cFlow: Flow-based Configuration Analysis Framework for Java Bytecode Based Cloud Systems

Zhanghao Chen, Zijie Lu
University of Illinois at Urbana-Champaign
{zc32,zijielu2}@illinois.edu

Abstract

Modern software systems, especially open-source cloud systems, are highly customizable via configuration. Misconfiguration can lead to costly problems such as performance degradation and system failures. Existing attempts to address this issue usually rely on static taint analysis to track the usage of configuration options in the system. The static taint analysis component in existing attempts is usually deeply coupled with their upstream analyses and does not support taint propagation path recovery. In this project we propose cFlow, a flow-, field-, and context-sensitive static taint analysis framework for Java bytecode-based cloud systems. cFlow tracks the flow of configuration options from loading points to user-specified sink points, and outputs the taint propagation paths for upstream configuration analyses. Since there is no existing benchmark for taint tracking in cloud systems, we apply cFlow on Hadoop Common and manually check the taint propagation path starting from selected configuration loading points. Our preliminary evaluation shows cFlow can correctly track the flow of configuration options starting from 60 out of 150 inspected sources, and 73 of the 90 incorrect cases are due to uncovered library modeling and implicit information flow, which is not supported by design.

1 Introduction

Many modern software systems are highly customizable, exposing a wide range of configuration options to users. Software systems rely on configuration to deliver expected functionalities and services. Errors in configuration settings can result in incorrect system states, including performance degradation, unintended system behaviors and even system failures. As use of open-source software and cloud computing infrastructure surges, misconfiguration are ubiquitous and come with an expensive cost as illustrated in [1–3].

A configuration analysis tool can help evaluate and understand different configurations' impact on performance as well as detect, prevent and troubleshoot configuration errors despite the system complexity. Existing tools [6, 11, 15] use static taint analysis approaches to detect configuration dependencies and misconfiguration. However, the taint analysis component in these tools are usually deeply coupled with their domain-specific analyses and do not provide taint propagation information that is needed by upstream configuration analyses. We have also tried to modify existing static taint analysis frameworks for our needs. One of the most influential work is FlowDroid [4], a tool that uses taint analysis to find sensitive data leaks in Android applications. However, our attempt to apply FlowDroid on configuration analysis resulted in non-deterministic taint flows and the authors of FlowDroid confirmed that it is not planned to be fixed in the near future.

In this project we thus present cFlow, a flow-, field-, and context-sensitive static taint analysis tool for Java bytecode-based cloud systems. cFlow takes as input Jimple intermediate representation of Java bytecode, and tracks how configuration options flow through a software system from their loading points to user-specified sinks such as external API calls. Based on the taint analysis, cFlow starts from source points and generates a directed taint propagation graph. Finally, a path reconstructor module outputs the reconstructed taint propagation paths.

The paper is organized as follows. Section 2 discusses some of the existing static taint analysis tools for software systems and their limitations. Section 3 explains the design and implementation of cFlow's infrastructure in detail. Section 4 shows the evaluation approaches and results. Section 5 reviews cFlow's limitation and points out future improvement direction. Finally, Section 6 concludes the project.

2 Related Work

Static taint analysis is widely applied in vulnerability detection for Android applications [4, 8, 10]. Several recent works have also applied static taint analysis to analyze software configurations for improved software reliability. [11] applies it for easier troubleshooting of configuration errors. [15] utilizes it to detect latent configuration errors. [6] uses it to discover the software configuration dependencies in cloud systems. However, each of these works builds their own static taint analysis infrastructures which are usually deeply coupled with their domain-specific analyses, and their infrastructures do not support recovering the taint propagation path information needed by some upstream configuration analyses like usage inconsistency detection. Our work aims to develop a static taint analysis framework for configuration analysis that provides a clean API to be reused and supports recovering taint propagation path.

