and diagnosis studies, to some extent, utilize taint analysis techniques on configuration variables. Taint analysis is a common form of information-flow analysis, which tracks data and control flow to identify vulnerabilities and diagnose bugs. However, existing taint analysis or static slicer tools [14]–[16] are not specially designed and suitable for configuration analysis. On one hand, scaling these existing tools to implement configuration analysis is challenging because they do not provide interfaces for intermediate results of taint analysis. For instance, inferring configuration constraints involves the utilization of tainted variables to analyze code context and identify specific code patterns. On the other hand, these works do not cover certain



```
/* mysql mysqld.cc */
if ( log_queries_not_indexes )
  specialflag |= LOG_NOT_INDEXES;
/* mysql sql_parse.cc */
if (specialflag & LOG_NOT_INDEXES)
  mysql_slow_log.write()
```

If the option "log_queries_not_indexes" is enabled, queries that are expected to retrieve all rows are logged.

(a) Example of implicit data flow.



```
/* postgresql xlog.c */
for (;;){
  pg_usleep( CommitDelay );
  ...
  XLogWrite(WriteRqst, false);
}
```

Setting the option "CommitDelay" adds a time delay before a WAL flush is initiated.

(b) Example of implicit control flow.

Fig. 1: Example of the difference between configuration taint analysis and classic taint analysis.

‡ Teng Wang and Haochen He are co-first authors.
* Xiaodong Liu and Shanshan Li are the corresponding authors.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on December 31,2023 at 22:10:36 UTC from IEEE Xplore. Restrictions apply.

propagation policies of configurations necessary for various configuration analyses. Figure 1 illustrates the aspects that should be analyzed in configuration taint analysis, but are ignored in classic taint analysis. In Figure 1(a), the option `log_queries_not_indexes` implicitly affects the variable `specialflag` to control whether the software logs slow queries. However, the existing taint tools typically do not consider implicit data flows (`specialflag`) because it often leads to overtainting in vulnerability detection [17], [18]. In Figure 1(b), the option `CommitDelay` implicitly controls the execution interval and frequency of `XLogWrite()` by using a `sleep` statement. And the implicit control dependency cannot be recognized by the existing tools.

To address the knowledge gap, we conducted the first empirical study on the propagation policy of configuration options. We distinguished *explicit* and *implicit* propagation in both data and control flow. Explicit data flows directly propagate values, whereas implicit data flows affect other variables through conditional statements (§ III-B). Additionally, explicit control flows directly determine the execution of basic blocks, while implicit control flows affect the execution frequency of these blocks (§ III-C). These four types of propagation are all crucial for configuration analysis. Our study reveals that 34.2% of configuration options utilize implicit data flows. We found that configurations are frequently used in conditional statements to modify program behavior, and typically affect only a small number of variables within a single branch. This results in such propagation being a common occurrence for configuration options, and presents a favorable opportunity to decrease the likelihood of false positives. Furthermore, 14.2% of options exhibit implicit control dependence, indicating the prevalence of the propagation for configurations. Notably, to the best of our knowledge, previous studies have not investigated implicit control flows.

In this paper, we report our experience designing and implementing a taint analysis infrastructure for configurations, CONFTAINTER. It can support various kinds of configuration analysis, e.g., explicit or implicit analysis for data or control flows. CONFTAINTER is inter-procedural, field sensitive, and designed for C/C++ programs. To support applicability, CONFTAINTER also provides many interfaces for intermediate results of taint analysis. Based on the infrastructure, researchers and developers can easily implement analysis techniques for different configuration-related targets. CONFTAINTER conducts both data-flow and control-flow analysis for configuration options. One challenge is to achieve precise field and object-sensitive data-flow analysis in C/C++ programs, because objects are usually passed through pointers multiple times. To address this, we conduct a study towards the propagation feature of fields storing option values, and leverage the feature to solve it. Another challenge is to identify implicit data-flow propagation, particularly as there are usually many nested branches under configuration-dominated conditional statements. To achieve it, we utilize the feature of *phi* nodes in IR and define rules for identifying implicit data flow.

We evaluated the effectiveness of CONFTAINTER on 5 popular open-source systems. The result shows that the accuracy rate of data-flow analysis is 96.1%, and recall is 94.2%. Moreover, the accuracy of control-flow analysis is 97.7%, and recall is 95.5%. CONFTAINTER has comparable capabilities to existing tools in analyzing explicit data flows, and can identify implicit flows that existing tools could not analyze. For applicability, we apply CONFTAINTER to misconfiguration detection and configuration-related bug detection. The result indicates CONFTAINTER is highly applicable for configuration-related tasks with a few lines of code.

To summarize, this paper makes the following contributions:

- We conducted an empirical study on the propagation policy of configuration options. And we concluded four rules of how configurations affect program behaviors, i.e., explicit or implicit propagation of data or control flow.
- We designed and implemented CONFTAINTER, a taint analysis infrastructure to support various kinds of configuration analysis. The source code and data can be found in the repository:
  https://github.com/wangteng13/ConfTainter
- We evaluated the effectiveness of CONFTAINTER on 5 systems. The result shows that the accuracy rate of data- and control-flow analysis is 96.1% and 97.7%, and the recall rate is 94.2% and 95.5%, respectively. We also applied CONFTAINTER to misconfiguration detection and configuration bug detection, and the result shows CONFTAINTER is highly applicable for configuration tasks with a few lines of code.

II. BACKGROUND

In this section, we give some brief background on static taint analysis, and then we introduce different types of configuration-related studies which apply taint analysis.

A. Static Taint Analysis

Taint analysis [16], [17], [19]–[21] is a form of information-flow analysis, which tracks some selected data of interest as entry points(i.e., *taint seeds*), propagates them along program execution paths according to a customized policy (i.e., *taint propagation policy*), and then checks the taint status at certain critical location (i.e., *taint sinks*). It is a key technique in software security, which has been shown effective in dealing with software attack prevention [22], and data leak detection [23]. The taint seeds, propagation policy, and sinks are usually determined by the requirement of the tasks.

Static taint analysis propagates taint values following all possible paths with no need for concrete execution, and therefore it has no impact on runtime performance. Although static taint analysis may lead to potential imprecision (undertainting or overtainting), it is widely used in defect detection and fault diagnosis in various fields, not limited to security. Static taint analysis is also applied in many configuration-related studies, i.e., misconfiguration detection, misconfiguration diagnosis, and configuration-related bug detection. In these works, taint seeds are usually *configuration variables*, which load configuration values in the application and are used during the

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on December 31,2023 at 22:10:36 UTC from IEEE Xplore. Restrictions apply.

program execution. While, taint propagation policy and sinks are customized respectively.

B. Taint Analysis in Configuration Studies

Misconfiguration Detection. Misconfiguration detection [7], [10], [13] aims at detecting potential misconfigurations before deployment. Many studies discover configuration errors by checking against configuration constraints, which are inferred from source code using static program analysis. SPEX [7] infers configuration constraints by tracking the data and control flow of configuration options, and identifying predefined code patterns. CDep [10] focuses on configuration dependencies between multiple options in Java programs. For data-flow analysis, they propagate the tainted variables through assignments, arithmetic, and string operations, until reaching customed sink statements. For control-flow analysis, they find the conditional branch which involves configuration options, and take its dominating statements as the scope of its control-flow propagation.

