

Project TRANSPROSE: Reconciling Mobile-Code Security With Execution Efficiency*

Wolfram Amme
wolfram@ics.uci.edu

Niall Dalton
ndalton@ics.uci.edu

Peter H. Fröhlich
phf@acm.org

Vivek Haldar
vhaldar@ics.uci.edu

Peter S. Housel
housel@acm.org

Jeffery von Ronne
jronne@ics.uci.edu

Christian H. Stork
cstork@ics.uci.edu

Sergiy Zhenochin
serega@ics.uci.edu

Michael Franz
franz@uci.edu

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

Abstract

Project TRANSPROSE is a comprehensive research project investigating techniques for transporting programs securely over potentially insecure channels. The central focus of this project is the development of a blueprint for a next-generation mobile-code distribution format. A problem of previous approaches to mobile-code security has been that the additional provisions for security lead to a loss of efficiency, often to the extent of making an otherwise virtuous security scheme unusable for all but trivial programs. TRANSPROSE strives to deviate from the common approach of studying security in isolation and instead focuses simultaneously on multiple aspects of mobile-code quality. Besides security, such aspects include encoding density, speed of dynamic code generation, and the eventual execution performance. This paper gives a high-level overview of project TRANSPROSE and presents initial results, which include a highly-effective syntax-based compression scheme for Java programs, as well as a performance-oriented intermediate program representation providing guaranteed security.

*Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL), Air Force Materiel Command, USAF, under agreement number F30602-99-1-0536. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The Views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, AFRL, or the U.S. Government.

1 Introduction

The ability to send mobile code from one machine to another is one of the most important enabling technologies of the Internet age. Mobile code, especially forms of mobile code that are target machine-independent, greatly alleviate many previously existing problems of software distribution, version control, and maintenance. Mobile code also provides the means for entirely new approaches, such as “executable content” within documents.

Unfortunately, using mobile code is fraught with risks. If an adversary succeeds in deceiving us into executing a malicious program supplied by him or her, this may have catastrophic consequences and may lead to loss of confidentiality, loss of information integrity, loss of the information itself, or a combination of these outcomes. Hence, we must at all costs avoid executing programs that can potentially cause such harm.

The *first line of defense* against such incidents is to shield all computer systems, all communications among them, as well as all of the information itself against intruders using physical and logical access controls.

The *second line of defense* is to use cryptographic authentication mechanisms to detect mobile code that has not originated with a known and trusted code provider or that has been tampered with in transit.

Project TRANSPROSE concerns itself with the *third line of defense* that is independent of and complementary to the first two mentioned above: Assume that an intruder has successfully managed to penetrate our system (breaking defense #1) and is able to present us with a mobile program

that falsely authenticates itself as being uncompromised and originating from a trusted party (circumventing defense #2), how do we nevertheless prevent it from causing any damage?

To answer this question, we have been studying a particular class of representations for target-machine independent mobile programs that can provably encode only legal programs. Hence, there is no way an adversary can substitute a malicious program that can corrupt its host computer system: Every well-formed mobile program that is expressible in such an encoding is guaranteed to map back to a source program that is deemed legal in the original source context, and mobile programs that are not well-formed can be rejected trivially. Further, such an encoding can be designed to guarantee not only referential integrity and type-safety within a single distribution module, but also to enforce these properties across compilation-unit boundaries.

A problem of previous approaches to mobile-code security has been that the additional provisions for security lead to a loss of efficiency, often to the extent of making an otherwise virtuous security scheme unusable for all but trivial programs. Conversely, project TRANSPROSE from the outset has been striving to deviate from the common approach of studying security in isolation, and instead has focused on satisfying multiple goals of mobile-code quality simultaneously. Some such additional qualities to consider are the mobile code format's *encoding density* (an important factor for transfer over wireless networks) and the ease with which high-quality native code can be generated by a just-in-time compiler at the eventual target site.

The remainder of this paper outlines various facets of project TRANSPROSE, starting with definitions and scope of the project and its architecture, and then presenting some preliminary results.

2 Definitions and Scope

We understand the term *mobile code* from a compiler construction perspective as any *intermediate representation* that fulfills the following criteria:

- **Completeness:** The intermediate representation has an executable semantics independent of external information.
- **Portability:** The intermediate representation is free of assumptions about the eventual execution platform (processor family, operating system).
- **Security:** The intermediate representation can be shipped safely over insecure channels without the potential for compromising the execution platform.

