Memory Requirements of Java Bytecode Verification on Limited Devices

Karsten Klohs [a,1,2] Uwe Kastens [a,3]

[a] *Department of Computer Science*
University Paderborn
Paderborn, Germany

Abstract

Bytecode verification forms the corner stone of the Java security model that ensures the integrity of the runtime environment even in the presence of untrusted code. Limited devices, like Java smart cards, lack the necessary amount of memory to verify the type-safety of Java bytecode on their own. Proof carrying code techniques compute, outside the device, tamper-proof certificates which simplify bytecode verification and pass them along with the code. Rose has developed such an approach for a small subset of the Java bytecode language.

In this paper, we extend this approach to real world Java software and develop a precise model of the memory requirements on the device. We use a variant of interval graphs to model liveness of memory regions in the checking step. Based on this model, memory-optimal checking strategies are computed outside the device and attached to the certificate.

The underlying type system of the bytecode verifier has been augmented with multi-dimensional arrays and recognizes references to uninitialized Java objects.

Our detailed measurements, based on real world Java libraries, demonstrate that the approach offers a substantial improvement in size of certificate over the similar approach taken by the KVM verifier. Worst case memory consumption on the device is examined as well and it turns out that the refinements based on our model save a significant amount of memory.

Key words: Proof Carrying Code, Bytecode Verification, Limited Devices, Java Card

[1] This work was conducted in cooperation with the ORGA Kartensysteme GmbH, Am Hoppenhof 33, 33104 Paderborn

[2] Email: taiko@upb.de, Fax: 0049 5251 60 6697

[3] Email: uwe@upb.de

1 Introduction

Chip cards are used in many application domains where security is an important issue. They serve as banking cards to access cash points, as subscriber identity modules (SIMs) in mobile phones, or as cards of clients in the public health system.

Java Cards are a special variant of Smart Cards equipped with a microprocessor that runs a Java Card Virtual Machine to execute Java programs. The platform independence of Java eases development of applications. Additionally, other Java concepts - like applet isolation and securely downloadable code even after a card is issued - fit the needs of the market.

Downloaded code may originate from untrusted sources or may be communicated over untrusted channels. Therefore, any Java Virtual Machine verifies that the code is safe before it is released for execution. This process is known as *bytecode verification*.

Type safety of program code is an essential issue. It has to be guaranteed that every bytecode instruction operates *always* on program objects which have the *correct type*, e.g. that arithmetic instructions do never operate on object references because pointer arithmetic would compromise the security of program execution.

Proofs of type safety require types of local variables and instruction operands to be reconstructed. Unfortunately, data flow analysis, the classical approach to solve such problems, is too complex for the limited resources of Java Cards. One of the suggestions to solve this challenge is based upon the general concept of *proof carrying code* introduced by Necula [Nec97]. Proof carrying code deals with the same scenario of downloading code from an untrusted producer over an untrusted channel to a consumer. This method can also be used to relief the "code consuming" Java Card from the memory intensive data flow analysis.

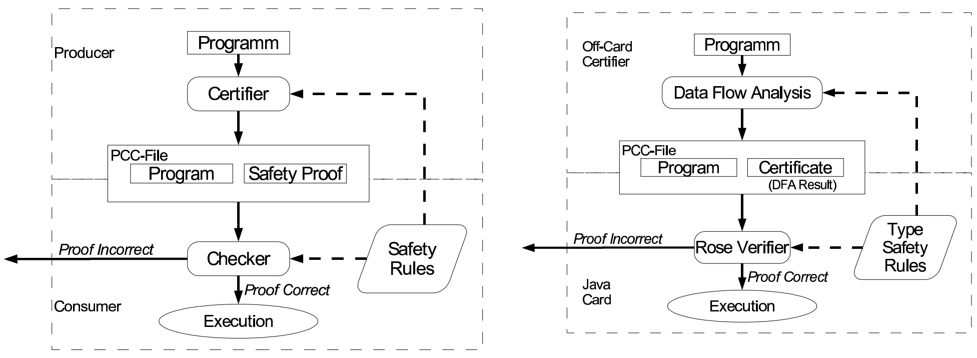

Fig. 1. Proof Carrying Code adapted to Bytecode Verification

The proof carrying code scenario is illustrated in Figure 1. The producer attaches a proof to the code according to some safety rules defined by the consumer.

Checking the attached proof against the code guarantees that the safety rules hold, unless the check fails. Thereby, any tampering of the code or the proof during creation or transport will be detected. Moreover, the consumer's check that a given proof is correct is usually much easier than generating the proof which is done by the producer.

Rose adapted the concept of proof carrying code to achieve a new variant of bytecode verification [Ros98] as illustrated in Figure 1. Type safety of Java's intermediate bytecode is regarded as the safety rules of the PCC method. The proof is the result of a data flow analysis. The Java Card acts as the consumer. It uses the certificate as an oracle which can predict the final result of the data flow analysis whenever needed. This way, the check of the type safety can be done efficiently in a single pass.