Misconfiguration Diagnosis. Misconfiguration diagnosis [8], [11], [12] refers to locating the root causes of configuration errors that caused failures, performance issues, and incorrect behaviors. ConfDiagnoser [8] diagnoses configuration errors by using dependence analysis on configuration options to identify configuration-affected control flow. Specially, for each option, ConfDiagnoser statically determines its affected predicates in conditional statements, and identifies the execution paths it affects in source code.

Configuration-related Bug Detection. Static taint analysis is also widely used in bug detection. Toddler [24] identifies redundant memory access patterns in codes to detect performance bugs. SCIC [9] detects invalid configuration bugs by statically analyzing the data flow of configuration options and detecting whether the options are used in the source code. Moreover, taint analysis can also be applied to fuzzing techniques to improve test quality and efficiency [25].

III. Understanding the Propagation Policy of Configuration

In order to learn how configurations affect program behaviors, we conduct an empirical study on the propagation policy of configuration. In this section, we will first describe the study methodology, then introduce our findings including the data-flow and control-flow propagation.

A. Study Methodology

The study methodology includes the criteria to choose study subjects, and the methods to collect and analyze the propagation policy.

Studied Subjects. Table I describes 6 software systems used in our study. We chose these projects because: a) they cover different domains, including database, web and FTP server; b) they are widely used and studied by the existing works [7], [8], [13], [26]; c) they are highly configurable and all have various types of options; d) they are open-source and well maintained by the community. These criteria ensure that

```
1 /* vsftpd ftpdataio.c*/
2 static unsigned int get_chunk_size(){
3   ret = tunable_trans_chunk_size * BLCKSZ ;
4   return ret;
5 }
6 void do_file_recv(...){
7   unsigned int chunk_size = get_chunk_size();
8   int retval = ftp_read_data(..., chunk_size);
9 }
```

Fig. 2: Example of explicit data-flow propagation.

we can learn the propagation policy of various categories of configuration options in various systems.

Collection and Analysis. For each study system, we randomly select 100 options to learn the propagation policy. We first use *ConfMapper* [27] to locate the original configuration variables. Then, we collect and analyze the data-flow and control-flow propagation of the variables, both in source code and LLVM Intermediate Representation (IR). Each option was inspected by two inspectors. When they diverged, a third inspector was consulted for additional discussions until a consensus was reached. It took two months to collect and analyze the propagation policy. The results of the empirical study are shown in Table I. We concluded four rules of how configurations affect program behaviors, i.e., explicit and implicit propagation of data or control flow.

B. Data-flow Propagation

The data-flow propagation of configuration options can be divided into explicit and implicit data flow, based on whether the tainted variable explicitly propagates its value (e.g., by assignment operations) or not.

1) Explicit Data-flow Propagation: Variables are usually propagated through operation statements (e.g., assignment or arithmetic), or parameters and return values of functions in programs. In LLVM IR, the four forms can be traced back through the def-use chain [28] of the configuration variable. We refer to this type of propagation as *explicit data-flow* propagation. All options have explicit data-flow propagation.

Figure 2 shows an example of explicit data-flow propagation in VSFTPD. The configuration variable `tunable_trans_chunk_size` performs an arithmetic operation in Line 2, and the result is assigned to the variable `ret`. As a result, variable `ret` becomes a new taint seed. After that, the taint is propagated through the return statment (Line 4) to the variable `chunk_size` (Line 7). Then the taint is propagated as the parameter of the function `ftp_read_data` (Line 8).

2) Implicit Data-flow Propagation: Except for explicit data-flow propagation, configuration options can also affect variables implicitly through conditional statements. In this scenario, configuration options are not explicitly used as parameters in functions or operation statements. Instead, it determines which branch of the program would be executed, thereby implicitly altering the values of other variables and forming implicit associations. Moreover, this particular form of propagation cannot be traced through the def-use chain in LLVM IR. We refer to it as *implicit data-flow* propagation.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on December 31,2023 at 22:10:36 UTC from IEEE Xplore. Restrictions apply.

TABLE I: Studied software systems and the result of empirical study.

Project	Desc.	LOC	# Option	Explicit Data.	Implicit Data.	Explicit Control.	Implicit Control.
MySQL	SQL database	3714K	981	100	31	39	9
PostgreSQL	SQL database	2846K	275	100	21	77	19
HTTPD	Web Server	284K	669	100	32	54	13
Nginx	Web Server	144K	664	100	21	24	7
Squid	Web Server	180K	424	100	60	81	20
VSFTPD	FTP Server	16K	125	100	40	79	17
Total	-	-	-	600 (100.0%)	205 (34.2%)	354 (59.0%)	85 (14.2%)

However, this type of propagation is often overlooked by existing taint analysis tools [14]–[16], [29], as it often leads to false positives in vulnerability detection [17], [18]. Additionally, handling implicit flows requires a delicate balance between undertainting, overtainting, and efficiency [18], as there might be many affected variables in a single branch.

Nonetheless, this propagation is frequently (34.2%) observed for configuration options. Configurations are often used in conditional statements to modify program behavior. We find that configuration variables, on average, only affect three variables within a single branch. This makes such propagation a common occurrence for configuration options and presents a beneficial opportunity to reduce the likelihood of false positives. Figure 1(a) shows an example of implicit data-flow propagation in MySQL. The configuration variable `log_queries_not_indexes` affects the value of another variable `specialflag` through the conditional branch (Line 2-3). Especially, if `log_queries_not_indexes` is true, variable `specialflag` would be initialized with 'LOG_NOT_INDEXES'. Thus, `specialflag` has implicit data-flow dependency on the option.

Finding 1: Data-flow propagation of configurations can be realized through explicit operations (e.g., *assignments*), or through implicit means by conditional statements. The latter (34.2%) is common for configurations, but often ignored by existing tools.

C. Control-flow Propagation

The control-flow propagation of options can be divided into explicit and implicit control flow, based on whether the tainted variable explicitly dominates the branches or not.

1) Definitions: Before we delve into the detailed feature of control-flow propagation, we first introduce a set of definitions on control dependence from the previous study [30], [31].

Def 1. A node (basic block) Y ***post-dominates*** another node X (Y pdom X) if and only if all paths from X to an exit node of the control flow graph (CFG) must go through Y.

Def 2. A node (basic block) Y is ***control-dependent*** on another node X if and only if (a) Y post-dominates a successor of X; (b) Y does not post-dominate all successors of X.

Def 3. ***Transitivity of control dependency***: If node (basic block) A is control-dependent on another node B, and B is control-dependent on another node C, then A is control-dependent on C.

