SafeTSA: A Type Safe and Referentially Secure Mobile-Code Representation Based on Static Single Assignment Form

Official ACM version at http://doi.acm.org/10.1145/378795.378825.

Wolfram Amme
wolfram@ics.uci.edu

Niall Dalton
ndalton@ics.uci.edu

Jeffery von Ronne
jronne@ics.uci.edu

Michael Franz
franz@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. Despite these advantages, SafeTSA is more compact than Java bytecode.

1. INTRODUCTION

The Java Virtual Machine's bytecode format ("Java bytecode") has become the de facto standard for transporting mobile code across the Internet. However, it is generally acknowledged that Java bytecode is far from being an ideal mobile code representation—a considerable amount of preprocessing is required to convert Java bytecode into a representation more amenable to an optimizing compiler, and in a dynamic compilation context this preprocessing takes place while the user is waiting. In addition, it has been shown that the rules for bytecode verification don't exactly match those of the Java Language Specification, so that there are certain classes of perfectly legal Java programs that are rejected by all compliant bytecode verifiers[29].

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-Java bytecode 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 Java bytecode 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 Java bytecode could in principle perform CSE and store the resulting expressions in additional, compiler-created local variables, but this approach is clumsy.

The approach taken with SafeTSA[1] is radically different from Java'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 explicit assignments to local and temporary variables, 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.

ACM COPYRIGHT NOTICE. Copyright © 2001 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or permissions@acm.org.

[1]SafeTSA stands for Safe Typed Single Assignment Form

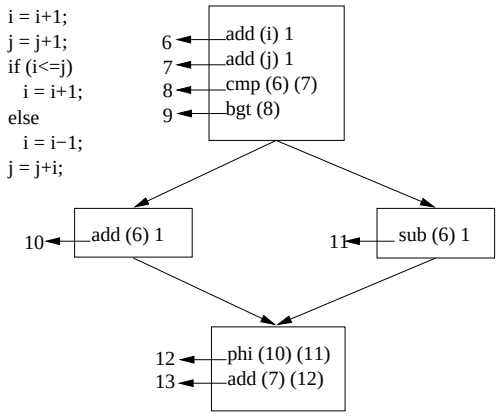

Figure 1: Program in SSA Form

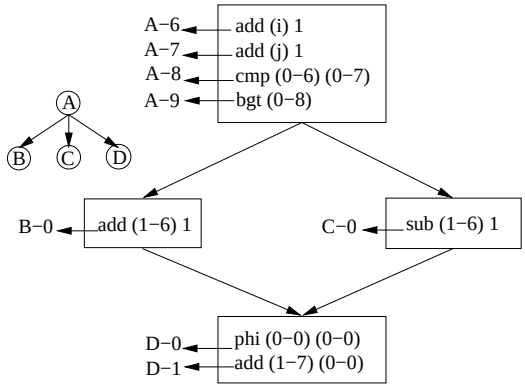

Figure 2: Reference-Safe SSA

Finally, SafeTSA programs are transmitted *after* high-level common subexpression elimination (address calculations are not explicit at this stage), which removes redundancies and thereby often leads to smaller and more efficient programs.

The following sections introduce various aspects of the SafeTSA encoding, discuss the current status of our implementation, present preliminary results, and give some references to related work.

2. REFERENTIAL INTEGRITY

A program in SSA form contains no explicit 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. 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 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 is given in Figure 2.

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 doesn't 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 (Figure 3).