We have tried to modify existing static taint analysis frameworks for our needs. One of the most influential work is FlowDroid [4], a tool that uses taint analysis to find sensitive data leaks in Android applications. However, when applying

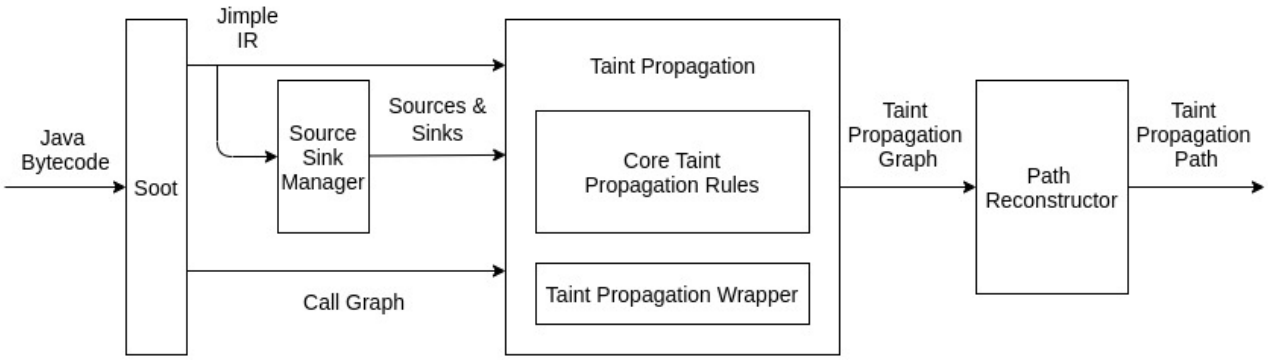

Figure 1. cFlow's architecture.

FlowDroid on configuration analysis, we found that FlowDroid produces non-deterministic taint flows. We observe that approximately 46% of the taint flows appear in one run but cannot be found in the other, which affects about 33% of the configuration parameters. The similar problem is also reported in [5]. The authors of FlowDroid confirmed that it is not planned to be fixed in the near future. Therefore, we decide to build our own framework with incorporated facilities for identifying configuration loading points in modern cloud systems.

3 Approach

In this section, we present the design of cFlow, a flow-, field-, and context-sensitive static taint analysis tool that tracks how configuration option values flow through a software system from their loading points to the user-specified sink points where the values are used (e.g. an external API call).

3.1 Overview

Figure 1 shows an overview of cFlow's architecture. cFlow is based on the Soot [13] Java bytecode optimization framework and can operate on any Java bytecode based software. The software to be analyzed is first compiled into Java bytecode. Soot then transforms the Java bytecode into the Jimple intermediate representation [14] and computes a call graph. cFlow provides a source sink manager component that allows users to specify the patterns for identifying configuration loading points as well as the sink points. cFlow then performs static taint analysis that starts from the configuration loading points and follows its propagation through the software system using the taint propagation engine. Each tainted object is represented using an internal data structure called taint abstraction. Every time there is an information flow from a tainted object t to another object s (e.g. by assignment), the taint propagation engine will create a new taint abstraction for s, and draw a directed edge from the taint abstraction of t towards it. Therefore, the output of the taint prorogation phase is a directed taint propagation graph. Finally, the path reconstructor performs realizable path reconstruction in the graph and outputs taint propagation paths from configuration loading points to the user-specified sink points.

3.2 Field-Sensitive Taint Abstraction

We begin our detailed design discussion with how cFlow represents a tainted object. The biggest challenge here is how to model field accesses, which is essential for our analysis due to the ubiquitous usage of fields in Java bytecode based software. There are two types of fields, instance fields and static fields. Here, we focus our discussion on instance fields.

Techniques for modeling instance field accesses can be distinguished into field-based and field-sensitive approaches. Field-based approaches treat fields as entities independent of the base objects they belong to, while field-sensitive approaches treat fields as dependants of their base objects. A field-based approach will cause over-tainting as the same field of multiple base objects of the same type may be tainted in some base objects while not tainted in some others.

To address this issue, we adopt a field-sensitive approach. However, since there could be arbitrarily deep field access path in the form of $b.f_1.f_2.\cdots.f_n$, the analysis have to keep track of an unbounded field access path to maintain precision, which not only threatens the scalability of the analysis but also complicated the implementation. We trade off precision for scalability and simplicity in implementation and limits the field access path length to 1.