Rose showed in her master thesis [Ros98] for a non trivial subset of the Java language that the above approach is tamper proof. We have developed a precise model of the temporary memory requirements of the approach. This model leads to an efficient implementation and reveals new optimization possiblities. Furthermore, we have implemented a prototype that supports the full Java language and we have performed experimental measurements on real world software. The results show that the minimization of the temporary memory requirements is crucial for the approach and that the optimized variant becomes viable for today's Java Cards.

The remainder of this paper is structured as follows. First, we concentrate on the fundamental extensions of the Rose approach and our new technique for memory optimizations. Then we present the results of our evaluation. The section on related work focuses on alternative approaches to bytecode verification in resource-constrained environments.

2 Refined Verification Methods

We extended the approach suggested by Rose in three ways to be able to cope with the full instruction set of Java Bytecode.

The most challenging extension is the support for multi-branches because they influence the memory requirements of the verification significantly. We have developed an abstract model for the memory requirements of the approach. We use this model to improve the formulation of Rose's verification algorithm and to obtain a memory-efficient implementation.

We extended the type system of the verifier, too. Arrays introduce larger extensions to the type system itself, while the special size of the data types `long` and `double` influences the implementation of the verifier slightly. In addition, we discriminate references to objects that have not been initialized yet within our type system. This is an elegant way to solve a special verification task completely transparent to the verification algorithm.

The instruction set of the Java Virtual Machine has a couple of instruction groups like arithmetic integer instructions which have the same semantics at the type level. Rose's formal proofs require the definition of inference rules for each instruction. To reduce the number of rules Rose restricts herself to a few representative instruction like `IADD` and `ISUB` of an instruction group only. Our prototype supports *entire* instruction groups.

In the following section we summarize briefly how classic data flow analysis works and how Rose's variant has been derived from the classical approach. Then we describe our extensions in detail.

2.1 From DFA Verification to Lightweight Verification

The bytecode verification solves the problem of type inference: regardless of the flow of control each instruction has to find values of an appropriate type on the operand stack or in the local variables.

To ease the discussion of the different approaches we use a common terminology:

Definition 2.1 A *frame type* is a type based representation of the state of the method frame at a certain instruction.

The term frame type emphasizes the relationship to the method frame of the method in question that holds the operand stack and the local variable registers at runtime[4].

The elementary operation is the manipulation of frame types by an instruction. It can be regarded as a type based simulation of the execution of the instruction.

Definition 2.2 A *type transfer function* of an instruction maps a given frame type to a frame type that characterizes the effect of execution according to the type based semantics of the instruction. Additionally, the type transfer function states the preconditions that have to hold if the instruction is executed.

The preconditions prevent the execution of instructions if their operands do not have suitable types or if the execution would lead to a stack overflow etc.

Finally, we discriminate different kinds of frame types.

Definition 2.3 An *input frame type* is the input of the type transfer function of an instruction; an *output frame type* characterizes the result of the transfer function. Furthermore, we use the attribute *initial* to denote a frame type of the first pass of the data flow analysis. Similarily, a *final* frame types correspond to the final result of the data flow analysis.

Input and output frame types correspond to the situation right before and after an instruction while the final input frame type denotes the type inference

[4] "Method frame" is the Java term for the activation record of a method. We use the term "local variable registers" to emphasize that we talk about the local variables in the method frame of the virtual machine and not about local variables of the Java source language

result for an instruction. If all instructions can be executed on their final input frame type the program is type safe. Otherwise, the verification fails.

2.1.1 Type Inference by Conventional Data Flow Analysis

Data flow analysis solves a type inference problem by an algorithm that iterates over the control flow graph. A node of the control flow graph represents an instruction sequence of straight line code, i.e. code whose inner instructions are not target of a branch nor themselves branches. Control flow nodes are connected by directed edges whenever may be a transfer of control between the two instruction sequences.

The algorithm starts with the first instruction of the method body; its initial input frame type reflects the situation right after the invocation of a method when the first local variable registers of the method frame contain the parameters and the operand stack is empty. The transfer function of the instruction is applied to this initial input frame type and the initial output frame type is propagated to all successor instructions. This step is successively applied to all instructions whose input frame type has been changed, until a fixed point is reached. This process may require several iterations because backward branches may propagate new information to instructions that have already been processed.

The verification within a control flow node is straight forward because all output frame types are identical with the input frame types of the successor instruction. Hence, all frame types especially the output frame type of the last instruction of the sequence can be computed directly from the input frame type of the control flow node. Consequently, the instruction sequence of a control flow node can be regarded as a “super instruction” with a complex transfer function and we can focus on the input and output frame types of control flow nodes from now on.

Multiple successors of a control flow node and branches that target already visited nodes complicate the type inference algorithm as depicted in Figure 2. Branches with multiple targets like the edges to the nodes **C** and **D** cause the change of several input frame types. As a consequence, all pending input frame types have to be stored when the next one is processed.

