Maple-IR: An IR-based Framework for Analyzing Java Bytecode

ecx86[1] and Bibl[2]

***Abstract*— It is crucial to develop efficient control flow analysis algorithms due to their critical role in static analysis. While mature control flow analysis algorithms exist, they do not address the challenges presented by the unique characteristics of Java bytecode, especially in the case of exception ranges. While building the control flow graph (CFG), current optimization frameworks either do not consider exception control flow or do so poorly, resulting in the need for costly and difficult-to-implement downstream patches. In this research, we present an approach that integrates exception range information into the CFG. This is done via a novel block splitting approach to more efficiently handle the implications of exception ranges. As a result, later optimizations and analyses can be performed much more easily. Finally, we introduce our analysis framework, Maple-IR, which incorporates our work.**

I. INTRODUCTION

Java source code is compiled to Java bytecode [1] which the JVM executes. Java bytecode is a popular target langauge for many languages, such as Scala, Clojure, and Groovy. In addition, the popular Android platform makes use of Java bytecode. This makes Java bytecode a very appealing target for static analysis and optimization, and two key analyses for this are data flow analysis [6] and control flow analysis [2]. It is important to design correct, efficient, and reliable methods to model both data flow and control flow because almost all optimizations depend on them. However, Java bytecode has some idiosyncrasies which make such modeling difficult. Two of these issues are stack-based operation and exception control flow. In this paper, we will discuss and attack each of these problems.

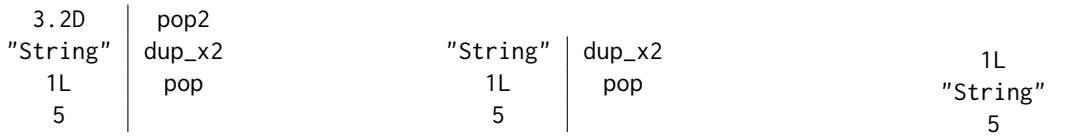
The stack-based nature of Java bytecode poses a problem for data-flow analysis. Bytecode represents method code linearly with postfix (concatenative) notation [5], and the virtual machine operates on a stack and a set of local variables. The stack is modified implicitly through bytecode instructions and only the topmost few stack elements are accessible. Because of the implicit nature of stack access, proper data-flow analysis can prove to be difficult. On the other hand, local variables can be accessed randomly and are zero-indexed and behave much like regular variables. Stack elements and local variables have vastly varying characteristcs, leading to two distinct classes of variables: *stack variables* and *local variables*. When writing an analysis framework for Java bytecode, this distinction must be kept in mind.

To solve this obstacle to data-flow analysis, it is necessary to perform a stack analysis to convert implicit stack references into explicit stack variable references. This is not as simple as it seems as the stack can store values of differing sizes, allowing for a discrepancy between the stack's height and size in elements. Furthermore, instructions accessing the stack may reference different positions within the stack depending on the size of the elements present. An example of this is shown in Figure 1. Moreover, method parameters are passed in using a left-to-right calling convention as local variables, indexed starting from 0. In non-static methods, the first parameter is a reference to the class instance (i.e. the `this` pointer). These problems must be handled with care to ensure proper data-flow analysis. To do so, we use a method based on stack simulation and synthetic copy insertion similar to the one presented by Vallée-Rai et al. [3].

Another major Java-specific challenge for accurate code generation and analysis are exception ranges. Control flow in bytecode is controlled through explicit labeled control flow instructions, implicit natural flow from two sequential labels, and exception flow. Exceptions are represented in the bytecode via the exception table, whose entries are exception ranges, each containing the range's start and end labels, its handler block, and exception type. When an exception occurs in a block contained by the range, the control flow immediately transfers to the handler block, discarding stack variables. Because control flow can transfer from anywhere inside a protected (contained in an exception range) block to the handler block, this poses significant problems in modeling the control flow of the program. In addition, a block may have multiple handlers and handlers themselves may have handlers. Lastly, `finally` blocks have type `null` and are executed after all protected and handler blocks within a `try-catch` chain regardless of the exceptions thrown. Currently, there is no clear way to model control flow while taking into account exception flow. We propose a block splitting-based method to address this issue and to to facilitate easy implementation of classical compiler algorithms without consideration exception flow.