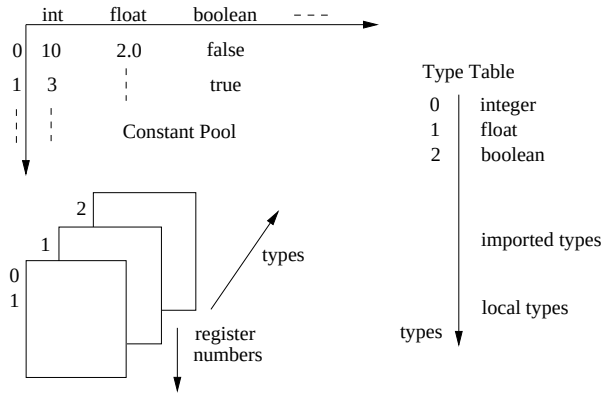

Figure 3: Implied Machine Model of SafeTSA

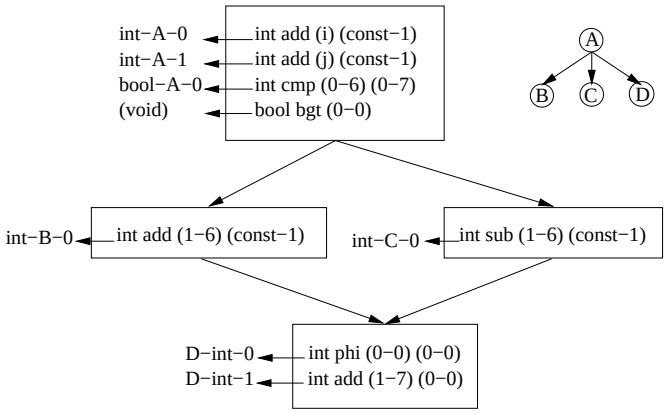

Figure 4: Type-Separated Reference-Safe SSA

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 naming scheme for any basic block (a contiguous numbering facilitates compact externalization). 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 are shown in Figure 4.

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. This is always translated into an explicit runtime null-check on the target machine and is the only way to convert a normal *ref* value into a *saferef* value. Similarly, the index-check operation takes an array and the number of an integer register, checks that the integer value is within bounds (with a concomitant runtime check), and if the check succeeds, copies 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 success, it will copy the value being cast to the next available free register on the plane of the target type (the dynamic check will always result in actual target machine code; the copy operation will disappear if the register allocator manages to coalesce the affected live ranges). The *downcast* operation, however, is only allowed where the type cast is known to be statically safe (e.g. casts from a class's *safe-ref* type to its *ref* type, or from a class's *ref* type to

[2]Because of the need to support dynamically-sized arrays, safe-index types are actually bound to array *values* rather than to their static types. A more detailed discussion of this issue can be found in Appendix A.

its superclass's *safe-ref*), and therefore has no runtime overhead.

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.

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

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 using on-the-fly code generation.

Currently our compiler can process programs written in the Java language and produce SafeTSA intermediate code. Work is progressing on JIT compilers targeting the SPARC and Intel platforms, and we are confident that our approach will yield a competitive runtime system. Our front-end is based on the Pizza[25] compiler.

The front-end of the compiler takes as input either Java classes or packages in source form and for each class in the input, produces a file containing a compressed version of the SafeTSA representation of that class. The transformation of a Java class to its SafeTSA representation is performed in three main steps. After successful syntactic and semantic analysis, the program is transformed into a Unified Abstract Syntax Tree (UAST). Next, an SSA generator transforms the UAST into a SafeTSA graph, which is finally encoded into a binary stream and written to a file.

The motivation for the use of an UAST is the future extensibility of the system to handle input languages other than Java. In the current implementation, the UAST combines the structural elements of Java, Fortran95, and Ada95 in a single data structure. Therefore, it will be easy to support the compilation of Fortran95 and Ada95 programs in the future. The principles behind the UAST structure can be seen as a generalization of the Abstract Syntax Tree used in Crelier's OP2[9] compiler. The essential idea in this type of Abstract Syntax Tree is to integrate the dominator and control flow information in the same structure. The use of a binary tree simplifies code generation and optimization.

Many algorithms exist for the transformation of a high level program into its corresponding SSA form and for the determination of the dominator relation[10, 11, 28, 21]. Our compiler takes the method of Brandis and Mössenböck [6], which constructs the SSA form and the dominator relation of the program in a single pass from the source code, and adapts this method to work on the UAST. Furthermore, we improved the handling of *return*, *continue* and *break* instructions to avoid inserting phi nodes where there are fewer than two infeasible paths. To eliminate superfluous phi instructions, we perform dead code elimination based on the calculation of live variables as suggested by Briggs et al. [7] leading to a reduction of 31% on average in the number of phi instructions.

In our implementation, transformation from the UAST into SafeTSA form is limited to expressions and assignments. This leads to the partitioning of the SafeTSA graph into a Control Structure Tree, i.e. the structural part of the UAST, and the SafeTSA part[3]. From the Control Structure Tree a coherent control flow graph and dominator tree can be derived efficiently, facilitating high quality code generation by providing high level program and blocks of SafeTSA code. This, for example, eases the determination of induction variables for use in software pipelining [30].

