Grapple: A Graph System for Static Finite-State Property Checking of Large-Scale Systems Code

Zhiqiang Zuo
State Key Lab for Novel Software Technology, Nanjing University
zqzuo@nju.edu.cn

John Thorpe
UCLA
jothor@cs.ucla.edu

Yifei Wang, Qiuhong Pan, Shenming Lu
State Key Lab for Novel Software Technology, Nanjing University

Kai Wang, Guoqing Harry Xu
UCLA
{wangkai,harryxu}@cs.ucla.edu

Linzhang Wang, Xuandong Li
State Key Lab for Novel Software Technology, Nanjing University

Abstract

Many real-world bugs in large-scale systems are related to object state that is supposed to obey a specified finite state machine (FSM). They are triggered when unexpected events occur on objects in certain states, making these objects transition in a way that violates their specifications. Detecting such FSM-related bugs with static analysis is challenging, especially in distributed systems that have large codebases.

This paper presents a single-machine, disk-based graph system, called Grapple, which was designed to conduct precise and scalable checking of finite-state properties for very large codebases. Grapple detects bugs through *context-sensitive, path-sensitive* alias and dataflow analyses, which are both formulated as dynamic transitive-closure computations and automatically parallelized by the system. We propose a novel path constraint encoding/decoding algorithm to attach a path constraint to a graph edge, allowing the graph engine to efficiently recover a path and compute its constraint during the computation. We have implemented Grapple and conducted a comprehensive evaluation over widely deployed distributed systems. Grapple reported a total of 376 warnings, of which only 17 are false positives. Our results also demonstrate the scalability of Grapple: it took between 51 minutes and 33 hours to finish all the analyses on a low-end desktop with 16G memory and 1T SSD space, while the traditional approaches ran out of memory in all cases.

CCS Concepts • **Software and its engineering** → **Software reliability**; *Formal software verification*;

Keywords static analysis, graph processing, bug detection

ACM Reference Format:
Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, and Linzhang Wang, Xuandong Li. 2019. Grapple: A Graph System for Static Finite-State Property Checking of Large-Scale Systems Code. In *Fourteenth EuroSys Conference 2019 (EuroSys'19), March 25–28, 2019, Dresden, Germany.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3302424.3303972

1 Introduction

Large-scale software systems — from operating systems [27], through web browsers [52], to databases [23, 25] and data processing engines [2, 24, 26] — form the backbone of modern computing. As these systems are widely used in a spectrum of areas, ensuring their reliability is critical. Despite ceaseless efforts from the industry and research community to make these systems more reliable, bugs are still regularly seen in all kinds of systems [13, 32, 44, 46, 53, 76].

1.1 Problem

One popular category of bugs is *state related* — a bug manifests after a (finite) sequence of events occur on an object of interest, driving the object into an erroneous state. Common examples include acquired locks that are not released after no longer being used, opened file handlers that are not closed after file accesses are done, or allocated memory regions that are not freed when the contained data is no longer used, or are double freed. Objects involved in such bugs often have a *finite-state machine* (FSM) description of their possible states. Any event that makes the object transition to an unacceptable state indicates a bug. As reported by Chou et al. [13], bugs with FSM properties constitute the dominant category among all bugs studied in (an old version of) Linux kernel.

Over the twenty years since the study [13] was published, thanks to the significant progress made on bug detection and fixing [9, 10, 20, 33, 47, 48, 50, 51, 54, 56, 57, 73, 75], the numbers of bugs in traditional single-machine systems such as the Linux kernel have gone down dramatically [53, 67]. However, as modern computing is entering the "Big Data"

EuroSys'19, March 25–28, 2019, Dresden, Germany

ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303972

```
public void configure(InetSocketAddress addr, int maxcc) throws
      IOException {
   ...
   this.ss = ServerSocketChannel.open();
   ...
   LOG.info("binding to port " + addr);
   ss.socket().bind(addr);
   ss.configureBlocking(false);
   acceptThread = new AcceptThread(ss, addr, selectorThreads);
}
public void reconfigure(InetSocketAddress addr){
   ServerSocketChannel oldSS = ss;
   try {
      this.ss = ServerSocketChannel.open();
      ...
      LOG.info("binding to port " + addr);
      ss.socket().bind(addr);
      ss.configureBlocking(false);
      ...
      oldSS.close();
      acceptThread.wakeupSelector();
      try {
         acceptThread.join();
      } catch (InterruptedException e) {
         ...
      }
      acceptThread = new AcceptThread(ss, addr, selectorThreads);
      acceptThread.start();
   } catch(IOException e) {
      LOG.error("...");
   }
}
```

Figure 1. The code snippet showing a ServerSocketChannel leak in class NIOServerCnxnFactory of ZooKeeper 3.5.0.

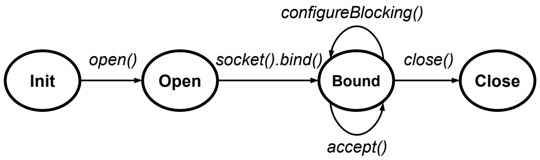


Figure 2. A (partial) finite state machine defining the state transitions of a ServerSocketChannel object.

era, the last few years have seen a proliferation of distributed computing systems developed for various analytical needs. Recent studies [32, 43, 76] on bugs in popular distributed systems show, surprisingly, that FSM-related bugs are still the dominant category. For instance, almost all bugs in error handling and error propagation studied [32, 76] and several "deep bugs" found [43] have finite-state properties.

Example. To illustrate, consider Figure 1, which depicts a simplified code snippet from the class NIOServerCnxnFactory of Apache ZooKeeper [35], a widely deployed open-source system designed to provide centrialized services for distributed applications. The code snippet shows a socket channel leak, which is due to an inappropriate use of the Java NIO class ServerSocketChannel, whose (partial) FSM specification is shown in Figure 2.

During initialization, method configure is first invoked. This method initializes a SocketChannel object this.ss by invoking ServerSocketChannel.open(). Based on the FSM in Figure 2, ss transitions to the state *Open*. Next, socket().bind() is invoked on ss to bind the channel's socket to an address *addr*. The non-blocking mode can be enabled on a bound channel via the method call configureBlocking(false). Once bound, the socket channel can be made ready to listen to connections via a call to accept(). In the example, the object acceptThread is ready to accept connections at the end of method configure.

Method reconfigure is called later to bind to another port. reconfigure first saves the old socket object in oldSS and then overwrites this.ss with a newly created socket channel object. oldSS is not closed until several statements later. This is problematic — upon any exception thrown from these middle statements, the execution would either go to the last catch block in the method or another catch block that handles the exception in a caller of the method. In either case, oldSS would remain open indefinitely due to the loss of reference.

State of the Art. Effectively detecting such bugs is of critical importance for reliable executions of distributed cloud systems that often last for weeks and months for continuous processing of online requests. While dynamic analysis [7, 9, 22, 28, 42, 74] is often the weapon of choice for precisely capturing bugs in large programs, it needs representative workloads to achieve coverage, which is often a daunting task for distributed systems. Static analysis [1, 16, 19, 31, 38, 47, 55, 62], as an alternative, has attracted much attention in the past as it has a potential to find bugs early on during development and yet does not need any input to run the program.

However, finding FSM-related bugs with a static analysis is challenging, especially for distributed systems that have large codebases. This is first because static detection of such bugs requires precise tracking of state transitions for each object of interest. As each such object can flow a long way through many methods and control-flow branches, to accurately report true bugs (as apposed to many false warnings), the static analysis needs to be both *context sensitive* and *path sensitive*: context sensitivity distinguishes analysis results based on calling contexts while path sensitivity tracks control-flow branches and eliminates infeasible flows that can never happen in actual executions.

In the above example, for instance, path sensitivity enables us to track the flow of these socket-channel objects in various control-flow paths and report a bug only in that particular exception-handling path. Without path sensitivity, the checker would either over-approximate that bugs exist for all objects in all paths (because behaviors in different control-flow paths are not distinguishable) or under-approximate that there is no bug (because method close is indeed called at the end). Clearly, neither is acceptable for large-scale software systems where the checker can generate too many warnings to be manually verified by a human developer. Despite a large body of prior work on context-sensitive [41, 63, 70, 72] and path-sensitive [1, 4, 18, 34, 62] analysis algorithms, these algorithms are often sequential, hard to implement, and unable to scale to modern systems such as the Apache Hadoop software stack.

"Systemizing" Static Analysis. Pioneered by Graspan [67], a recent line of work attempted to develop Big Data systems for scaling sophisticated static analysis. This direction is particularly promising due to the following three benefits. First, complicated analysis algorithms get reduced to *simple data computations* that can be automatically parallelized by the underlying system; the concern on efficiency and scalability is shifted from analysis developers' shoulders to the system, which can leverage massive amounts of (CPU, memory, and disk) resources to scale the analysis workloads running on top. Second, the implementation of a client analysis requires only the development of simple user-defined functions (UDFs), enabling regular developers to easily prototype and maintain an analysis without worrying about how to tune its performance.

1.2 Our Contributions

This work is another quest in this direction: we developed a new system, called Grapple, which can perform precise and scalable static checking of finite-state properties for very large codebases. Grapple takes as input (1) a program graph, (2) a set of types of interest (*e.g.*, files, locks, tasks, *etc.*), and (3) a set of FSMs describing the appropriate states and transitions for these types. Grapple tracks the flow of each object of each specified type, in a fully context-sensitive and path-sensitive manner, to identify the possible and feasible sequences of events that may occur on the object.