- **Density:** The intermediate representation can be encoded in a compact form to minimize the impact of bandwidth-limited channels.
- **Efficiency:** The intermediate representation is well-suited for generating efficient, directly executable object code using just-in-time compilation techniques.

Apart from these fundamental criteria, the following properties are also desirable:

- **Extensibility:** The intermediate representation should be general enough to allow multiple source languages to be transported effectively.
- **Pipelability:** The intermediate representation should enable pipelining of decoding (decompression) and code-generation.

Within the TRANSPROSE project, we are developing an infrastructure for a mobile-code transportation standard. This infrastructure takes the form of a well-documented extensible component framework. The architecture of this framework is similar to the architecture of a modern compiler, however it is usually deployed in a distributed fashion:

- *Code producers* use compilers for various source languages, which could be called “front-ends” of the architecture, to compile applications into the mobile code intermediate representation.
- *Code consumers* use compilers for this intermediate representation, which could be called “back-ends” of the architecture, to generate native object code suitable for execution on their machines.

Code producers and code consumers are separated in time and/or space, and mobile code is shipped from producers to consumers using a variety of channels.

Within the design space bounded by these requirements, we have made the following fundamental design decisions:

- Cryptographic approaches to security are orthogonal to our project. Such techniques rely on *trust* relationships between code producers and code consumers. Although they potentially help detect malicious code, they do not protect the execution platform against being compromised.
- Compilation time requirements are more stringent at the code consumer's site than at the code producer's site. It is therefore beneficial to off-load time-consuming compilation tasks to the code producer, provided that the computed information can be shipped compactly and in a tamper-proof manner.

- Compressing mobile code is viable if the time for transmission and decompression is lower than the transmission time of uncompressed code. The overall transmission speed is that of the slowest link along the path from producer to consumer, which in many important emerging application areas (including military ones) is often a low-bandwidth wireless one.

Given these definitions and scope, one immediately wonders how this compares to the currently dominant industry solution, the Java Virtual Machine (JVM). The JVM has quickly become the de-facto standard for encoding mobile programs for transport across the Internet. Unfortunately, the solution embodied by Java fails to deliver the execution efficiency of native code at reasonable levels of dynamic compilation effort.

The main reason for this deficiency is that the JVM's instruction format is not very capable in transporting the results of program analyses and optimizations. As a consequence, when Java byte-code 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. Java also fails in preserving programmer-specified parallelism when transporting programs written in languages such as Fortran-95, leading to loss of information that is essential for optimizations and that cannot be completely reconstructed at the code recipient's site.

The main reason why Java byte-code has these deficiencies is to allow verification by the recipient. Untrusted mobile code needs to be verified prior to execution to protect the host system from potential damage. The most fundamental validation step is to ascertain that an arriving mobile program is type-safe, since a breach of the type system can be used to subvert any other security policy. The use of a type-safe source language does not by itself remove the necessity of verification. This is because barring any additional authentication mechanism, it cannot be guaranteed that any given piece of mobile code ever originated in a valid source program in the first place—it might instead have been explicitly hand-crafted to corrupt its host.

Verifying the type-safety of a piece of Java virtual-machine code is a non-trivial and time-consuming activity. Interestingly enough, and as we elaborate below, in the course of the TRANSPROSE project we have identified certain mobile-program representations that not only remove the need for verification altogether, but that can also transport, in a tamper-proof manner, the results of program analyses benefiting code generation on the eventual target machine. As already hinted at in the introduction, the key idea for doing away with verification is to use a representation that can provably encode only legal programs in the first place. This still leaves the generation of native code to be done by the code consumer; however, it significantly reduces the amount of time spent on program analysis at the

target site, allowing this time to be spent on better optimization instead.

3 Policy Assumptions and Guarantees

Security in our approach is based on *type safety*, using the typing model of the source language (but not restricted to any particular language's type-safety model). A type-safe source language is a requirement. The underlying idea is then the following:

A security guarantee exists at the source language level; just preserve it through all stages of code transportation.

The requirements for a mobile-code transportation system thereby become:

- all mobile programs need to be programmed in a type-safe language
- each host system needs to publish its policies in terms of a type-safe API

The mobile-code transportation system then guarantees type safety throughout: all of the host's library routines are guaranteed to be called with parameters of the correct type(s) by the mobile program. Also, capabilities (object pointers) owned by the host can be manipulated by the mobile client application only as specified in the host's interface definition (for example, using visibility modifiers such as *private*, *protected*, ...), and they cannot be forged or altered.

