Efficient Bytecode Verification for Constrained Devices

Philipp Adler and Wolfram Amme
Friedrich-Schiller-University Jena
Institute for Computer Sciences
Ernst-Abbe-Platz 2
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. In the paper we present the basic principles of type-separated bytecode and show how its properties can be used to achieve a more efficient bytecode verification algorithm. Different verification techniques are illustrated and compared. The best algorithm presented achieves a linear verification time without a substantial overhead in program size.

1. INTRODUCTION

Mobile code like Java Bytecode is used to transfer program code from one computer to another. To avoid execution errors after transmitting the program each mobile code system performs some kind of verification before running the program code. Many different levels of verification exist, for example complex and resource consuming algorithms (i.e. program proofs) or fast and efficient, although not completely reliable, signature approaches.

With Java Bytecode a tradeoff was made. During bytecode verification the program is checked for type and reference safety. There are also checks to guarantee that the stack cannot over- and underflow and that all operands are used correctly. Therefore, a certain level of security is established with a medium resource overhead.

On modern computers time and space consumption during Java Bytecode verification is mostly irrelevant. Plenty of memory is available and increasing computing power reduces the verification time. Indeed, the possibility of exploiting the time behavior of the verification algorithm to mount an attack on virtual machines is existing [4], but this danger can often be ignored.

With 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 resources 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 bytecode verification is often higher than on normal systems. Both problems are not completely solved, leading frequently to compromises where one resource is exchanged with another. For example, to lower the verification time for Java Bytecode more space is needed [11] and vice versa [3, 8].

In this paper we present the basic concepts of an alternative intermediate representation which is currently based on Java Bytecode. This alternative representation combines type-separation with bytecode. In doing so, the same verification goals as in Java Bytecode can be achieved with less memory consumption and a shorter verification time.

An overview of current bytecode verification techniques is given in section 2. Our intermediate representation is called Type-Separated Bytecode and will be briefly explained in 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 their 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

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, that the stack cannot over- or underflow and that all operands are used correctly [9].

Since Java Bytecode belongs to the stack based intermediate representations it has one stack and register set which are untyped. Therefore, the values on the stack and in each register are of arbitrary type. To guarantee type safety a verification algorithm has to check that all types are used correctly. This examination is done using a data flow analy-

sis combined with abstract interpretation. During analysis, the algorithm has to remember the type of each value on the stack and in the registers, inferring types if necessary (e.g. at instructions where control flow joins). If different stack and register states must be merged type inference can get very costly. As a side-effect multiple iterations of the data flow analysis may be necessary. For this reason the verification process often is time and memory consuming.

Specifically, since the verification uses a data flow analysis where multiple iterations can occur, the complexity class of the standard verification 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 branches the memory consumption during verification rises quickly into regions of thousands of bytes [7]. 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. Verification performance can be improved if problem areas which present special challenges to bytecode verification are eliminated. Subroutines are such a challenge and can be dealt with differently [5, 12], for example by inlining them into the generated code. The second approach tries to speed up the verification process, e.g. with modified algorithms or by adding special information to the bytecode [2, 10, 11].

For example, to simplify and speed-up the data flow analysis and to reduce main memory usage stackmaps can be added to the program. In fact, for constrained devices stackmaps are already used and in the future will become an obligation for each generated Java Bytecode file. The main disadvantage of using stackmaps are the sizes of the bytecode files, which easily grow by two-digit percentages. Although these techniques can reduce the amount of verification time, as a compromise the program size often increases due to restructuring code or adding additional information.

Memory consumption during verification is targeted by various researchers. The problem emerges particularly on constrained devices, but suggested solutions can be used even for other systems. To reduce memory usage one can use special constraints for the program code, leading to more advantageous code with less control points [8]. For example, one constraint for generated bytecode are empty stacks at branches. If in the original bytecode the stack is not empty, the values must be stored into registers and thereafter put back onto the stack. This code restructuring produces an additional overhead. Another technique modifies the verification algorithm and uses multiple verification passes, each solving subproblems [3]. This technique saves main memory, however verification time increases since verification must be done for every type. Altogether, memory conserving techniques often lead to increased code size or verification time.