Some optimization frameworks do not model exception flow whatsoever and simply leave it unstructured. However, this pushes the burden of considering exception flow onto the compiler writer, leading to a decrease in performance or even code correctness. Vallée-Rai et al. [3] explored an intermediate representation (IR) approach to analyzing Java bytecode which focused on converting Java bytecode first to Jimple IR before performing optimizations. Then, the Soot optimization framework [4] was built on top of Jimple and other IRs similar to Jimple. This approach solves the first issue addressed earlier, the stack-based nature of the JVM; however, it does not address the second. Namely, Soot

[1]ecx@mapleir.org
[2]bibl@mapleir.org



(a) Pre-stack and code (b) Post-step stack and code (c) Post-stack

Fig. 1: Implicit stack access via instructions.

(a) A simple stack containing only single-size elements. (b) A more complicated stack.

Fig. 2: Explicit representation of stack elements as stack variables.

and Jimple leave exception control flow unstructured; that is to say, exceptions are represented in the "start-end-handler-type" style found in bytecode. We believe this representation to be unwieldy and unconducive to optimization. Therefore, we will structure the exception flow by integrating it into the standard control flow graph (CFG) methods used by compilers and optimizers.

Other other hand, some work has been done to address the problem of exception control flow. For example, Choi et al. assume handler edges can be taken from anywhere within a basic block. They claim that due to large number of possible points for an exception to be thrown, it is better for the protected blocks to be represented in this fashion [13]. As a result, CFGs produced by Choi et al.'s approach do not accurately reflect exception control flow, which forces downstream analysis algorithms to be modified. Our work expands upon ideas first explored by the Krakatau disassembler project [12], namely, properly structuring control graphs to account for exception control flow.

With these two major problems solved, there remain no major impediments for analysis and optimization. Before, an optimizer would need to consider exception flow explicitly, which would lead to a large amount of code specifically for handling the case of exception flow. In addition, if stack elements were not made explicit via stack variables, it would be nearly impossible to construct an IR to optimize to begin with. For these reasons, we propose a method to first convert stack-based bytecode into a variable-based IR, which will then be updated to take into account exception-based control flow properly.