The loop branch to the visited node **A** shows that an already computed input frame type may change afterwards. Thus, the data flow analysis requires multiple iterations to reach the fixed point and the verifier has to keep all input frame types in memory.

The memory required to store one frame type for every control flow node easily exceeds the RAM resources of a Java Card. Some of the more complex methods have 30 local variable registers and about 100 control flow nodes. This leads to a memory requirement of 6 kBytes which is actually supplied by the most powerful Java Cards only. Therefore the classical approach is considered impractical for Java Cards.

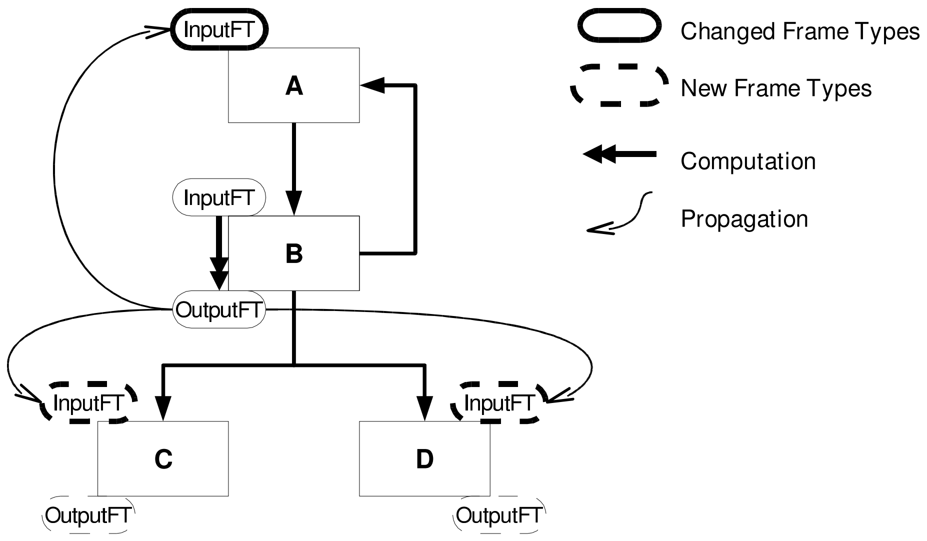

Fig. 2. Propagation of a Result Frame Types along Backward and Multi-Branches

2.1.2 Lightweight Verification

Here the proof carrying code approach comes in: Rose [Ros98] simplifies the algorithm by storing type information in an additional certificate whenever the verifier does not calculate the final input frame type immediately during the first pass. This idea combines two essential aspects: First, the certificate predicts information which would be computed by a classical verifier during the inspection of other control flows later. Second, the certificate contains entries only if the information of the initial input frame type differs from the final input frame type.

Figure 3 shows an example. The *final* input frame type of **B** is derived immediately by combining the actual output frame type of **A** and the difference information stored in the certificate. The approach requires a single pass

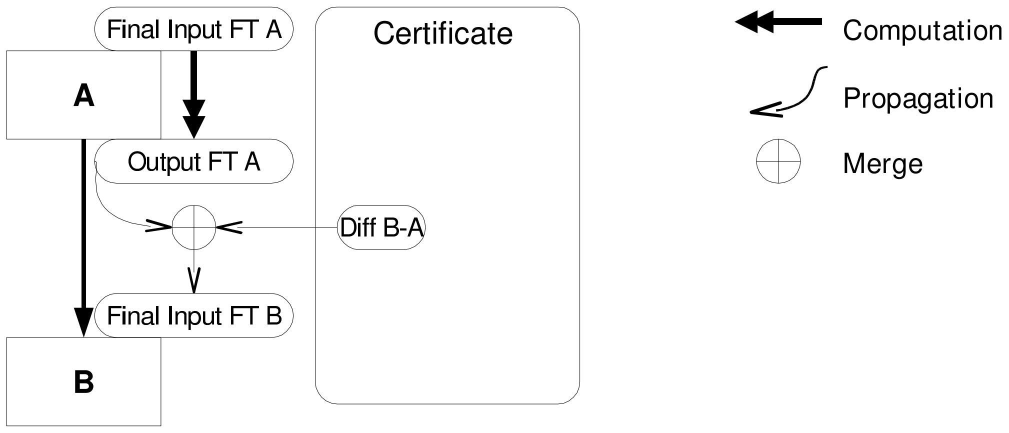

Fig. 3. Computation of Frame Types with Certificate Entries

over the bytecodes only, because the final result of the data flow analysis, i.e. the final input frame type of the actual control flow node, is always at hand.

An additional off-card phase determines the certificates by the classical data flow analysis without the resource restrictions of the Java Card. The certificate that is transmitted to the card holds the pieces of information that are essential to reconstruct the data flow analysis result from a initial input frame type computed by the verifier. Usually, the certificate entries are empty and a certificate fits into about 100 Byte.