Grapple checks these sequences against the FSM specification provided by the user — a bug report is generated if there exists any event sequence that can drive an object into an undefined or erroneous state of the specification. Grapple is effective at detecting a broad range of bugs, including source-sink problems [12] (*e.g.*, resource leaks), a subset of distributed-concurrency bugs [44] (*e.g.*, Hadoop-MapReduce concurrency control), exception-handling errors [76] (*e.g.*, missing error handling code), or typestate-related bugs [21, 64] (*e.g.*, inappropriately used file handlers). All of them are common bugs in modern distributed systems.

Problem Statement. To statically detect the sequences of events that occur on an object *o* in an enormous scope, Grapple requires three important pieces of information about *o*. First, we need to identify all the program points (and variables) to which *o* can flow. Because the events we care about are mostly method calls, finding these program points would allow us to extract all possible sequences of calls invoked on *o*. To this end, we need a *dataflow analysis* [58].

Second, many variables can alias along *o*'s flow. The second piece of information we need is the aliasing relationships among these variables, that is, we want to know the set of all variables that can potentially reference *o*. This information cannot be obtained by a dataflow analysis because many of the aliasing relationships are due to complicated heap reads and writes, and cannot be detected by a dataflow analysis that tracks only stack operations. For this piece of information, we need a *pointer/alias analysis* [63].

Third, a static flow of *o* may go through many control-flow branches; some of these branches may conflict with others. For example, if a program has the following two branches: `if(b) a.m(); if(!b) a.n()`, it is clear that the two events *n* and *m* can never occur in the same sequence, since it is impossible for the object to flow into both branches at run time. To effectively filter out infeasible flows, we need *path sensitivity*, which, in turn, requires the modeling and checking of path constraints (*e.g.*, $b \wedge !b$ in the above example).

Putting them all together, to accurately report FSM-related bugs, we need a fully context-sensitive, path-sensitive alias analysis and a fully context-sensitive, path-sensitive dataflow analysis. Context sensitivity is needed to distinguish analysis results based on calling contexts (*i.e.*, results under different contexts do not get merged) for precise bug reports, while path sensitivity is critical for eliminating irrelevant paths to reduce false warnings. Context sensitivity and path sensitivity are both difficult to achieve — the numbers of distinct calling contexts and control-flow paths both grow exponentially with the size of the program, not to mention the multiplicative effects that arise when both need to be done simultaneously. We have not seen any evidence that a fully context-sensitive and path-sensitive analysis could scale to a real-world distributed system such as Hadoop.

The Grapple System. We developed Grapple, the first graph system that can perform fully context-sensitive and path-sensitive finite-state property checking for very large programs. The major research question is how to express these sophisticated analysis algorithms with a simple computation model that can be performed in a *mechanical* manner by the system. To answer this question, we leverage Graspan's insight [67] that alias and dataflow analyses can both be modeled as a *dynamic transitive-closure computation* over a graph representation of the program.

Challenges and Solutions. Grapple uses this same model for the transitive-closure computation to obtain the aliasing and dataflow information. However, as a significant departure from Graspan, Grapple employs a series of new techniques to encode, decode, and solve path constraints during processing. Making these techniques work for large systems requires a fundamentally new system design to overcome two major challenges: (1) how to represent constraints (*i.e.*, large boolean formulas) in a graph efficiently and (2) how to quickly find a path and compute its constraint during the computation.

To overcome the first challenge, we developed a technique that builds an interprocedural control-flow execution tree (ICFET) as an *index engine* using symbolic execution. Instead of letting each edge carry a large boolean path constraint, we encode the path (based upon the ICFET) concisely and losslessly into a sequence of intervals, which can uniquely identify the path (§3). This sequence is associated with each graph edge to reduce the space requirement.

To overcome the second challenge, we developed an efficient encoding/decoding algorithm that can quickly find an interprocedural path from its interval-based encoding, compute and solve its constraint, as well as compose a new constraint if a transitive edge is added (§4).

We chose to implement Grapple as a single-machine, out-of-core system. As a bug-finding tool, Grapple is intended to be used by developers on a daily basis. Hence, a single-machine system is desirable as developers can perform code checking on their own desktops/laptops without needing access to a cluster. However, computing the aforementioned analysis information can be both CPU and memory intensive. If a single machine's memory cannot satisfy the computational need, Grapple leverages support from fast SSDs to store, on disk, part of the input and intermediate data, and efficiently swap data between memory and disk.

Summary of Results. We have implemented Grapple in C++ and made it publicly accessible at https://github.com/grapple-system. We performed an extensive evaluation over four widely deployed distributed systems: Apache Hadoop, HBase, HDFS, and ZooKeeper, all with large codebases. Using Grapple, we have checked four important FSM-related properties including Java I/O, error handlers, lock usage, and socket usage. Grapple reported a total of 376 warnings, of which only 17 are false warnings. Our results also demonstrate the efficiency of Grapple: it took Grapple between 51 minutes and 34 hours to check these complicated properties on a low-end desktop, while traditional approaches could not finish checking any of these programs — they all crashed with out-of-memory errors.

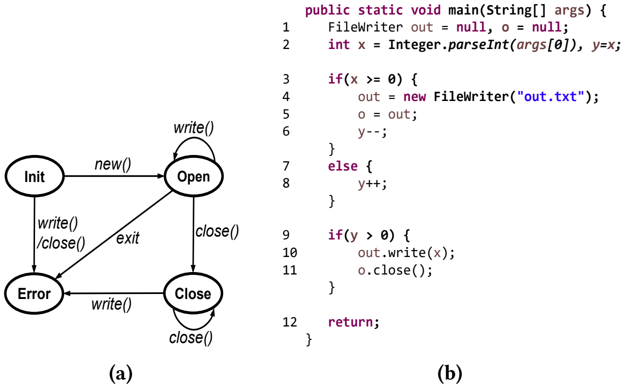
2 Background and Overview

We now provide some background related to static type-state analysis, and an overview of Grapple.

2.1 Background

Figure 3 shows a FSM specification (3a) and a program (3b) that we use as example to describe how our analysis works to find FSM-related bugs. The FSM specifies the possible states during the lifespan of each object of type `java.io.FileWriter` and the transitions among them. At the beginning, the object is at state `Init`. It transitions to *Open* upon the invocation of the constructor, and remains at *Open* for any number of calls to `write` until method `close` is invoked on it, leading it to the *Close* state. Invoking `write` on *Close* drives the object into *Error*.

The program in Figure 3b has two control-flow branches, resulting in four possible paths. Along each path, we extract an event sequence of the object created at Line 4. The first path $(1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12)$ contains an event sequence `new`→ `write` → `close`; in the second path $(1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 9 \rightarrow 12)$, the sequence of events only includes `new`; the third path $(1 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12)$ contains `write`→`close`; and finally, no event occurs on the fourth path $(1 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 12)$ where the false branch is taken for both conditionals.



(a)

```
   public static void main(String[] args) {
1     FileWriter out = null, o = null;
2     int x = Integer.parseInt(args[0]), y=x;

3     if(x >= 0) {
4         out = new FileWriter("out.txt");
5         o = out;
6         y--;
      }
7     else {
8         y++;
      }

9     if(y > 0) {
10        out.write(x);
11        o.close();
      }

12    return;
   }
```

(b)

Figure 3. Example of a finite-state-machine related property and a buggy program. Figure (a) shows a FSM describing the properties objects of type `java.io.FileWriter` and Figure (b) shows a buggy Java program related to the property.

Among these four paths, the third one is infeasible because if $x < 0$, y must be ≤ 0 at Line 9. For each of the other three paths, we check its event sequence against the specification in Figure 3. No problem is found for the first and the fourth paths. For the second path, although the event `new` does not lead the object into the erroneous state, the object would not be in the accept state (*Close*) when the program finishes and hence, a potential bug is reported.

Note that we would have reported an additional bug had the third path been considered, but it would be a false warning as the path can never be reached at run time. This clearly demonstrates the importance of path sensitivity in eliminating false reports.

Analyses Under the Hood. Three analyses work together to extract the set of feasible event sequences as seen above. First, an *alias analysis* [63] analyzes the program to understand, for each object of interest, what variables can reference it. In the example, the pointer/alias analysis determines that the variables *o* and *out* are aliases as they both point to the object created at Line 4. Hence, method calls invoked on *o* and *out* become the events we are interested in.

Second, a *dataflow analysis* [58] runs to identify the states the tracked objects can be in at each program point. The four states in Figure 3 form the set of *dataflow facts*. Based on a *control-flow graph* of the program, the analysis computes a subset of facts for each program point (*i.e.*, before and after each statement) based on the effect of the statement. At each control-flow join point (*e.g.*, before Line 9 or before Line 12), the sets coming from different control-flow branches are combined to form a new set. For example, before Line 9, the two sets of dataflow facts from the true and false branches of the conditional $x \geq 0$ are {*Open*} and {*Init*}, and hence, the new set of states reaching Line 10 is {*Open*, *Init*}. If a loop exists, the computation is done on the loop body repeatedly until a fixed point is reached.

Third, as discussed earlier, path conditions need to be taken into account explicitly in both alias and dataflow analyses to eliminate infeasible aliasing relationships and infeasible dataflows. In our example, `out` and `o` are feasible aliases because the path condition for the statement causing the aliasing (`o = out`) is $x \geq 0$, which is satisfiable. By contrast, the dataflow from the `else` branch at Line 7 into the `if` branch at Line 9 is infeasible due to the unsatisfiable path condition ($x < 0 \wedge y > 0 \wedge y = x + 1$).