The above definitions state the control dependency between basic blocks, which means that the execution of Y depends on the control flow decision of X. Based on this, we propose an extension of the definition to include control dependency between basic blocks and configurations:

```
1 /* nginx os/unix/ngx_solaris_sendfilev_chain.c */
2 ngx_chain_t * ngx_solaris_sendfilev_chain(...){
3   if (!c-> sendfile )
4     return ngx_writev_chain(c, in, limit);
5 }
```

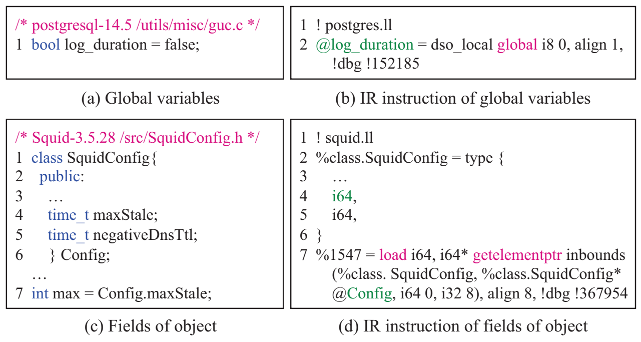
Fig. 3: Example of explicit control-flow propagation.

Def 4. A node (basic block) Y is ***control-dependent* on a configuration option** C if and only if (a) the branching instruction of node X is tainted by C; (b) Y is control-dependent on X.

2) Explicit Control-flow Propagation: Configuration options are commonly used as switches in software, enabling users to adjust various functional features of the system. These options often appear in conditional statements (e.g., *if*, *switch*, *for*, and *while*), and determine which branch of the program will be executed. As a result, the dominated branches and blocks are control-dependent on the configuration option. We refer to this type of control dependency as *explicit control-flow* propagation. In LLVM IR, configuration options explicitly influence the execution of basic blocks in different paths using branching (*br*) instructions. We find 59.0% of configuration options have explicit control-flow propagation in our study. Figure 3 shows an example of explicit control-flow propagation in Nginx. The call statement of function `ngx_writev_chain` (Line 4) is explicitly control-dependent on the option `sendfile`.

3) Implicit Control-flow Propagation: Except for explicit control-flow propagation, configuration options could also implicitly propagate control dependency and dominate program blocks, e.g., using delay statements. Although these options do not determine the execution of basic blocks, they can affect the execution frequency of these basic blocks. Especially, when a delay statement (e.g., *sleep* function) occurs in a loop and the configuration variable affects the parameter of the delay statement, the basic blocks in the loop can be considered as implicit control-flow dependencies. Another pattern is that inside a loop, the configuration variable controls a conditional statement and explicitly dominates a delay statement. Similarly, the basic blocks in the loop are implicitly control-dependent on the option. We refer to this type of control dependency as *implicit control-flow* propagation. We find 14.2% of configuration options have implicit control-flow propagation. To the best of our knowledge, previous studies have not investigated implicit control flows.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on December 31,2023 at 22:10:36 UTC from IEEE Xplore. Restrictions apply.



```
/* postgresql-14.5 /utils/misc/guc.c */
1 bool log_duration = false;
```

(a) Global variables

```
1 ! postgres.ll
2 @log_duration = dso_local global i8 0, align 1,
     !dbg !152185
```

(b) IR instruction of global variables

```
/* Squid-3.5.28 /src/SquidConfig.h */
1 class SquidConfig{
2   public:
3     …
4     time_t maxStale;
5     time_t negativeDnsTtl;
6     } Config;
  …
7 int max = Config.maxStale;
```

(c) Fields of object

```
1 ! squid.ll
2 %class.SquidConfig = type {
3     …
4     i64,
5     i64,
6 }
7 %1547 = load i64, i64* getelementptr inbounds
     (%class. SquidConfig, %class.SquidConfig*
     @Config, i64 0, i32 8), align 8, !dbg !367954
```

(d) IR instruction of fields of object

Fig. 4: Examples of configuration variable mapping and the IR instructions.

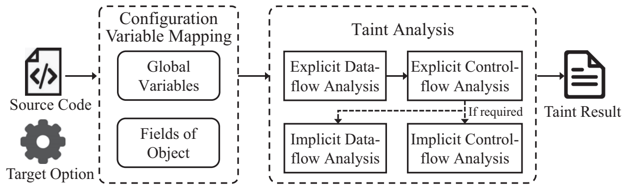

Source Code
Target Option
Configuration Variable Mapping
Global Variables
Fields of Object
Taint Analysis
Explicit Data-flow Analysis
Explicit Control-flow Analysis
If required
Implicit Data-flow Analysis
Implicit Control-flow Analysis
Taint Result

Fig. 5: Framework of CONFTAINTER.

Figure 1(b) shows an example of implicit control-flow propagation in PostgreSQL. The option `CommitDelay` affects the frequency of loop execution, and thus the frequency of other statements in the loop, e.g., `XLogWrite()`. Therefore, we consider statement `XLogWrite()` is implicitly control-dependent on option `CommitDelay`.

Finding 2: Control-flow propagation of configurations can be realized by explicitly determining the execution of basic blocks, or implicitly controlling the execution frequency. However, the latter (14.2%) could not be recognized by existing tools.

IV. DESIGN OF CONFTAINTER

In this section, we describe the design of CONFTAINTER, a taint analysis infrastructure for configurations, which can support various kinds of configuration analysis. CONFTAINTER is inter-procedural (tracking values across methods), field sensitive (configuration values could be stored in a field of a class or structure), and designed for C/C++ programs. Based on the infrastructure, researchers and developers can easily implement analysis techniques for different configuration-related targets. Figure 5 shows the overview of CONFTAINTER, which takes the source code and target option as input, and outputs the taint results. CONFTAINTER first does the mapping between the target option and the original configuration variable. After that, CONFTAINTER conducts both data-flow and control-flow analysis for configuration options. CONFTAINTER can analyze implicit data flow and control flow or not, according to the user's intention.

Algorithm 1: Explicit data-flow analysis

```
Input: $OriginalInst$ is the set of original instructions
       from Configuration Variable Mapping.
Output: $TaintSet$ is a set of tainted data-flow instructions.
1  $TaintSet \leftarrow \varnothing$;
2  $TQueue \leftarrow OriginalInst$;
3  while $TQueue \neq \varnothing$ do
4      $TaintIR \leftarrow TQueue$.pop();
5      insert $TaintIR$ into $TaintSet$;
6      $UseChain \leftarrow TaintIR$.getUsers();
7      for each $User \in UseChain$ do
8          if $User$.Type() $=$ $GetElementPtr$ instruction then
9              Conduct field-sensitive analysis;
10         else if $User$.Type() $=$ $call/return$ instruction then
11             Conduct inter-procedural analysis;
12         else
13             Conduct intra-procedural analysis;
```

A. Configuration Variable Mapping

Before conducting taint analysis, CONFTAINTER needs to select the configuration variables as taint seeds. Therefore, we first integrate ConfMapper [27] to find the original variable, which first loads the target option. ConfMapper is suitable for systems, whose mapping practice from options to program variables is clustered in specific source files and code snippets [27]. The accuracy of ConfMapper could reach nearly 95% in many C/C++ programs.

With the configuration variables found by ConfMapper, CONFTAINTER analyzes the LLVM IR to find the original IR instruction of these variables. There are two types of configuration variable mapping, i.e., global variables and fields of object. And CONFTAINTER can support both the two types of configuration variable mapping.

1) Global Variables: Figure 4a and 4b show an example of mapping by global variables in PostgreSQL. The global variable `log_duration` is the original variable used to load the option `log_duration`. Since global variables are prefixed with the '@' character and the string of variable name in IR, CONFTAINTER iterates over all global variables and searches for the string `@log_duration` to find the IR instruction.