To enforce correct semantics of Java threads, only local variables can be considered as values, in contrast to global variables which must be accessed via getfield or setfield instructions as the contents of such variables may be changed at any time. A problematic feature of translating Java to SSA form is the encoding of the try-catch-finally construct[13]. In our approach, at any point where an exception may occur, we split basic blocks into linked subblocks where each of the subblocks has only a single entry and exit point. An implicit control-flow edge is created from each potential point of exception to a special exception-handling phi-node that is the predecessor of all of the catch blocks. Because SafeTSA always has a default, possibly empty, catch block, the finally block is a successor of the catch blocks and also any break, continue, or return nodes inside the try block.

SafeTSA has been designed so that it can be externalized as a sequence of symbols, where each symbol is chosen from a finite set determined only by the preceding context. The encoder transmits these symbols in three phases. First, it encodes the Control Structure Tree (CST) as a sequence of productions of its grammar. Second, it transmits the SafeTSA blocks in a fixed order, derived from the CST, corresponding to a pre-order traversal of the dominator tree. Each instruction is encoded as opcodes, types, and operands as were discussed earlier, but only the type information of phi-nodes is encoded during this stage. Third, it transmits the phi-node operands. Phi-node operands are postponed, because they may reference instructions that do not precede them in the pre-order traversal. Since each of these symbols is chosen from a finite set, any dictionary encoding scheme can be used to convert the symbol sequence into a binary stream. Our present prototype uses a simple prefix encoding, which is similar to what would result from using Huffman[17] encoding with fixed equal probabilities for all symbols.

8. PRELIMINARY RESULTS

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.

Unfortunately, in general, some data dependencies are hidden within the SSA form of a program due to memory accesses, i.e. field and array operations. For example, a store operation may write into a field that will later be read by a load operation. Although the SSA form will not express the data dependence between these operations explicitly, the original program's semantics must be enforced[4]. To guarantee that all data dependencies are maintained during the optimization phase, the compiler has to trace the memory accesses of the program.

[3]Java has short-circuit operators that alter control-flow. These are handled by translation into if-else statements and allowing these if-else statements in all expression contexts.

[4]Note that type safety is not the issue here, only read-write ordering of data items that all have the same type. In SafeTSA, type safety is enforced by type separation.

Class Name	File Size			Number of Instructions		
	Java Bytecode	SafeTSA	SafeTSA optimized	Bytecode	SafeTSA	SafeTSA optimized
sun.tools.javac						
BatchEnvironment	18399	14605	13931	2516	1640	1462
BatchParser	4939	3832	3796	394	286	276
CompilerMember	1192	401	397	50	29	28
ErrorMessage	305	90	90	14	3	3
Main	11363	11265	10813	1734	1410	1281
SourceMember	13809	11888	11246	1735	1333	1169
sun.tools.java						
AmbiguousClass	422	147	147	18	5	5
AmbiguousMember	751	217	214	46	13	12
ArrayType	837	260	260	35	15	15
BinaryAttribute	1716	944	854	121	77	64
BinaryClass	8156	6008	5727	873	617	527
BinaryCode	2292	1536	1479	133	77	62
Parser	23945	23678	22901	2578	1732	1614
Scanner	10540	11695	11222	4240	2912	2779
sun.math						
BigDecimal	6140	5309	4926	935	702	612
BigInteger	19309	20009	18393	5638	3463	3080
BitSieve	1557	1155	1080	277	153	140
MutableBigInteger	9667	10757	9823	3415	2223	1925
SignedMutableBigInteger	896	427	424	116	53	52
Linpack						
Linpack	3336	3512	3042	1097	638	524

Figure 5: SafeTSA class files compared to Java class files.

There are many ways in which to solve this problem, we use the approach of introducing a special variable *Mem* which describes the state of the memory. During the optimization phase, a store to memory will produce a new value for *Mem*, reflecting the fact that the memory has been updated. Additionally, load operations take an extra parameter which is the current value of the *Mem* variable. As the current implementation does not include inter-procedural optimizations, global updating of the memory will be approximated by having each function call return an updated value of the *Mem* variable. Furthermore, if the current value of *Mem* is different on two incoming edges of a block, then a phi node must be inserted in the join block to merge the values into a new current value for *Mem*. This artificial mechanism expresses all data dependencies, formerly implicit in the SSA form, in an approximated, conservative manner.[5] This mechanism is used solely during the optimization phase and is not part of the transmitted code, and it allows common subexpression elimination while respecting the semantics of the original program and the typing rules of SafeTSA.

