A Type-Safe Mobile-Code Representation Aimed at Supporting Dynamic Optimization At The Target Site

Wolfram Amme wolfram@ics.uci.edu
Niall Dalton ndalton@ics.uci.edu
Michael Franz franz@uci.edu
Jeffery von Ronne jronne@ics.uci.edu

Department of Information and Computer Science
University of California
Irvine, CA 92697-3425

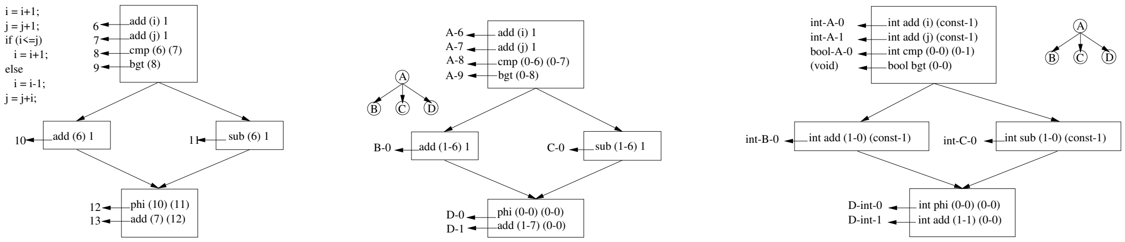
Abstract

We introduce SafeTSA, a type-safe mobile code representation based on static single assignment form. We are developing SafeTSA as an alternative to the Java Virtual Machine, over which it has several advantages: (1) SafeTSA is better suited as input to optimizing dynamic code generators and allows CSE to be performed at the code producer's site. (2) SafeTSA provides incorruptible referential integrity and uses "type separation" to achieve intrinsic type safety. These properties reduce the code verification effort at the code consumer's site considerably. (3) SafeTSA can transport the results of type and bounds-check elimination in a tamper-proof manner.

1 Introduction

The Java Virtual Machine's bytecode format (JVM-code) has become the de facto standard for transporting mobile code across the Internet. However, it is generally acknowledged that JVM-code is far from being an ideal mobile code representation—a considerable amount of preprocessing is required to convert JVM-code into a representation more amenable to an optimizing compiler, and in a dynamic compilation context this preprocessing takes place while the user is waiting. Further, due to the need to verify the code's safety upon arrival at the target machine, and also due to the specific semantics of JVM's particular security scheme, many possible optimizations cannot be performed in the source-to-JVM-code compiler, but can only be done at the eventual target machine—or at least they would be very cumbersome to perform at the code producer's site.

For example, information about the redundancy of a type check may often be present in the front-end (because the compiler can prove that the value in question is of the correct type on every path leading to the check), but this fact cannot be communicated safely in the JVM-code stream and hence needs to be re-discovered in the just-in-time compiler. By "communicated safely", we mean in such a way that a malicious third party cannot construct a mobile program that falsely claims that such a check is redundant. Or take common subexpression elimination: a compiler generating JVM could in principle perform CSE and store the resulting expressions in additional, compiler-created local variables, but this approach is clumsy at best.



(a) in SSA Form

(b) in Reference-Safe SSA Form

(c) in Typed Reference-Safe SSA Form

Figure 1: An Example Program

The approach taken with SafeTSA[1] is radically different from JVM's stack-based virtual machine. The SafeTSA representation is a genuine static single assignment variant in that it differentiates not between **variables** of the original program, but only between unique **values** of these variables. SafeTSA contains no assignments or register moves, but encodes the equivalent information in phi-instructions that model dataflow. Unlike straightforward SSA representations, however, SafeTSA provides intrinsic and tamper-proof *referential integrity* as a well-formedness property of the encoding itself.

Another key idea of SafeTSA is "type separation": values of different types are kept separate in such a manner that even a hand-crafted malicious program cannot undermine type safety and concomitant memory integrity. Interestingly enough, type separation also enables the elimination of type and range checks on the code producer's side in a manner that cannot be falsified.

