Type-Separation as a Method for Efficient Bytecode Verification

Philipp Adler and Wolfram Amme

Friedrich-Schiller-Universität Jena, Ernst-Abbe-Platz 1-4, 07743 Jena, Germany,
{phadler,amme}@informatik.uni-jena.de

Abstract. Java Bytecode is applied on a large variety of different architectures. Still, one problem arising is verification of class files. This is especially true for constrained devices like cell phones and embedded systems. For larger programs the verification algorithm is costly in either runtime or used memory. In this paper we present the basic principles of type-separated bytecode and show how its properties can be used to achieve a more efficient verification algorithm. Different techniques which can significantly decrease time and memory consumption during the verification process without creating a substantial overhead in the program file are illustrated and compared.

1 Introduction

No other mobile code format reached such a popularity like Java bytecode. It is used in many different domains although it has some serious flaws. The most two important drawbacks are the space and time consuming verification process and an intermediate representation which is difficult to optimize.

On modern computers the consumption of time and space during the verification process is mostly irrelevant. There is plenty of memory available and computing power increases more and more. Indeed, the possibility of exploiting the time behavior of the verification algorithm to mount an attack on virtual machines is existing [5], but on modern desktop computers this increase in time is barely noticeable. Also, the increasing capabilities of personal computers lead to many improvements to execute bytecode efficiently. Beside the formerly important interpretation there exist just-in-time (JIT) and adaptive compilers which produce native machine code on the fly. The generated code is considerably faster than interpreted execution, but these compilers have an noticeable overhead.

On constrained devices the situation is not as simple as on modern computers. Their computing power and memory have increased, too, but they have far less ressources than their larger counterparts. The verification algorithm must not excess available memory which is just a fraction of the amount usable on desktop systems. Because of the low computing power the time spend during verification is also a multiple of the normal time used. These problems are not completely solved, leading frequently to compromises where one ressource is exchanged with the other. To lower the verification time more space is needed [15]

and vice versa [2, 9]. Another difficult problem is program execution. Since modern JIT and adaptive compilers can often not be employed on constrained devices interpretation is inevitable. Every single instruction takes up precious time, so it would be helpful if optimizations could eliminate unnecessary operations. Using different optimization frameworks like SOOT [17] this is partially feasible. But more complex optimizations must be omitted since they cannot be transferred to the virtual machine in a secure way.

In this paper we present the basic concepts of an alternative intermediate representation which is currently based on Java Bytecode. To obtain an overview of current verification techniques section 2 presents related work. Our intermediate representation is called Type-Separated Bytecode and will be briefly explained in the following section 3. Using type-separation bytecode can be made more robust against manipulation and may be used to transport information for program optimizations via verifiable program annotations. Afterwards, in section 4 we focus on different verification methods which can be used to verify type-separated bytecode. The algorithms including runtime behavior and memory consumption are illustrated. Concluding with section 5 we point to current tasks and future research concerning our project.

2 Related Work

Beside Java Bytecode there exist other techniques to transport mobile code from one environment to another. For example there are syntaxoriented representations like Architecture Neutral Distribution Format (ANDF) [13], Slim-Binaries [7] and SafeTSA [1] as well as proof carrying ones like Proof-Carrying-Code (PCC) [12] and Typed Assembly Language (TAL) [11]. Although these intermediate representations have advantages their use is not widespread, at least not commercial.

One mobile code representation similar to Java Bytecode is the Common Intermediate Language (as part of the Common Language Infrastructure (CLI) [4]). It was developed as a more sophisticated intermediate representation and is mainly used with Microsofts .NET environment. Therefore, although it obtains more and more attention, this language is mainly used on windows-based computers.

Currently, Java Bytecode is still one of the primarily used mobile code representations for transmitting program code. During this transfer the program may be altered because of transmission errors or even manipulation. To guarantee that language properties remain intact each program is checked for integrity using a verification algorithm. Besides structural and logical correctness this algorithm also verifies that type and reference safety is ensured [10].

Each verification of program code usually starts by utilizing a data flow algorithm. The complexity class of the standard algorithm is $O(n^2)$ with n being the number of code instructions. Additionally, a lot of memory is needed to store the states of the stack and register set at each control flow node. Especially for programs with many backward jumps the memory consumption rises quickly into

regions of thousands of bytes [8]. With this overhead the standard algorithm is simply not suited for constrained devices.

Many workarounds were proposed to solve the time and memory issues. Reducing the time factor is mainly achieved using two approaches. The first one deals with simplifying the bytecode, for example on the problem of subroutines [6, 16], while the second one tries to speed up the verification process, e.g. with modified algorithms or by adding special information to the bytecode [3, 15, 14]. These techniques can reduce the amount of verification time, but as a compromise the program size increases due to restructuring code or adding additional information.