3.3 Taint Policies

The core of cFlow is to perform a static taint analysis that tracks the flow of configuration option values through the software. We present cFlow's taint policy in the following that specifies how a taint is introduced at sources, propagated through the software, and finally reaches sinks.

3.3.1 Taint Sources and Sinks

Taint Sources. Configurations are loaded by reading from an external source and storing option values in program variables. Those program variables at the loading sites serve as the initial taints for the analysis and all the other variables are initialized as untainted. Mature cloud systems have been found to adopt a key-value model for configuration management and have a set of well-defined APIs for loading and setting configurations [7, 12]. For example, Hadoop projects has a specific class `org.apache.hadoop.conf.Configuration` for configuration management, which provides a set of getter/setter methods (e.g. `getInt`, `setInt`) that takes the configuration option name, and loads/sets the option values. cFlow provides an interface for identifying configuration getter/setter methods that allows users to implement for a particular software system. cFlow also pre-installs the interface implementations for Hadoop, HBase and Spark.

Taint Sinks. cFlow allows users to specify how to identify sink statements. It also provides a default implementation that considers all the external APIs as sinks with a few exceptions to filter out the logging- and printing-related APIs. Any taint abstractions that flows into a sink statement will be reported later with the corresponding propagation path.

3.3.2 Taint Propagation

cFlow formulates taint analysis as a forward dataflow analysis and propagates taint abstractions as dataflow facts in the interprocedural control flow graph (ICFG) of the analyzed program. Here, we do not track implicit information flow that takes control flow dependency into consideration and only focus on tracking explicit information flow resulted from sequence of assignments. Taint abstractions are introduced at taint sources, and flow through the program statements forwardly along ICFG edges to simulate the path-insensitive execution of the program. The effect of each program statement on the dataflow facts can be specified with a set of flow functions that compute the taint abstractions flowing out of a statement given a set of taint abstractions flowing into the statement. Further, since taint abstractions are propagated individually, we can express the taint propagation rules in terms of pointwise flow functions as below:

1. **Assignment Flow Function.**
   For an assignment of the form $x.* = y.*$ [1] | Unop($y.*$) | Binop($y.*, z.*$) | Cast($y.*$) | Instanceof($y.*$), where Unop and Binop are arbitrary numerical unary and binary operations:
   - For an incoming taint $T = x.*$, the new taint set is $\emptyset$.
   - For an incoming taint $T = y.* \mid z.*$, the new taint set is $T, x.*$.
   - For all other incoming taints, pass them on.
2. **Call Flow Function.**
   A call flow function receives incoming taints from the previous statement in the caller and outputs taints to the entry node of its callees.
   For a call site of the form $o.m(a_0, a_1, \cdots, a_n)$:
   - For an incoming taint $T = a_i.*$, the outgoing taint set is $p_i.*$, where p_i is the variable that stores the i-th parameter of the method m.
   - For an incoming taint $T = o.*$, the outgoing taint set is $this.*$, where $this$ is the this object in the callee.
   - For an incoming taint $T = S$, the outgoing taint set is T, where S is a static field.
3. **Return Flow Function.**
   A return flow function receives incoming taints from the exit node of the callees and output taints to the next statement in the caller.
   For a call site of the form $o.m(a_0, a_1, \cdots, a_n)$:
   - For an incoming taint $T = p_i.*$ in the callee, the outgoing taint set in the caller is $a_i.*$, where p_i is the variable that stores the i-th parameter of the method m, and p_i is not of a primitive type (e.g. int, char).
   - For an incoming taint $T = this.*$ in the callee, the outgoing taint set in the caller is $o.*$.
   - For an incoming taint $T = S$ in the callee, the outgoing taint set is T in the caller, where S is a static field.