Finally, SafeTSA programs are transmitted *after* common subexpression elimination, which removes redundancies, leading to smaller and more efficient programs.

2 Referential Integrity

A program in SSA form contains no assignments or register moves; instead, each instruction operand refers directly to the definition or to a "phi" function which models the merging of multiple values based on the control flow. However, straightforward SSA is unsuitable for application domains that require verification of referential integrity in a context of possibly malicious code suppliers. This is because SSA contains an unusually large amount of references needing to be verified, far more than the original source program, making the verification process very expensive.

As an example, consider the program in Figure 1(a). The left side shows a source program fragment and the right side a sketch of how this might look translated into SSA form. Each line in the SSA representation corresponds to an instruction that produces a value. The individual instructions (and thereby implicitly the values they generate) are labeled by integer numbers assigned consecutively; in this illustration, an arrow to the left of each instruction points to a label

[1]SafeTSA stands for Safe Typed Single Assignment Form

that designates the specific target register implicitly specified by each instruction. References to previously computed values in other instructions are denoted by enclosing the label of the previous value in parentheses - in our depiction, we have used (i) and (j) as placeholders for the instructions that compute the initial values of i and j. Since there are no uses of uninitialized variables in Java, such instructions must always exist—in most cases, these would correspond to values propagated from the constant pool.

The problem with this representation lies in verifying the correctness of all the references. For example, value (10) must not be referenced anywhere following the phi-function in (12), and may only be used as the first parameter but not as the second parameter of this phi-function. A malicious code supplier might want to provide us with an illegal program in which instruction (13) references instruction (10) while the program takes the path through (11)—this would undermine referential integrity and must be prevented.

The solution is based on the insight that in SSA, an instruction may only reference values that dominate it, i.e., that lie on the path leading from the entry point to the referencing instruction. This leads to a representation in which references to prior instructions are represented by a pair (l-r), in which l denotes a basic block expressed in the number of levels that it is removed from the current basic block in the dominator tree hierarchy, and in which r denotes a relative instruction number in that basic block. For phi-instructions, an l-index of 0 denotes the appropriate preceding block along the control flow graph (with the nth argument of the phi function corresponding to the nth incoming branch), and higher numbers refer to that block's dominators. The corresponding transformation of the program from Figure 1(a) is given in Figure 1(b).

The resulting representation using such (l-r) value-references provides referential integrity intrinsically without requiring any additional verification besides the trivial one of ensuring that each relative instruction number r does not exceed the permissible maximum. The latter fact can actually be exploited when encoding the (l-r) pair space-efficiently.

3 Type Separation

The second major idea of our representation is *type separation.* While the "implied machine model" of ordinary SSA is one with an unlimited number of registers (=values), SafeTSA uses a model in which there is a separate *register plane* for every type (disregarding, for a moment, the added complication of using a two-part (l-r) naming for the individual registers, and also temporarily disregarding type polymorphism in the Java language—both of these are supported by our format, as explained below). The register planes are created implicitly, taking into account the predefined types, imported types, and local types occurring in the mobile program.

Type safety is achieved by turning the selection of the appropriate register plane into an implied part of the operation rather than making it explicit (and thereby corruptible). In SafeTSA, ***every instruction automatically selects the appropriate plane for the source and destination registers***; the operands of the instruction merely specify the particular register numbers on the thereby selected planes. Moreover, the destination register on the appropriate destination register plane is also chosen implicitly—on each plane, registers are simply filled in ascending order.

For example, the operation *integer-addition* takes two register numbers as its parameters, *src1* and *src2*. It will implicitly fetch its two source operands from register *integer-src1*, *integer-src2*, and deposit its result in the *next available integer register* (i.e., the register on the integer plane, having an l-index of zero and an r-index that is 1 greater than the last integer result produced in this basic block). There is no way a malicious adversary can change integer addition to operate on operands other than integers, or generate a result other than an integer, or even cause "holes" in the value numbering scheme for any basic block. To give a second example, the operation *integer-compare* takes its two source operands from the integer register plane and will deposit its result in the next available register on the Boolean register plane.