In our measurements we compare the size and number of instructions for programs compiled to Java bytecode, 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 bytecode produced using version 1.2.2 of Sun javac using options to generate no debug information (javac -g:none).

Figure 5 shows 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 bytecode 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 instruction count 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 is usually smaller than the respective Java bytecode files and sometimes substantially so. We know of two reasons contributing to the failure of file sizes to show as much of an improvement as instruction count: a substantial amount of each file consists of symbolic linking information and constants, and many SafeTSA instructions have multiple operands where the corresponding bytecode would utilize separate stack manipulation instructions.

[5] A more sophisticated alias analysis, for example, could disambiguate between two references a[i] and a[j] wherever it could prove that $i \neq j$, thereby finding more common subexpressions and leading to an even denser encoding.

Class Name	Phi Instructions			Null-Checks			Array-Checks		
	Before	After	Δ%	Before	After	Δ%	Before	After	Δ%
sun.tools.javac									
BatchEnvironment	131	75	-43	425	206	-51	11	9	18
BatchParser	19	16	-16	53	46	-13	N/A	N/A	N/A
Main	330	301	-9	246	155	-37	53	49	8
SourceClass	356	200	-44	926	605	-35	N/A	N/A	N/A
SourceMember	221	123	-44	327	261	-20	12	12	N/A
sun.tools.java									
BinaryAttribute	12	7	-42	19	12	-37	N/A	N/A	N/A
BinaryClass	56	35	-37	131	62	-52	2	2	N/A
BinaryCode	6	3	-50	15	4	-73	1	1	N/A
Scanner	58	47	-19	101	58	-42	8	8	N/A
Parser	351	263	-25	196	151	-23	11	11	N/A
sun.math									
BigDecimal	54	35	-35	119	73	-39	26	16	38
BigInteger	382	296	-23	451	257	-43	188	169	10
BitSieve	18	15	-17	15	11	-26	3	3	N/A
MutableBigInteger	205	169	-18	400	172	-52	136	132	3
Linpack									
Linpack	138	88	-36	70	43	-39	67	54	19

Figure 6: Number of Phi-, Null-Check and Array-Check instructions before and after optimization.

Figure 6 gives more 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 achieved, using the conservative CSE mechanism described above. 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 all of our SafeTSA sizes contain explicit null-checks, type-checks, and index checks, while these need not be transported in Java bytecode, but also cannot be removed as a consequence.

Although these are encouraging results, we can identify much scope for improvement. A dramatic improvement would be the integration of alias information into the memory handling. This can be done, for example, with a simple form of field analysis as described in [15], partitioning *Mem* by field name. An alternative is the integration of more precise alias analysis techniques, e.g. [12, 3], which will lead to a typed partition of the memory. Naturally, the use of interprocedural analyses would lead to even better results. Note that all of these alias analyses could be performed safely due to our construction of memory safety.

9. RELATED WORK

Because of the simplicity of code generation, 0-address architectures have been a popular intermediate target architecture in compiler design. A well known example of such a stack-based intermediate language is pCode [24], used for code generation in the earlier Pascal compilers. pCode has recently received renewed attention because of its influence on the Java Virtual Machine.

Java's platform independent bytecode format is either interpreted by a virtual machine (resulting in unacceptable performance) or compiled by a just-in-time compiler at the target site. Unfortunately, such just-in-time compilation must occur in real time, while an interactive user is waiting. The JVM's particular design leads to quite expensive verification and initial bytecode analysis phases, often leaving insufficient time to perform good code generation, or forcing the use of optimization algorithms that favor speed over the best achievable code quality; for example, the use of linear scan register allocation instead of graph coloring [26].

The difficulty of processing Java bytecode partly results from the underlying stack model, as well as the fact that many bytecode operations intrinsically include sub-operations, e.g. iaload includes the address computation, array checks and the actual load of the array element. The stack-based model which limits the access to the top element of the stack, prevents the reuse of operands and code reordering [18]. In our SafeTSA approach we have split these composed bytecode operations into elementary units, which—after performing some standard optimizations—leads to a significant reduction in code size and eventually in execution time. Further, SafeTSA's is more appropriate for structural optimizations at the consumer side.