   For a call site with a return value of the form $x = o.m(a_0, a_1, \cdots, a_n)$, there are two additional rules:
   - For a incoming taint $T = x.*$, the outgoing taint set is $\emptyset$.
   - For a incoming taint $T = r.*$ in the callee, the outgoing taint set is $x.*$ in the caller, where r is the return value in the callee.
4. **Call-to-Return Flow Function.**
   A call-to-return flow function receives incoming taints from previous statement in the caller and output taints that are preserved during the method invocation to the next statement in the caller.
   For a call site of the form $o.m(a_0, a_1, \cdots, a_n)$:
   - For an incoming taint $T = a_i.*$, where a_i is of a primitive type, the outgoing taint set is $a_i.*$.
   - For all other incoming taints not processed by the call flow function, pass them on.

For all other statements not processed by the above flow functions, all incoming taints are passed on.

3.4 Scalable Interprocedural Analysis

Naively propagating taint abstractions using the above flow functions does not scale to large software systems. In this section, we describe the techniques used by cFlow for scalable interprocedural analysis.

[1] $.*$ represents a possible field access. In Jimple, at most one side of an assignment statement can be a field access.

3.4.1 Summary-based Analysis

Many methods are called multiple times in the codebase. We adopt a summary-based analysis to cache the analysis results of each method for scalability. To maintain the context-sensitivity of the analysis, separate analysis results for different calling contexts are cached, where the calling contexts are the taint abstractions that are passed from the call site to the entry node of the possible callees. However, a method can be called with many different sets of incoming taint abstractions, each of which needs to be analyzed separately for caching. Since the taint abstractions are propagated individually, it suffices to compute a summary of the method for each possible incoming taint abstractions. For a method call with multiple incoming taint abstractions as the calling context, its summary is equivalent to the union of the method call summaries with each single taint abstraction as the calling context.

3.4.2 Source-Independent Taint Abstraction

An important design decision to make is whether to store the source information in a taint abstraction. If the source information is stored in each taint abstraction, then two taint abstractions that taint the same object but are propagated from different sources are considered to be different and need to be propagated separately. In this way, it can be known at no cost whether there exists a taint propagation path between each pair of sources and sinks. However, it hurts the re-usability of method call summaries.

With the current design, we choose not to store the source information in a taint abstraction. In this way, taint abstractions are independent of any information irrelevant to the actual taint abstraction and redundant taint abstractions can be merged during propagation. However, a graph reachability problem needs to be solved to answer whether there exists a taint propagation path between a pair of source and sink. Nonetheless, since our goal is to output the propagation path information, we need to perform path reconstruction anyway which automatically solves the graph reachability problem.

3.4.3 Library Modeling

Modern software systems typically rely on libraries to provide their functionality. However, directly analyzing the libraries as usual may not be feasible, and precisely modeling the library calls requires much human labor. FlowDroid [4] adopts a shortcut approach that models common library calls using a few heuristic-based rules, which is a special case for the call-to-return rule. We provide a set of predefined rules that handle the common string, collection and math operations as well as a few common third-party library calls used in Hadoop based on FlowDroid's predefined rules. For example, taint the entire map when adding a tainted element to the map. Users can also provide their own rules when needed.

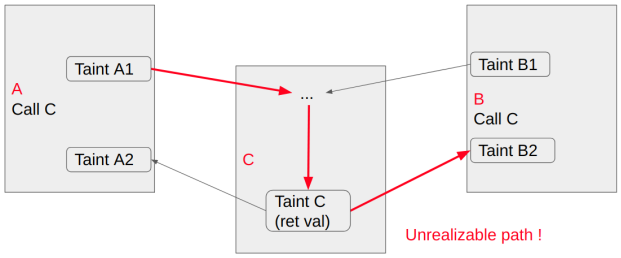

Figure 2. Example of an unrealizable path.

3.4.4 Call Graph Considerations

Dynamic polymorphism is extensively exploited in Java bytecode based programs. Reflection is also heavily used in some programs. To properly handle them, an accurate program call graph is needed. cFlow relies on the SPARK toolkit [9] in Soot to generate the call graph. However, building an accurate call graph will significantly increase the running time and memory consumption of cFlow. cFlow by default uses a simple call graph that does not take into account dynamic polymorphism or reflection, but gives users the option to enable SPARK.