Graph Formulation. Both the pointer/alias analysis and dataflow analysis can be formulated as a grammar-guided graph-reachability problem [67]. The program is turned into a graph, with vertices and edges representing program entities of interest. A (context-free) grammar is given by the developer to specify the constraints the analysis has to follow, and during execution the analysis traverses the graph to find paths whose labels match the grammar rules and add transitive edges over such paths.

To illustrate, consider the formulation of Java pointer analysis [63]. The graph is constructed in such a way that vertices represent variables and edges represent assignments. Each edge has a label representing the semantics of the represented assignment. Figure 4(a) shows the four types of statements important to the analysis and their corresponding edge representations.

Type	***Stmt***	***Edge***	
object initialization	$x = new\ O()$	$x \xleftarrow{new} o$	(1)
assignment	$x = y$	$x \xleftarrow{assign} y$	(2)
field store	$x.f = y$	$x \xleftarrow{store[f]} y$	(3)
field load	$x = y.f$	$x \xleftarrow{load[f]} y$	(4)

(a) Statements and edge representations

$$flowsTo ::= new\ (assign \mid store[f]\ alias\ load[f])^* \quad (5)$$

$$alias ::= \overline{flowsTo}\ flowsTo \quad (6)$$

(b) Context-free grammar

Figure 4. Graph edges and grammar for the Java pointer analysis proposed by Sridharan and Bodik [63].

Given the graph representation, the pointer/alias analysis uses the context-free grammar shown in Figure 4(b). The non-terminal *flowsTo* is a relation over $O \times V$ where V is the variable set and O is the set of objects (*i.e.*, allocation sites). Object o may flow to variable v (*i.e.*, v may reference o), if there is a path from o to v in the graph and the sequence of labels on the path can be reduced to *flowsTo* based on the grammar. As can be seen from the first part of rule (5), a path is a *flowsTo* path if it contains a *new* edge and an arbitrary sequence of *assign* edges.

It becomes trickier when field accesses are considered. A flow exists only when an object is written into a heap location and later read out from the same location. In Java, the representation of a heap location $a.f$ has two components: a base object (a) and a field (f). Consider a pair of statements $a.f = b$ and $d = c.g$. An object flow exists from b to d only if f and g are the same field and a and c are aliases. This explains the second part of rule (5) ($store[f]\ alias\ load[f]$), where *alias* is a relation that is recursively defined — if object o can flow to both variable u and v (*i.e.*, $(o, u) \in flowsTo$ and $(o, v) \in flowsTo$), u and v are aliases. Note that we use a bar edge $\overline{flowsTo}$ to represent the reverse of *flowsTo*: if $(o, u) \in flowsTo$, then $(u, o) \in \overline{flowsTo}$.

At the core of this analysis is to compute a transitive closure over the edges whose labels are compliant with the grammar rules. Since the computation can be very expensive for large programs (especially if context sensitivity is considered and callees are *cloned* into callers), Graspan solves the scalability problem by exploiting disk support — it first partitions the large program graph into multiple partitions based on vertex intervals; every computational iteration loads two partitions into memory, checking each pair of consecutive edges to find opportunities to add transitive edges. Newly added edges are flushed into the partitions defined by their source vertices and oversized partitions get dynamically repartitioned to guarantee that the computation never runs out of memory. The dataflow analysis has a different graph representation and grammar; we refer the interested reader to Graspan [67] for details.

Graph Cloning for Context Sensitivity. To achieve context sensitivity, Graspan adopts the *top-down approach* [61] — for each method, its intraprocedural graph representation is *cloned* at each call site that invokes the method. The clone of the callee method is included into the graph of the caller method that contains the call site. The callee's formal parameters and the actual parameters at the call site are connected explicitly with edges. Graph cloning is done in a reverse-topological order — the cloning of a graph not only copies the edges and vertices in one method; it does so for *all* edges and vertices in its (direct and transitive) callees.

Graspan follows the standard treatment [71] to handle recursion — strongly connected components (SCC) are identified on a pre-computed call graph, and then the graphs for methods in each SCC are collapsed into a single graph and treated context insensitively.

An alternative way to achieve context sensitivity, which was used by most existing context-sensitive techniques, is the *bottom-up* approach [61] that computes a *summary* for each method to summarize the interesting behavior of the method. The summary is *applied* at each call site invoking the method. This approach does not clone methods and is hence more scalable than the cloning-based approach, since it doesn't increase memory use significantly. However, the summary-based approach cannot provide complete information about the program and has only limited ability to answer user queries. For example, since it does not explicitly represent calling contexts, it is not able to answer queries such as "what objects does a variable point to under a particular context?". In addition, as path sensitivity is required in property checking, a summary-based approach would additionally need to compute separate summaries for distinct control-flow paths in a method, which is difficult to do for large complicated programs. In fact, we are not aware of any existing technique that can summarize both path-sensitive aliasing and dataflow effects.

The cloning-based approach can cause the size of the graph to grow dramatically and hence the generated graph is often large. However, this is *not* a concern as the out-of-core support in Graspan efficiently handles graphs that are too large to fit into main memory.

2.2 Overview of Grapple

The main contribution of this paper is to add path sensitivity into the alias and dataflow analyses by incorporating path constraints into the graph representation, and constraint solving into dynamic transitive-closure computation.

At a high level, each edge carries a boolean formula, representing the conditions of a set of control-flow branches taken to reach the statement represented by the edge. A transitive edge is added over a graph path if (1) the sequence of the labels on the path matches the grammar and (2) the conjunction of the formulas on the path is satisfiable. Using this approach, both the pointer/alias and dataflow analyses can be made path sensitive, producing precise results for finite-state property checking.

Workflow of Grapple. Grapple has a frontend and a backend. The frontend consists of two compiler-based graph generators, which turn a program into two different graph representations, one for pointer/alias analysis [63] and a second for dataflow analysis [58]. The backend is Grapple's single-machine, disk-based graph engine that leverages fast SSDs to process very large input graphs.

To find FSM-related bugs, Grapple has a three-phase workflow. The first phase performs path-sensitive alias computation, as discussed earlier in this section, over the first program graph (prepared for the alias analysis). The computation produces an expanded graph at the end, with many transitive edges added. Aliasing pairs can be easily computed by enumerating edges with an *alias* label. The second phase conducts path-sensitive dataflow computation over the second program graph (prepared for the dataflow analysis). During this phase, the aliasing results produced by the first phase are held in memory to answer alias queries for dataflow analysis. The final phase extracts state information at each program point computed by the second phase and checks it against all applicable FSMs to detect bugs.

Both the alias and dataflow analyses are context sensitive — analysis solutions are separately maintained for distinct calling contexts, leading to significantly increased precision in final results. For both analyses, context sensitivity is achieved through aggressive method cloning, as discussed earlier at the end of §2.1.

3 Constraint Representation

To enable path sensitivity in graph processing, Grapple needs to embed boolean constraints in the graph representation of the program. A naïve approach is to represent a formula in its original format (with variables, and boolean and arithmetic operators) and associate it with each edge. When a transitive edge is added, a new constraint is generated by combining multiple input constraints into a conjunctive form. The constraint is then stored as edge data for further processing. However, this approach can be prohibitively expensive for processing real-world graphs. A path constraint is a combination of a set of control-flow conditions. Since an interprocedural path can be arbitrarily long, representing this combination in its original form would require a large amount of storage space for each edge in the graph.

3.1 Intraprocedural Path Constraints

To solve this problem, we develop a novel encoding scheme to represent path constraints. This subsection discusses the encoding for intraprocedural path constraints. We will discuss an extension of this encoding shortly to represent interprocedural paths.

Our idea is to create a static *control-flow execution tree (CFET)* using symbolic execution to represent all possible control-flow paths in a method. A control-flow path can be uniquely represented as an interval on the CFET, which can be encoded concisely as a pair of integers. Instead of carrying a boolean formula, each graph edge contains an interval-based encoding of a constraint, which can be used as an index to quickly locate the path and compute its constraint during the graph computation.

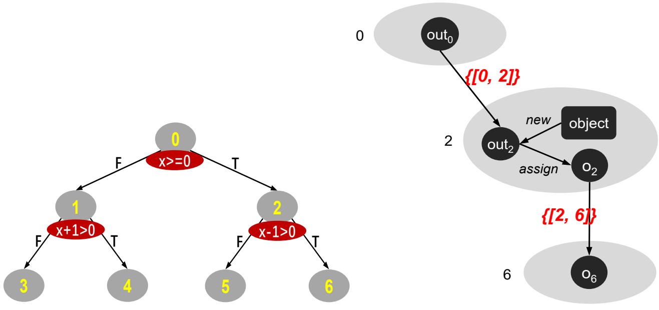
A CFET is a binary tree, where each node represents an "extended basic block", which corresponds to one basic block[1] or multiple blocks fused together along a fall-through edge. A leaf node ends at the procedure exit, while each non-leaf node always ends at a branch conditional. The conditional represents a branching point and hence, the node has two child nodes indicating the true and false branches of the conditional. An example of the CFET will be discussed shortly.

[1] https://en.wikipedia.org/wiki/Basic_block

Each node in the CFET has a unique integer ID, computed using the following algorithm similar to Eytzinger's method[2]:

- The root node has the ID 0;
- For a node with ID = n, the IDs computed for its children are, respectively, $2 * n + 1$ for its false child, and $2 * n + 2$ for its true child.