3. TYPE-SEPARATED BYTECODE

Amme et al. [1] present a mobile code representation which uses type-separation, but is based on static single assignment form. In type-separated bytecode the technique of type-separation is applied to normal bytecode. Instead of having one untyped stack and register set there are as many stacks and register sets as types exist. For example, if there are five different types used in a method there would be five distinct stacks and register sets, each associated to its appropriate type. Every instruction knows per se where the operands of an operation must come from (source types) and the result should go to (destination type). 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 stacks and register sets for the generated bytecode are displayed after execution of each instruction. In this program three types are used: *int*, *Object* and *[Object* (an array of *Object*). For each type a stack and register set exists, hence there are three stacks and register sets. Since in this example only one register of the *[Object* type is used it is the only one shown. The other register sets are not used and remain empty.

If an instruction is executed it takes its operands directly from its associated source stack(s) or register set and puts the result back onto the appropriate destination stack or into the destination register set. For example, instruction 52 pushes the integer value 10 onto the *int* stack. Instruction 54 takes the dimension of the array from the *int* stack and puts the created array onto its corresponding array type stack (in this case onto the *[Object* stack). Instruction 56 takes the top element of the *[Object* stack and puts it into its associated register 0 (also of type *[Object*). The more complex instruction 64 takes an array of type *[Object*, an element of type *Object* and an integer value to store the *Object* value into the array of type *[Object* at an index indicated by the integer value.

During basic verification of type-separated bytecode it is not necessary to store the types of the values on the stacks or in the register sets. Since stacks and register sets are type-separated the type of every value is implicitly known. Therefore, it is sufficient to store only the height of each stack and the register status for each register set (if a value was written into a register). During verification (using abstract interpretation) it is simply required to check for the given instruction that the source stacks contain at least the amount of used values and that the used register of the source register set is already initialized. In the example type-separated bytecode in figure 1 before verifying instruction 64 it is known that the stack height for the stacks of the three types *int*, *Object* and *[Object* is one. During abstract interpretation of instruction 64 the used stack's heights are reduced by one. Additionally, it is observed that no stack over- or underflows. Therefore, abstract interpretation of this single instruction was successful.

4. VERIFICATION OF TYPE-SEPARATED BYTECODE

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

Java Program Code				
`Object list[] = new Object[10];` `list[5] = new Object();`				
Type-Separated Bytecode				
Bytecode	Stacks			Registers
	int	Object	[Object	[Object$_0$
`[52] BIPUSH 10`	10	—	—	—
`[54] NEWARRAY [Object`	—	—	ref_{54}	—
`[56] REFSTORE_0 [Object`	—	—	—	ref_{54}
`[58] NEWOBJECT Object."<init>":()V`	—	ref_{58}	—	ref_{54}
`[60] REFLOAD_0 [Object`	—	ref_{58}	ref_{54}	ref_{54}
`[62] BIPUSH 5`	5	ref_{58}	ref_{54}	ref_{54}
`[64] ARRAYSTORE [Object`	—	—	—	ref_{54}

Figure 1: Verification of Type-Separated Bytecode

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

4.1 Basic Verification

During basic verification of Java Bytecode a data flow analysis (DFA) is used to guarantee proper use of types and references. For each instruction a state must be inferred which consists of a stack map and a register map (where for each value on the stack or in the registers the type must be stored). When performing the DFA it is possible that two states must be merged. For every stack position and each register of the states it must be guaranteed that the types of the values can be joined. This is done using (time-consuming) type inference. Storing states for instructions poses an additional overhead in memory usage.

The algorithm for the simple verification of type-separated bytecode is comparable to the standard algorithm which is used for Java Bytecode verification. It consists of a simple data flow analysis and a current state. The current state is used for abstract interpretation of instructions and contains for each type only the current stack height and register status (which registers currently are initialized). Depending on the instruction it is verified that no stack can over- or underflow and only registers which are properly initialized are accessed. To simplify the DFA, subroutines do not exist and are resolved during code generation.