A further significant performance problem with the Java virtual machine is the time consuming verification phase. In particular, checking that all operand accesses to the stack are valid - which requires a data flow analysis - decreases the runtime of applications significantly. In SafeTSA this verification phase is done by checking if a value has already been defined, which can be implemented using simple

counters holding the numbers of defined values for each type in each basic block.

There are many JIT compilers for Java; two examples are HotSpot and Jalapeno. The HotSpot Server compiler [16] developed at Sun uses an SSA-based internal representation similar to the one described in Click's dissertation [8]. It turns Java exceptions into explicit control flow in its IR and uses a full flow pass to discover types from the Java bytecode. In contrast our representation always explicitly marks all types, and therefore no type inference is required.

Jalapeno [5, 1, 2] is an extensive Java virtual machine/dynamic compilation system from IBM research that is itself written in Java and indeed even uses its own services. Jalapeno is another compiler that uses SSA internally, but unlike HotSpot, it is not entirely based on SSA but uses SSA only for flow sensitive optimizations. Jalapeno uses three different levels of IR, a high-level form that is essentially equivalent to bytecode, an intermediate-level form that exposes the underlying object model, and a low-level form that is very close to the target machine.

In the last few years, several native code optimizing Java compilers that use an intermediate representation based on SSA form have been developed. Each of them requires either Java source code or Java bytecode as input and produces native machine code as output. However, there is no compiler that conserves the information of the SSA based intermediate representation in any kind of class file that could support a dynamic code generation process. Below we will discuss in more detail two optimizing compilers of particular interest.

The Swift compiler [27] has been designed and implemented at the Western Research Laboratory of Compaq Computer Corporation. Swift translates Java bytecode to optimized machine code for the Alpha architecture and uses SSA form for its intermediate representation. The intermediate language used by the compiler is relatively simple, but allows for straightforward implementation of all standard scalar optimizations and other advanced optimization techniques, i.e. method resolution and inlining, interprocedural alias analysis, elimination of run time checks, object inlining, stack allocation of objects, and synchronization removal. Each value in the SSA graph also has a program type. Comparable with our approach, the type system of Swift can represent all of the types present in Java program. In contrast to our approach the Swift intermediate representation contains no structural information, i.e. the control structure tree of our SafeTSA graphs. Also, compared to the instruction set of our SafeTSA, the instructions used by Swift are very specialized and adapted to its target architecture.

Marmot [13] is a research compiler from Microsoft that transforms Java bytecode into native machine code. The organization of the compiler can be divided into three parts: conversion of Java class files to a typed high level intermediate representation based on SSA form, high level optimization, and code generation. Type information in the high level representation of the Marmot compiler (there is also a low-level IR) is derived by type elaboration. This process produces a strongly-typed intermediate representation in which all variables are typed, all coercion and conversions are explicit, and all overloading of operators is resolved. Marmot doesn't support Java's essential facility of *dynamic class loading*.

The intermediate representation for Microsoft's recently announced ".NET" platform is a further improvement of the stack based virtual machine. It has a provision for including a second description based on SSA form. This approach assumes an external authentication-based security mechanism because it is not safe: First, there is no guarantee that the program represented as a virtual-machine program corresponds to the one represented in SSA form—the two representations are completely independent of each other and it appears that the loader on the target platform may elect to use either format if both are present. Second, the SSA part of the representation has no provisions for safety and is apparently never verified.

The Architecture Neutral Distribution Format (ANDF[4]), originally developed by the Defence Research Agency in the UK (DRA) is also a tree based representation, the TDF intermediate language. TDF is a tree structured language, defined as a multi-sorted abstract algebra, which preserves more program structure information that other languages. However, it has a weaker type system than typical high level languages. It was originally designed for the compilation of sequential languages such as C and Lisp. Indeed the overall level of the language is similar to C but also includes support for other features such as for garbage collection. Programs represented in ANDF are compiled to native code at installation time. As such, ANDF was designed solely as a distribution format rather than a mobile code format.