However, this simplicity comes at the cost that the correctness of the information in the certificate has to be checked: Every final input frame type has to comply with the final output frame type of all predecessor nodes. The merge process guarantees this condition for the direct successor node because the difference information in the certificate can only produce a more restrictive final input frame type. Additionally, the verifier has to ensure that the final input frame types of all other successor nodes are more restrictive than the actual final output frame type, too. These final input frame types have either been computed already or they are computed later in the verification process. Thus, the verifier has to store some frame types temporarily.

2.2 Runtime Memory Requirements of the Lightweight Verification

In this section we reduce the problem of dealing with temporary frame types to the classical graph problem of finding a minimal cut for a linear arrangement of graph nodes.

Such a reduction is well know from register allocation techniques: The evaluation order of an expression can be modeled as a linear arrangement of a directed acyclic graph whose nodes represent single operations. These operation nodes are connected by a directed edge whenever the output value of one operation is used by the other operation. As a consequence, these intermediate values have to be stored in registers until their final use. A cut in the linear arrangement separates the nodes in treated and remaining ones as depicted in Figure 4. Hence, the edges that cross a cut correspond to the

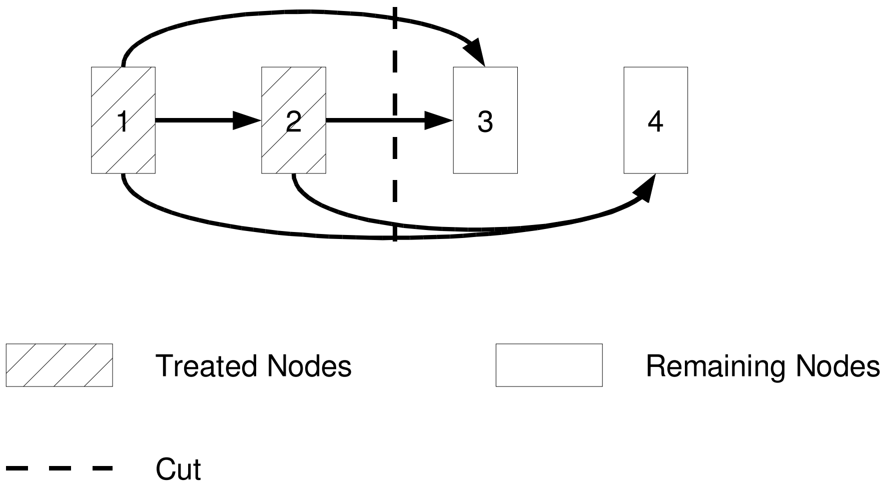

Fig. 4. A Cut in a Linear Arrangement of a Graph

number of intermediate values that are stored before the next operation is executed and the cut with the maximum number of values determines the register requirement for the given evaluation order.

The runtime memory requirements of the verifier can be determined in a similar way. The order in which the nodes of the control flow graph are considered during the verification forms the linear arrangement.

A cut in the linear arrangement corresponds to a transition of the verifier from one control flow node to the next. The nodes on the left hand side of the cut have already been verified and the nodes on the right hand side still remain.

Every edge in the control flow graph causes a test for compliance because the final output frame type of the source node has to be checked against the final initial frame type of the target node. Such a check can be done as soon as both frame types have been computed i.e. when their nodes have been verified. Hence, a frame type has to be stored for every edge that crosses the cut because then one of the participating nodes is not computed yet.

Figure 5 depicts the two situations: if a branch originates at a remaining

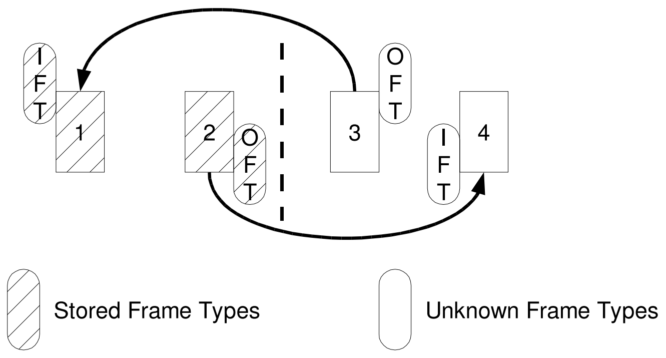

Fig. 5. Control Flow Edges and Temporary Frame Types

control flow node, the final input frame type of the target node is stored for the check. The situation is similar when a control flow edge targets a remaining node but the final output frame type has to be stored then.

The orientation of the edges does not model dependency information like in the register allocation model but determines only which kind of frame type will be stored. Therefore, our model differs slightly because there is no need to arrange source nodes before target nodes.

Multi-branches like LOOKUP-switches and the fact that a node can be target of multiple nodes complicate the model as depicted in Figure 6.

Although node 1 is targeted by two edges that cross the cut it is only necessary to store the final input frame type of node 1. The two edges just indicate that the frame type will be used twice. The problem is once more similar for stored output frame types. If a node has multiple successors in the sequence of unvisited node the output frame type is used for multiple compliance tests.

