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

Keywords static analysis, graph processing, bug detection

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.

1 Introduction

Large-scale software systems — from operating systems [30], through web browsers [59], to databases [26, 28] and data processing engines [2, 27, 29] — 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 [15, 36, 50, 52, 60, 69].

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. [15], 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 [15] was published, thanks to the significant progress made on bug detection and fixing [10, 11, 22, 37, 53, 54, 57, 58, 61, 63, 64, 86, 89], the numbers of bugs in traditional single-machine systems such as the Linux kernel have gone down dramatically [60, 79]. However, as modern computing is entering the "Big Data"
“deep bugs” found [49] have finite-state properties.

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 [8, 10, 25, 31, 48, 87] 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, 18, 21, 35, 44, 53, 62, 72], 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 opposed 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 [47, 73, 82, 84] and path-sensitive [1, 4, 20, 39, 72]
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 [79], 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 [14] (e.g., resource leaks), a subset of distributed-concurrency bugs [50] (e.g., Hadoop-MapReduce concurrency control), exception-handling errors [90] (e.g., missing error handling code), or typestate-related bugs [24, 75] (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 [67].

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 [73].

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∧!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 [79] 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.

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\textrightarrow{} write \textrightarrow{} 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\textrightarrow{} 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.

Among these four paths, the third one is infeasible because if \(x < 0\), \(y\) must be \(\leq 0\) 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 [73] 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 [67] 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 ($o = \text{out}$) 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 ($x < 0 \land y > 0 \land y = x + 1$).

**Graph Formulation.** Both the pointer/alias analysis and dataflow analysis can be formulated as a grammar-guided graph-reachability problem [79]. 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 [73]. 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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Stmt</th>
<th>Edge</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>object initialization</td>
<td>$x = \text{new } O()$</td>
<td>$x \xleftarrow{\text{new}} o$</td>
<td>(1)</td>
</tr>
<tr>
<td>assignment</td>
<td>$x = y$</td>
<td>$x \xleftarrow{\text{assign}} y$</td>
<td>(2)</td>
</tr>
<tr>
<td>field store</td>
<td>$x.f = y$</td>
<td>$x \xleftarrow{\text{store}[f]} y$</td>
<td>(3)</td>
</tr>
<tr>
<td>field load</td>
<td>$x = y.f$</td>
<td>$x \xleftarrow{\text{load}[f]} y$</td>
<td>(4)</td>
</tr>
</tbody>
</table>

(a) Statements and edge representations

$\text{flowsTo} ::= \text{new} (\text{assign} \mid \text{store}[f] \mid \text{load}[f])^*$ (5)

$\text{alias} ::= \text{flowsTo} \text{flowsTo}$ (6)

(b) Context-free grammar

**Figure 4.** Graph edges and grammar for the Java pointer analysis proposed by Sridharan and Bodik [73].

Given the graph representation, the pointer/alias analysis uses the context-free grammar shown in Figure 4(b). The non-terminal $\text{flowsTo}$ 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 $\text{flowsTo}$ based on the grammar. As can be seen from the first part of rule (5), a path is a $\text{flowsTo}$ path if it contains a new edge and an arbitrary sequence of $\text{assign}$ 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) ($\text{store}[f] \mid \text{alias load}[f]$), where $\text{alias}$ is a relation that is recursively defined — if object $o$ can flow to both variable $u$ and $v$ (i.e., $(o, u) \in \text{flowsTo}$ and $(o, v) \in \text{flowsTo}$), $u$ and $v$ are aliases. Note that we use a bar edge $\overline{\text{flowsTo}}$ to represent the reverse of $\text{flowsTo}$: if $(o, u) \in \text{flowsTo}$, then $(u, o) \notin \text{flowsTo}$.

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 [79] for details.

**Graph Cloning for Context Sensitivity.** To achieve context sensitivity, Graspan adopts the top-down approach [71] — 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 [83] 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 [71] 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 [73] and a second for dataflow analysis [67]. 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 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.

1https://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\textsuperscript{2}:

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

![Figure 5](image_url)

**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 [12, 43]. 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.

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 $[\text{ID}_{\text{start}}, \text{ID}_{\text{end}}]$, 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., $[\text{ID}_{\text{start}}, \text{ID}_{\text{end}}]$, a unique path can be determined by a backward traversal starting at the end node $\text{ID}_{\text{end}}$ until the start node $\text{ID}_{\text{start}}$ 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 in decoding must deterministically reach the start node of the internal at the end.