2) Fields of Object: Figure 4c and 4d show an example of mapping by fields in Squid. Line 1-6 of Figure 4c declare the class `SquidConfig` and initialize the variable `Config`. Option `max_stale` is stored in the eighth field of `Config`, i.e., `maxStale`. In LLVM IR, fields of object are loaded by `GetElementPtr` instructions (line 7 in Figure 4d). CONFTAINTER iterates over all `GetElementPtr` instructions of the program, and searches for the specified class type and offset to find the IR instructions that load the configuration variable.

B. Explicit Data-flow Analysis

CONFTAINTER conducts explicit data-flow analysis first, and the others are based on explicit data-flow propagation.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on December 31,2023 at 22:10:36 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 shows the main process of explicit data-flow analysis. The input is the original IR instructions from Configuration Variable Mapping. And the output is the set of tainted instructions by explicit data-flow propagation of the target option. CONFTAINTER uses the def-use chain [28] and call graph in LLVM IR to implement data-flow taint tracking. In particular, we first initialize the output set, i.e., *TaintSet* (Line 1), and save the original IR instructions from input into *TQueue* (Line 2). Then, CONFTAINTER fetches each instruction from the *TQueue* (Line 4), and inserts the fetched instructions into *TaintSet* (Line 5), until the *TQueue* is empty.

For each fetched instruction, CONFTAINTER traverses its def-use chains to propagate taints (Line 6-13). A *User* is an instruction that uses the *def instruction* [28]. To simplify, CONFTAINTER will deal with different types of *Users*, respectively. **a)** If the *User* involves `GetElementPtr` instructions, which is used to load or store fields of object, CONFTAINTER conducts field-sensitive analysis to taint a field of object (Line 8-9). **b)** If the *User* involves call or return instructions, which is used to call some functions with the tainted variable as a parameter, CONFTAINTER conducts inter-procedural analysis (Line 10-11). **c)** In other cases, CONFTAINTER conducts intra-procedural analysis (Line 12-13), and inserts the newly tainted variable of the current *User* into the *TQueue*. For example, in Figure 4d, Squid loads the value of `Config.maxStale` to a temporary variable '%1547'. Thus, CONFTAINTER taints the temporary variable '%1547', and put it into *TQueue*.

1) Field-sensitive Analysis: It is challenging to conduct precise field and object-sensitive data-flow analysis in C/C++ programs, primarily due to the complexity of pointer analysis when objects are passed through pointers multiple times. To address this, we investigated how fields, used to store configuration variables, are propagated in source code. Our findings indicate that 32.5% (195/600) of configuration options are propagated to fields of specific structures. Furthermore, in 92.3% (180/195) of cases where a configuration option is propagated to a field of a specific structure, that field is only used to store the value of that option for all objects with the same structure. To enable CONFTAINTER to track such taints, we record the struct type and offset when propagating taints to fields. Then, when CONFTAINTER encounters `GetElementPtr` instructions with the same struct type and offset, it taints the result variable of those instructions and adds it to *TQueue* for further analysis.

2) Inter-procedural Taint Tracking: CONFTAINTER uses the call graph of the program to implement inter-procedural taint tracking. If taints are propagated to a function parameter, CONFTAINTER would take CFG of the called function, and make the relevant parameter as new taint seeds. CONFTAINTER only analyzes the source code of the target system, and stops tracking taints until entering external functions. On the other hand, if taints are propagated to return value of the function, CONFTAINTER would use the call graph to find all callers of the function, and make the call site as new taint seeds. Moreover, CONFTAINTER also supports inter-procedural analysis by references.

Algorithm 2: Explicit control-flow analysis and implicit taint analysis

```
Input: TaintSet is set of explicit data-flow instructions.
  DelayFuncSet is the set of user-defined delay functions
Output: ExplicitCSet is set of explicit controlled blocks.
  ImplicitCSet is a set of implicit controlled blocks.
  ImplicitTSet is a set of implicit data-flow instructions.
1  ExplicitCSet, ImplicitCSet, ImplicitTSet ← $\varnothing$;
2  for each TaintIR $\in$ TaintSet do
3      if TaintIR.Type() $=$ br instruction then
4          insert TaintIR.leftBranch into ExplicitCSet;
5          insert TaintIR.rightBranch into ExplicitCSet;
6          Conduct implicit data-flow analysis;
7          if TaintIR.dominates.contains(DelayFunc) then
8              Conduct implicit control-flow analysis;
9      else if TaintIR.Type() $=$ call instruction $\wedge$
         TaintIR.getFunction() $\in$ DelayFuncSet then
10         Conduct implicit control-flow analysis;
```

C. Explicit Control-flow Analysis

CONFTAINTER performs explicit control-flow analysis on the basis of explicit data-flow propagation. Algorithm 2 shows the main process of control-flow analysis and implicit taint analysis. The input is the results of data-flow analysis, i.e., the set of explicit tainted data-flow instructions. CONFTAINTER also requires a set of user-defined delay functions as inputs. And the outputs are the set of explicit, implicit control-flow blocks, and implicit data-flow instructions. CONFTAINTER uses the CFG in LLVM IR to implement control-flow taint tracking. In particular, we first initialize the three output sets (Line 1). Then, CONFTAINTER traverses the *TaintSet* and fetches each instruction from *TaintSet* to find the branching instructions, which are dominated by the target option (Line 2-3). When found, CONFTAINTER leverages transitivity of control dependency to find the controlled blocks, and collects them with with its branching instructions into *ExplicitCSet* (Line 4-5). CONFTAINTER also records the called functions in the controlled blocks. After that, CONFTAINTER conducts implicit taint analysis, i.e., implicit data-flow analysis (Line 6) and implicit control-flow analysis (Line 7-10).

D. Implicit Taint Analysis

CONFTAINTER performs implicit taint analysis according to the user's specifications.

1) Implicit data-flow analysis: Implicit data-flow analysis builds upon explicit data-flow and control-flow analysis. According to the definition of implicit data-flow propagation, CONFTAINTER should identify the variables which are assigned with different values in different configuration-dominated branches. The most intuitive approach to is to iterate through all the basic blocks in different branches, and collect all the variables that appear and are assigned in each branch. However, this can be difficult and error-prone because

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on December 31,2023 at 22:10:36 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Effectiveness of CONFTAINTER on data-flow and control-flow analysis.

Project	# Option	Discovered Data Flow	Accuracy	Recall	Discovered Control Flow	Accuracy	Recall
HTTPD	33	195	185/195 (94.9%)	185/201 (92.0%)	66	64/66 (97.0%)	64/68 (94.1%)
PostgreSQL	18	121	117/121 (96.7%)	117/124 (94.4%)	32	32/32 (100%)	32/32 (100%)
Squid	24	166	158/166 (95.2%)	158/167 (94.6%)	46	44/46 (95.7%)	44/48 (91.7%)
SQLite	12	55	53/55 (96.4%)	53/55 (96.4%)	14	14/14 (100%)	14/14 (100%)
Coreutils	13	72	72/72 (100%)	72/74 (97.3%)	16	16/16 (100%)	16/16 (100%)
Total	100	609	585/609 (96.1%)	585/621 (94.2%)	174	170/174 (97.7%)	170/178 (95.5%)

there may be many nested branches and basic blocks under configuration-dominated conditional statements.