Despite its simplicity, the algorithm has a number of advantages. First, it is monotonic, providing a guarantee that each node in a CFET is assigned a *unique ID*. Moreover, it is rather easy and inexpensive to compute the ID of a node's parent by simply bit-shifting its own ID. As discussed shortly, these IDs will be needed to perform *online* path recovery and, hence the efficiency of the path computation is critical to the overall system performance.



(a) **(b)**

Figure 5. (a) The CFET for the example code in Figure 3b; (b) the program graph for the alias analysis.

To illustrate, Figure 5a depicts the CFET for Figure 3b. A root node 0 is first created with conditional $x \geq 0$ to represent the entry basic block (*i.e.*, Lines 1 – 3 in Figure 3b). After symbolically evaluating the conditional at Line 3, we create two child nodes 1 and 2 and connect them to node 0 through F and T edges, respectively. In the true branch (*i.e.*, Lines 4, 5, 6, and 9), since y equals $x - 1$, we compute and associate with node 2 a symbolic condition $x - 1 > 0$. For node 1 that represents the false branch (*i.e.*, Lines 8 – 9), y equals $x + 1$ and hence the symbolic condition computed is $x+1 > 0$. Similarly, two child nodes are created for both node 1 and node 2, indicating the different execution branches.

Note that a node in CFET can represent multiple basic blocks. For example, node 4 corresponds to the basic block containing Line 10 and 11, as well as the return statement at Line 12. As another example, node 1 corresponds to the basic block containing Line 4 – 6 as well as the conditional `y > 0` at Line 9, while node 2 includes the basic block containing Line 8 and the same conditional at Line 9.

The CFET representation is similar in spirit to the block-level symbolic execution tree used by symbolic execution engines [11, 37]. In a symbolic execution tree, a path constraint and the symbolic values of the involved variables are maintained at each node. CFET differs from this tree in the following three ways. First, the symbolic execution tree is mostly intraprocedural (to perform function-level symbolic execution); its goal is to make control-flow paths explicit for a symbolic analysis to traverse. By contrast, as discussed shortly, CFETs for individual methods are connected to form an interprocedural CFET, which is used as an index for the graph engine to perform online lookup for constraints of interprocedural paths. Second, each CFET has an integer ID that is uniquely maintained and can be used to easily find its parents. Finally, in CFET, we do not maintain the constraint of a long path at each node. Instead, only the conditional of the local branch represented by the node is stored. A path is retrieved online and its constraint is computed during the graph processing.

Algorithm 1: A high-level description of path decoding.

```
Input: An interval encoding of a path $[ID_{start}, ID_{end}]$
Output: Its path constraint $\mathcal{P}_c$
begin
    $\mathcal{P}_c \leftarrow true$
    $Id_{current} \leftarrow ID_{end}$
    repeat
        $Id_{parent} \leftarrow$ GetParent($ID_{current}$)
        $c \leftarrow$ GetConstraint($ID_{parent}$)
        $\mathcal{P}_c \leftarrow \mathcal{P}_c \wedge c$
        $Id_{current} \leftarrow ID_{parent}$
    until $Id_{current} == ID_{start}$
    return $\mathcal{P}_c$

function GetParent(ID)
    return $ID >> 1$
```

Based on CFET and the node-numbering algorithm, we devise a novel interval-based path encoding/decoding technique to efficiently represent path constraints:

Path encoding: each path in the CFET can be represented as a pair of IDs $[ID_{start}, ID_{end}]$, which denote, respectively, the IDs of the start and end nodes of the path.

Path decoding: given an interval-based encoding of a path, *i.e.*, $[ID_{start}, ID_{end}]$, a unique path can be determined by a backward traversal starting at the end node ID_{end} until the start node ID_{start} is reached. The detailed decoding algorithm is given in Algorithm 1.

To handle loops, we bound the number of loop iterations to avoid the infinite growth of CFET. Particularly, we statically unroll the loop a certain number of times, effectively transforming each loop into a piece of cycle-free code. Given a cycle-free CFET, an ID interval *uniquely* identifies a CFET path — since CFET is a binary tree, each node can only have a single parent and, thus, the backward traversal conducted

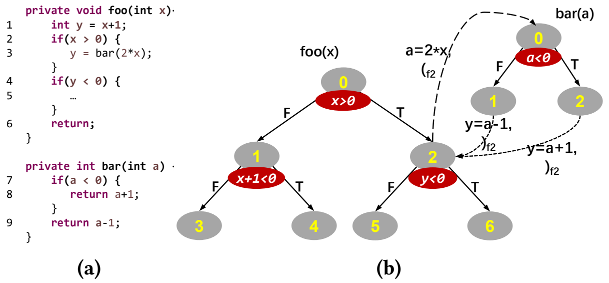
[2] https://en.wikipedia.org/wiki/Genealogical_numbering_systems

in decoding must deterministically reach the start node of the internal at the end.

3.2 Interprocedural Path Constraints

To encode constraints for paths crossing multiple methods, we build interprocedural CFET (ICFET)[3] by augmenting CFET with call/return edges that connect callers and callees. In particular, two extra edges are created at each call site of method M that invokes method N: (1) a *call edge* from the node (say n) in M's CFET representing the basic block that contains the callsite, to the root node of N's CFET; and (2) a *return edge* from each leaf node in N's CFET to node n. Each call/return edge is annotated with two pieces of information: a call site ID and the symbolic equation for parameter passing under which the call/return is made. The call site ID indicates the calling context while the symbolic equation is used to "pass parameters" to compute interprocedural path constraints.

Since a node in an ICFET may have multiple parents (i.e., predecessors in the graph), the interval-based path encoding we used in CFET does not generalize to ICFET. To effectively encode an ICFET path, we extend the interval-based algorithm by representing a path as a list of intervals, each of which represents a path fragment in a method. A path can be decoded by repeatedly executing Algorithm 1 on each fragment. The conjunction of the constraints for all sub-paths thus becomes the constraint for the ICFET path.



```
    private void foo(int x)·
1       int y = x+1;
2       if(x > 0) {
3           y = bar(2*x);
        }
4       if(y < 0) {
5           …
        }
6       return;
    }

    private int bar(int a) ·
7       if(a < 0) {
8           return a+1;
        }
9       return a-1;
    }
```

Figure 6. An example of interprocedural path encoding/decoding: (a) shows a code snippet and (b) shows its corresponding ICFET.

Compared to representing a path constraint in its original form (*i.e.*, a boolean formula), our interval-sequence-based representation is much more concise. Although it still requires variable-sized storage for each edge, this storage is bounded by the depth of method calls on a path. In practice, this depth is often small.

Example. To illustrate, consider the simple code snippet in Figure 6a and its corresponding ICFET in Figure 6b. Given that method `bar` is invoked in the true branch of the conditional at Line 1, a (dashed) call edge is created to connect node 2 of `foo` and the root node 0 of `bar`. Two (dotted) return edges are created to link each leaf (exit) node (*i.e.*, 1 and 2) of `bar` back to node 2 of *foo*. These edges are labeled with the IDs of the call sites as well as the symbolic equations for parameter passing. To distinguish calls from returns, we use a left parenthesis $(_i$ to denote the call edge at the call site i and a right parenthesis $)_i$ to denote the return edge back to the call site i. During an online ICFET traversal, left and right parentheses are matched to guarantee that we end up obtaining a context-sensitive ICFET path.

Following the encoding algorithm, the execution path for the statements 1->2->3->7->8->4->6 is encoded as a sequence of intervals connected by call/return edges: $[foo_0, foo_2] \xrightarrow{a=2*x,(_{f2}} [bar_0, bar_2] \xrightarrow{y=a+1,)_{f2}} [foo_2, foo_5]$. The first interval $[foo_0, foo_2]$ indicates the sub-path from the root node of `foo` to node 2 containing the call site. $[bar_0, bar_2]$ denotes the execution path within method `bar`, while $[foo_2, foo_5]$ represents the last fragment of the graph in `foo` from the branch containing the call site to the leaf node 5.

As for path decoding, we first extract the constraints $x > 0$, $a < 0$ and $\neg(y < 0)$, respectively, for the three path fragments. By computing symbolic values for each variable using symbolic execution and passing them between `foo` and `bar` with call/return edges, we obtain the final path constraint, *i.e.*, $x > 0 \ \& \ a = 2 * x \ \& \ a < 0 \ \& \ y = a + 1 \ \& \ \neg(y < 0)$, which is the conjunction of the constraints for all three sub-paths (*i.e.*, $x > 0$, $a < 0$, and $\neg(y < 0)$) as well as the equations modeling parameter passing (*i.e.*, $a = 2 * x$ and $y = a + 1$). An off-the-shelf SMT solver can be employed next to solve this interprocedural constraint using x as the input variable.

3.3 Constructing ICFET

ICFET is *not* a representation of the input program; rather, it is created as an *index* to make the information of control-flow paths explicit so that a path can be encoded concisely and retrieved efficiently. The program graph to be analyzed is the same as that used in Graspan [67] except that each edge now contains an additional path encoding. The ICFET is constructed simultaneously as the program is transformed into the program graph.

To construct the ICFET, we first create an individual CFET for each method and connect CFETs using call/return edges based on a pre-computed call graph. To compute a CFET from a method, we perform symbolic execution on the method body using the method's formal parameters as symbolic variables. For call sites whose return values are assigned to variables (*e.g.*, `a = m()`), these variables (*e.g.*, *a*) are also used as symbolic variables as their values are not known until the callees are analyzed.