3.5 Path Reconstruction

After the taint propagation is completed, a directed taint propagation graph is generated. To report the taint propagation path from sources to sinks, cFlow traverse the taint propagation graph and reconstructs the propagation path. The main challenge here is how to maintain context-sensitivity during the traversal to avoid reconstructing unrealizable taint propagation paths. Figure 2 shows an example of unrealizable path, where both method A and B call method C, and there is an unrealizable path in red that connects the taint A1 in method A to the taint B2 in method B through taints in method C.

The core observation is that when the control flow returns from the callee back to the caller, it must return to the same call site from which it enters the callee. Therefore, we can keep track of the call sites in a stack when traversing the taint propagation graph. Every time a new callee is entered, we push the current call site into the stack. When the return site of the callee is reached, we only return back to the call site on the top of the stack. We add this simple trick on top of the standard recursive depth first search algorithm and solves the context-sensitivity problem.

Another problem is the cycles introduced by loops and recursive calls in the taint propagation graph. We choose to keep track of all visited taint abstractions for each active method invocation so that we unroll each loop only once. For recursive calls, we keep track of the whole calling stack, and stops traversing when a recursive call is detected.

4 Evaluation

We evaluate the project via addressing the following research questions

- **RQ1**: Can cFlow track taint intra-procedurally and inter-procedurally?
- **RQ2**: Does cFlow support field-sensitivity?
- **RQ3**: What is cFlow's overall performance on real world application such as Hadoop?

In this section, we address each question in detail. Specifically, we apply cFlow on Hadoop v3.3.0 and manually inspect the resulting taint paths. For RQ1 and RQ2, we showcase an example of taint paths and validate the correctness.

4.1 RQ1: Intra- and inter-procedural taint tracking

As mentioned in 3.3.1, cFlow considers `Configuration` object invoking getter methods as configuration loading point and creates a source taint. The code snippet listed in Listing 1 along with the cFlow result in Listing 2 in the appendix illustrates how taint flows intra- and inter-procedurally: Within method `getShutdownTimeout`, `duration`, which corresponds to `l1` in Jimple intermediate representation (IR), is the source. `$b0$` is a copy of `duration` in the if statement, which is in turn tainted. In addition, the taint is included in the output taint set at the return statement.

Then the taint is propagated to variable `shutdownTimeout` based on call-to-return rule in method `shutdownExecutor`. The variable corresponds to `l0` in Jimple IR. The taint tracking terminates prior to `LOG.error` or `awaitTermination` because these functions are external to Hadoop. The path can be extended once we include these functions in the taint wrapper. Other than that, we can observe that cFlow correctly tracks taint both intra- and inter-procedurally.

4.2 RQ2: Field-sensitivity in cFlow

We still use the same example in Listing 1 to show cFlow's capability of tracking tainted field. One of the constructors of `HookEntry` invoked method `shutdownTimeout` and passed the taint as one of the arguments when calling another constructor method, in which `HookEntry` object's field `timeout` is subsequently tainted. This is accurately reflected in the taint paths 2. The path continues from `l1` being returned by method `shutdownTimeout`, and then taints `$l1$`, which was assigned the returned value. `$l1$` is then passed as an argument when invoking another constructor method, which taints the base `HookEntry` object `r0`'s field `timeout`. After returning from the invoked constructor, the field `timeout` of `r0` still remains tracked in the original constructor method. From this selected example, cFlow shows its capability to accurately track field tainting.