SafeTSA combines this type separation with the concept of referential integrity discussed in the previous section. Hence, beyond having a separate register plane for every type, we additionally have one such complete two-dimensional register set for every basic block. The results of applying both type separation and reference safe numbering to the program fragment of Figure 1(a) are shown in Figure 1(c).

4 Construction of Memory Safety

For every reference type *ref*, our "machine model" provides a matching type *safe-ref* that implies that the corresponding value has been null-checked. Similarly, for every array *arr* we provide a matching type *safe-index-arr* whose instances may assume only values that are index values within legal range[2]

Null-checking then becomes an operation that takes an explicit *ref* source type and an explicit register number on the corresponding register plane. If the check succeeds, the *ref* value is copied to an implicitly given register (the next available) on the plane of the corresponding *safe-ref* type, otherwise an exception will be generated. Similarly, the index-check operation will take an array and the number of an integer register, check that the integer value is within bounds, and if the check succeeds, copy the integer value to the appropriate *safe-index* register plane.

The beauty of this approach is that it enables the transport of null-checked and index-checked values across phi-joins. Phi-functions are strictly type-separated: all operands of a phi-function, as well as its result, always reside on the same register plane. Whenever it is necessary to combine a *ref*-type and the corresponding *safe-ref* type in a single phi-operation, the *safe-ref* type needs to be *downcast* to the corresponding unsafe *ref* type first. The downcast operation is a modeling function of SafeTSA and will not result in any actual code on the eventual target machine.

Null-checking and index-checking can be generalized to include all type-cast operations: an *upcast* operation involves a dynamic check and will cause an exception if it fails. In the case of

[2] Because of the need to support dynamically-sized arrays, safe-index types are actually bound to array *reference values* rather than to their static types.

success, it will copy the value being cast to the next available free register on the plane of the target type (only the dynamic check will result in actual code at the target machine, but not the copy operation). The *downcast* operation never fails and will never result in any actual target code.

All memory operations in SafeTSA require that the storage designator is already in the *safe* state; i.e., these operations will take operands only from the register plane of a safe-ref or safe-index type, but not from the corresponding unsafe types. There are four different primitives for memory access:

getfield ref-type object field
setfield ref-type object field value
getelt array-type object index
setelt array-type object index value

where *ref-type* denotes a reference type in the type table, *object* designates a register number on the plane of the corresponding safe-ref type, *field* is a symbolic reference to a data member of *ref-type*, and *value* designates a register number **on the plane corresponding to the type of *field***. Similarly, for array references, *object* designates a register on the plane of the array type that contains the array's base address and *index* designates a register on the array's safe-index plane that contains the index.

The *setfield* and *setelt* operations are the only ones that may modify memory, and they do this in accordance with the type declarations in the type table. This is the key to type safety: most of the entries in this type table are not actually taken from the mobile program itself and hence cannot be corrupted by a malicious code provider. While the pertinent information may be included in a mobile code distribution unit to ensure safe linking, those parts of the type table that refer to primitive types of the underlying language or to types imported from the host environment's libraries are always generated implicitly and are thereby tamper-proof.

This suffices in guaranteeing memory-safety of the host in the presence of malicious mobile code. In particular, in the case of Java programs, SafeTSA is able to provide the identical safety semantics as if Java source code were being transported to the target machine and compiled and linked locally. Our prototype compiler is capable of encoding all of this information in approximately the same space as the equivalent Java bytecode instructions

5 Primitive Operations

The preceding discussion mentioned built-in operations such as *integer-add* and *integer-compare*, bringing up the question of which primitives are actually built into our machine model. In fact, primitive operations in SafeTSA are subordinate to types and there are only two generic instructions:

primitive base-type operation operand1 operand2...
xprimitive base-type operation operand1 operand2...

where *base-type* is a symbolic reference into the type table, *operation* is a symbolic reference to an operation defined on this type, and *operand1... operandN* designate register numbers on the respective planes corresponding to the parameter types of the operation. In each case, the result is deposited into the next available register on the plane corresponding to the result type of the operation.