The symbolic execution computes, for each variable, a symbolic value expressed in terms of the symbolic variables

[3] Strictly speaking, ICFET is *not* a tree due to call and return edges, but rather a collection of connected trees. We call it ICFET just for presentation purposes.

(*i.e.*, formal parameters of the method). Upon encountering a control-flow divergence point, we create two child nodes and associate with them the symbolic representation of the conditional guarding the branch. At each call site, we compute the symbolic expression for each argument. At the return statement, we compute the symbolic expression for the variable returned. These symbolic expressions are subsequently used to connect CFETs to form an ICFET.

Discussion. While the ICFET contains information about calling contexts, we do *not* perform method inlining on it — instead of cloning methods aggressively, the CFETs for methods are *not* cloned to form the ICFET; they are *trivially connected using call/return edges at call sites.* As a result, the encoding of each ICFET path has to contain call/return information to uniquely identify an ICFET path and compute a context-sensitive path constraint. By contrast, the program graph to be processed is a fully inlined representation of the program where context sensitivity is explicitly modeled using cloning. Recursion is *not* an issue for the construction of ICFET, while for the program graph, we collapse the methods in each strongly connected component (representing recursively-invoked methods) and treat them in a context-insensitive manner.

Clearly, the program graph is *explicitly context sensitive* while the ICFET is not. This difference in handling is due to an important insight — the explicit context sensitivity enabled by method inlining in the program graph significantly simplifies our computation model because there is no need to let each edge carry call-site information and match calls with returns during computation; any solution computed over the program graph is context sensitive *by nature*. Although inlining can blow up the size of a program graph and make it exceed the memory capacity, the increased memory use can be mitigated by leveraging fast SSDs and developing efficient scheduling algorithms. ICFET, on the other hand, is created primarily for encoding and indexing purposes; it has to be small enough to stay in memory throughout the computation to provide quick path/constraint lookups. Hence, we choose not to perform inlining for ICFET, and context sensitivity is achieved by matching calls and returns (*i.e.*, left and right parentheses) during a path lookup.

4 The Grapple Graph Computation

In this section, we first discuss how the program graph is generated (§4.1). We next present Grapple's graph computation model (§4.2), and in particular, how to add transitive edges and compute their path constraints. We finally discuss Grapple's system design to support this model (§4.3).

4.1 Program Graph Generation

We build the ICFET using symbolic execution as discussed in §3. Next, to turn a program into a graph for Grapple to process, we generate a sub-graph for each basic block, *e.g.*, by using the rules shown in Figure 4. Here, we discuss our graph generation algorithm in the context of alias analysis while a similar algorithm can be easily derived to generate a graph for dataflow analysis. Initially, each edge is labeled with its assignment type and a sequence with only one interval $\{[i, i]\}$, where i is the ID of its containing basic block.

For each variable v that appears in multiple basic blocks (*e.g.*, $b_1, b_2, \dots, b_n$), we create a separate vertex v_i for each such basic block b_i and an artificial assignment edge from v_i to v_j if there exists a path on the ICFET from block b_i to b_j. For instance, Figure 5b shows the program graph for the alias analysis of the example in Figure 3b. At block 2, two edges $object \rightarrow out_2$ and $out_2 \rightarrow o_2$ are created due to the *new* and *assign* statements. Because variable *out* appears in both block 0 and 2, two separate vertices out_0 and out_2 are created, one for block 0 and a second for block 2.

Next, we add an edge labeled *assign* and a sequence with only one interval $\{[0, 2]\}$ to connect out_0 and out_2. Similarly, another *assign* edge is added to link o_2 and o_6. In Figure 5b, we omit the $[i, i]$ intervals for ease of presentation; label *assign* is also not shown on the artificial edges.

Cloning for Context Sensitivity. Given the intraprocedural program graphs, we perform aggressive inlining by cloning the graph of a callee for each of its invoking call sites and including a clone into the graph of each caller. This is done in a *bottom-up manner* based on a pre-computed context-insensitive call graph. Two special types of edges are created during inlining to (1) connect the value flow between a caller and a callee and (2) incorporate the call site information into the graph. A parameter-passing edge connects an actual parameter in the caller to its corresponding formal parameter in a clone of the callee. It is annotated with an *assign* label and a single-element list $\{c_{id}\}$, where c_{id} is the corresponding call edge ID in the ICFET. Note that for each parameter-passing edge here, there exists a corresponding call edge in the ICFET. We simply use the ID of this call edge as an identifier. Similarly, a value-return edge connects a return variable in the clone of the callee to the left-hand-side (LHS) variable at its invoking call site in the caller. It is annotated with an *assign* label and $\{r_{id}\}$, where r_{id} is the ID of its corresponding return edge in the ICFET.

To summarize, before the computation, each edge in the program graph is annotated with its assignment type and a path encoding represented by a sequence containing exactly one element. For each edge connecting two methods, this sequence contains the ID of its corresponding call/return edge in the ICFET. For regular edges inside a method, it contains one single interval, encoding a control-flow path on the ICFET. More complicated sequences are computed upon the addition of transitive edges.

4.2 Constraint-guided Edge Induction

Grapple performs dynamic transitive-closure computation over the program graph based on two kinds of constraints:

(1) the labels of each pair of consecutive edges match the grammar rules and (2) the conjunction of the constraints carried by these edges is satisfiable. Since Graspan [67] already checks constraints of (1), we focus on how to retrieve and solve constraints of (2).

Before the computation starts, Grapple loads the ICFET entirely into memory. The size of the ICFET, even for a large program, is reasonably small (*e.g.*, about 3GB for Hadoop) and can easily fit into the main memory. Similarly to Graspan, Grapple checks a pair of edges at a time — this significantly simplifies the computation model and yet does not lose any generality because any context-free grammar can be transformed into an equivalent grammar such that the right hand side of each production rule contains only two terms [58], similar to the Chomsky normal form.

For a pair of consecutive edges $x \xrightarrow{\langle l_a, i_1 \rangle} y$ and $y \xrightarrow{\langle l_b, i_2 \rangle} z$ in the program graph where l_a and l_b are their assignment types, and i_1 and i_2 are their interval sequences representing two ICFET paths, the key research question to be answered here is how to find their combined path constraint and form a path encoding for a new edge.

Compute Combined Constraints. Finding the two path constraints requires decoding of i_1 and i_2. The decoding process is straightforward if i_1 and i_2 both contain a single interval, as described in Algorithm 1. If i_1 and/or i_2 contain a call/return edge ID or are already sequences of multiple intervals connected by call/return edge IDs, these call/return IDs are suppose to be matched ensuring that the ICFET traversal during decoding passes exactly the same call and return edges. In other words, the interprocedural ICFET path found must match the particular clone of the method where these edges are located in the program graph. Next, the constraints for the two sub-paths involved are merged into a conjunctive form, representing the constraint of the combined path.

Compute a New Encoding. The combined path constraint is sent to an SMT solver to check its satisfiability. If it is satisfiable, a new edge is added from x to z. The two paths represented by i_1 and i_2 are merged into a single path, whose encoding is written into the new edge as a label together with a nonterminal in the context-free grammar to which l_a and l_b can be reduced. Generating the new encoding requires far beyond concatenating the two sequences. We need to consider the following four cases:

1. Neither i_1 nor i_2 contains any call/return edge ID: for example, if i_1 and i_2 are, respectively, {[a, b]} and {[b, c]}, the new encoding is {[a, c]}.
2. i_1 or i_2 contains just one single call/return edge ID: for example, if i_1 and i_2 are, respectively, {[a, b]} and {c_i}, the new encoding is {[a, b], c_i, [0, 0]}, where 0 means the entry basic block of the callee.
3. i_1 and i_2 both contain multiple intervals, and the concatenated sequence of call/return edge IDs in i_1 and i_2 contain *matched* call/return pairs: for example, if i_1 and i_2 are, respectively, {[a, b], c_i, [0, 0]} and {[0, d], r_i, [b, c]}, the new encoding is {[a, c]}.
4. i_1 and i_2 both contain multiple intervals, and the concatenated sequence of call/return edge IDs in i_1 and i_2 do *not* contain matched call/return pairs: for example, if i_1 and i_2 are, respectively, {[a, b], c_i, [0, 0]} and {[0, d], c_j, [0, 0]}, the new encoding is {[a, b], c_i, [0, d], c_j, [0, 0]}.

The cases (1) and (2) are straightforward. Case (3) describes a scenario in which the interprocedural path starts from a caller, goes through a callee, and eventually comes back to the caller. In this case, the intervals between this pair of matched call and return edge IDs (*i.e.*, c_i and r_i) are removed in the new encoding, because the part inside the callee has been "completed" and does not need to be represented. In case (4), both sequences involve a call and, thus, the extended call sequence needs to be modeled in the new encoding.

4.3 System Implementation

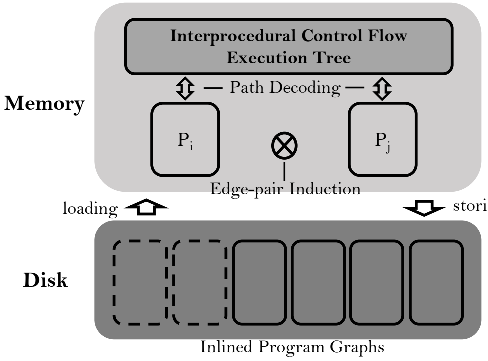


Figure 7. Grapple's computing engine.

We implemented Grapple's graph engine based upon Graspan's edge-pair-centric computation model [67]. Figure 7 illustrates Grapple's architecture.