To make the number of edges that cross a cut equal to the number of

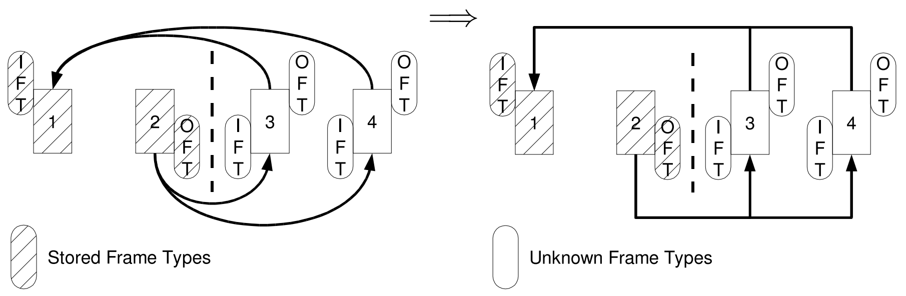

Fig. 6. From Multi-Edges to Special Edges

stored frame types, we model edges that originate in a set of remaining nodes or that target a set of remaining nodes by a special kind of edge as depicted in Figure 6.

Finally, each temporary frame type and the corresponding compliance tests are represented by a single edge. This edge determines the lifetime of its frame type. The frame type remains in memory as long as the edge crosses the cut of the actual transition. The number of cut edges is the number of simultaneously stored frame types of the actual transition. The cut with the maximum number [5] of edges determines the memory consumption of the whole verification pass when frame types are stored just as long as they are needed.

Our model shows another possibility to reduce the memory requirements of the verification now: the off-card phase can determine the linear arrangement with the *minimal* cut if the on-card verifier is able to deal with flexible verification passes. Our evaluation in Section 3 shows the expected effects.

2.3 Extensions of the Type System

A type system can be modeled as a lattice [Grä98] over the partially ordered set T of all types. The partial order defines the *subtype* relation on types. Additionally, the relation $\mathit{LeastUpperBound}\colon T \times T \rightarrow T$ determines the nearest common super class of two types.

The verifier uses the type lattice in two different ways. First, it ensures that the type transfer function is evaluated only if the precondition holds that the actual input frame type contains subtypes of the operand types. Moreover, the verifier merges frame types by applying the *LeastUpperBound* relation successively to all stack elements and local variables.

A new data type leads to two modifications: We insert the type in the lattice and we formulate the transfer functions of the new instructions. The introduction of long arithmetic types and arrays that are not covered by the Rose's sublanguage is straight forward.

More interestingly, we use the type system also to adress the verification of object initialization. This solution is elegant because it is completely trans-

[5] We do not consider the fact that frame types may differ in size, because we expect that the optimal solution is the one with a minimal number of frame types in most cases.

parent to the checker algorithm and keeps the semantic description of all other bytecode instructions unchanged.

On the bytecode level the creation of new objects is decomposed into two distinct instructions. The `NEW` instruction allocates memory for the object while its constructor is invoked later with the `INVOKESPECIAL` instruction. The verifier has to ensure that *references to an uninitialized object* are only used in a very restricted way. For example, other method invocations than constructor calls with `INVOKESPECIAL` are not allowed.

We solve this verification aspect in a similar way like Freund [FM99]. We introduce new *uninitialized types* into the type system. These new types make up a completely new part of the lattice which is not related to the original class hierarchy. The verifier will reject the code if uninitialized references are used as operands of conventional instructions because the subtype checks fail. The type transfer functions of the `NEW` and `INVOKESPECIAL`-instruction produce and consume the uninitialized variants of types.

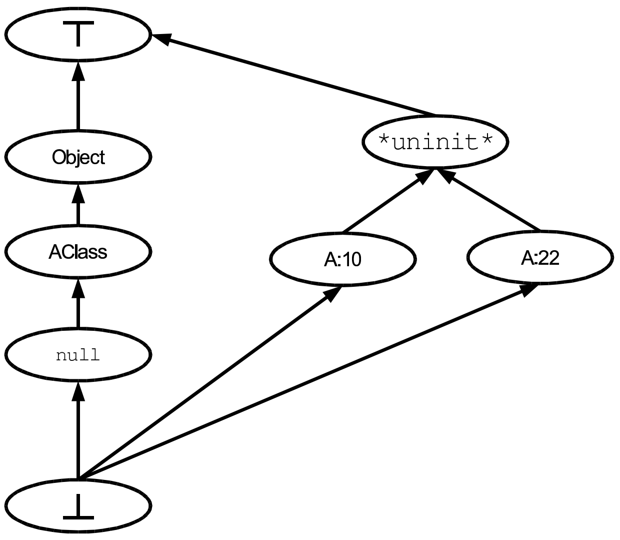

Fig. 7. Types for Uninitialized References

The conversion of uninitialized types into their initialized counterparts has to convert all copies of an uninitialized reference. To find these copies in the actual input frame every `NEW` instruction produces an own type. These types are distinguished by the offset of the `NEW` instruction.