Memory consumption during verification is also targeted by various researchers. Since this problem emerges particularly on constrained devices many solutions are mainly for these environments although they can be used for other systems, too. To reduce memory usage one can use special requirements or constraints for the program code, leading to simpler code with less control points [9]. Another technique modifies the algorithm and uses multiple verification passes, each solving subproblems [2]. While conserving main memory these techniques often lead to increases in code size or verification time.

3 Type-Separated Bytecode

Java Bytecode belongs to the stack based intermediate representations. It has one stack and register set which are untyped. Hence, the stack and every register can contain any in the program existing type. To guarantee type safety a verification algorithm has to check that all types are used correctly. Further, the algorithm has to remember the type of each value on the stack or in the registers, inferring types if necessary. This process is mostly time and memory consuming, especially if different stack and register states must be merged.

Type inference during abstract interpretation can be avoided if the idea of type-separation is used. Amme et al. [1] present a mobile code representation which uses type-separation but is based on static single assignment form. Hence, we apply this technique to normal bytecode. The basic idea behind type-separation is to separate every single type from all different types. Thus, instead of having one untyped stack we have as many stacks as types exist, each associated with its appropriate type. To work with these stacks we must also have typed register sets, again one for each type. Each instruction knows per se where the operands of an operation must come from and the result should go to. Because of this property it is not possible to undermine the type system, thus the type-separated bytecode becomes type secure.

In figure 1 a piece of a Java program is shown with its corresponding type-separated bytecode. To visualize type separation the verification process is displayed for each bytecode line. In this program four types are used: *int*, *String*, *Object* and *[Object* (an array of *Object*). Two of the types have associated register sets, namely type *int* has two registers int_0 and int_1 whereas type *[Object*

Java Program Code

```
Object list[] = new Object[10];
for (int i = 0; i < list.length; i++)
  list[i] = new String();
```

Type-Separated Bytecode

Bytecode	Stacks (Height)				Initialized Registers		
	int	String	Object	[Object	int_0	int_1	$[Object_0$
[52] BIPUSH 10	1	0	0	0	—	—	—
[54] NEWARRAY [Object	0	0	0	1	—	—	—
[56] REFSTORE_0 [Object	0	0	0	0	—	—	i
[58] ICONST_0	1	0	0	0	—	—	i
[59] ISTORE_0	0	0	0	0	i	—	i
[60] REFLOAD_0 [Object	0	0	0	1	i	—	i
[62] ARRAYLENGTH [Object	1	0	0	0	i	—	i
[64] ISTORE_1	0	0	0	0	i	i	i
[65] ILOAD_0	1	0	0	0	i	i	i
[66] ILOAD_1	2	0	0	0	i	i	i
[67] IF_ICMPGE 87	0	0	0	0	i	i	i
[70] NEWOBJECT String.<init>:()V	0	1	0	0	i	i	i
[72] DOWNCAST String Object	0	0	1	0	i	i	i
[75] REFLOAD_0 [Object	0	0	1	1	i	i	i
[77] ILOAD_0	1	0	1	1	i	i	i
[78] ARRAYSTORE [Object	0	0	0	0	i	i	i
[80] ILOAD_0	1	0	0	0	i	i	i
[81] ICONST_1	2	0	0	0	i	i	i
[82] IADD	1	0	0	0	i	i	i
[83] ISTORE_0	0	0	0	0	i	i	i
[84] GOTO 60	0	0	0	0	i	i	i
[87] RETURN	0	0	0	0	i	i	i

Fig. 1. Verification of Type-Separated Bytecode

has one $[Object_0$. The remaining types do not need a register since values are never stored.

During standard verification only the height of the stacks and the register status (if a value was written into it) is important. The type of the value used in an instruction is always known. For example, the instruction 52 pushes an integer value on the *int* stack (increasing the stack height) and instruction 72 transfers a value from one stack to another. In the latter case it must be checked that this cast is possible (using a downcast this check can be done statically) and that a value is available.

4 Verification of Type-Separated Bytecode

In the following subsections we present the verification algorithm for type-separated bytecode. First, the base algorithm is given. To improve time and memory consumption it is further refined and additional techniques are shown.

Therefore, the last verification algorithm presented is currently the most effective one for type-separated bytecode.