For example, the host's file system interface might have a procedure

Open(...): File

that returns an abstract file object. A type-safety based security scheme is able to guarantee that the mobile client program cannot alter such a file object in any way prohibited by its specification, or access its contents unless this is allowed by explicit visibility modifiers. Conversely, additional security policies such as "mobile program X can open files only in directory Y" need to be implemented on the host's side.

Hence, the semantics of such a transportation scheme are identical to "*sending source code*", which incidentally is the model that almost all programmers (falsely) assume anyway. Note that, for efficiency reasons and to guard trade secrets embedded in the mobile code, the approach of actually sending source code is usually not an option.

As a consequence, subtle problems arise whenever the semantics of the intermediate language are different from those of the source language; for example, there are programs that can be expressed using the Java Virtual Machine's byte-code language that have no equivalent at the

Java source language level. Situations like these should be avoided.

3.1 Is Type Safety Sufficient?

Interestingly enough, the model of “transporting the equivalent of source code” enables automatic support for *any* future user-specified security policy that can be cast into a language construct.

Take for example Andrew Myers *Java Information Flow* language [28]. This language provides an additional modifier (an additional dimension in the type space) to variables that specifies a security attribute of the variable’s owner. Information flow can then be restricted statically among components to occur only from low security to high security, but not vice versa.

Clearly, this is a new security property that is not supported by current mobile code transportation schemes. However, this property and any other property that can be mapped onto a language construct can easily be supported by grammar-based mobile-code transportation schemes such as the one developed in the TRANSPROSE. The key is to use an approach that completely preserves the type semantics of the underlying source language.

4 Encoding Only Legal Programs

We have been investigating a class of encoding schemes for mobile programs that rely on semantic information to guarantee that only legal programs (under the syntactic and typing rules of some underlying grammar) can be encoded in the first place. This is essentially achieved by constantly adjusting the “language” used in the encoding to reflect the semantic entities that are legally available under the aforementioned rules at any given point in the encoding process. A program encoded in this manner cannot be tampered with to yield a program that violates these rules. Rather, any modification in transit can in the worst case only result in the creation of another legal program that is guaranteed to conform to the original rules. Because the encoding schemes we are exploring are related to data compression, they also yield exceptionally high encoding densities as a side-effect.

4.1 Example: Probabilistic Encoding

As a concrete example of an encoding that has the property of being able to encode only legal programs, consider a probabilistic encoding mechanism that parses the intermediate abstract syntax tree representation of a program and encodes it based on the intermediate representation’s grammar and a continuously adapted probabilistic model.

At each step in the encoding, the alternatives that can possibly occur at this point are enumerated. For each type of grammatical class, there is a different probability distribution, and both the encoder and the decoder keep track of these distributions. For example, a statement can be either of (*assignment, if-statement, while-statement, ...*), and the probability of it being an assignment is usually highest. Similarly, if it is indeed an assignment, then the left hand side can either be a local variable, or a global or external variable, or an array access, or a pointer access, etc. If it is a local variable, the choices are limited by the declarations in the program, and their relative probability can be based on past history.

An encoding can then be found using a form of arithmetic coding. The probabilities of the various occurring constructs define sub-intervals of the range [0-1), and the bit-pattern that represents the encoding corresponds to the real number designating the sub-interval. For example, if assignment has a probability of 1/2, we would designate the interval [0-0.5) to represent assignment. Assume that with probability of 0.8, the target of an assignment is a local variable, this would lead to [0-0.4) to stand for “assign to local”. If we had three local variables *i, j, and k*, and assignment to them had equal probability, these assignments could now be represented as [0-0.1333), [0.1333-0.2666), and [0.2666-0.4) and so on. At each step, the interval is narrowed down in such a manner that highly probable events contribute less information than less probable ones. Any real number in the final interval represents the encoded construct. The probabilities themselves are constantly adjusted as the encoding proceeds. As a consequence, individual constructs contribute just a fraction of a bit to the final encoding in programs that exhibit a high degree of self-similarity. Note that the bit pattern that is ultimately emitted need not be buffered indefinitely, but instead can be output progressively, by rescaling the interval appropriately.

Note that this encoding can provably encode only legal programs, because elements that are forbidden at any given point have zero probability and hence cannot be represented. On the other hand, any given bit-pattern maps back onto a legal program according to the original rules, because the corresponding real number falls into some interval. Also note that performance-enhancing (but semantically irrelevant) annotations can easily be superimposed on the encoding stream simply by incorporating them into the probabilistic model.