Slim Binaries, an intermediate representation developed by Franz and Kistler[14, 20] are also based on a tree structured language. In contrast to ANDF, Slim Binaries were designed with mobile code applications in mind and to be suitable for dynamic code generation on target machines. Slim Binaries have been shown to be much smaller than ANDF and Java bytecode files - an important feature for mobile code applications. Slim Binaries have been used successfully in studying continuous program optimization[19]. Unlike SafeTSA, Slim Binaries are based on encoding the source-level abstract syntax tree of a program. As a consequence of being so far removed from the internal representations typically used in compilers, Slim Binaries cannot communicate the results of program analyses and target machine independent optimizations as well as SafeTSA can.

Proof carrying code[23] is a new subject of research aimed at the safe execution of untrusted, possibly mobile, code. There is a burden of proof on the code to be executed, to obey the target site's security policy. Typically this is in the form of an attached proof. Upon validation of the proof at the target site, the code is assumed safe to run and no dynamic checks need be performed. Proof carrying code provides memory and type safety similar to SafeTSA, however, we do not need to generate, transport and verify proofs. SafeTSA is safe by construction, and cannot be manipulated to give unsafe programs.

TAL (Typed Assembly Language) [22] uses an approach to type safety that is semantically equivalent to our concept

of type separation. TAL annotates assembly language or object code with type information, memory management primitives, and a sound set of typing rules. The type annotations can be checked before assembly or execution of the code to verify its memory, control flow and type safety. A particular aim of the TAL project is the compilation of type-safe functional languages based on the polymorphically typed second order lambda calculus—known as System F—to a typed target language via a sequence of type-preserving transformations. This aids in verifying correct compilation, as well as verifying that compiled code is well-behaved. Unlike TAL, SafeTSA provides cross-platform portability, since it defers code generation to the target machine, while still allowing rapid verification and code generation. Due to their high-level representation and suitability for compression, SafeTSA programs are likely to be more compact than TAL's assembly or object code with annotations.

10. CONCLUSION AND OUTLOOK

The Java Virtual Machine's instruction format is not very capable in 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 Java bytecode language. Our representation, SafeTSA, can provide the identical 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 Java bytecode. Also, SafeTSA removes the need to perform type/range check elimination at the code receiver's side, genuinely enabling shifting this workload to the code's producer without jeopardizing safety. Surprisingly, despite its advantages, SafeTSA is no more voluminous than JVM code.

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 Java bytecode, they will be using these better alternatives. The authors believe to have identified such an alternative in SafeTSA.

11. ACKNOWLEDGEMENTS

The authors would like to thank Peter Housel and the anonymous reviewers for their helpful comments on this paper. We also thank Michael Phillipsen for providing the source code of the Pizza compiler.

Parts of this effort are sponsored by the National Science Foundation under grant CCR-9901689, and 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.

This paper is dedicated to the memory of Bratan Kostov.

12. REFERENCES

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

[2] 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.

[3] W. Amme and E. Zehendner. Data dependence analysis in programs with pointers. *Parallel Computing*, 24(3–4):505–525, May 1998.

[4] Architecture Neutral Distribution Format (XANDF) Specification. Open Group Specification P527, January 1996.

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

[6] M. M. Brandis and H. Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Trans. Prog. Lang. and Sys.*, 16(6):1684–1698, Nov. 1994.

[7] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software Practice and Experience*, 28(8):859–881, July 1998.

[8] C. Click. *Combining Analyses, Combining Optimizations*. PhD thesis, Rice University, Houston, Texas.

[9] R. Crelier. OP2: A Portable Oberon Compiler. Technical Report 1990TR-125, Swiss Federal Institute of Technology, Zurich, Feb., 1990.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. K. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL '89*, 1989.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. and Sys.*, 13(4):451–490, Oct. 1990.

[12] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. *ACM SIGPLAN Notices*, 33(5):106–117, May 1998.

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

[14] M. Franz and T. Kistler. Slim Binaries. *Communications of the ACM*, 40(12):87–94, Dec. 1997.

[15] S. Ghemawat, K. H. Randall, and D. J. Scales. Field analysis: Getting useful and low-cost interprocedural information. In *PLDI '00*, 2000.

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

[17] D. A. Huffman. A method for the construction of minimum redundancy codes. In *Proceedings of the IRE, 40*, pages 1098–1101, 1951.

[18] 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.

[19] T. Kistler. Continuous Program Optimization. *Phd Dissertation, University of California, Irvine*, 1999.