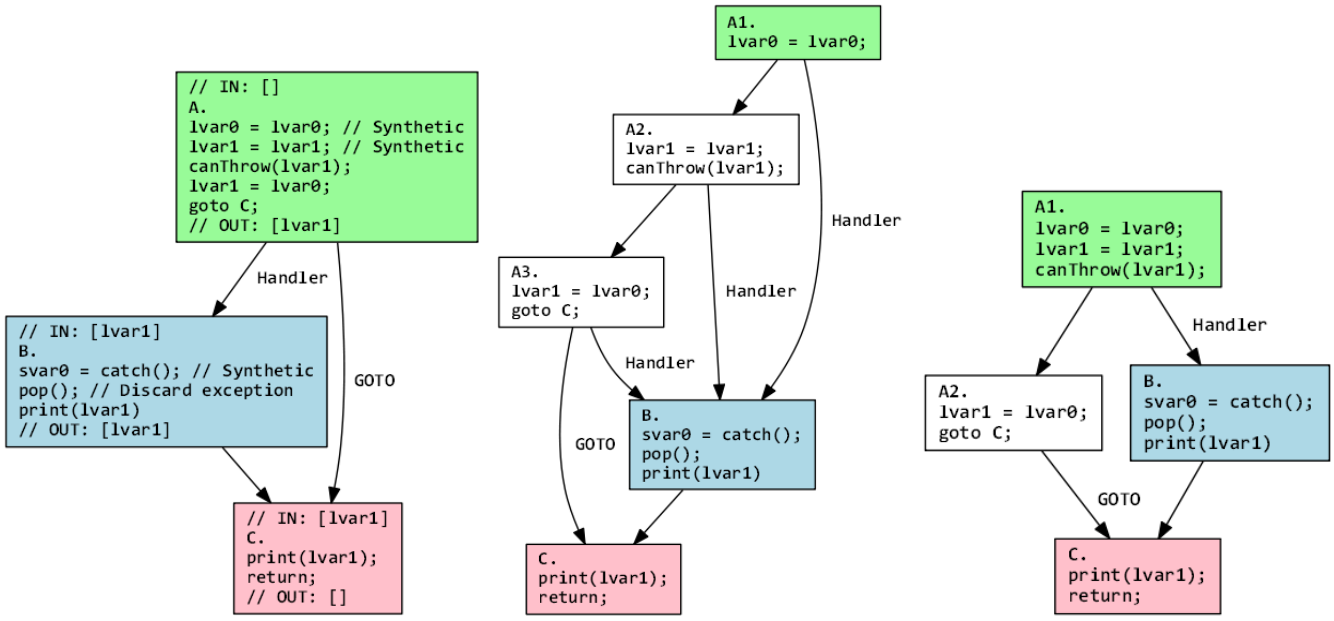
II. METHODS

In Java bytecode, each method is a discrete unit of code, so we consider only the code of a single method. It is necessary to first properly model control flow and data flow before constructing the IR. First, we construct a preliminary control flow graph (CFG) with protected blocks intact. We use a worklist method to build the control flow graph, and we initalize the worklist by adding the entry and handler labels. Then, we process the worklist by iterating through instructions sequentially beginning at each label in the worklist, creating empty basic blocks and control flow graph edges whenever labels or jump instructions are encountered. Newly created empty basic blocks are added to the worklist. As we iterate through the instructions, we also build the intermediate representation (IR). Blocks are also ended when an `athrow` instruction is reached, but no successor edge is created yet; exception flow is not considered until after the IR has been built. Finally, after all blocks are processed and the IR is built, a handler edge to the handler block is added to all blocks within the exception range. After the IR has been built, the CFG is naturalized to incorporate exception flow comprehensively. However, this is not trivial as accurately modeling the control flow while maintaining proper basic block structure involves taking into account when it is possible for an exception to be thrown in the method.

A. Building the IR

To build the IR, we must consider the problem of implicit stack data flow. For every block, we keep track of the incoming stack into the block from preceding blocks. At the end of a block, we copy to the incoming stack of successor blocks the current stack, and add the successor blocks to a worklist. If a successor block's stack is already initialized, we can skip copying to it. This relies on the assumption that the incoming stack of a block is consistent for all of its preceding blocks, which holds if the code is verifiable by the JVM. If code is unverifiable (i.e. invalid), the JVM will refuse to run it. Because the worklist essentially enforces a preorder traversal of the control flow graph, any label we visit will have its incoming stack already computed. The benefit of this method is that the control flow graph and IR can be built in $O(n)$, where n is the number of labels in the code.

Beginning with the initial block, we sequentially process the instructions in each block, simulating the stack by implementing each bytecode instruction in code. Every JVM bytecode instruction must be implemented in code for a comprehensive stack analysis [3]. The stack pointer is also simulated, increasing from 0 as the stack grows. The stack



(a) Before splitting, with comments and liveness. (b) Naïve block splitting. (c) Optimized block splitting.

Fig. 3: An example of block splitting via naïve and optimized methods.

pointer is the sum of the sizes of all of the elements on the stack. Double-size elements (doubles and longs) take up two stack indices, while single-size elements take up one stack index. The initial block is initialized with an empty stack, and handler blocks are initialized with a stack of only the caught exception. We represent local variables simply as variables in the IR. Stack variables are allocated to represent stack elements based on their position on the stack. The index of a stack variable is the stack pointer at the point in which it was defined. The size of a variable, whether local or stack, is the size of the value it stores. Local variables representing method parameters at the start of a method are initialized to themselves by inserting a synthetic copy at the beginning of the entry block. Similarly, a synthetic copy is inserted for the stack variable that represents the caught exception in the start of handler blocks.

B. Naturalizing the Control Flow

After the intermediate representation and preliminary CFG are built, it is now necessary to simplify the graph and take into account exception control flow. First, we remove redundant edges in the CFG by combining successive blocks that have only 1 incoming and outgoing edge. Handler edges can be ignored unless one block has a handler edge while another block does not. This is because currently handler edges can be taken from anywhere in a block, which allows for combining of blocks with identical handler edges. As blocks are merged, we also update the variable definition map.