The only difference between *primitive* and *xprimitive* concerns exceptions. Operations that may potentially cause an exception (such as integer divide) must be referenced using the *xprimitive* instruction. Each occurrence of an xprimitive instruction in a basic block automatically leads to an additional incoming branch to the phi-functions in the appropriate exception-handling join-blocks.

Note that it is entirely up to the type system of the language being transported by SafeTSA to specify what operations on which types may actually cause exceptions. For example, the type *Java.lang.primitive-integer* provides *add*, *subtract*, and *multiply* among its primitives and *divide* among its xprimitives, but another language that is less lenient about arithmetic overflow conditions might define all four operations *add,subtract*, *multiply*, and *divide* only as xprimitives for its particular integer type.

There are no primitive operations for accessing constants and parameters. Instead, these are implicitly "pre-loaded" into registers of the appropriate types in the initial basic block of each procedure. Note that this "pre-loading" is yet another example of an operation that merely occurs on the SafeTSA level and that doesn't correspond to any actual code being generated on the target machine.

6 Method Invocation

Just as a set of operations is associated with each primitive type, a table of methods can be associated with a reference type. This table is a built from local method definitions and from a list of imported methods. Two primitives provide method invocation with and without dynamic dispatch:

xcall base-type receiver method operand1 operand2...
xdispatch base-type receiver method operand1 operand2...

where *base-type* identifies the static type of the receiver object, *receiver* designates the register number of the actual receiver object on the corresponding plane, *method* is a symbolic reference to the method being invoked, and *operand1 ... operandN* designate register numbers on the respective planes corresponding to the parameter types of the method. The result will be deposited into the next available register on the plane corresponding to the result type of the method.

The symbolic *method* may reference any method which can be invoked on the static type denoted by *base-type*. For xcall, this determines the actual code that will be executed, but for xdispatch, it only determines a slot in the static type's dispatch table that will be polymorphically associated with a method by the dynamic type of the instance referenced by *receiver*. For Java programs, the code producer is required to resolve overloaded methods and insert explicit *downcast* operations for any operands whose static type does not match the type of a method's corresponding formal parameter.

7 Implementation Status And Preliminary Results

We have been building a system consisting of a compiler that takes Java source files and translates them to the SafeTSA representation, and a dynamic class loader that takes SafeTSA code distribution units and executes them on SPARC using on-the-fly code generation.

Currently our compiler can process programs written in the complete Java language and produce SafeTSA intermediate code. The class loader and dynamic code generator do not yet produce competitive results, but work on them has progressed sufficiently that we are confident of the correctness of our approach. Our front-end is based on the Pizza[9] compiler.

SafeTSA provides a safe mechanism for the transportation of optimized code. We take advantage of this fact to perform optimizations that will reduce the size and eventually the execution time of the transmitted code. As a proof of concept, we currently implement constant propagation, common subexpression elimination and dead code elimination at a local level.

In the following measurements we compare the size and number of instructions for programs compiled to Java byte-code, SafeTSA, and optimized SafeTSA. As benchmarks, we use programs from the Sun Java Development Kit. These include classes from the Java compiler, javac, the Java interpreter, java, as well as some classes from the Math and Linpack packages. The latter classes are used to demonstrate reductions of array checking instructions. Where we compare to Java, we refer to byte-code produced using version 1.2.2 of Sun javac.

The first three columns of Figure 2 show the sizes and numbers of instructions in SafeTSA files as compared to Java class files—in most cases SafeTSA has less than 40% of the number of instructions that Java byte-code requires. The above-mentioned optimizations can reduce significantly the number of instructions in SafeTSA form, by more than 10% in most cases, and up to 19% for some programs. Constant propagation leads to an improvement of only 1% or 2% in the program size. Dead code elimination generally is most effective in reducing the number of phi instructions - between 3% and 7% of the number of instructions at most. The majority of the code size reduction is due to common subexpression elimination. In our measurements the reduction due to this was between 5% and 14%. The size of the SafeTSA files lie between 81% and 97% of the Java byte-code files. This is closer to Java byte-code size than would be expected from the difference in instruction counts because symbolic information accounts for 60% of the space required in SafeTSA.