[20] T. Kistler and M. Franz. A Tree-Based alternative to Java byte-codes. *International Journal of Parallel Programming*, 27(1):21–34, Feb. 1999.

[21] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Prog. Lang. and Sys.*, 1(1):121–141, July 1979.

[22] 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.

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

[24] K. V. Nori, U. Ammann, et al. Pascal-P implementation notes. In D. W. Barron, editor, *Pascal – The Language and its Implementation*, pages 125–170. John Wiley and Sons, Ltd., 1981.

[25] 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.

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

[27] 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.

[28] V. C. Sreedhar and G. R. Gao. A linear time algorithm for placing [phi]-nodes. In ACM, editor, *Conference record of POPL '95*, pages 62–73, New York, NY 10036, USA, 1995. ACM Press.

[29] R. F. Stärk and J. Schmid. Java bytecode verification is not possible. Technical report, ETH Zürich, Nov. 2000.

[30] J. Wang, C. Eisenbeis, M. Jourdan, and B. Su. Decomposed software pipelining: A new perspective and a new approach. *International Journal of Parallel Programming*, 22(3):351–373, June 1994.

APPENDIX

A. JAVA ARRAY REFERENCES

In Java, the size of an array may often not be known statically, but once that an array object has been created, its size remains constant. As a consequence, an index that is safe to use with a given array reference will remain safe throughout the lifetime of that SSA array reference (which is not necessarily the same as the lifetime of the underlying array variable).

SafeTSA approaches type safety in a conservative manner: for each array-ref value, we create a safe-index type that signifies a "value that can safely be used as an index for this array-ref value".

As a further consequence of the approach used with SafeTSA, a safe-index value can only pass though phi nodes that are dominated by the corresponding safe-ref value for the underlying array (the safe-type must cease to exist when the value ceases to exist). This results in the need to give types a limited scope based on the dominator relationship, just like SSA values, which however doesn't present any major implementation challenges.

B. A MORE DETAILED EXAMPLE

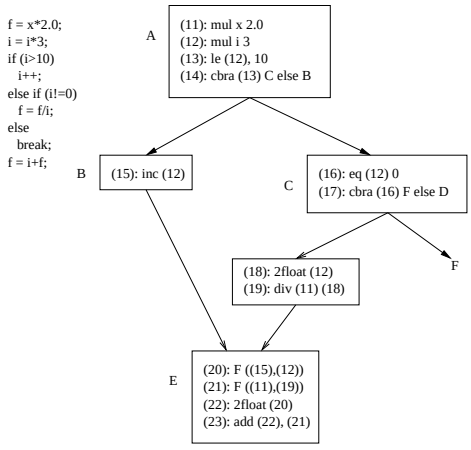

Figure 7: Program Fragment in SSA Form

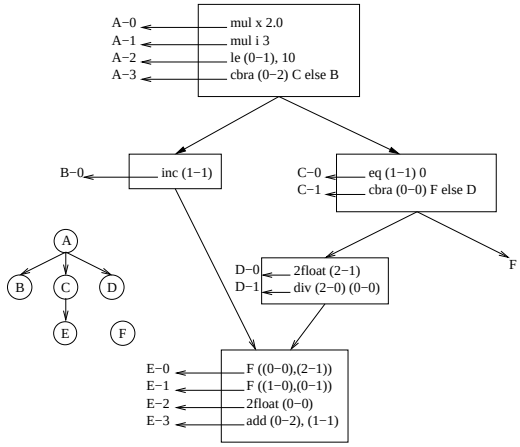

Figure 8: Referentially Secure SSA Form

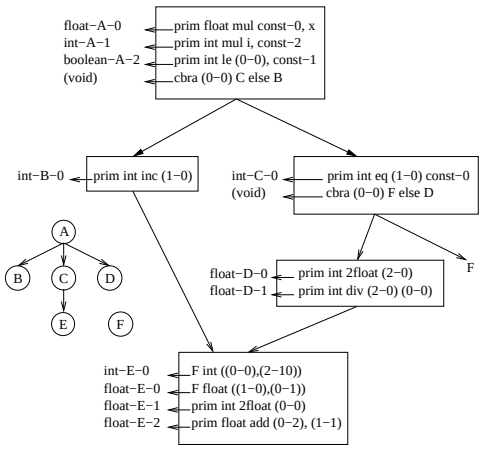

Figure 9: SafeTSA Form