Now that the CFG is now cleaned such that there are no redundant blocks, exception flow can be handled. At this point, a handler edge can be taken from anywhere within a basic block, which is problematic for analysis. If exception flow is not corrected, later processes that depend on control flow information will need special patches for exception flow, which is undesirable. Therefore, we will correct the CFG so that handler blocks can only be taken from the end of a block. To do this, we will split protected blocks by transferring all statements above the split point to a new block and updating the corresponding CFG edges. To do this, we transfer all predecessor edges into the new block, connect the new block and existing block with an edge, and duplicating all handler edges of the existing block in the new block.

What remains is the problem of deciding at what points to split within a protected block. Because almost every instruction can lead to an exception [1], the most conservative method would be to split after every instruction. However, this is extremely inefficient and leads to huge CFGs. Instead, we consider how exception flow impacts the state of the program incoming to the handler block. Stack variables are discarded as control flows through a handler edge, so we can ignore them. Local variables are preserved, but they may not necessarily be used in the handler. To decide which local variables must be considered, we use liveness information [7]: **the set of variables which must be preserved is the set of variables live into the handler block**. Stack variables are never live into a handler block. To preserve the state of local variables while only allowing control flow over handler edges at block endpoints, it is necessary to split protected blocks before each assignment to a handler-live local variable.

C. Improved block splitting

While this produces a correct, representative IR in which handler edges are well-behaved, the control flow graph is still rather large. This is because before optimizations [6] take place, the IR is verbose and contains a large number of

copies. A large block may contain a large number of copies to local variables, which can lead to a large increase in CFG size, which has negative performance impacts downstream in the compiler toolchain. While it is very difficult to accurately model which IR statements can cause an exception [1], it is possible to optimize the splitting algorithm a bit. First, we introduce the notion of a *simple copy*: **a copy is simple iff it is a copy from a single variable to another, or it is a synthetic copy**. An easy-to-implement but highly powerful optimization is to simply only split if a statement that is not a simple copy has been encountered since the last split point.

In addition to this stricter definition of splitting locations, we also remove unnecessary handler edges after splitting. Consider the handler edge of block A3 in Figure 3b. Clearly, it is not possible for an exception to be thrown in A3. In order to take advantage of situations like this, we must first clearly define what criteria a block must meet in order to have its handler edges pruned. Handler edges can be removed from a block B if all statements in B are simple copies, excluding the control flow statement if present. Because propagation has not occurred yet, complex expressions cannot occur within a control flow statement, so it can be ignored.

Algorithm 1 Optimized block splitting

Data: Set of blocks `blocks`, set of exception ranges `ranges`, live-in sets `liveIn`

```
1:  locals ← empty_map
2:  for range r ∈ ranges do
3:      for b ∈ r.blocks do
4:          locals(b) += liveIn(r.handler)
5:  for block b ∈ blocks do
6:      if locals(b) = ∅ then
7:          continue
8:      i ← 0
9:      checkSplit ← FALSE
10:     for statement s ∈ b do
11:         if checkSplit = TRUE and s is a copy whose
        target ∈ locals(b) then
12:             split(b, i)
13:             i ← 0
14:             checkSplit ← FALSE
15:         else if s is not a simple copy then
16:             checkSplit ← TRUE
17:         i ← i + 1
```

Algorithm 1 performs the optimized block splitting algorithm described. It keeps track of a map of which local variables must be preserved within each protected block, and then processes each protected block. For each block, it proceeds through the statements sequentially, keeping track of whether a statement other than a simple copy has been encountered since the last split. When a split point is detected, it calls the `split` function to split the block at the detected point.

Function 2 takes care of the bookkeeping of the block splitting process. It moves statements from the beginning of

Function 2 `split`(b, i)

Data: Set of edges (source, destination) `edges`, set of exception ranges `ranges`

```
1:  b′ ← new_block
2:  Add b′ to the CFG
3:  Transfer the first i statements from b to b′

4:  for edge (p, b) ∈ edges do
5:      if edge is a handler edge then
6:          r ← ranges.findByEdge(edge)
7:          r.handler = b′
8:      edges -= (p, b)
9:      edges += (p, b′)
10: if the last statement in p is a control flow statement then
11:     Replace targets to b with one to b′.

12: for handler edge (b, s) ∈ edges do
13:     edges += (b′, s)
14: if checkPrune(b)) then
15:     for handler edge (p, b) ∈ edges do
16:         edges -= (p, b)
17: edges += (b′, b)
```