---

\textsuperscript{2}https://en.wikipedia.org/wiki/Genealogical_numbering_systems
3.2 Interprocedural Path Constraints

To encode constraints for paths crossing multiple methods, we build interprocedural CFET (ICFET) 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 call site, 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.

![Figure 6](image.png)

**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 \( r \) 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: \([\text{foo}_0, \text{foo}_2] \xrightarrow{a=2x} [\text{bar}_0, \text{bar}_2] \xrightarrow{y=a+1} [\text{foo}_2, \text{foo}_3] \). The first interval \([\text{foo}_0, \text{foo}_2] \) indicates the sub-path from the root node of \( foo \) to node 2 containing the call site. \([\text{bar}_0, \text{bar}_2] \) denotes the execution path within method \( bar \), while \([\text{foo}_2, \text{foo}_3] \) 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., \( x > 0 \land a = 2x \land a < 0 \land y = a + 1 \land \neg(y < 0) \), 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., \( a = 2x \) 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 [79] 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 (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. ICET, 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., \(b_1, b_2, \ldots, b_n\)), 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 \(out_2 \rightarrow out_2\) 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 \(o_1\) and \(o_2\). 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 [79] 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 [67], similar to the Chomsky normal form.

For a pair of consecutive edges \( x \xrightarrow{(l_a, i_1)} y \) and \( y \xrightarrow{(l_b, i_2)} z \) 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 supposed 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 such that \( l_a \) 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 \([c]\), the new encoding is \([a, b], [0, 0]\), 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, \([a, b], c_1, [0, 0]\) and \([0, d], r_1, [b, c]\), 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, \([a, b], c_1, [0, 0]\) and \([0, d], c_1, [0, 0]\), the new encoding is \([a, b], c_1, [0, d], c_1, [0, 0]\).

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., \( c_1 \) and \( r_1 \)) 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

We implemented Grapple’s graph engine based upon Graspan’s edge-pair-centric computation model [79]. 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 iteration. 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 Grasp’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, exhibiting 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 Grasp 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.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Version</th>
<th>#LoC</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZooKeeper</td>
<td>3.5.0</td>
<td>206K</td>
<td>distributed coordination service</td>
</tr>
<tr>
<td>Hadoop</td>
<td>2.7.5</td>
<td>568K</td>
<td>data-processing platform</td>
</tr>
<tr>
<td>HDFS</td>
<td>2.0.3</td>
<td>546K</td>
<td>distributed file system</td>
</tr>
<tr>
<td>HBase</td>
<td>1.1.6</td>
<td>1.37M</td>
<td>distributed database</td>
</tr>
</tbody>
</table>

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. [90]), 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 [78] (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.

<table>
<thead>
<tr>
<th>Checker</th>
<th>I/O lock except</th>
<th>socket</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZooKeeper</td>
<td>2 0 0 59</td>
<td>0 4 0 65</td>
<td>0 5</td>
</tr>
<tr>
<td>Hadoop</td>
<td>0 0 0 54</td>
<td>2 0 0 54</td>
<td>2 2</td>
</tr>
<tr>
<td>HDFS</td>
<td>1 1 1 43</td>
<td>3 4 1 49</td>
<td>5 5</td>
</tr>
<tr>
<td>HBase</td>
<td>15 2 0 176</td>
<td>8 0 0 191</td>
<td>10 10</td>
</tr>
</tbody>
</table>

**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. [90], 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 exception 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.

**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 [94] 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).

<table>
<thead>
<tr>
<th>Subject</th>
<th>#V (K)</th>
<th>#EB (K)</th>
<th>#EA (K)</th>
<th>PT</th>
<th>CT</th>
<th>TT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZooKeeper</td>
<td>2,420</td>
<td>12,860</td>
<td>24,066</td>
<td>47s</td>
<td>0h06m15s</td>
<td>0h07m02s</td>
</tr>
<tr>
<td>Hadoop</td>
<td>8,349</td>
<td>17,426</td>
<td>30,206</td>
<td>1m25s</td>
<td>51m49s</td>
<td>53m14s</td>
</tr>
<tr>
<td>HDFS</td>
<td>7,610</td>
<td>17,977</td>
<td>29,354</td>
<td>56s</td>
<td>0h33m56s</td>
<td>0h45m52s</td>
</tr>
<tr>
<td>HBase</td>
<td>26,090</td>
<td>70,860</td>
<td>125,852</td>
<td>9m51s</td>
<td>33h42m08s</td>
<td>33h51m59s</td>
</tr>
</tbody>
</table>

Figures 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.