4.1 Basic Verification

The algorithm for the normal verification of type-separated bytecode is closely related to the standard algorithm which is used for Java Bytecode verification. It is a simple data flow analysis (DFA) where each instruction is simulated on an active map containing the stacks and corresponding register sets. Depending on the instruction it is verified that no stack can over- or underflow and only registers which do exist and are properly initialized are accessed. To simplify the DFA no subroutines (bytecode instructions jsr/ret) are allowed and therefore these are resolved during code generation.

```
finish = false

get first instruction
initialize active map
{{
  if (map exists at instruction)
    join current map and stored one
    if (not possible)
      abort verification
    if (map has changed)
      make joined map the active one
    unmark instruction
  simulate instruction
  if (error during simulation)
    abort verification with error
  if (instruction exits cflow)
    finish = true
  if (instruction is branch)
    if (target has a map)
      join current and target maps
      if (not possible)
        abort verification
      if (map has changed)
        mark target instruction
        store joined map at target
    else /* no map present */
      store active map at target
      mark target instruction
  get following instruction
 } while (finish is false)
 if (marked instruction exist)
   get marked instruction
   unmark instruction
   make stored map the active one
   finish = false
} while (finish is false)
```

Fig. 2. Standard Verification Algorithm

```
finish = false
mark all instructions with maps
get first instruction
initialize active map
{{
  if (map exists at instruction)
    join current map and stored one
    if (not possible)
      abort verification
    if (map has changed)
      abort verification
    unmark instruction
  simulate instruction
  if (error during simulation)
    abort verification with error
  if (instruction exits cflow)
    finish = true
  if (instruction is branch)
    if (target has a map)
      join current and target maps
      if (not possible)
        abort verification
      if (map has changed)
        abort verification

    else /* no map present */
      abort verification

  get following instruction
 } while (finish is false)
 if (marked instruction exist)
   get marked instruction
   unmark instruction
   make stored map the active one
   finish = false
} while (finish is false)
```

Fig. 3. StackMap-Based Algorithm

In figure 2 the base algorithm is outlined. It uses one active map (named *current map*) where all simulation of currently processed instructions occurs. Additionally, for each jump target another map is stored which contains the (data

flow) input to this instruction. If such an instruction is reached during normal verification either the active map is stored at the target and the instruction is marked for further verification or a join between the stored and the active map occurs. Should a join not be possible (perhaps since stack sizes differ) the verification aborts. If the number of initialized registers in the active map is smaller than those in the target map, the resulting map changes. In this case it must be stored at the target and the destination instruction gets marked. This guarantees that this program section is verified again with the recently stored map. In case we encounter an instruction which "exits control flow" (for example returns and unconditional jumps) the verification for the currently processed program section is finished and another marked instruction (if existing) must be taken to continue verification.

Relying on a data flow analysis without special modifications is costly. Since a fixpoint cannot always be reached in one iteration it is possible that multiple iterations occur. Therefore, the worst case runtime is $O(n^2)$. This circumstance does not happen very often, in most cases two to three iterations are sufficient. The advantage with type separation is that since no type inference is needed additional passes cost less time than with type inference.

Another important point with data flow analysis is memory consumption. For our verification algorithm we need for each type in the method to verify one byte for the stack height and one bit for each register used (for r registers these are $(r + 7)/8 = n$ bytes). A map now contains this information for all necessary types. During verification at least one map is needed to simulate the instructions. Besides this active map one needs to store a map at each jump target (where control flow splits or merges). Since the amount of jump targets can get very large a lot of memory may be consumed. This makes the algorithm not a suitable candidate for verification on constrained devices.

4.2 StackMap-Based Verification

For constrained devices and upcoming in the newest JDK the verification algorithm is splitted into two parts: one at the producer and one at the consumer side. At the producer side after doing a normal verification information regarding the fixpoint of the data flow analysis is added to the program. This information can be used one the consumer side to do a verification where multiple iterations can no longer occur since always the least upper bound is annotated. If this is not the case either the program or the map has been altered and the verification aborts. Therefore, the worst case runtime of the verification algorithm is $O(n)$ and the memory consumption during verification can be significantly reduced (see [15, 14]). It must be noted that the information added to the program code can considerably increase the file size and should not be neglected.

The algorithm used for type-separated bytecode can also be adapted to the stackmap approach (see figure 3). To be concrete, for each jump target the incoming data for the DFA is added to the program. The worst case runtime is also $O(n)$. For verification one active map is needed, but other maps need not to be saved. If another map exists at an instruction this map has been annotated by

the code producer. A merge between the active and the stored map still occurs, but the merge must not yield a different map. After merging, the transmitted map for this instruction is loaded and the verification continues with this map.