Function 3 `checkPrune`(b)

Output: True if handler edges from b can be removed.

```
1:  for statement s ∈ b do
2:      if s is a control flow statement then
3:          continue
4:      if if s is not a simple copy then
5:          return FALSE
6:  return TRUE
```

the existing block b to a new block b' and updates the CFG edges accordingly. To update the CFG edges, the predecessor edges of b are redirected into b'. If b is an exception range handler, the exception range that corresponds to the handler edge must have its handler updated to b'. Then, if the control flow statement has target b, it must be updated to target b' instead. After predecessor edges are considered, we now consider successor edges. We first copy handler edges from b to b' because following our strict definition of a split point, newly split blocks will always need handler edges. Next, we remove handler edges from b if they are no longer necessary. Finally, we connect b' and b with an edge.

After the CFG naturalization process is complete, exception flow has now been successfully modeled and integrated into the basic block structure of the CFG. In other words, exception flow no longer needs to be treated specially and handler edges can be treated as ordinary edges with no specific semantic differences. At this point, the IR is ready for condensing, optimization, and analysis.

For the practical implementation of the algorithms, we designed, implemented, and tested the algorithms ourselves with no outside assistance. The algorithms were built on top

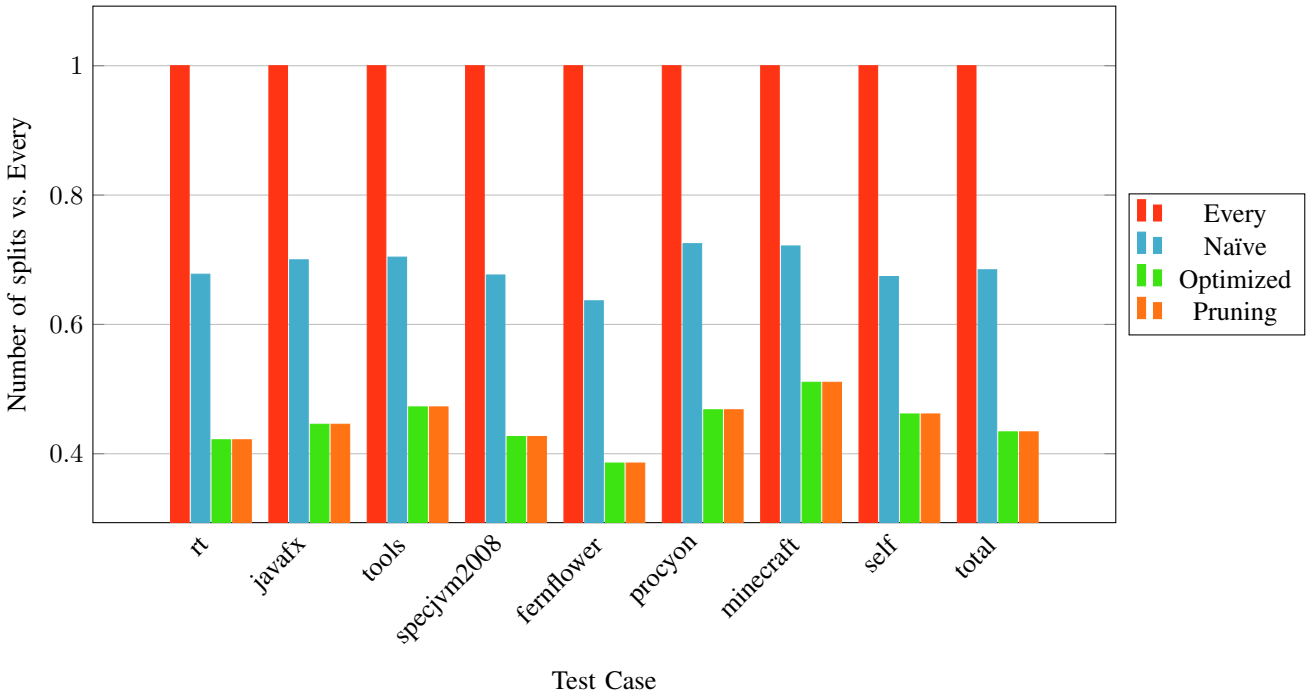


Fig. 4: CFG quality in terms of blocks created due to splitting.

of a Java analysis framework created from scratch leveraging the ObjectWeb ASM bytecode manipulation library [8]. A friend assisted in the initial programming of the analysis framework and otherwise played no part in the project. They contributed neither to the design nor the implementation of the algorithms and methods discussed in this report.