→ 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.
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.](image)

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 Grasp spec [79], 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>
<thead>
<tr>
<th></th>
<th>#Const.</th>
<th>#Hits</th>
<th>Rate</th>
<th>TOC(s)</th>
<th>TWC(s)</th>
<th>Saving</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZooKeeper</td>
<td>536579</td>
<td>321385</td>
<td>59.9%</td>
<td>68622</td>
<td>24808</td>
<td>63.7%</td>
</tr>
<tr>
<td>Hadoop</td>
<td>494856</td>
<td>385894</td>
<td>78.0%</td>
<td>51715</td>
<td>6883</td>
<td>86.7%</td>
</tr>
<tr>
<td>HDFS</td>
<td>1074770</td>
<td>647960</td>
<td>60.3%</td>
<td>131742</td>
<td>38882</td>
<td>70.5%</td>
</tr>
<tr>
<td>HBase</td>
<td>22054460</td>
<td>15895429</td>
<td>72.1%</td>
<td>2895804</td>
<td>770077</td>
<td>73.4%</td>
</tr>
</tbody>
</table>

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 [72], 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.

<table>
<thead>
<tr>
<th>Program</th>
<th>#Partition</th>
<th>#Iteration</th>
<th>#Constraint (K)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZooKeeper</td>
<td>2</td>
<td>18</td>
<td>11</td>
<td>142</td>
</tr>
<tr>
<td>Hadoop</td>
<td>4</td>
<td>24</td>
<td>24</td>
<td>157</td>
</tr>
<tr>
<td>HDFS</td>
<td>4</td>
<td>38</td>
<td>20</td>
<td>546</td>
</tr>
<tr>
<td>HBase</td>
<td>20</td>
<td>215</td>
<td>215</td>
<td>6159</td>
</tr>
</tbody>
</table>

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 [21, 53, 62] and security vulnerabilities [9]. Engler et al. [22] 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 [60] 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 [6, 18], CodeSonar [35], and KlocWork [44], 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 [42, 45, 51].

Earlier work from Hallem et al. [37] 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 [37] 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 [37] does not even use an alias analysis, which can lead to large numbers of false positives and negatives. In addition, that work [37] does not perform symbolic execution to compute path constraints and, hence, it can offer only a very limited form of path sensitivity.

**Path-sensitive Analysis.** To precisely report true bugs, researchers proposed various sophisticated path-sensitive analysis techniques. BLAST [39], SLAM [4], and CBMC [17] support path-sensitive alias analysis 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 [16]. Saturn [1] supports intraprocedural path sensitivity. Calysto [3] improves the scalability of a path-sensitive analysis by exploiting program structure information, reducing the analysis work by employing staged analyses and reusing previous results. ESP [19] and Saber [76] use a sparse value-flow representation for programs to improve analysis efficiency. Pinpoint [72] uses a holistic design for a sparse value-flow analysis. However, when it analyzes large-scale codebases, it still needs a large amount of memory; the analysis would fail if the memory requirement cannot be satisfied.

Moreover, a number of tools [10, 11, 31, 70] employ symbolic execution [43] 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 [75]. There exists a body of work [7, 24] 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.

**Dynamic Transitive Closure Computation.** Dynamic transitive closure computation can be solved by context-free-language (CFL) reachability analysis [88]. Reps [66] used CFL-reachability to formulate a series of dataflow problems. A variety of static analyses can be considered as instances of the CFL-reachability problem. Examples are program slicing [68], interprocedural dataflow analysis [23, 67], pointer [73, 84] and alias analysis [85, 91, 92], shape analysis [41, 65], specification inference [5], context translation [13], library summarization [77], or information flow analysis [55]. CFL-reachability has a cubic-time worst case complexity, which is referred to as cubic bottleneck [38]. Researchers have proposed various optimizations to improve its efficiency, including compositional analysis [32], sparse-representation-based analysis [82], and demand-driven analysis [74, 85].
Grapple: A Graph System for Static Finite-State Property Checking

Graph Systems. There exists a large body of work on distributed [33, 34, 56] and single-machine graph processing [46, 69, 79–81, 93], among which Grasp [79] is the only one designed for static analysis. Grapple is based upon Grasp, 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 (#20214380047). 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