Another advantage using this approach is based on the fact that annotated stackmaps only need to be read. This is important for constrained devices. Since no writing occurs the maps can be stored in persistent memory and precious scratch memory is saved. The main disadvantage is that the stackmaps can increase file size considerably. Stackmaps for Java Bytecode can easily result in overheads of approximately 20% [8]. But in contrast to normal stackmap-based verification our algorithm should be faster again since no type inference need to be done.

4.3 Verification Using Constraints

To improve the verification process constraints can be made for bytecode. For example typical constraints are preinitialized registers and empty stacks at branches [9]. Fortunately, these constraints can also be used with type-separated bytecode and enable a large performance boost to the verification.

Preinitialized Registers Having preinitialized registers simply means that all registers (except those holding parameter values) are initialized with a neutral value prior entering the method (for example zero for primitive types and null for reference types). This is a constraint the virtual machine must ensure. It is not needed to store neutral values "by hand" into registers, simply zeroing the memory for the registers suffices. If a register is read though no value has been stored into it, the neutral value is taken. This value has always the type associated with the register. Hence, no type violation can occur. If the value is of reference type any access is preceded by a nullcheck so once again type safety is not broken.

If we use preinitialized registers with type-separated bytecode two important overheads are eliminated. First, we do not need to check if a register is initialized. Other checks like type of register can be omitted too, since the type of the register set is always known. Second, since we do not need to consider the status of registers we do not have to store them, resulting in less memory wasting. The remaining issue which still has to be done for registers is to check that only the given amount of registers is used.

Preinitialized registers also induce a reduction in verification runtime. Using type-separated bytecode multiple verification passes can only occur when two maps are joined and the resulting map has less initialized registers than the stored map. Since we do not need to examine registers at all there is no way multiple iterations can occur. Therefore, the worst case runtime for an algorithm implementing preinitialized registers is $O(n)$. Additionally, the memory consumption is reduced. All maps used during verification just have to store the stack height for each type. Thus, the verifier needs for a method with t types, j jump targets and one active map $t * j + t$ bytes.

Empty Stacks at Branches In general it is possible that at branch instructions and branch targets the stack is not empty. During normal bytecode verification both stacks (the incoming and the stored one) must be merged which is a costly process. Type-separated bytecode simplyfies this matter a lot since it must only be assured that the stack heights for identical types are equal. If it is known that the stack heights at branches are zero it is sufficient to compare each stack with zero. It is no longer required to store the stack heights at the branches. Still, we need some memory to store the positions of branch targets (to check if stacks are essentially zero) and if no other constraints are active we must store the used registers.

Using just empty stacks at branches does not solve the problem of multiple iterations. Therefore, the worst case verification runtime is $O(n^2)$. Memory consumption is slightly lower than verification without constraints, but still is too much for constrained devices. Since most of the time the stack is empty at branch targets, the program code overhead for this constraint is very low. If the stack is not empty, all remaining values must be saved in registers before the branch and restored afterwards. This results in a slight increase in program code.

Special Instructions In general, the verification algorithm needs to know were jump targets are. At these points control flow typical merges which results in verification states stored there. This information can be retrieved on the fly (resulting in possibly multiple iteration passes) or in advance (traversing all instructions once before verification).

Using special instructions at jump targets divides the program into basic blocks and for this reason precisely reveals branch targets. Therefore, program code need not be traversed prior verification and multiple iterations can be avoided. The verification algorithm must ensure that each instruction at a branch target is indeed a special instruction. The instruction itself can simply be skipped. All other verification parameters stay the same, except that the program code increases by n bytes, where n is the number of jump targets.

Normal Code	Modified Code
09: ...	09: ...
10: iload_1	10: iload_1
11: iload_2	11: **jmp_target**
	12: iload_2
12: iadd	13: iadd
13: ifeq 20	14: ifeq 21
16: iload_0	17: iload_0
17: goto 11	18: goto 11
20: iload_1	21: **jmp_target**
	22: iload_1
21: ...	23: ...

Fig. 4. Inserting Special Instructions

In figure 4 a piece of code is displayed. On the left is the unmodified program, whereas on the right the instruction *jmp_target* has been used as a branch target marker. Since *jmp_target* is embedded into the program code instruction numbers for targets have to be adjusted. Otherwise, the code is unchanged. It is now possible to tell on the fly where basic blocks start and where for example verification maps should be stored. This gives a slight increase in performance, yet the main characteristics of verification stay the same. It is also possible to store the number of *jmp_target*s into the class file for improved safety and memory management.