III. Results and Discussion

We performed experiments to determine the efficacy of our method. The experiments were performed on a variety of test cases, which we selected based on their large size and complexity. The test cases specifically were:

- `rt`, `javafx`, `tools` — Standard library and runtime jars packaged with the `jdk1.8.0_102`. Total size: 95.0MB
- `specjvm2008` — The SPECjvm2008 series of benchmarks, including all libraries. Total size: 8.1MB
- `fernflower`, `procyon` — Two popular Java decompiler softwares. The latest versions as of September 2016 were used. Total size: 2.3MB
- `minecraft` — The game client of the popular Java video game Minecraft, version 1.10.2. Total size: 4.2MB (classes only)
- `self` — Our implementation of the optimization framework described, compiled using `javac`. Total size: 1.1MB

First, we evaluated the impact of split point accuracy on CFG quality by implementing four methods of protected block splitting. Each method is defined by the way it decides when it is necessary to split a block. For a baseline, the simple **Every** variant is used.

- **Every**: Split a protected block after every statement.
- **Naïve**: Split a protected block before each copy to a variable live-in to a handler.
- **Optimized**: Split a protected block before each copy to a variable live-in to a handler only if a statement that is not a simple copy has been encountered since the last split point.
- **Pruning**: Same as **Optimized**, in addition to handler pruning as described in Section II-C.

Figure 4 shows the ratio of blocks created due to range splitting compared to the most primitive method, **Every**, for each method of splitting. It can be seen that even the **Naïve** variant, which only considers variable liveness, is significantly better than splitting a protected block after each statement. Overall, this method provides a 31.6% over the **Every** method. **Optimized** provides another 36.6% improvement over the **Naïve** method and a 56.6% improvement over the **Every** method. It can be seen that while **Naïve** is a significant improvement over the **Every** variant, it is still rather inefficient due to the large number of copies within an unoptimized CFG. However, optimization cannot take place before exception flow is modeled, so a large number of copies is unavoidable. Therefore, optimized block splitting is important as it avoids many unnecessary new blocks and handler edges.

Because handler edge pruning does not affect the number of blocks created, **Pruning** and **Optimized** have the same results for this assessment. To achieve a more comprehensive analysis on the value of each block splitting method, let us also consider the number of handler edges created due to block splitting. For this, the CFG before splitting (unsplit) was used as a baseline to compare against. We first record the