Figure 2 also gives detail on the practical influence of optimizations performed prior to transmission of the code. It contains information on the reduction of phi instructions, null-checks, and array checks. These are of particular interest as they lead to less information that needs be transmitted as well as eventually to faster execution. As can be seen, the number of phi instructions was reduced by more than 30% in most cases. Surprisingly, we can eliminate and safely transport a program with, in most cases, 30% fewer null-checks, and in some cases up to 70% reduction is

Class Name	Instruction Count			Phi Instruction			Null-Checks			Array-Checks		
	JVM	STSA	Opt.	with	w/o	Δ%	with	w/o	Δ%	with	w/o	Δ%
sun.tools.javac												
BatchEnvironment	2516	1640	1462	131	75	-43	425	206	-51	11	9	-18
BatchParser	394	286	276	19	16	-16	53	46	-13	N/A	N/A	N/A
Main	1734	1410	1281	330	301	-9	246	155	-37	53	49	-8
SourceClass	5396	3869	3381	356	200	-44	926	605	-35	N/A	N/A	N/A
SourceMember	1735	1333	1169	221	123	-44	327	261	-20	12	12	N/A
sun.tools.java												
BinaryAttribute	121	77	64	12	7	-42	19	12	-37	N/A	N/A	N/A
BinaryClass	873	617	527	56	35	-37	131	62	-52	2	2	N/A
BinaryCode	233	77	62	6	3	-50	15	4	-73	1	1	N/A
Scanner	4240	3912	3779	58	47	-19	101	58	-42	8	8	N/A
Parser	3578	1732	1614	351	263	-25	196	151	-23	11	11	N/A
sun.math												
BigDecimal	935	702	612	54	35	-35	119	73	-39	26	16	-38
BigInteger	5638	3463	3080	382	296	-23	451	257	-43	188	169	-10
BitSieve	277	153	140	18	15	-17	15	11	-26	3	3	N/A
MutableBigInteger	3415	2223	1925	205	169	-18	400	172	-52	136	132	-3
Linpack												
Linpack	1097	638	424	138	88	-36	70	43	-39	67	54	-19

Figure 2: Number of Phi-, Null-Check and Array-Check instructions before and after optimization.

achieved. Perhaps even more surprisingly, our optimizations are based only on knowledge of safe values and common subexpression elimination and not on any context sensitive analysis. Most of our benchmarks do not include a lot of array manipulation. However, for those that do, we see a reduction of up to 38% in the number of array check instructions. Note that our SafeTSA sizes contain explicit null-checks, type-checks, and index checks, while these need not be transported in Java byte-code, but also cannot be removed as a consequence.

8 Related Work

It is difficult to generate quality native code from JVM-code[6]. This situation is exacerbated in JIT compilers: because they need to operate while a user is waiting, they often need to favor compilation speed over code quality (e.g. by using linear scan register allocation[10] rather than graph coloring.) JVM is also hard to verify. In particular, checking that all operand accesses to the stack are valid requires a data flow analysis. SafeTSA promises to alleviate both of these concerns.

In the last few years, several native code optimizing Java compilers that use an intermediate representation based on SSA form have been developed: the Swift compiler [11], Marmot [5], and the HotSpot Server compiler [1]. Jalapeno [4, 2, 3] also uses SSA for certain optimizations.

The intermediate representation for Microsoft's recently announced ".NET" platform offers an improvement over the stack based virtual machine, allowing for a second SSA form description to be added to the stack based representation. Not all of .NET's details have been released yet, and it is unclear what provisions .NET may have to guaranty the consistency between the stack and

SSA based representations, as well as the type safety of the SSA based representation.