In figure 2 the simple verification algorithm is outlined. It uses one current state which is used for the abstract interpretation of instructions. This current state is modified depending on the instruction, for example stack heights are modified and register states get updated. Additionally, for each branch target a state is stored which contains the (data flow) input to this instruction. If such a branch instruction is reached during normal verification either the current state is stored at the target and marked for further verification or a join between the stored and the current state occurs. During the join stack heights for all stacks are compared and must be equal, otherwise the verification fails. Afterwards, register sets are joined. If the number of initialized registers in the current state is smaller than those in the stored (target) state, the resulting joined state changes (in comparison to the stored state). In this case the joined state must be stored at the branch target and the target instruction gets marked. This guarantees that this program section is verified again with the recently stored (joined) 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.

```
finish = false
unmark all instructions
get first instruction
initialize current state
do
  do
    if (stored state exists at instruction)
      join current state and stored one
      if (not possible) abort verification
      if (joined state != stored state)
        stored  state = joined state
        current state = joined state
      unmark instruction
    simulate instruction
    if (error occurs during simulation)
        abort verification with error
    if (instruction exits control flow)
      finish = true
    if (instruction is branch)
      if (target instruction has a stored state)
        join current and target state
        if (not possible) abort verification
        if (joined state != target state)
          mark target instruction
          target state = joined state
      else /* no stored state present */
        store current state at target
        mark target instruction
    get following instruction
  while (finish is false)
  if (marked instruction exist)
    get marked instruction
    unmark instruction
    current state = stored state
    finish = false
while (finish is false)
```

Figure 2: Standard Verification Algorithm

Relying on a data flow analysis without modifications can get costly. Since a fixpoint cannot always be reached in one iteration multiple iterations can occur. However, using type-separated bytecode multiple iterations can only happen due to changing register sets (if they get smaller during a join). If only stacks could be considered one linear pass over the program code would be sufficient. Multiple iterations can result in a worst case runtime of $O(n^2)$. This does not happen

very often, in most cases two to three iterations are sufficient. The advantage with type separation is (since no type inference is needed) additional passes cost less time compared to Java Bytecode verification (with type inference). Additionally, in type-separated bytecode verification simple checks (comparing only stack heights and register status) replace complex checks (type comparison and inference) resulting in an overall verification speed-up without sacrificing any of Java Bytecode's security aspects.

Another important point using DFA is memory consumption. For our basic verification algorithm we need for each type (in the method to verify) one byte for the stack height (for t types these are t bytes) and one bit for each register used (for r_i registers of type i these are $\lfloor \frac{r_i+7}{8} \rfloor$ bytes). A state now contains this information for all necessary types t. During verification at least one state is needed for abstract interpretation. Besides this current state one needs to store a state at each branch target (where control flow splits or joins). For j branch targets this is a total consumption of

$$\left(\left(\sum_{i=1}^{t} \left\lfloor \frac{r_i + 7}{8} \right\rfloor \right) + t \right) * (j+1) = n$$

bytes. Since the amount of branch 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 soon with JDK 6 [6]) the verification algorithm for Java Bytecode is splitted into two parts: one at the code producer and one at the code consumer side. At the producer side, after performing a verification on the bytecode, information regarding the last iteration of the data flow analysis is added at special control flow points to the program. This information can be used on the consumer side to perform a simplified verification. Precisely, the last iteration of the data flow analysis is repeated using the attached information. Multiple iterations can no longer occur since that would indicate that either the program or the added information was modified (and in both cases verification has to abort). Therefore, the worst case runtime of the verification algorithm is $O(n)$. Additionally, the memory consumption during verification can be significantly reduced (see [10, 11]) since only one current state must be maintained. However, it must be noted that the data flow information added to the program code can considerably increase the file size.

The base algorithm used for type-separated bytecode can also be adapted to the stackmap approach (see figure 3). For each branch target the incoming data of the last iteration of the DFA is added to the program code as so-called stackmaps. The stackmaps contain the current state before abstract interpretation of the target instruction and therefore can be used directly for a simplified verification. The worst case runtime is now $O(n)$.