Figure 7 illustrates the resulting lattice for a method that contains two `NEW A` instructions at the offsets 10 and 22.

2.4 Comparison and Discussion

In the proof carrying code approach for Java bytecode verification an off-card analysis computes the result of the data flow problem of type inference analysis. A verifier can prove the type safety of the program with this information efficiently.

Our approach reveals two degrees of freedom in the combination of proof carrying code and a data flow analysis: the type system and the certificate i.e. the information that is transfered to the checker.

Many subproblems of the verification can be encoded in the type system. However, this should not be overstressed in a resource-constraint environment: multiple subtyping introduced by interface types would lead to a large set-based type system and its much easier to augment the spare `INVOKEINTERFACE`-instructions by a runtime check.

The approach taken by the Kilo Virtual Machine (KVM) transfers the entire data flow analysis result to the verifer. As a consequence all compliance tests can be done on the fly and no intermediate results are needed. In contrast, Rose's approach minimizes the size of the certificate. It exploits that the verifier computes much data flow information during the check anyway. However, an implementation of Rose's pseudo code algorithm would store intermediate results unnecessarily. Our model determines how long the checker needs certain pieces of information and shows how the checking order can be arranged to achieve a memory optimal pass.

This way, the minimal certificate approach finally outperforms the straightforward KVM approach.

3 Memory Requirements

We discuss the size of the certificate and the temporary memory requirements of the on-card algorithm separately. In the first subsection we compare Rose's certificates to certificates used by another approach based on the general proof carrying code concept: The verifier that is implemented in the Kilo Virtual Machine (KVM) stores the final input frame type of every control flow node in its certificates. With all final input frame types at hand none has to be additionally stored. Therefore, the size of the KVM certificates show the overall memory requirements of this approach. In the second subsection we investigate the expected additional memory consumption for variants of Rose's approach with different storing strategies for temporary frame types.

We focus our investigations on the worst cases because they must remain manageable within the few kilo bytes of memory in a Java Card. The average case is usually not a challenge because between 40% and 60% of the analyzed methods contain only a single control flow node. We have analyzed the standard API supplied by the Java Card Development Kit, two packages of the standard Java Runtime Library, and the jDFA data flow analysis framework on which our certifier prototype is based.

3.1 Comparison between RR-Certificate and Full Certificates

Figure 8 shows the relationship between the largest certificates of the KVM and of Rose's verifier. The sizes of the Rose certificates are derived from

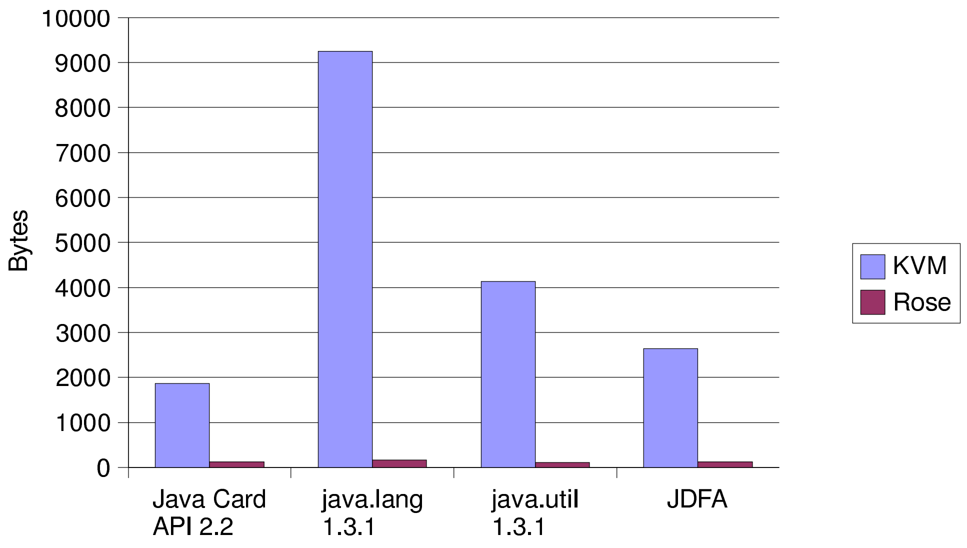

Fig. 8. KVM vs. Rose: Largest Certificate Sizes

Dirk Jansen's prototype implementation of a certifier [Jan03]. We measure the size of frame type by the maximum number of local variable registers of the method and a factor of 2 for the encoding of a single type[6]. The size of the largest certificate of Rose does not exceed 200 Bytes for all code styles. In contrast, the estimated worst cases of the KVM certificates that range in several kilo bytes impose severe challenges on a Java Card. This comparison shows the impressive effects of Rose's minimization idea. Another observation is that the Java Card API code has comparatively small KVM certificates.