4.4 Combining Multiple Techniques

Each previous mentioned constraint by itself has the potential to increase the performance of verification. If all are combined the result is clearly noticeable. The algorithm integrating all three presented constraints works as depicted in figure 5.

```
start with first instruction
do
  simulate instruction
  if (error occurs during simulation)
      abort verification
  if (instruction is branch)
    if (current stack is not empty)
      abort verification
    if (target is not jmp_target)
      abort verification
    if (instruction is unconditional
        branch) AND (next instruction
        is not jmp_target)
      abort verification
  else if (instruction is jmp_target)
    if (current stack is not empty)
      abort verification
  else /* normal instruction */
    /* no special treatment */
  get next (in code order) instruction
while (instruction is available)
```

Fig. 5. Combined Verification Algorithm

Special instructions in the program code mark branch targets (instruction *jmp_target*). Although it is no longer needed to store anything at these points (registers are preinitialized, stacks are empty) we still need to check wether the stacks are essentially empty. Gathering this information during verification is feasible because of the special instructions inserted.

To simulate instruction behavior a current map is needed. This is the only map required for verification. In comparison to normal verification this map must simply store the stack heights, any register status can be omitted since registers are preinitialized.

Altogether the combined verification algorithm has a linear runtime with worst case $O(n)$ and a verification overhead of t bytes, where t is the number of different types for the method verified (which in general are far less than 100). The program code increases slightly by at least j bytes with j being the number of jump targets and some additional bytes to guarantee the empty stacks.

5 Outlook

Currently, two main research areas are pursued. The first one deals with verification as stated in this paper. At this time, all algorithms for verification of type-separated bytecode are implemented. Now, we will focus on exhaustive evaluations regarding memory consumption and runtime behavior of the verification algorithms as well as measuring the program code overhead because of the proposed constraints.

The second research area deals with execution of type-separated bytecode. First analysis show that plain execution of type-separated bytecode takes roughly the same time as execution of Java Bytecode. Yet, additional research to improve the runtime performance and evaluations regarding memory consumption need to be done.

Acknowledgments This work is supported by the DFG (Deutsche Forschungsgemeinschaft) through grant AM 150/2-1.

References

1. Wolfram Amme, Niall Dalton, Michael Franz, and Jeffery von Ronne. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'2001)*, volume 36 of *ACM SIGPLAN Notices*, pages 137–147, Snowbird, Utah, USA, June 2001. ACM Press.
2. Cinzia Bernardeschi, Giuseppe Lettieri, Luca Martini, and Paolo Masci. A space-aware bytecode verifier for Java cards. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode'2005)*, pages 216–232, 2005.
3. Ian Bayley and Sam Shiel. JVM bytecode verification without dataflow analysis. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode'2005)*, pages 185–202, 2005.
4. ECMA (European Computer Manufacturers Association) International. Common Language Infrastructure (CLI), Standard ECMA-335, December 2002. Published on Internet under
5. Andreas Gal, Christian W. Probst, and Michael Franz. A denial of service attack on the Java bytecode verifier. Technical Report 03-23, School of Information and Computer Science, University of California, Irvine, October 2003.

6. Masami Hagiya and A. Tozawa. On a new method for dataflow analysis of Java Virtual Machine subroutines. *Lecture Notes in Computer Science*, 1503, 1998.
7. Thomas Kistler and Michael Franz. A Tree-Based alternative to Java byte-codes. *International Journal of Parallel Programming*, 27(1):21–34, February 1999.
8. Xavier Leroy. Java bytecode verification: An overview. In *Proceedings of the International Conference on Computer Aided Verification (CAV'2001)*, pages 265–285, London, UK, June 2001. Springer-Verlag.
9. Xavier Leroy. Bytecode verification on Java smart cards. *Software – Practice and Experience*, 32:319–340, 2002.
10. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, second edition, April 1999.
11. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.
12. George C. Necula. Proof-carrying code. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'1997)*, ACM SIGPLAN Notices, pages 106–119, New York, NY, USA, January 1997. ACM Press.
13. OpenGroup. Architecture Neutral Distribution Format (XANDF) Specification. *Open Group Specification P527*, page 206, January 1996.
14. Eva Rose. Lightweight bytecode verification. *Journal of Automated Reasoning*, 31(3–4):303–334, 2003.
15. Eva Rose and Kristoffer Høgsbro Rose. Lightweight bytecode verification. In *Proceedings of the Workshop on Formal Underpinnings of the Java Paradigm (OOPSLA'1998)*, October 1998.
16. Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'1998)*, pages 149–160, New York, NY, January 1998. ACM.
17. Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Soot: A Java Optimization Framework, 1999. Sable Research Group of McGill University,