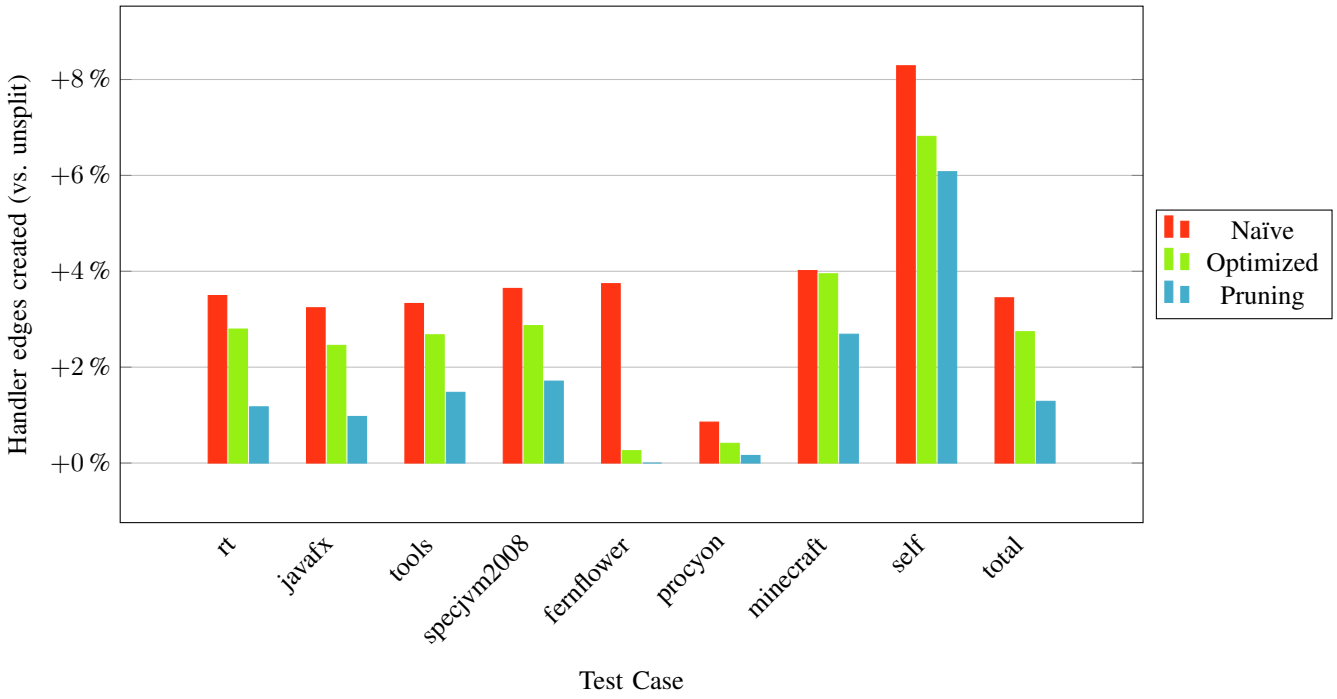


Fig. 5: CFG quality in terms of handler edges created.

number of handler edges present in the CFG before block splitting, and take the difference of handler edges present after block splitting with this pre-split count to find the number of handler edges created as a result of the splitting process.

Figure 5 shows the ratio of number of edges created during block splitting to the number of edges before splitting. The **Every** variant has been excluded due to the enormous number of edges created: while the other methods create edges amounting to less than 10% of the unsplit count, **Every** creates edges up to 400% of the unsplit count. This would severely disproportion the graph axis and thus has been excluded. Ignoring **Every**, it is shown that **Optimized** produces 21% less edges overall than **Naïve**, a noticeable improvement. Furthermore, when pruning is performed on top of optimized block splitting, 63% less copies are produced than when naïve splitting is performed overall. Edge pruning yields a 53% improvement over only optimized block splitting, which is a significant improvement for little cost.

Optimized block splitting is an improvement upon the definition of a block split point, so the decrease in block split count caused is to be expected. Similarly, edge pruning is an improvement upon the definition of when a handler edge is necessary, so a decrease in number of handler edges created is unsurprising. However, it is interesting to note that it also causes a noticeable decrease in number of handler edges created while this is not true for pruning. The explanation for this phenomena is that because optimized block splitting leads to less new blocks created, there will be less new blocks to copy handler edges to. Another interesting point is the relatively low impact of the **Every** method on block split count in contrast to its huge impact on handler edge count. This is due to the fact that while the number of statements in a block grows linearly, because handler edges must be copied each time a block is split, the number of handler edges grow quadratically while the number of block splits grows linearly.

The results of these experiments demonstrate the importance of accurate block splitting and handler edge pruning. Without strict split point definitions and edge pruning, integrating exception flow into the control flow graph and basic block structures would be impractical. With our techniques, it is practical and efficient to include exception flow properly into control flow information while preserving the semantics of the program. This allows for later optimizations and analysis done on the program to be done without having to consider exception flow specially, which reduces the programming effort required for the optimizer. In addition, it is slower to have to consider all of the implications of exception flow on-the-fly such as the state of local variables than to factor them into the control flow data before optimization or further analysis begins.

Furthermore, because many optimization algorithms grow quadratically with CFG size [9] [10], it is important to reduce the complexity of the CFG early in the compiler toolchain to lower costs later on. For example, consider the case of liveness information. Traditionally, liveness information is stored in sets for each basic block and computed through solving data-flow equations [6] [11]. As the number of basic blocks increases, the volume of liveness information stored increases in addition to the running time of algorithms to compute it. If exception flow was incoroprated into the

CFG sloppily (such as in the **Every** method), leaving many unnecessary edges and blocks, algorithm performance would decrease sharply for methods containing `try-catch` ranges.