3.2 Temporarily Stored Frame Types

We investigate the memory requirements of three different implementation variants of Rose's approach. The *RoseDirect* variant implements the original description canonically. The *RosePass* variant uses our analysis of the lifetime of frame types. The *OptPass* variant additionally exploits that a pre-computed flexible visiting order of control flow nodes can reduce the number of frame types further.

The diagrams in Figure 9 show the estimated memory requirements for the for coexisting final input frame types (I_{max}), final output frame types (O_{max}), and the sum of both types. Every value shows the result of the method that produced the worst case in the category.

The different values for input and output frame types of the *RoseDirect* and *RosePass* show the application of our lifetime model to the fixed verification pass. All figures show a decrease in the number of input frame types because we reuse the memory of input frame types that are no longer needed. In the presence of multi-branches that arise from `SWITCH`-instructions the original

[6] This is an upper bound because there are usually optimization possibilities for the encoding and the number of local variable registers in a frame type.

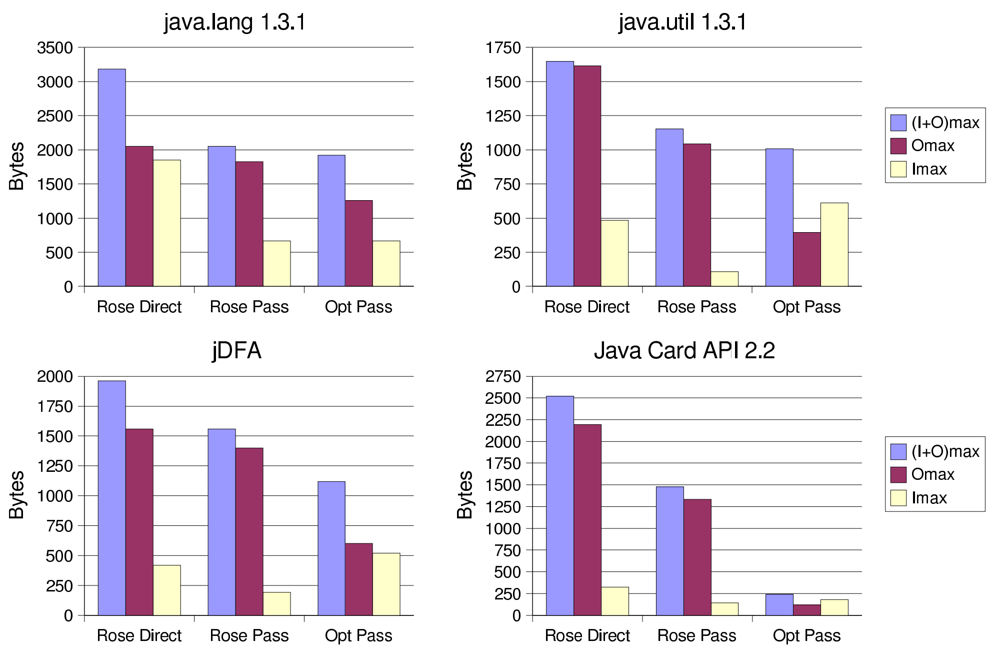

Fig. 9. Worst Cases of the Maximal Memory Consumption

idea of propagating the output frame type to all successor nodes would lead to copies of the same frame type. Again our model determines the lifetime of every output frame type and stores it until its last use. The effect is show by the Java Card API where the number of result frame types of the worst case decreases significantly.

However, all *RosePass* worst cases suffer from a large number of output frame types that arises from the depth first traversal. The comparison to the *OptPass* values show the expected effects if an optimizer determines a flexible traversal in combination with our lifetime model. The worst case of the Java Card API shows the most significant improvement that arises from the early choice of nodes with many predecessors like exception handlers.

Pre-computed lifetimes of frame types, a flexible visiting order, and minimized certificate entries in combination lead to an acceptable memory consumption for the verification process even in the worst cases. We conclude, that the KVM certificates contain a large number of redundant type information that is reduced effectively by the use of minimized certificates.

4 Related Work

Our work combines a couple of different ideas and concepts: *Proof Carrying Code* is applied to conduct *lightweight bytecode verification*, and our model of memory consumption applies a variant of *register allocation*. Furthermore, we solved the verification of uninitialized references by using a *specialized extension of a type system*. The following presentation of related work is organized along these topics.

Proof-Carrying-Code: The concept of Proof-Carrying-Code was originally applied to ensure the safety of user written extensions of operating system kernels [NL96]. With the attached proof the kernel can easily ensure that device drivers do not access certain memory areas.

SafeTSA [ADvRF01] uses separate register planes for values of distinct types. Operations implicitly retrieve their operands from register planes with values of the expected types. The proof of type correctness is encoded in auxiliary instructions that reinterpret types.

Lightweight Bytecode Verification: Other approaches for a verification suitable for Java-Cards were suggested by Leroy [Ler02], Deville [DG02], and Naccache [NTTT02]. The KVM verifier is specified in [BLTY03].