To this end, CONFTAINTER uses *phi* nodes and instructions [32] of IR to implement implicit data-flow analysis. The *phi* instructions are used to choose one value based on a condition, and consist of several predecessor nodes with corresponding values. When CONFTAINTER identifies a conditional statement (CS) that is dominated by the target option, CONFTAINTER will check whether there is a *phi* instruction after the junction of two branches. Especially, we identify the specific *phi* instructions if **a)** there exists a predecessor node P1, which post-dominates a successor node of CS; and **b)** there exists another predecessor node P2, which is post-dominated by another successor node of CS. If found, CONFTAINTER would taint the result variable of the *phi* instruction as implicit data-flow propagation, and insert them into *ImplicitTSet*.

2) Implicit control-flow analysis: CONFTAINTER checks the pattern of implicit control flow described in § III-C3. CONFTAINTER first identifies whether the tainted function is a delay function and inside a loop (Line 9); also, CONFTAINTER checks whether the dominated blocks of the target option contain a delay function and inside a loop (Line 7). Then, CONFTAINTER regards the blocks in the loop as implicit control-flow propagation, and inserts them into *ImplicitCSet*. For example in Figure 1(b), CONFTAINTER finds the delay function `pg_usleep` is tainted by the target option `CommitDelay`, and the function is inside the `for` loop. Thus, CONFTAINTER collects the blocks of the loop as implicit control-flow propagation.

E. Implementation

We implement CONFTAINTER using the LLVM compiler infrastructure with version 10.0. CONFTAINTER uses Clang [33] and WLLVM [34] to build the IR of the target systems. Some parameters are necessary during building IR. Using '-fno-discard-value-names' ensures that the actual name of variables is present regardless of the build mode. '-g' ensures the standard debugging information of the target systems. '-O0' ensures turning off compile-time optimizations to preserve semantic information of the source code. '-mem2reg' ensures *phi* nodes for SSA optimization. We tried our best to make CONFTAINTER support various types of IR instructions, including terminator, binary, bitwise binary, memory, and other instructions. For applicability, CONFTAINTER provides many interfaces for intermediate results of taint analysis, e.g., `getExplicitDataflow()`. To make the output readable, CONFTAINTER also provides interfaces to output the process of how taints are propagated from the original variable to any tainted variables, both in source code and IR level.

V. EVALUATION

All experiments are conducted on a 64-bit Ubuntu 18.04 machine (8 cores, Intel Core i7-9700K, and 32GB RAM). To evaluate CONFTAINTER, we consider the following three research questions:

- **RQ1:** How effective is CONFTAINTER in analyzing data-flow and control-flow propagation? This question evaluates the accuracy rate and recall rate of CONFTAINTER in both data-flow and control-flow analysis.
- **RQ2:** Can CONFTAINTER outperform the existing popular tools for taint analysis? This question compares CONFTAINTER with DG [14] and SVF [16].
- **RQ3:** How applicable is CONFTAINTER to apply to configuration-related tasks? This question evaluates the applicability by applying CONFTAINTER to misconfiguration detection and configuration-related bug detection.

A. RQ1: Effectiveness on Analyzing Data-flow and Control-flow Propagation

1) Experimental Setup: To evaluate the effectiveness of CONFTAINTER in analyzing data- and control-flow propagation, we choose 3 large-scale systems (HTTPD, PostgreSQL, and Squid) as the target systems. To avoid over-fitting, we also choose SQLite [35] and Coreutils [36], which are not included in our study. SQLite is a popular embedded database engine, and widely used in web browsers and operating systems. Coreutils are the union package of GNU core utilities, and we select global variables of Coreutils to simulate configuration options. We use SQLite and Coreutils to evaluate the effectiveness of CONFTAINTER in medium- and small-scale systems.

We randomly sample 100 options from the five systems as evaluation targets, based on the proportion of options in each system. Please note that the selected options do not overlap with the options we studied in Section III. Then, we manually analyze and collect the data- and control-flow propagation of the target options as ground truth. The analysis and collection were independently done by two authors, and took five days. When they diverged, a third inspector was consulted for additional discussions until consensus were reached. The ground truth is shown in Table II. For data-flow analysis, we check the accuracy and recall rate on the code statements CONFTAINTER finds. For control-flow analysis, we check the accuracy and recall rate on the basic blocks CONFTAINTER finds.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on December 31,2023 at 22:10:36 UTC from IEEE Xplore. Restrictions apply.

2) Results and analysis: The evaluation results are shown in Table II. Overall, CONFTAINTER performs well both in data-flow and control-flow analysis. The average accuracy rate of data-flow analysis is 96.1% (585/609), and the average recall rate is 94.2% (585/621). The average accuracy of control-flow analysis is 97.7% (170/174), and the average recall is 95.5% (170/178). And the execution time of CONFTAINTER for a single option is within 20 minutes.

The false positive of CONFTAINTER is mainly due to inaccurate pointer analysis and lack of path-sensitive analysis. In this paper, we find configuration options are usually stored in some field variables of specific structures, and one field would only be used to store the corresponding configuration option. Thus, CONFTAINTER approximates pointer analysis by tainting the fields with the same struct type and offset, which would lead to false positives. For example, in HTTPD, the option `limit_xml_body` is assigned to the field `limit_xml_body` of structure `core_dir_config`. And CONFTAINTER would taint all the statements where the field `limit_xml_body` of structure `core_dir_config` appears. The false positive occurs as HTTPD initializes the field during startup. While there is no data-flow propagation of the option at this time.

The false negative of CONFTAINTER is mainly due to the non-availability of some functions, e.g., standard library or external library functions. CONFTAINTER only analyzes the source code of the target system, which would break potential data- and control-flow propagation. Moreover, the inaccuracy of pointer analysis also makes CONFTAINTER misses some data and control-flow propagation.

Answer to RQ1: This result indicates that CONFTAINTER is effective in data- and control-flow analysis, with an accuracy of 96.1% and 97.7%, and a recall of 94.2% and 95.5%, respectively.

B. RQ2: Comparison with the Existing Tools

1) Experimental Setup: Taint analysis is often used for fault diagnosis. In order to evaluate the effectiveness of CONFTAINTER in real-world cases, we picked 20 configuration-related bugs from 4 popular systems (MySQL, HTTPD, Squid, and SQLite). We selected the bug-induced options as target options, and extracted the code statements tainted by target options from the relevant patches as ground truth. The analysis and collection were independently performed by two authors to ensure accuracy.

We compare CONFTAINTER with DG [14] and SVF [16]. DG is also one of the most popular open-source frameworks and supports both data- and control-flow analysis. SVF is a static data-flow analysis framework and has more than 1k stars in Github. We executed the three tools to conduct taint analysis on the target options on the bug-fixed versions, and examined how many explicit data flows and implicit flows they could identify in the ground truth.