IV. Conclusion

The idiosyncrasies of the Java bytecode Virtual Machine are summarily its stack-based nature, as well as the control flow due exception handlers. The first issue is easily addressed via symbolic execution, which is a common approach in static analysis. Other projects which implement similar strategies include Hex-Ray's IDA Pro and Binary Ninja, for example. The second problem is more specific to the JVM and more difficult to tackle. To solve it, we introduced an efficient block-splitting and edge-pruning algorithm.

A small downside of block splitting is that its efficiency can vary based on the contents of protected basic blocks. While bytecode produced by typical Java compilers is suitable for block splitting, this may not be true for heavily obfuscated or manually protected bytecode. Nevertheless, in our experience, we have not encountered such bytecode yet. We theorize that protected blocks requiring many split points are rare.

There exist some possible improvements for our method, specifically in deciding whether a statement can cause an exception. In this study, we considered any IR statement that was not a simple copy able to throw an exception. This definition is conservative, and leads to more block split points than actually required. In the future, better modeling of the possible locations for an exception to be thrown could allow for more compact CFGs. Another possible optimization is to only consider exceptions matching the type of the relevant exception range. Furthermore, it may be only necessary to consider certain types of exceptions, such as runtime exceptions and checked exceptions.

For future work, we consider another problem present in Java bytecode, subroutines. Currently, subroutines are present in the JVM as a backwards-compatibility feature for code produced by older Java compilers. In the past, subroutines were used as a way to compile `finally` blocks while avoiding code duplication. Subroutines present many of the same problems as the ones presented by exception ranges: they allow for control flow to jump to arbitrary points within basic blocks, thus causing the basic block to be poorly-structured. These "side" edges can be handled via the block splitting method described in this paper.

ACKNOWLEDGMENT

The authors would like to thank our friends in the Runescape and Minecraft reverse-engineering communities. This work would not have been possible without them.

References

[1] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, "The Java Virtual Machine Specifications," Oracle, 2015.

[2] F. E. Allen, "Control Flow Analysis," in *SIGPLAN '70 Symposium on Compiler Construction*, 1970, pp. 1-19.

[3] R. Vallée-Rai and L. J. Hendren, "Jimple: Simplifying Java Bytecode for Analyses and Transformations," Sable Research Group, McGill University, Montreal, Quebec, 1998.

[4] R. Vallée-Rai, "Soot: A Java Bytecode Optimization Framework," McGill University, Montreal, Quebec, 2000.

[5] J. Purdy, (2012, February 12). *Why Concatenative Programming Matters*. [Blog]. Available: https://evincarofautumn.blogspot.com

[6] A. V . Aho and J. D Ullman, "More About Data-Flow Analysis," in *Principles of Compiler Design*, 2nd ed. Reading, MA: Addison-Wesley Publishing Company, ch. 14, sec. 1-5, pp. 478-491.

[7] K. D. Cooper and L. Torczon, "Data-Flow Analysis" in *Engineering a Compiler*, 2nd ed: Morgan Kaufmann ch. 9, pp. 475-538.

[8] E. Kuleshov, "Using the ASM framework to implement common Java bytecode transformation patterns," in *Aspect-Oriented Software Development*, Vancouver, British Columbia, 2007.

[9] B. Boissinot, A. Darte, F. Rastello, B. Dupont de Dinechin, C. Guillon, "Revisiting Out-of-SSA Translation for Correctness, Code Quality, and Efficiency," in *International Symposium on Code Generation and Optimization (CGO'09*, IEEE/ACM, 2008, pp. 114-125.

[10] V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam, "Translating out of static single assignment form," in *Static Analysis Symposium (SAS'99)*, Italy, 1999, pp. 194 – 204.

[11] B. Boissinot, S. Hack, D. Grund, B. D. de Dinechin, and F. Rastello, "Fast Liveness Checking for SSA-Form Programs," in *International Symposium on Code Generation (CGO'08)*. IEEE/ACM, 2008, pp. 35-44.

[12] R. Grosse, "Krakatau Java Bytecode Tools", 2016, Github repository, https://github.com/Storyyeller/Krakatau

[13] J. Choi, D. Grove, M. Hind, and V. Sarkar, "Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs," in *SIGSOFT Software Engineering Notes*, vol. 24, no. 5, pp. 21-31, 1999.