4.3 RQ3: Overall performance on cloud software

In this section, we apply cFlow to the entire Hadoop Common codebase and reason if and how the current design and implementation needs to be modified to better achieve completeness and precision. Since there is no existing benchmark, or dataset such as DroidBench [4] to base our evaluation on, we decide to manually inspect the resulting taint paths. Here,

```
public final class ShutdownHookManager {

    private static void shutdownExecutor(final
    ↪  Configuration conf) {
        try {
            EXECUTOR.shutdown();
            long shutdownTimeout =
            ↪  getShutdownTimeout(conf);
            if (!EXECUTOR.awaitTermination(
                shutdownTimeout, TIME_UNIT_DEFAULT)) {
                LOG.error("ShutdownHookManger shutdown
                ↪  forcefully after" + " {}
                ↪  seconds.", shutdownTimeout);
                EXECUTOR.shutdownNow();
            }
            ...
    }

    static long getShutdownTimeout(Configuration conf)
    ↪  {
        long duration = conf.getTimeDuration(
            SERVICE_SHUTDOWN_TIMEOUT,
            SERVICE_SHUTDOWN_TIMEOUT_DEFAULT,
            TIME_UNIT_DEFAULT);
        if (duration < TIMEOUT_MINIMUM) {
            duration = TIMEOUT_MINIMUM;
        }
        return duration;
    }

    static class HookEntry {
        ...
        private final long timeout;

        HookEntry(Runnable hook, int priority) {
            this(hook, priority,
            getShutdownTimeout(new Configuration()),
            TIME_UNIT_DEFAULT);
        }

        HookEntry(Runnable hook, int priority, long
        ↪  timeout, TimeUnit unit) {
            ...
            this.timeout = timeout;
        }
```

Listing 1. Hadoop example

we propagate the taint as far as possible and output all found taint propagation paths not necessarily terminate at sinks. Due to the time limit, we inspected 150 out of 404 sources and their corresponding taint paths. Among the 150 sources, 60 sources have their paths terminated correctly. The other 90 sources encounter incorrect termination, and we categorized the reasons in Table 1. The majority of the early termination is due to methods that are not included in the taint wrapper rule. As mentioned in Section 3.4.3, currently we only

Problem	Reason	Count
Under-tainting	Not included in library modeling	54
	Implicit information flow	19
	Abstract method calls	7
	Part of boolean operation	3
	Can't detect aliasing	3
	Static field access	2
	Config used as default value of another config parameter	1
Over-tainting	HashMap overtainting	1

Table 1. Under/over-tainting in Hadoop Common

consider common string, collection and math operations, and a few third-party library calls. The early termination problem can be alleviated once more relevant methods are included. The second major reason of early termination is that by design, we do not track implicit information flow that takes control flow dependency into consideration to avoid over-tainting. However, this design also ignores boolean operations, since Jimple converts them to if-statements in IR. Given that Hadoop codebase commonly uses boolean operation including ternary operators in assign statements, we might need a more fine-grained handling of implicit information flow. When it comes to polymorphic behaviors such as dynamic binding, we can turn on SPARK option for cFlow to cover abstract method calls. Aliasing detection and static field tracking is discussed in Section 5. We also encounter one corner case where a configuration option value is used as the default value of another configuration parameter. So far we only detected one over-tainting problem caused by a tainted HashMap object, which is consistent with the taint wrapper rule discussed in 3.4.3.

Running cFlow on Hadoop Common without using SPARK takes 29 iterations to converge, and finishes path reconstruction under a minute. With SPARK option turned on, cFlow takes on average 2 minutes and 34 seconds to finish. Overall, cFlow is stable and relatively fast when applied to analyzing Hadoop Common, and shows promising accurate configuration analysis result. With the modification mentioned above in the future patches, cFlow will be able to achieve even better completeness and precision.

5 Limitations

cFlow is neither sound nor complete for the design tradeoffs as well as the inherent limitations of static analysis. It also has some performance issues to be improved for future work.

Implicit information flow cFlow focuses on tracking explicit information flow resulted from sequence of assignments and does not support tracking implicit information flow that takes control flow dependency into consideration, which is likely to cause massive over-tainting.

Static field and nested field access For scalability reasons, cFlow does not support tracking static field accesses and nested instance field accesses with field access path larger than 1. Static fields have global scope and prolonged lifetime, and once tainted, need to be propagated across method boundaries for each method invocation. On the other hand, tracking nested instance field accesses requires tracking a possibly unbounded field access path.