2) Results and analysis: The evaluation results in real cases are shown in Table III. We found that CONFTAINTER has comparable capabilities to DG and SVF in analyzing explicit data flows. Specifically, it was able to identify 96.0% (144/150) of the explicit data flows in the ground truth, which is comparable to DG (98.0%) and SVF (98.7%). CONFTAINTER lacks precise pointer analysis, which leads to fewer discoveries of some data flows compared to DG and SVF. However, CONFTAINTER is capable of identifying a significant number of implicit data and control flows that DG and SVF were unable to detect. This suggests that classic taint analysis tools may not be fully equipped for configuration analysis.

TABLE III: Taint analysis results in real cases compared with DG [14] and SVF [16].

Bug ID	**Option**	**LOCE**†	**Ours**	**DG**	**SVF**	**Implicit**‡
MySQL #28808	log_queries_not_indexes	4	4	4	4	2
MySQL #29015	tmpdir	9	9	9	9	0
MySQL #51631	general_log	10	10	10	10	0
MySQL #45689	interactive_timeout	7	7	7	7	2
MySQL #44100	max_connections	6	6	6	6	0
HTTPD #43502	CustomLog	4	4	4	4	0
HTTPD #44736	VirtualHost	4	4	4	4	0
HTTPD #56226	KeepAliveTimeout	8	7	7	8	2
HTTPD #35256	AllowEncodedSlashes	12	12	12	12	3
HTTPD #61355	DirectorySlash	11	8	10	10	0
Squid #1703	diskd_program	7	7	7	7	0
Squid #579	useragent_log	4	4	4	4	2
Squid #918	cache_dir	5	5	5	5	0
Squid #4827	netdb_filename	7	5	7	7	0
Squid #1931	allow_underscore	4	4	4	4	1
SQLite #a1fa	journal_size_limit	10	10	10	10	0
SQLite #c48d	count_changes	11	11	10	10	0
SQLite #eb94	reverse_unordered	7	7	7	7	2
SQLite #3c9e	integrity_check	10	10	10	10	0
SQLite #a340	case_sensitive_like	10	10	10	10	2
Total	-	150	144	147	148	16

† LOCE is short for Line of Code of Explicit data flows of the options in the patch of the bug.
‡ Implicit: implicit data and control flows found by CONFTAINTER.

Answer to RQ2: This result indicates CONFTAINTER has comparable capabilities to existing tools in analyzing explicit data flows. Additionally, CONFTAINTER can identify implicit flows that existing tools are not able to analyze.

C. RQ3: Applicability of CONFTAINTER in Configuration-related Tasks

In this section, we evaluate the applicability by applying CONFTAINTER to two types of configuration-related tasks, i.e., misconfiguration detection and configuration-related bug detection.

1) Misconfiguration detection: Some studies of misconfiguration detection leverage static analysis to infer configuration constraints from source code. For example, SPEX [7] infers constraints (e.g., value relationship and control dependency) by tracking the data and control flow of configuration options, and identifying custom code patterns. The code patterns are as followed: a) If two variables from different options' data-flow paths are compared with each other, SPEX infers value relationship of the two options. b) If the conditional statement and its dominating statements involve two variables from different options' data-flow paths respectively, SPEX infers control dependency of the two options.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on December 31,2023 at 22:10:36 UTC from IEEE Xplore. Restrictions apply.

TABLE IV: Configuration constraints inferred by TInfer and SPEX. Each number indicates constraints inferred by the respective tool, and the percentage indicates the accuracy.

Project	Value Rel.		Control Dep.	
	TInfer	SPEX	TInfer	SPEX
PostgreSQL	8 (88.9%)	6 (85.7%)	42 (91.3%)	44 (91.7%)
OpenLDAP	3 (75.0%)	2 (50.0%)	2 (66.7%)	0 (N/A)
Squid	9 (100.0%)	9 (100.0%)	15 (83.3%)	14 (77.8%)
Total	**20 (90.9%)**	17 (85.0%)	**59 (88.1%)**	58 (87.9%)

In this section, we tried a prototype tool by scaling CONFTAINTER to infer constraints, by using the same patterns in SPEX. Based on CONFTAINTER infrastructure, We implement the prototype tool, TInfer, with no more than **90** lines of code in C++ language. We evaluate TInfer and SPEX on three systems (HTTPD, OpenLDAP and Squid). The results are manually confirmed by two authors by checking out the official documentation and source code. As shown in Table IV, the result indicates TInfer has comparable capabilities in inferring constraints compared with SPEX. And CONFTAINTER can infer **4** more constraints than SPEX with higher accuracy. The prototype experiment indicates CONFTAINTER is highly applicable for configuration analysis.

2) Configuration Update Bug Detection: Many software systems support updating the options dynamically to provide flexibility and persistent services. However, dynamic configuration updates may also affect the system reliability at the same time. Many bug reports [37]–[41] show that, it may lead to unexpected results like software crashes or functional errors, even if the new option values are valid. We collected 75 real-world bugs from 5 widely used systems, and found nearly half (45%) of bugs fail to assign configuration-related variables with updated values [42]. Therefore, we designed a method of metamorphic testing to detect the bugs with the oracle: The values of program variables related to configuration options should be the same, no matter whether the options are loaded since the system startup or updated dynamically.

In this section, we tried a prototype tool, TCub, by scaling CONFTAINTER to detect configuration update bugs. Specially, we conduct two executions for each test; the first execution loads the option at the startup phase, while the second execution updates the option to the same value at runtime. We record the values of configuration-related variables during the two executions, and check whether the same or not to identify the bugs. Benefiting from the high applicability and program interface of CONFTAINTER, we implement the submodule of TCub to find the variables tainted by (explicit and implicit) data flow of target options and their locations, with no more than **40** lines of code in C++ language.

We evaluate TCub on 33 reproduced bugs from 5 systems to detect known bugs, and the latest version of the same systems to detect unknown bugs. The results are shown in Table V. TCub successfully detected most (31/33=94%) of the known bugs. In the 2 cases left, taint analysis fails to get configuration-related variables due to complicated pointer analysis. Moreover, TCub found 5 new bugs in 4 systems, all of which have been confirmed or fixed by developers. The prototype experiment indicates CONFTAINTER is highly applicable for configuration bug detection.

TABLE V: Effectiveness of TCub on known and new bugs.

Project	Reproduced Bugs	Detected by TCub	New bugs
MySQL	11	11	1
MariaDB	6	5	1
Redis	13	13	1
PosrgreSQL	1	1	0
Squid	2	1	2
Total	33	**31**	**5**

TABLE VI: Coverage of configuration fuzzing.

Coverage	TFuzz	Baseline
Configuration coverage	26/27 (96.3%)	23/27 (85.2%)
Basic block coverage	113/224 (50.4%)	93/224 (41.5%)
Branch coverage	96/197 (48.7%)	82/197 (41.6%)

3) Fuzzing for configuration bugs: Fuzzing is a promising approach for vulnerability detection and has been applied to various software. However, the existing fuzzing technology usually ignores configuration options as part of fuzzing seeds, which may miss some configuration-related bugs.