Graph Engine. There are three phases in program graph processing. First, a preprocessing step partitions the input graph into a set of smaller partitions stored on disk. Each partition is defined by a logical interval of vertices and contains all edges whose source vertices fall in the interval. Edges are sorted by their destination vertices. Since Grapple's computation is performed on a pair of partitions in each iteration, the partitioning is done in a way such that any two partitions, if loaded together, would not exceed the memory capacity.

Next, the edge-pair-centric computation is started to add transitive edges. In each iteration, two edge partitions are loaded into memory. This computation is similar in spirit to table joining in relational algebra, but it differs from table joining in the matching criteria — we need to consider the constraints of both assignment semantics and paths when joining edges. New edges are written into the partitions that contain their source vertices.

Third, a postprocessing step runs to take care of work at the end of each iteraton. For example, we need to repartition oversized partitions to guarantee all partitions are balanced and would not grow to exceed the memory size after new edges are added. A scheduler then picks two other partitions to load into memory for the next iteration. The computation iterates until no new edges can be found.

To implement Grapple, we augmented Graspan's engine by building the ICFET and holding it in memory throughout the execution as read-only data, which can be concurrently accessed by multiple edge-induction threads.

A challenge here is that Grapple's edge data has variable sizes due to the need of carrying interval sequences. Instead of creating separate interval-sequence objects and linking them to each edge via pointers, we inline all intervals and call/return edge IDs explicitly in the storage for each edge. The first byte of the edge storage contains the length of the sequence. This design makes it hard to perform random edge accesses, because the location of an edge cannot be easily computed using its ID. This is *not* a concern in Grapple, however, as most edge accesses are sequential.

Another impact of variable-sized edge data is that it is easy for Grapple to produce unbalanced partitions. To overcome this challenge, we conduct *eager repartitioning* during an iteration — Grapple repartitions an edge partition as soon as we observe that the size of its edge data exceeds a threshold instead of waiting until the end of the iteration to do the repartitioning.

Constraint Memoization. It is easy to see that edges located in the same program scope may share common paths, exhibitting *temporal locality.* Hence, memoizing the results of constraint solving can significantly improve the computation efficiency. To implement memoization, Grapple leverages the *least recently used (LRU)* caching policy. We implemented the LRU cache by maintaining a hash map and using encoded paths as the keys. Before retrieving and solving the actual constraint, Grapple first checks if an encoding has been solved recently. The result is reused if the encoding can be found in the hash map. Least used keys are moved away.

5 Evaluation

Our implementation of Grapple consists of approximately 15.2K lines of code in Java (for the ICFET generator and graph generators) and C++ (for the graph engine). We reused about 1.5K lines of code from Graspan when implementing Grapple's backend. Microsoft's Z3[4] was used as the SMT solver. We have conducted a comprehensive set of experiments to understand Grapple's usefulness and performance. Our evaluation sets out to answer the following questions:

- Q1: Is Grapple useful? (§5.1)
- Q2: How well does Grapple perform? (§5.2)
- Q3: How does Grapple compare against other implementations of finite-state property checkers? (§5.3)

Table 1. Characteristics of subject programs.

Subject	Version	#LoC	Description
ZooKeeper	3.5.0	206K	distributed coordination service
Hadoop	2.7.5	568K	data-processing platform
HDFS	2.0.3	546K	distributed file system
HBase	1.1.6	1.37M	distributed database

We selected four large-scale distributed systems — Apache Hadoop, HDFS, Apache HBase, and Apache ZooKeeper — as our target programs. Table 1 reports the characteristics of each program including the version, the number of lines of code, and a short description. Using Grapple, we implemented four different finite-state property checkers: a Java I/O resource checker, a lock-usage checker , an exception-handler checker (that finds mishandling of the thrown exceptions as proposed by Yuan et al. [76]), and a socket-usage checker. These properties are important in distributed-system implementations since misuse of any of these resources can cause service failures, deadlocks, performance degradation, or even data loss. Their FSMs can be easily understood and specified — it took one developer one day to read the related API information to acquire these FSMs.

The front end contains Java compiler support, implemented using the Soot Java compiler infrastructure [66] (https://github.com/Sable/soot) that translates each program into a graph for alias analysis and a second graph for dataflow analysis. Each of these graph generators has around 1500 lines of Java code. The implementation of the ICFET generator is also based on Soot and has approximately 3200 lines of Java code. All experiments were conducted on a commodity desktop with an Intel Xeon W-2123 4-Core CPU, 16GB memory, and 1T SSD, running Ubuntu 16.04.

5.1 Bugs Found

Table 2. The numbers of bugs reported for the I/O checker, the lock-usage checker, the exception-handling checker, and the socket-usage checker; TP and FP report the numbers of true bugs and false positives, respectively.

Checker	I/O		lock		except.		socket		total	
	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
ZooKeeper	2	0	0	0	59	0	4	0	65	0
Hadoop	0	0	0	0	54	2	0	0	54	2
HDFS	1	1	1	0	43	3	4	1	49	5
HBase	15	2	0	0	176	8	0	0	191	10

Bug Statistics. We ran the four checkers on each program. Table 2 reports the numbers of true bugs and false positive warnings produced by Grapple. For each warning generated, we manually inspected the program code to understand whether it is a true bug or a false warning. Note

[4] https://github.com/Z3Prover/z3

that even true bugs detected by a static analysis may not cause real problems during the execution since they may not be triggered (or even are never triggered due to certain dynamic constraints). The Java I/O resource checker reported 21 warnings, of which 18 are real bugs. All of these bugs are due to the missing of a call to method *close* on certain control-flow paths. The 3 false positives were reported due to the lack of support for the `try-with-resources` construct in Java 8, which automatically closes a stream at the end of the block it guards.

The developers appeared to be very careful with lock usage: our lock checker reported one bug for HDFS where the two methods `lock` and `unlock` are mis-ordered. As for exception handling, Grapple found more than 300 cases where explicitly thrown exceptions never have handlers. According to Yuan et al. [76], these exceptions can lead to various kinds of failures; they need to be handled in a catch block in either their throwing methods or the callers of these methods. The false positives reported here are primarily due to the imprecise control flow graphs generated by Soot for nested try-catch blocks. The socket checker found eight real socket leaks. In HDFS, one false positive was reported because the checker failed to recognize the initialization of a socket object as the object is fetched from a collection.

In total, Grapple found 359 true bugs through 4 checkers on four large-scale distributed systems, with a 4.7% false positive rate.

Example Bugs. Figures 8(a) and 8(b) show two additional bugs found in these systems. For example, the code in Figure 8(a) attempts to establish a connection using a `for` loop that tries at most five times. Each iteration of the loop invokes method `sockConnect`, which may or may not throw an `IOException`. If it does, the `catch` block needs to handle it before trying another time. The major exception handling logic is in the last `else` branch in the `catch` block — a new socket object is created and a timeout is set on it. The problem here is the method call `sockConnect` in the `try` block attempts to advance the state of the socket object from *Init* to *Open*, and eventually to *Ready*. An exception can be thrown when the object is in state *Open* before it transitions to *Ready*. Such an exception would cause those sockets to remain in *Open* although none of them can be used.

Figure 8(b) shows a simple bug due to the missing of exception handling logic. This is actually a known bug, which led to a server performance problem in HDFS. The bug manifests when a large file is uploaded to a `DataNode`. When the block scanner starts, shutting down the `DataNode` progresses extremely slowly. The code snippet shows that, within method `shutdown`, `blockScannerThread` is sent an interrupt (via a call to `interrupt`) and then waiting to complete (`join`). The bug manifests under the following call stack: `DataBlockScanner.run` → ... → `BlockSender.sendBlock` → `BlockSender.sendPacket` → `DataTransferThrottler.throttle`. A simplified code snippet of method `throttle` is also shown in Figure 8(b). When method `throttle` receives the interrupt sent from `shutdown`, it does not execute any handling logic. Consequently, the interrupt gets ignored and the `while` loop continues iterating, leading to a long wait in `shutdown`.

```
void connectToLeader(InetAddress
    addr){
  sock = new Socket();
    ...
  for (int tries = 0; tries < 5;
      tries++) {
    try {
      sockConnect(sock, addr);
      sock.setTcpNoDelay(nodelay);
      break;
    }
    catch (IOException e) {
      if (...) {
        LOG.error("...");
        throw e;
      }
      else if (tries >= 4) {
        LOG.error("...");
        throw e;
      }
      else {
        //The most common case
        LOG.warn("...");
        sock = new Socket();
        ...
      }
    }
    Thread.sleep(1000);
  }
}
```

(a)

```
void shutdown() {
  synchronized (this) {
   if (blockScannerThread != null) {
    blockScannerThread.interrupt();
   }
  }
  if (blockScannerThread != null) {
   try {
    blockScannerThread.join();
   }
   catch (InterruptedException e) {
         ... }
  }
}
void throttle(long numOfBytes) {
  while (...) {
    long now =
          System.currentTimeMillis();
    long curPeriodEnd =
          curPeriodStart + period;
    if (now < curPeriodEnd) {
      try {
       wait(...);
      } catch (InterruptedException
            ignored) {
         //Should handle interrupt
               and stop the loop!
      }
    }
  }
}
```

(b)

Figure 8. Bugs found: (a) a representative socket leak bug in ZooKeeper, and (b) missing error handling in HDFS causing significant performance degradation.

Confirmed Bugs. We have reported all the bugs found to their corresponding bug repositories. As of February 2019, four bugs have been confirmed and others are pending. Among them, one bug was determined as a "blocker"-level bug (*i.e.*, highest severity level), one as "critical"-level (*i.e.*, second highest severity level), and two others as major. We will update the list of the bugs in our GitHub repository [78] as more bugs are confirmed.