Loops and recursive calls To resolve the cycles introduced by loops and recursive calls in the taint propagation graph that result in theoretically unlimited taint propagation path length, we unroll each loop and recursive call only once. This approach might overlook complex tainting logic and cause under-tainting or over-tainting.

Library modeling Given the nature of heuristics-based library modeling, our modeling may not cover all the library calls. The current coarse heuristic rules applied may lead to either under- or over-tainting for the modelled library calls.

Aliasing, dynamic polymorphism, and reflection These behaviors have long been major challenges for static analysis. We completely ignore the problem of aliasing for now. Although Soot automatically resolves some intra-procedural aliasing problem when transforming the Java bytecode into the Jimple IR, cFlow cannot deal with any non-trivial aliasing problem. We rely on the SPARK call graph toolkit [9] to resolve dynamic polymorphism and reflection, and inherits its limitations. For performance reasons, the cFlow also disables the usage of SPARK by default.

Path reconstruction scalability Our realizable path reconstructor is based on a modified version of recursive depth first search. Although the implementation is straight forward and accurate, it poorly scales to codebase larger than HDFS, and requires improvement.

6 Conclusion

We have presented cFlow, a flow-, field-, and context-sensitive static taint analysis tool that tracks how configuration option values flow through a software system from their loading points to the user-specified sink points where the values are used (e.g. an external API call). It incorporates identification of configuration loading points, supports recovering taint propagation path, and provides a clean API to be integrated in upstream configuration analyses. Our preliminary evaluation shows cFlow can correctly track the flow of configuration options starting from 60 out of 150 inspected sources, and 73 of the 90 incorrect cases are due to uncovered library modeling and implicit information flow, which is not supported by design.

The cFlow tool is publicly available at: https://github.com/xlab-uiuc/cflow. The presentation video can be found at https://illinois.zoom.us/rec/share/LLgsjMslpFHZIkCGpfc-lmNykHf-m-Hm1H9J9YXzTQ0N_JB3ehNmcnPVSNwb3Yvu.Ij3HcUr8U1zOEMne.

References

[1] Linkedin.com inaccessible on thursday because of server misconfiguration. https://www.straitstimes.com/singapore/linkedincom-inaccessible-on-thursday-because-of-server-misconfiguration.

[2] Microsoft: Misconfigured network device caused azure outage. http://www.datacenterknowledge.com/archives/2012/07/28/microsoft-misconfigured-network-device-caused-azure-outage/.

[3] Thanks, amazon: The cloud crash reveals your importance. https://www.pcworld.com/article/226033/thanks_amazon_for_making_possible_much_of_the_internet.html.

[4] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices 49*, 6 (2014), 259–269.

[5] Benz, M., Kristensen, E. K., Luo, L., Borges Jr, N. P., Bodden, E., and Zeller, A. Heaps'n leaks: how heap snapshots improve android taint analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (2020), pp. 1061–1072.

[6] Chen, Q., Wang, T., Legunsen, O., Li, S., and Xu, T. Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems. In *In Proceedings of the 2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)* (Virtual Event, November 2020).

[7] Dong, Z., Andrzejak, A., Lo, D., and Costa, D. Orplocator: Identifying read points of configuration options via static analysis. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)* (2016), IEEE, pp. 185–195.

[8] Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS) 32*, 2 (2014), 1–29.

[9] Lhoták, O., and Hendren, L. Scaling java points-to analysis using s park. In *International Conference on Compiler Construction* (2003), Springer, pp. 153–169.

[10] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. CHEX: statically vetting android apps for component hijacking vulnerabilities. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012* (2012), T. Yu, G. Danezis, and V. D. Gligor, Eds., ACM, pp. 229–240.

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

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

[13] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 2010, pp. 214–224.

[14] Vallee-Rai, R., and Hendren, L. J. Jimple: Simplifying java bytecode for analyses and transformations.