In this section, we tried a prototype experiment and tool, TFuzz, by applying CONFTAINTER to directed grey-box fuzzing [43] for configuration-related bug detection. In particular, we first use CONFTAINTER to find configuration-related code statements, including the data-flow and explicit control-flow propagation. And then we instrument the basic blocks involved, which are used to collect the coverage of basic blocks related to configurations. After that, we use Aflgo [43] with Squirrel [44] fuzzing framework to generate test cases (SQL statements). We also design mutation rules for configurations to increase coverage, e.g., randomly mutating the options that dominate conditional statements.

Based on CONFTAINTER infrastructure, we implement the submodule to record the basic blocks tainted by configurations with no more than **100** lines of code in C++ language. To evaluate the effectiveness, we collect 27 options in SQLite for the fuzzing test. We set the fuzzing method without configuration-related code guidance (Aflgo + Squirrel) as the baseline. The results are shown in Table VI. With configuration-related code guidance, the coverage of configuration options is increased from 85.2% (23/27) to 96.3% (26/27). And the basic block coverage of configuration options increased from 41.5% (93/224) to 50.4% (113/224). Moreover, branch coverage of configuration options increased from 41.6% (82/197) to 48.7% (96/197). On the other hand, we also find four hangs and crashes [1] caused by configurations, which could not be found by previous works. The prototype experiment indicates CONFTAINTER is highly applicable for configuration bug detection.

Answer to RQ3: This result indicates CONFTAINTER is

[1]The details of the four hangs and crashes are shown in our repository.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on December 31,2023 at 22:10:36 UTC from IEEE Xplore. Restrictions apply.

highly applicable and scalable for configuration-related tasks with a few lines of code.

D. Discussion

In this section, we discuss the limitations of CONFTAINTER. The first limitation is the potential inaccuracy of configuration variable mapping. We integrate ConfMapper [27] to find the original configuration variable. Although the accuracy of ConfMapper could reach 95% in many C/C++ programs, potential errors of ConfMapper might reduce the accuracy of CONFTAINTER. To this end, we also provide interfaces to accept user-provided variable mapping. This also enables CONFTAINTER to support taint analysis on regular variables. The second limitation is that the analysis of ConfTainter is not path-sensitive, which might lead to overtainting. Some variables should not be tainted because they have not stored option values at runtime. Our future work will focus on supporting path-sensitive analysis to reduce false positives. Another limitation is the potential omissions in the propagation policy of configurations. In this paper, we concluded the implicit control-flow propagation, which had not been investigated by previous studies. To avoid missing some implicit propagation, we selected various categories of systems and configuration options for the empirical study.

VI. LESSONS LEARNED

In this section, we discuss our experience and lessons learned while developing and applying CONFTAINTER.

Leveraging the propagation feature of taint seeds is able to improve the efficiency of field-sensitive analysis. Achieving precise field and object-sensitive analysis in C/C++ programs is typically challenging. To achieve this goal, taint analysis tools need to track all traces of the tainted object and inspect whether the specific field propagates the taint. The lack of precise pointer and path-sensitive analysis also leads to false positives and false negatives. To this end, we conduct a study on the propagation feature of fields that store option values. We find that configuration options are usually stored in some specific fields of particular structures, with one field designated to store the corresponding option. Therefore, we record the struct type and offset if taints are propagated to fields. CONFTAINTER would then taint all the fields with the same struct type and offset, which significantly improved the efficiency, compared to other taint tools.

A good practice in configuration design is to consider the interplay between user-defined configurations, developer-defined constants, and workload-related variables. Throughout our investigation of configuration propagation, we frequently observed that configurations interact with constants and workload-related variables, thereby influencing program behaviors. For instance, in the following code, the maximum value among the configuration option (i.e., `srv_page_size`), constant (i.e., `IO_BLOCK_SIZE`), and workload-related variable (i.e., `io_size`) is used to calculate the IO buffer size for file sorting. By taking these factors into account, developers can enable end-users to customize the behavior of the system while preventing unexpected side effects from user-defined configurations. This practice also ensures that the system is adaptable to varying workload.

```
int merge_io_buffer_size(int n_buffers){
  int io_size = load_io_buffer_size(n_buffers);
  return max(max( srv_page_size ,IO_BLOCK_SIZE),io_size);}
```

Scalable infrastructure and convenient interfaces for intermediate results can effectively help developers to perform configuration analysis. There are many existing taint analysis tools [14]–[16], [29]. These techniques, however, are rarely used in configuration analysis due to a lack of supporting some propagation policies of configurations and interfaces for intermediate results. Therefore, we implemented CONFTAINTER as a taint analysis infrastructure to support various kinds of configuration analysis. For applicability, we provide many interfaces for intermediate results of taint analysis, e.g., `getExplicitDataflow()`. Based on the infrastructure and intermediate results, developers can easily implement analysis techniques for different configuration-related targets. The evaluation in Section V-C shows that, it is easy to apply CONFTAINTER infrastructure to misconfiguration and bug detection with only a small amount of C++ code.

VII. RELATED WORK

Static taint analysis. Static taint analysis [14]–[16], [29] tracks data and control flow to identify bugs, with no need for concrete execution. DG [14] conducts data and control dependence analysis to construct dependence graphs [45] of the program. SUTURE [15] conducts static points-to and high-order taint analysis to discover high-order taint vulnerabilities. These methods, however, are not suitable for configuration analysis due to the lack of certain propagation policies of configurations. In this paper, we conduct an empirical study on the propagation policy of configurations, and design CONFTAINTER to support various kinds of configuration analysis, e.g., explicit and implicit analysis for data or control flow.

Some works [46]–[51] focus on improving the precision and efficiency of static analysis. ConDySTA [46] uses dynamic analysis results as supplements to static analysis to reduce false negatives. Fusion [47] improves the scalability of the path-sensitive analysis by conducting an optimized SMT-solving method, which works directly on the program dependence graph. Dillig et al. [48] improve path sensitivity by considering the variable observability and sufficient conditions of original path constraints. P/Taint [49] unifies points-to analysis and taint analysis to improve precision and recall. Pearce et al. [51] extend the set-constraints language to support an efficient field-sensitive pointer analysis for C programs. These methods can also be used as supplements and improve the effectiveness of CONFTAINTER.

Dynamic taint analysis. Dynamic taint analysis [18], [52]–[54] is also a form of information flow analysis, useful for identifying the origin of data during execution. Phosphor [52] conducts dynamic taint tracking for the Java Virtual Machine (JVM), and simultaneously achieves high performance, soundness, precision, and portability. Neutaint [53] employs neural

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on December 31,2023 at 22:10:36 UTC from IEEE Xplore. Restrictions apply.

program embeddings to conduct the dynamic taint analysis, and utilizes saliency maps to reason about the most influential sources for different sinks. TaintInduce [54] automatically learns taint rules with minimal architectural knowledge by observing the execution behavior of instructions.

VIII. Conclusion

Taint analysis is a form of information-flow analysis, which tracks data and control flow to identify vulnerabilities and diagnose bugs. However, existing taint analysis tools are not suitable for configuration analysis due to the complex effects of configurations on program behaviors. We conducted an empirical study and concluded four propagation policies of configuration options. In this paper, we design and implement a taint analysis infrastructure for configurations, ConfTainter, which can support explicit and implicit analysis for data or control flow. Based on the infrastructure, researchers and developers can easily implement analysis techniques for different configuration-related targets. The evaluation shows ConfTainter is is highly effective and applicable.

Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments. This work is supported by the National Key R&D Program of China (No.2022YFA1005101), NSFC No.62272473, the Science and Technology Innovation Program of Hunan Province (No.2023RC1001) and NSFC No.62202474.

References

[1] 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.

[2] 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, pp. 307–319.

[3] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 159–172.

[4] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines*, 3rd ed. Morgan and Claypool Publishers, October 2018.

[5] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic configuration management at facebook," in *Proceedings of the 25th symposium on operating systems principles*, 2015, pp. 328–343.

[6] B. Maurer, "Fail at scale: Reliability in the face of rapid change," *Queue*, vol. 13, no. 8, pp. 30–46, 2015.

[7] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 244–259.

[8] S. Zhang and M. D. Ernst, "Automated diagnosis of software configuration errors," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 312–321.

[9] F. Behrang, M. B. Cohen, and A. Orso, "Users beware: Preference inconsistencies ahead," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 295–306.

[10] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu, "Understanding and discovering software configuration dependencies in cloud and datacenter systems," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 362–374.

[11] A. Rabkin and R. Katz, "Precomputing possible configuration error diagnoses," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 193–202.

[12] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: error diagnosis by connecting clues from run-time logs," in *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, 2010, pp. 143–154.

[13] S. Zhou, X. Liu, S. Li, Z. Jia, Y. Zhang, T. Wang, W. Li, and X. Liao, "Confinlog: Leveraging software logs to infer configuration constraints," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 94–105.

[14] M. Chalupa, "Dg: analysis and slicing of llvm bitcode," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2020, pp. 557–563.

[15] H. Zhang, W. Chen, Y. Hao, G. Li, Y. Zhai, X. Zou, and Z. Qian, "Statically discovering high-order taint style vulnerabilities in os kernels," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 811–824.

[16] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*, 2016, pp. 265–266.

[17] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[18] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, pp. 1–29, 2014.

[19] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Still: Exploit code detection via static taint and initialization analyses," in *2008 Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2008, pp. 289–298.

[20] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: effective taint analysis of web applications," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 87–97, 2009.

[21] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, "Taintpipe: Pipelined symbolic taint analysis," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 65–80.

[22] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks." in *USENIX Security Symposium*, 2006, pp. 121–136.

[23] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "Tainteraser: Protecting sensitive data leaks using application-level taint tracking," *ACM SIGOPS Operating Systems Review*, vol. 45, no. 1, pp. 142–154, 2011.

[24] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: Detecting performance problems via similar memory-access patterns," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 562–571.

[25] E. Geretto, C. Giuffrida, H. Bos, and E. Van Der Kouwe, "Snappy: Efficient fuzzing with adaptive and mutable snapshots," in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022, pp. 375–387.

[26] W. Li, Z. Jia, S. Li, Y. Zhang, T. Wang, E. Xu, J. Wang, and X. Liao, "Challenges and opportunities: An in-depth empirical study on configuration error injection testing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, pp. 478–490.

[27] S. Zhou, X. Liu, S. Li, W. Dong, and X. Yun, "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.

[28] "Iterating over def-use & use-def chains." https://llvm.org/docs/ProgrammersManual.html#iterating-over-def-use-use-def-chains, 2022.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on December 31,2023 at 22:10:36 UTC from IEEE Xplore. Restrictions apply.

[29] M. Sridharan, S. J. Fink, and R. Bodik, “Thin slicing,” in *Proceedings of the 28th ACM SIGPLAN conference on programming language design and implementation*, 2007, pp. 112–122.
[30] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, “Conversion of control dependence to data dependence,” in *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1983, pp. 177–189.
[31] T. Lengauer and R. E. Tarjan, “A fast algorithm for finding dominators in a flowgraph,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, no. 1, pp. 121–141, 1979.
[32] “Phi instruction in LLVM.” https://llvm.org/docs/LangRef.html#phi-instruction, 2022.
[33] “Clang: a C language family frontend for LLVM.” https://clang.llvm.org/, 2022.
[34] “WLLVM: Whole Program LLVM.” https://github.com/SRI-CSL/whole-program-llvm, 2022.
[35] “What Is SQLite?” https://www.sqlite.org/index.html, 2022.
[36] “GNU core utilities.” https://github.com/coreutils/coreutils, 2022.
[37] “MariaDB Bug #23988. SST failed: No route to host after set global wsrep_node_name on donor.” https://jira.mariadb.org/browse/MDEV-23988, 2020.
[38] “Squid Bug #2604. Changing icap_enable setting in squid.conf fails to take effect after calling squid -k reconfigure.” https://bugs.squid-cache.org/show_bug.cgi?id=2604, 2009.
[39] “PostgreSQL Bug #16160. Minor memory leak in case of starting postgres server with SSL encryption.” https://www.postgresql.org/message-id/16160-18367e56e9a28264%40postgresql.org, 2019.
[40] “Redis Bug #4545. dead loop AOF rewrite when config set appendonly no.” https://github.com/redis/redis/issues/4545, 2017.
[41] “Nginx Bug #796. nginx.pid is removed during reload if pid path is changed in nginx.conf but points to the same file through a symlink.” https://trac.nginx.org/nginx/ticket/796, 2022.
[42] T. Wang, Z. Jia, S. Li, S. Zheng, Y. Yu, E. Xu, S. Peng, and X. Liao, “Understanding and detecting on-the-fly configuration bugs,” in *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, 2023.
[43] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, “Directed greybox fuzzing,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.
[44] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee, and D. Wu, “Squirrel: Testing database management systems with language validity and coverage feedback,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 955–970.
[45] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The program dependence graph and its use in optimization,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.
[46] X. Zhang, X. Wang, R. Slavin, and J. Niu, “Condysta: Context-aware dynamic supplement to static taint analysis,” in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 796–812.
[47] Q. Shi, P. Yao, R. Wu, and C. Zhang, “Path-sensitive sparse analysis without path conditions,” in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 930–943.
[48] I. Dillig, T. Dillig, and A. Aiken, “Sound, complete and scalable path-sensitive analysis,” in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 270–280.
[49] N. Grech and Y. Smaragdakis, “P/taint: Unified points-to and taint analysis,” *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017.
[50] M. Das, S. Lerner, and M. Seigle, “Esp: Path-sensitive program verification in polynomial time,” in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002, pp. 57–68.
[51] D. J. Pearce, P. H. Kelly, and C. Hankin, “Efficient field-sensitive pointer analysis of c,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 30, no. 1, pp. 4–es, 2007.
[52] J. Bell and G. Kaiser, “Phosphor: Illuminating dynamic data flow in commodity jvms,” *ACM Sigplan Notices*, vol. 49, no. 10, pp. 83–101, 2014.
[53] D. She, Y. Chen, A. Shah, B. Ray, and S. Jana, “Neutaint: Efficient dynamic taint analysis with neural networks,” in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1527–1543.
[54] Z. L. Chua, Y. Wang, T. Baluta, P. Saxena, Z. Liang, and P. Su, “One engine to serve’em all: Inferring taint rules without architectural semantics.” in *NDSS*, 2019.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on December 31,2023 at 22:10:36 UTC from IEEE Xplore. Restrictions apply.