Table 3. Grapple's Performance: shown in the columns are the numbers of vertices (#V), the total numbers of edges before computation (#EB), the total numbers of edges after computation (#EA), the preprocessing time (PT), the computation time (CT), and the total running time (TT).

Subject	#V (K)	#EB (K)	#EA (K)	PT	CT	TT
ZooKeeper	2,420	12,860	24,066	47s	01h06m15s	01h07m02s
Hadoop	8,349	17,426	30,206	1m25s	51m49s	53m14s
HDFS	7,610	17,977	29,354	56s	01h53m56s	01h54m52s
HBase	26,090	70,860	125,852	9m51s	33h42m08s	33h51m59s

5.2 Grapple Performance

To understand Grapple's performance, we performed a variety of measurements. Table 3 reports various statistics including the numbers of vertices and edges in the original input program graphs, the numbers of edges at the end of computation, the preprocessing time, the computation time and the end-to-end running time. It took Grapple between 53 minutes and 33 hours to perform checking for these large-scale distributed systems. The running time depends on a number of factors including the code size, the number of methods and paths, and the depth of method calls.

We further broke down an execution into 4 different components — I/O, constraint encoding/decoding, SMT solving, and (in-memory) edge-pair-centric computation — and measured the time spent on each component. As these components can be performed in parallel, we had to sum up the time of each component across all threads, and then calculate the percentage of each component in the total time. Figure 9 depicts the cost breakdown.

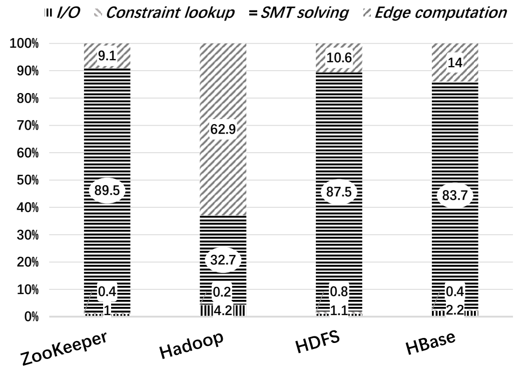


Figure 9. Performance breakdown: for each subject, shown *bottom-up* are the percentages of I/O time, constraint encoding/decoding time, constraint-solving time, and edge computation time.

The majority of the time is spent on SMT solving and edge computation. For Apache Hadoop, edge computation dominates the execution, taking more than 60% of the total time. We found that this is because there is a huge number of consecutive edges in the same basic blocks and all pairs of them are checked for edge induction.

By contrast, for the other programs, SMT solving takes most of the time because (1) there are more cross-block edge pairs than same-block edge pairs and (2) objects flow through many long interprocedural control-flow paths. Both factors contribute to increased complexity in path constraints and hence more time spent on constraint solving. As each edge is embedded with an encoded path, the size of the graph increases significantly, leading to increased I/O costs. The percentage of I/O in Grapple is larger than that in Graspan [67], where I/O takes about 1%. Compared with the other three components, constraint encoding/decoding is relatively inexpensive due to our concise representation and efficient encoding/decoding.

Table 4. Effectiveness of caching: reported are the total numbers of constraints solved during computation, the numbers of cache hits, the hit rates, the constraint-solving times without (TOC) and with caching (TWC), and the savings from caching (*i.e.*, 1-$\frac{TWC}{TOC}$), respectively. The total constraint-solving times reported here were obtained by summing up the times across all processing threads.

	#Const.	#Hits	Rate	TOC(s)	TWC(s)	Saving
ZooKeeper	536579	321385	59.9%	68622	24808	63.7%
Hadoop	494856	385894	78.0%	51715	6883	86.7%
HDFS	1074770	647960	60.3%	131742	38882	70.5%
HBase	22054460	15895429	72.1%	2895804	770077	73.4%

Constraint Caching. We have also evaluated the effectiveness of our constraint caching mechanism. Table 4 reports the results. Because many edges share the same path constraint, most of the hashmap lookups hit the cache. Clearly, caching provides substantial improvement for Grapple's computation efficiency.

5.3 Comparison with other analysis implementations

Traditional Implementations. Our initial goal was to compare Grapple with all the existing path-sensitive finite-state property checkers. However, there does not exist any path-sensitive analysis implementation for Java that is immediately available for comparison purposes. There are a number of them for C, though. For example, Saturn [1] is a path-sensitive finite-state checker that has been used to check various properties for large-scale systems such as the Linux kernel. However, Saturn was implemented more than a decade ago and its source code could not even compile on the Linux we used.

Furthermore, Saturn achieves its scalability using *function summaries* — no function inlining is explicitly made. As described earlier in §1, it *summarizes* the behavior of each function and *applies* the summary of a callee at each of its invoking call sites. Although this approach is scalable, it cannot provide a complete answer to many analysis questions. For example, questions such as whether there is resource leak under a given call stack cannot be answered. In addition, function summarization has many shortcomings such as the inability of precisely modeling heap effects, while none of them exist in a cloning-based technique such as Grapple. Finally, Saturn only performs intraprocedural path-sensitive alias analysis while Grapple enjoys full interprocedural path sensitivity. Another path-sensitive analysis tool, Pinpoint [62], is also designed for C and suffers from similar problems.

Because there was no tool available for comparisons, we implemented a path-sensitive alias analysis ourselves in a traditional (non-systemized) way. In this implementation, we represented the actual constraints using objects and saved them with edges via pointers. A worklist-based algorithm was employed to iteratively check existing edges and add new edges. This implementation could not successfully analyze any program in our set — it ran out of memory quickly after several iterations.

String-based Constraint Representations. Next, to validate the effectiveness of our path encoding/decoding mechanism, we compared Grapple with the systemized implementation that represents constraints as strings and embeds them directly in edges. The results are reported in Table 5.

As the need of space increases dramatically with the string-based implementation, more partitions (and more frequent repartitioning) are required to prevent the computation from running out of memory. This can be seen from the number of partitions in Table 5. For large programs such as HDFS, the number of partitions required by the string-based implementation is around 10× larger than that by Grapple. Since each iteration processes a much smaller amount of data, more iterations and constraint solving are needed for the computation to reach the fixed point. For HBase, the string-based implementation could not even terminate in 200 hours.

Table 5. The comparison with a naïve implementation that encodes constraints into strings; reported are # partitions, # computational iterations, # constraints solved (in thousand), and the total execution time, respectively.

	#Partition		#Iteration		#Constraint (K)		Time	
	Grapple	naive	Grapple	naïve	Grapple	naïve	Grapple	naïve
ZooKeeper	2	18	11	142	215	645	01h07m	03h25m
Hadoop	4	24	24	157	109	488	53m	03h02m
HDFS	4	38	20	546	427	4853	01h54m	22h15m
HBase	20	-	215	-	6159	-	33h51m	>200h

6 Related Work

There is a large body of work related to Grapple. Here we focus our discussion on those that are most closely related.

Static Analysis for Bug Detection. Static program analysis is widely used to detect software defects [19, 47, 55] and security vulnerabilities [8]. Engler et al. [20] used a simple pattern-based analysis to find bugs in the Linux kernel. A decade later, Palix et al. reimplemented the same bug detection tool using Coccinelle [53] and used it to check a later version of the kernel. During the past decade, a great number of commercial checkers, including *e.g.*, Coverity[5, 16], CodeSonar[31], and KlocWork[38], were also developed and used to find bugs in the wild. Most of these checkers are based on simple patterns/rules and/or intraprocedural analysis. As a result, they are inexpensive and scalable, but would likely miss bugs and report many false positives [36, 39, 45].

Earlier work from Hallem et al. [33] proposed a language and an analysis system that allows analysis designers to easily describe a static analysis using an FSM-based abstraction. While Grapple also focuses on FSM-related bugs, Grapple differs from the work [33] in that Grapple provides a precise and scalable solution to finite-state property checking. For example, our property checking is based on a context-sensitive, path-sensitive alias analysis while the system [33] does not even use an alias analysis, which can lead to large numbers of false positives and negatives.

Path-sensitive Analysis. To precisely report true bugs, researchers proposed various path-sensitive analysis techniques. BLAST [34], SLAM [4], and CBMC [15] support path-sensitive program verification for bug detection. Their techniques are all based on an algorithm called *counterexample-driven refinement*, which can significantly improve the scalability of a model checker [14]. Saturn [1] supports intraprocedural path sensitivity. Calysto [3] improves the scalability of a path-sensitive analysis by exploiting program structure information. ESP [17] and Saber [65] use a sparse value-flow representation for programs to improve analysis efficienty. Pinpoint [62] uses a holistic design for a sparse value-flow analysis. Moreover, a number of tools [9, 10, 28, 60] employ symbolic execution [37] to find bugs. However, these techniques are fundamentally limited by the exponential number of paths in a program and hence can only be used to check individual methods or very small programs. Finite-state property checking can also be thought of as typestate-based verification [64]. There exists a body of work [6, 21] that attempts to perform typestate checking in the presence of aliasing. However, no evidence has been shown that these techniques can scale to modern distributed systems.

Graph Systems. There exists a large body of work on distributed [29, 30, 49] and single-machine graph processing [40, 59, 67–69, 77], among which Graspan [67] is the only one designed for static analysis. Grapple is based upon Graspan, but incorporates techniques that can perform full-blown context-sensitive and path-sensitive analysis. None of the other systems were designed for such workloads.