Like our approach, proof carrying code (PCC)[8] aims at the safe execution of untrusted, possibly mobile, code. The target machine receives native code along with a proof that the native code complies with the target machine's security policy. Although PCC can be used to support arbitrarily complex security policies, those for which proofs can be made automatically are similar to the guarantees enforced by SafeTSA.

TAL (Typed Assembly Language) [7] guarantees a similar level of safety by overlaying a type system onto the machine code. Their compiler is also noteworthy for maintaining typing through several compiler phases and intermediate representations, some of which are similar to SSA.

9 Conclusion and Outlook

The Java Virtual Machine's instruction format is not very capable of transporting the results of program analyses and optimizations. As a consequence, when Java bytecode is transmitted to another site, each recipient must repeat most of the analyses and optimizations that could have been performed just once at the origin. The main reason why Java bytecode has these deficiencies is to allow verification by the recipient.

We have designed an alternative mobile-code representation that overcomes these limitations of the JVM bytecode language. Our representation, SafeTSA, can provide the same security guarantees as the Java Virtual Machine, but it can express most of them statically as a well-formedness property of the encoding itself. SafeTSA thereby obviates the need for an expensive dataflow analysis at the code recipient's site.

Further, SafeTSA preserves control and dataflow information as well as full typing information for each intermediate result. It is based on Static Single Assignment form, a representation that is also used internally by several state-of-the-art research compilers for Java. As a consequence, SafeTSA it is far easier to parse into a form useful for code optimization than JVM-code. Also, SafeTSA removes the need to perform CSE and type/range check elimination at the code receiver's side, genuinely enabling shifting this workload to the code's producer without jeopardizing safety. (Our current prototype eliminated approximately 30% of the null checks and up to 38% of the range checks when tested on several classes of Sun JDK's javac compiler.)

It is probably only a matter of time until the Java Virtual Machine will be displaced by alternative mobile-code transportation formats that better support optimization at the code receiver's site. Programmers will still be writing mobile programs using the Java source language (and alternative languages such as C#), but rather than compiling them into JVM-code, they will be using these better alternatives. The authors believe to have identified such an alternative in SafeTSA.

10 Acknowledgements

The authors would like to thank Peter Housel for his helpful comments on this paper and Michael Phillipsen for providing the source code of the Pizza compiler. This effort is partially supported by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-99-1-0536, by the National Science Foundation under grant CCR-9901689, and by the California MICRO Program, project No. 99-039 with a gift from Microsoft Research.

References

[1] Sun Hotspot compiler for Java. *http://java.sun.com/products/hotspot/*.

[2] B. Alpern, C. R. Attanasio, et al. The Jalapeno virtual machine. *IBM System Journal*, 39(1), February 2000.

[3] B. Alpern, A. Cocchi, D. Lieber, M. Mergen, and V. Sarkar. Jalapeno - a compiler-supported java virtual machine for servers. *Workshop on Compiler Support for Software System (WCSSS 99)*, May 1999.

[4] M. Arnold, S. Fink, et al. Adaptive optimization in the Jalapeno JVM. *ACM OOPLSA 2000.*

[5] R. Fitzgerald, T. B. Knoblock, et al. Marmot: An optimizing compiler for Java. Microsoft Technical Report 3, March 2000.

[6] J. Hummel, A. Azevedo, D. Kolson, and A. Nicolau. Annotating the Java bytecodes in support of optimization. *Concurrency: Practice and Experience*, 9(11):1003–1016, Nov. 1997. Special Issue: Java for computational science and engineering — simulation and modeling II.

[7] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Trans. Prog. Lang. and Sys.*, 23(3):528–569, May 1999.

[8] G. C. Necula. Proof-Carrying Code. In *POPL '97*, Paris, France, Jan. 1997.

[9] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97*, pages 146–159, Paris, France, 15–17 Jan. 1997.

[10] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Prog. Lang. and Sys.*, 21(5):895–913, September 1999.

[11] D. J. Scales, K. H. Randall, S. Ghemawat, and J. Dean. The Swift Java Compiler: Design and Implementation. WRL Research Report 2000/2, Compaq Research, April 2000.