For stackmap-based verification one current state is needed, but states need not to be saved at branch targets (since at these points stackmaps exist). If a stackmap exists at an instruction this map has been annotated by the code producer.

```
finish = false
unmark all instructions
mark all instructions with maps
get first instruction
initialize current state
do
  do
    if (map exists at instruction)
      join current state and stored map
      if (not possible) abort verification
      if (joined state != stored map)
        abort verification
      unmark instruction
    simulate instruction
    if (error occurs during simulation)
        abort verification with error
    if (instruction exits control flow)
      finish = true
    if (instruction is branch)
      if (target instruction has a map)
        join current state and target map
        if (not possible) abort verification
        if (joined state != target map)
          abort verification
      else /* no map present */
        abort verification
    get following instruction
  while (finish is false)
  if (marked instruction exist)
    get marked instruction
    unmark instruction
    current state = stored map
    finish = false
while (finish is false)
```

Figure 3: StackMap-Based Verification Algorithm

A merge between the current state and the stored map still occurs, but the result must be identical to the stored map. If this is not the case, verification fails. Otherwise, the stored map is loaded as the current state and verification continues normally. For t types and r_i registers of type i the memory consumption would be n bytes, where

$$n = \left(\sum_{i=1}^{t} \left\lfloor \frac{r_i + 7}{8} \right\rfloor \right) + t.$$

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% [7]. First estimations for type-separated bytecode show that the overhead would be similar.

4.3 Constraints for Improved Verification

To improve the verification process constraints can be made for generated bytecode. For example, constraints proposed for Java Bytecode are preinitialized registers and empty stacks at branches [8]. These constraints can also be used with type-separated bytecode and result in an improved verification process. Following three constraints were chosen for type-separated bytecode.

4.3.1 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 constraint is ensured by the virtual machine which sets the memory used by the registers to zero before executing program code. If a register is read though no value has been stored into it, the neutral value is taken. With Java Bytecode this constraint alone is not very usefull. Since registers can contain different types, it is inevitable to store the type of the values written into registers to maintain type safety.

Using preinitialized registers with type-separated bytecode is more convenient. Since registers are strictly typed, the type of the values written to or read from registers is always given. Hence, memory for storing register states is no longer needed. Additionally, no type violation can occur. If the value is of reference type any access to this value is preceded by a nullcheck so once again type safety is not broken. Therefore, two important overheads during verification are eliminated. First, we do not need to check if a register is initialized. Second, since we do not need to consider the status of registers we do not have to store them in the current state and at branch targets.

As a consequence preinitialized registers induce a reduction in verification runtime. Using type-separated bytecode, multiple verification passes can only occur when a current state and a stored state are joined and the resulting state has less initialized registers than the stored state. Since we do not need to examine registers at all, there is no way multiple iterations can occur. Therefore, the worst case runtime for a verification algorithm implementing preinitialized registers is $O(n)$ (in contrast to $O(n^2)$ for Java Bytecode verification without stackmaps). Additionally, the memory consumption is reduced. All states (current and stored ones) used during verification just store the stack height for each type. Thus, the verifier needs for a method with t types and j jump targets $t * j$ bytes and t additional bytes for the current state. Altogether these are $t * j + t$ bytes.

4.3.2 Empty Stacks at Branches

In general, it is possible that at branch instructions and branch targets the stack is not empty. During standard Java Bytecode verification both stacks (the incoming and the stored one) must be merged which is a costly process. If an empty stack is guaranteed at those points no merge is needed and storing the stack is superfluous. Nevertheless, the merge process must still be applied to the used registers and the register states must also be stored.

In type-separated bytecode at program points were a join of states is necessary only stack heights need to be compared. Using the basic verification algorithm with empty stacks it must be assured that the stacks at branches and branch targets are empty, essentially comparing each stack height with zero. It is no longer required to store the stack heights at branch targets. However, some memory is needed to store the positions of branch targets (to check if stacks are essentially zero) and register states.