7 Conclusion

This paper presents Grapple, the first single-machine, disk-based graph system for scalable context-sensitive, path-sensitive finite-state property checking. Using Grapple, we have checked a number of important FSM-based properties for four widely-deployed distributed systems. Grapple found numerous true bugs with reasonable checking time.

Acknowledgements

We thank the EuroSys reviewers for their thorough and insightful comments. We are especially grateful to our shepherd Petros Maniatis for his feedback, helping us improve the paper substantially. This work is partially supported by the National Key R&D Program of China (Grant #2017YFA0700604), the National Natural Science Foundation of China (Grants #61802168, #61632015, #61561146394), and the Fundamental Research Funds for the Central Universities of China (#020214380047). The UCLA authors acknowledge the support from National Science Foundation grants CNS-1613023, CNS-1703598, and CNS-1763172, as well as Office of Naval Research grants N00014-16-1-2913 and N00014-18-1-2037.

References

[1] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. 2007. An Overview of the Saturn Project. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '07)*. ACM, New York, NY, USA, 43–48. https://doi.org/10.1145/1251535.1251543

[2] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proc. VLDB Endow.* 7, 14 (Oct. 2014), 1905–1916. https://doi.org/10.14778/2733085.2733096

[3] Domagoj Babic and Alan J. Hu. 2008. Calysto: Scalable and Precise Extended Static Checking. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 211–220. https://doi.org/10.1145/1368088.1368118

[4] Thomas Ball and Sriram K. Rajamani. 2002. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. ACM, New York, NY, USA, 1–3. https://doi.org/10.1145/503272.503274

[5] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[6] Kevin Bierhoff and Jonathan Aldrich. 2007. Modular Typestate Checking of Aliased Objects. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 301–320. https://doi.org/10.1145/1297027.1297050

[7] D. L. Bird and C. U. Munoz. 1983. Automatic generation of random self-checking test cases. *IBM Systems Journal* 22, 3 (1983), 229–245. https://doi.org/10.1147/sj.223.0229

[8] Suhabe Bugrara and Alex Aiken. 2008. Verifying the Safety of User Pointer Dereferences. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP '08)*. IEEE Computer Society, Washington, DC, USA, 325–338. https://doi.org/10.1109/SP.2008.15

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[10] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*. 322–335.

[11] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90.

[12] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical Memory Leak Detection Using Guarded Value-flow Analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 480–491. https://doi.org/10.1145/1250734.1250789

[13] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An Empirical Study of Operating Systems Errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*. ACM, New York, NY, USA, 73–88. https://doi.org/10.1145/502034.502042

[14] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification*, E. Allen Emerson and Aravinda Prasad Sistla (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 154–169.

[15] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. 2004. Predicate Abstraction of ANSI-C Programs Using SAT. *Form. Methods Syst. Des.* 25, 2-3 (Sept. 2004), 105–127. https://doi.org/10.1023/B:FORM.0000040025.89719.f3

[16] Coverity. 2012. The Coverity Code Checker. http://www.coverity.com/.

[17] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-sensitive Program Verification in Polynomial Time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, New York, NY, USA, 57–68. https://doi.org/10.1145/512529.512538

[18] Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, Complete and Scalable Path-sensitive Analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 270–280. https://doi.org/10.1145/1375581.1375615

[19] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. 2000. Checking System Rules Using System-specific, Programmer-written Compiler Extensions. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4 (OSDI'00)*. USENIX Association, Berkeley, CA, USA, Article 1.

[20] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*. 57–72.

[21] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2006. Effective Typestate Verification in the Presence of Aliasing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA '06)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/1146238.1146254

[22] Justin E. Forrester and Barton P. Miller. 2000. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4 (WSS'00)*. USENIX Association, Berkeley, CA, USA, 6–6. http://dl.acm.org/citation.cfm?id=1267102.1267108

[23] Apache Software Foundation. 2018. Apache Cassandra. http://cassandra.apache.org/.

[24] Apache Software Foundation. 2018. Apache Hadoop. http://hadoop.apache.org/.

[25] Apache Software Foundation. 2018. Apache HBase. https://hbase.apache.org/.

[26] Apache Software Foundation. 2018. Apache Spark. https://spark.apache.org/.

[27] Linux Foundation. 2018. Linux. https://www.linux.org/.

[28] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[29] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 17–30. http://dl.acm.org/citation.cfm?id=2387880.2387883

[30] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 599–613. http://dl.acm.org/citation.cfm?id=2685048.2685096

[31] GrammaTech. 2012. The GrammaTech CodeSonar Static Checker. https://www.grammatech.com/products/codesonar.

[32] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. ACM, New York, NY, USA, Article 7, 14 pages. https://doi.org/10.1145/2670979.2670986

[33] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. 2002. A System and Language for Building System-specific, Static Analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. 69–82.

[34] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy Abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. ACM, New York, NY, USA, 58–70. https://doi.org/10.1145/503272.503279

[35] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*. USENIX Association, Berkeley, CA, USA, 11–11. http://dl.acm.org/citation.cfm?id=1855840.1855851

[36] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don'T Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 672–681.

[37] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[38] KlocWork. 2015. The KlocWork Static Checker. https://www.klocwork.com/products-services/klocwork.

[39] Ted Kremenek and Dawson Engler. 2003. Z-ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *Proceedings of the 10th International Conference on Static Analysis (SAS'03)*. Springer-Verlag, Berlin, Heidelberg, 295–315. http://dl.acm.org/citation.cfm?id=1760267.1760289

[40] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 31–46. http://dl.acm.org/citation.cfm?id=2387880.2387884

[41] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-sensitive Points-to Analysis with Heap Cloning Practical for the Real World. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 278–289. https://doi.org/10.1145/1250734.1250766

[42] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 216–226. https://doi.org/10.1145/2594291.2594334

[43] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: Semantic-aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 399–414. http://dl.acm.org/citation.cfm?id=2685048.2685080

[44] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 517–530. https://doi.org/10.1145/2872362.2872374

[45] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead Jr. 2013. Does Bug Prediction Support Human Developers? Findings from a Google Case Study. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 372–381. http://dl.acm.org/citation.cfm?id=2486788.2486838

[46] Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. 2013. A Characteristic Study on Failures of Production Distributed Data-parallel Programs. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 963–972. http://dl.acm.org/citation.cfm?id=2486788.2486921

[47] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*. 289–302.

[48] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID '06)*. 25–33.

[49] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.* 5, 8 (April 2012), 716–727. https://doi.org/10.14778/2212351.2212354

[50] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. 2007. MUVI: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. 103–116.

[51] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, 329–339. https://doi.org/10.1145/1346281.1346323

[52] Mozilla. 2018. Firefox. https://www.mozilla.org/en-US/firefox/.

[53] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. 2011. Faults in Linux: Ten Years Later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 305–318. https://doi.org/10.1145/1950365.1950401

[54] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. 25–36.

[55] William Pugh. 2015. The FindBugs Java Static Checker. http://findbugs.sourceforge.net/.

[56] Feng Qin, Shan Lu, and Yuanyuan Zhou. 2005. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA '05)*. 291–302.

[57] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. 2005. Rx: Treating Bugs As Allergies—a Safe Method to Survive Software Failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. 235–248. https://doi.org/10.1145/1095810.1095833

[58] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462

[59] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 472–488. https://doi.org/10.1145/2517349.2522740

[60] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[61] M. Sharir and A. Pnueli. 1981. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones (Eds.). 189–234.

[62] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 693–706. https://doi.org/10.1145/3192366.3192418

[63] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based Context-sensitive Points-to Analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/1133981.1134027

[64] R. E. Strom and S. Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering* SE-12, 1 (Jan 1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

[65] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static Memory Leak Detection Using Full-sparse Value-flow Analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. ACM, New York, NY, USA, 254–264. https://doi.org/10.1145/2338965.2336784

[66] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java Bytecode Optimization Framework. In *CASCON First Decade High Impact Papers (CASCON '10)*. IBM Corp., Riverton, NJ, USA, 214–224. https://doi.org/10.1145/1925805.1925818

[67] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 389–404. https://doi.org/10.1145/3037697.3037744

[68] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. 2015. GraphQ: Graph Query Processing with Abstraction Refinement—Scalable and Programmable Analytics over Very Large Graphs on a Single PC. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 387–401.

[69] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*.

[70] John Whaley and Monica S. Lam. 2004. Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, New York, NY, USA, 131–144. https://doi.org/10.1145/996841.996859

[71] John Whaley and Monica S. Lam. 2004. Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, 131–144. https://doi.org/10.1145/996841.996859

[72] Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming (Genoa)*. Springer-Verlag, Berlin, Heidelberg, 98–122. https://doi.org/10.1007/978-3-642-03013-0_6

[73] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. 2006. Using Model Checking to Find Serious File System Errors. *ACM Trans. Comput. Syst.* 24, 4 (Nov. 2006), 393–423.

[74] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[75] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. 2011. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. 159–172.

[76] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 249–265. http://dl.acm.org/citation.cfm?id=2685048.2685068

[77] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-scale Graph Processing on a Single Machine Using 2-level Hierarchical Partitioning. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. USENIX Association, Berkeley, CA, USA, 375–386. http://dl.acm.org/citation.cfm?id=2813767.2813795

[78] Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. 2019. The Confirmed Bug Lists. https://drive.google.com/file/d/1Xx6eO2_HMxm2SThz5sU9cT0a2Zx5fSN9/view?usp=sharing.