Leroy applies an off card transformation to the code. This transformation guarantees that the whole method can be verified with one single frame type. This frame type is determined on-card by an adapted data flow analysis. However, the transformation affects the runtime behaviour of a method because it increases the code size and the number of local variables slightly.

Deville's and Grimaud's idea is to conduct the classical data flow analysis on the card. They optimize the usage of memory by an efficient type encoding and suggest a memory manangement that evades in the EEPROM of the card whenever a verification requires more memory.

Naccache suggests to reduce the memory consumption of the standard verification process by reusing memory on the stack and local variables which are no longer alive. However, the effects of the additional administration overhead are not obvious, because the approach is not implemented.

Our evaluation in Section 3 compares the KVM approach and the one of Rose. The KVM stores all final input frame types rather than their differences. The differences of the certificate size are significant but the algorithm itself is simpler because the verifier can perform all checks against the certificate immediately.

Register Allocation: We model the memory consumption of the verification process similar to a register allocation method based on interval graphs [HGAM93]. Both approaches determine a *minimal cut in the linear arrangement of a graph.*

Calculation of Program Properties Based on Type Systems: There are several approaches that use type systems for Java Bytecode verification:

Stata [SA98] suggested an extension of a type system which is dedicated to the verification of subroutines.

Grimaud defined the intermediate language *Facade* [GG99] which can express type safe programs only. In this approach incorrect programs do not only produce conflicts in the type system, but the type system becomes a property of the language itself.

Moreover, type systems are used for other program analyses as well. One example is Volpano's approach to flow analysis [VSI96].

5 Conclusion

Rose's proof carrying code approach for bytecode verification on limited devices has served as the starting point of our research. Her approach relies on certificates, which are computed outside the device. Rose has realized her approach for a rather small subset of Java bytecode and has reported some encouraging results concerning the size of the resulting certificates.

In this paper, we have extended Rose's approach to real world Java programs. The two main additions have been extensions to the type system of the verifier and the investigation of the additional on-card memory requirements that arise from the validation of the minimized certificate. The type system has been augmented with multi-dimensional array types and with types that represent references to Java objects not fully initialized yet. Concerning the size of the intermediate results of the on-card verifier we have developed a realistic model of the actual memory requirements, which uses interval graphs to express liveness of memory regions.

The certificates of several real world bodies of Java code do not exceed 200 bytes while the size of the larges KVM certificates ranges in kilo bytes. However, memory usage on the device can be large, when Rose's original description is directly applied to store intermediate results, especially if multi-way branches and exceptions are supported. Memory minimal strategies which use our memory model reduce memory requirements by up to 80%.

All in all, our extended application of proof carrying code techniques puts full Java bytecode verification well within reach of today's severely memory-constrained Java smart cards.

References

[ADvRF01] Amme, Dalton, von Ronne, and Franz. SafeTSA: A Type Safe and Referentially Secure Mobile-Code Representation Based on Static Single Assignment Form. In *SIGPLAN'01 Conference on Programming Language Design and Implementation*, pages 137–147, 2001.

[BLTY03] Gilad Bracha, Tim Lindholm, Wei Tao, and Frank Yellin. *CLDC Byte Code Typechecker Specification*. SUN Microsystems, January 2003.

[DG02] Damien Deville and Gilles Grimaud. Building an "impossible" verifier on a Java Card, 2002.

[FM99] Stephen N. Freund and John C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.

[GG99] Jean-Jaques Vandewalle Gilles Grimaud, Jean-Louis Lanet. Facade: a typed intermediate language dedicated to smart cards. In *Proceedings of the 7th European Software Engineering Conference*, Lecture Notes in Computer Science, pages 476–493. Springer, 1999.

[Grä98] George Grätzer. *General Lattice Theory.* Birkhäuser Verlag, 1998.

[HGAM93] Hendron, Gao, Altman, and Mukerji. Register allocation using cyclic intervall graphs: A new approach to an old problem. Technical report, McGill University, 1993.

[Jan03] Dirk Jansen. Proof-Carrying Code Techniken zur speichereffizienten Verifikation von Java-Bytecode. Master's thesis, University Paderborn, 2003.

[Ler02] Xavier Leroy. Bytecode Verification on Java Smart Cards. In *Software Practice and Experience*, 2002.

[Nec97] George C. Necula. Proof-Carrying Code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 15–17 1997.

[NL96] Necula and Lee. Safe Kernel Extensions without Run-Time Checking. In *Second Symposium on Operating Systems Design and Implementations.* USENIX, 1996.

[NTTT02] Naccache, Tchoulkine, Trichina, and Tymen. Reducing the memory complexity of type-inference algorithms. e-Smart 2002, 2002.

[Ros98] E. Rose. Towards secure bytecode verification on a Java Card. Master's thesis, University of Copenhagen, 1998.

[SA98] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 149–160, New York, NY, 1998.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.