Since the problem of joining register sets is not addressed, using just empty stacks at branches does not solve the problem of multiple iterations. Therefore, the worst case verification runtime is $O(n^2)$. If the stack is not empty, all remaining values must be saved in registers before the branch and restored afterwards. This can result in a slight increase in program code.

4.3.3 Special Instructions

Typically, the verification algorithm needs to know were branch targets are. At these points control flow merges, which results in verification states that must be stored there. Branch targets can be retrieved on the fly (resulting in possibly multiple iteration passes) or in advance (traversing all instructions once before verification). For example, consider figure 4. On the left the unmodified Java Bytecode is displayed. During the first visit of instruction 11 the verification algorithm does not know that this instruction is a branch target and therefore no state is stored. When the verification process reaches instruction 17 it notices that a backwards branch to instruction 11 occurs. Since no information was stored at instruction 11 an additional verification pass is needed for this section of the program.

Using special instructions at branch targets divides the program into basic blocks and precisely reveals those branch targets. The program code need not be traversed prior verification and multiple iterations can be avoided respectively. The verification algorithm must ensure that each instruction at a branch target is indeed a special instruction. The instruction itself can simply be treated as a no-operation. All other verification parameters stay the same, except that the program code increases by n bytes, where n is the number of branch 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: ...

Figure 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 only instruction numbers have to be adjusted. It is now possible to identify on the fly where branch targets are and where for example verification states should be stored.

4.4 Basic Verification & Multiple Constraints

Each previous mentioned constraint by itself has the potential to improve verification performance. If all are combined within the basic verification algorithm 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)
```

Figure 5: Combined Verification Algorithm

Special instructions in the program code mark branch targets (instruction *jmp_target*). Although it is no longer necessary 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.

For abstract simulation of instruction behavior a current state is needed. This is the only state required for verification. In comparison to normal verification this state 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 memory consumption of t bytes, where t is the number of different types for the method verified (which in general are far less than 30). The program code increases slightly by at least j bytes with j being the number of jump targets and probably some additional bytes to guarantee empty stacks.

5. OUTLOOK

In this paper we have presented different verification techniques for type-separated bytecode. They guarantee the same security as Java Bytecode verification but use less resources. The combined verification algorithm has the lowest memory consumption (linear in number of types for the method) and fastest verification time (linear runtime in worst-case) of all algorithms presented. At this time, all algorithms for verification of type-separated bytecode are implemented and further exhaustive evaluations are performed.

Our next important research activity deals with execution of type-separated bytecode. Preliminary studies show that plain execution without additional requirements results in a slight overhead. To reduce this overhead we are working on a method to convert the multi-stack model to a single-stack model for program execution, which should make the runtime for type-separated bytecode comparable to Java Bytecode.

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

6. REFERENCES

[1] W. Amme, N. Dalton, M. Franz, and J. 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] I. Bayley and S. 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.

[3] C. Bernardeschi, G. Lettieri, L. Martini, and P. 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.

[4] A. Gal, C. W. Probst, and M. 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, Oct. 2003.

[5] M. Hagiya and A. Tozawa. On a new method for dataflow analysis of Java Virtual Machine subroutines. *Lecture Notes in Computer Science*, 1503, 1998.

[6] Java Community Process. JSR-000202 JavaTM Class File Specification Update Evaluation 1.0 Final Release. Technical report, Sun Microsystems, 2006.

[7] X. 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.

[8] X. Leroy. Bytecode verification on Java smart cards. *Software – Practice and Experience*, 32:319–340, 2002.

[9] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, second edition, Apr. 1999.

[10] E. Rose. Lightweight bytecode verification. *Journal of Automated Reasoning*, 31(3–4):303–334, 2003.

[11] E. Rose and K. H. Rose. Lightweight bytecode verification. In *Proceedings of the Workshop on Formal Underpinnings of the Java Paradigm (OOPSLA'1998)*, Oct. 1998.

[12] R. Stata and M. 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, Jan. 1998. ACM.