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

Appendix

```
source: l1 in l1 = virtualinvoke r0.<hadoop.conf.Configuration: long
↪  getTimeDuration(java.lang.String,long,java.util.concurrent.TimeUnit)>("hadoop.service.shutdown.timeout",
↪  30L, $r1) in method <hadoop.util.ShutdownHookManager: long getShutdownTimeout(hadoop.conf.Configuration)>
-$b0 in $b0 = l1 cmp 1L in method <hadoop.util.ShutdownHookManager: long
↪  getShutdownTimeout(hadoop.conf.Configuration)>
-l1 in return in method <hadoop.util.ShutdownHookManager: long getShutdownTimeout(hadoop.conf.Configuration)>
--[Return] $l1 in $l1 = staticinvoke <hadoop.util.ShutdownHookManager: long
↪  getShutdownTimeout(hadoop.conf.Configuration)>($r2) in method <hadoop.util.ShutdownHookManager$HookEntry:
↪  void <init>(java.lang.Runnable,int)>
---[Call] $l1 in specialinvoke r0.<hadoop.util.ShutdownHookManager$HookEntry: void
↪  <init>(java.lang.Runnable,int,long,java.util.concurrent.TimeUnit)>(r1, i0, $l1, $r3) in method
↪  <hadoop.util.ShutdownHookManager$HookEntry: void <init>(java.lang.Runnable,int)>
----l1 in identity_stmt in method <hadoop.util.ShutdownHookManager$HookEntry: void
↪  <init>(java.lang.Runnable,int,long,java.util.concurrent.TimeUnit)>
-----r0.<hadoop.util.ShutdownHookManager$HookEntry: long timeout> in
↪  r0.<hadoop.util.ShutdownHookManager$HookEntry: long timeout> = l1 in method
↪  <hadoop.util.ShutdownHookManager$HookEntry: void
↪  <init>(java.lang.Runnable,int,long,java.util.concurrent.TimeUnit)>
------r0.<hadoop.util.ShutdownHookManager$HookEntry: long timeout> in return in method
↪  <hadoop.util.ShutdownHookManager$HookEntry: void
↪  <init>(java.lang.Runnable,int,long,java.util.concurrent.TimeUnit)>
-------[Return] r0.<hadoop.util.ShutdownHookManager$HookEntry: long timeout> in specialinvoke
↪  r0.<hadoop.util.ShutdownHookManager$HookEntry: void
↪  <init>(java.lang.Runnable,int,long,java.util.concurrent.TimeUnit)>(r1, i0, $l1, $r3) in method
↪  <hadoop.util.ShutdownHookManager$HookEntry: void <init>(java.lang.Runnable,int)>
--------r0.<hadoop.util.ShutdownHookManager$HookEntry: long timeout> in return in method
↪  <hadoop.util.ShutdownHookManager$HookEntry: void <init>(java.lang.Runnable,int)>
---------[Return] $r3.<hadoop.util.ShutdownHookManager$HookEntry: long timeout> in specialinvoke
↪  $r3.<hadoop.util.ShutdownHookManager$HookEntry: void <init>(java.lang.Runnable,int)>(r0, i0) in method
↪  <hadoop.util.ShutdownHookManager: void addShutdownHook(java.lang.Runnable,int)>
--[Return] l0 in l0 = staticinvoke <hadoop.util.ShutdownHookManager: long
↪  getShutdownTimeout(hadoop.conf.Configuration)>(r1) in method <hadoop.util.ShutdownHookManager: void
↪  shutdownExecutor(hadoop.conf.Configuration)>
---$r6 in $r6 = staticinvoke <java.lang.Long: java.lang.Long valueOf(long)>(l0) in method
↪  <hadoop.util.ShutdownHookManager: void shutdownExecutor(hadoop.conf.Configuration)>
---l0 in $z0 = interfaceinvoke $r3.<java.util.concurrent.ExecutorService: boolean
↪  awaitTermination(long,java.util.concurrent.TimeUnit)>(l0, $r2) in method <hadoop.util.ShutdownHookManager:
↪  void shutdownExecutor(hadoop.conf.Configuration)>
```

Listing 2. Example cFlow path reconstruction