4.2 Technical Approach

Our technique is an improvement on the earlier “Slim Binary” method [17], a dictionary-based encoding scheme for syntax trees. In the Slim Binary scheme, a dictionary is grown speculatively and at any given time contains all sub-

expressions that have previously occurred and whose constituent semantic entities (variables, data members, ...) are still visible according to the source-language scoping rules. The encoding schemes we have been investigating under the TRANSPROSE project exert a much finer control over the encoding dictionary (or more general *context*), at each step temporarily removing from it all sub-expressions that are not applicable. Our research in this direction is elaborated in a separate section entitled “Compression of Abstract Syntax Trees”.

Unlike the Slim Binary method, we have also investigated applying the encoding to richer and more compiler-related starting representations than syntax trees. For example, the encoding could be applied to programs represented in a variant of Static Single Assignment (SSA) form [3, 32], after performing common sub-expression elimination and copy propagation. Such an SSA-based encoding disambiguates between different values of the same variable and not only simplifies the generation of high-quality native code at the receiver’s site, but also leads to fewer alternatives at each encoding step and consequently to a still denser encoding. Our work on encoding SSA has led to a genuinely new intermediate representation called SafeTSA which is described in the following section.

An alternative is to instead use a “bottleneck interface”. The core calculus approach is somewhat separate from our other research and presented in a separate section below.

5 Core Calculus

A source program generally goes through several different internal representations within the different stages of a compiler:

1. Strings of source program text
2. Streams of lexical tokens
3. Abstract syntax trees (ASTs) generated during the parsing process
4. A “medium-level” internal representation, which the compiler uses to analyze the source and transform it for optimization. To improve the retargetability of compilers, this representation is normally independent of the final compilation target.
5. A “low-level” internal representation, which makes explicit the actual machine instructions to be used. This representation is dependent on the compilation target, and is designed to allow scheduling, resource allocation, and additional target-dependent optimizations.
6. A machine-language representation, containing the actual bits to be executed by the target machine.

If a program is to be transported somehow from a code producer to a code consumer, any of these representation levels may be used. Several trade-offs may influence the choice of representation:

- Higher representation levels allow the program to be portable to a wider range of targets.
- Similarly, higher representation levels make it easier to optimize the program for specific target characteristics. This may be critical for good performance on modern architectures [24, 4].
- Higher representation levels make it easier for the consumer to prove or disprove the type safety of the program.
- Correspondingly, higher representation levels require more work on the part of the consumer, increasing the time required (and the energy required in embedded applications) before the program can be executed.
- Lower representation levels make the program more independent of the programming language in which the program is written.
- Lower representation levels increase the difficulty of reverse-engineering the program.

The representation level of Java Bytecodes is somewhat mixed, containing many features that are specific to the Java language, as well as some features more typical of a low-level encoding. Overall it has proven to be adequate for transporting code in a number of source languages to a variety of popular machine architectures (as well as direct interpretation), while allowing validation by the code consumer.

However, mismatches in the representation level and the semantics of the representation can result in bad performance for languages other than Java. Potential sources for problems include insufficient flexibility for object storage (resulting in inefficient use of space); differences in the semantics of subtyping, object dispatch or primitive operations; and lack of support for parallelism or other specialized language constructs.

Shortly after the JVM was introduced, Shivers and Fahlman [34] proposed solving this problem by providing a mechanism for extending the Java Virtual Machine. The mechanism would allow new language-specific instructions to be added to the JVM for use by compilers. The new instructions would be described using a low-level format such as a Register-Transfer Language (RTL). Due to the unsafe, low-level nature of the extension language, extension programs would become part of the trusted computing base on the code consumer’s side, and would therefore require some

sort of cryptographic authentication mechanism. This proposal was never implemented, and so there is still no adequate transport mechanism for mobile code in many advanced programming languages.

An alternative method of solving this problem is used by the *TRANSPROSE core calculus* approach. In the base approach, programs are encoded using a high-level intermediate representation based on the source-language specific Abstract Syntax Tree (AST), just as with the Slim Binaries system. In addition to transporting the encoding AST, which has proven to be a good target for compression, a *mapping* is transported, giving the semantics of the source language AST in terms of a core calculus.

Programs are transmitted using this approach as follows: the code producer uses a traditional compiler front-end to produce an abstract syntax tree for the source program, and then directly encodes this tree for transmission. A mapping describing the semantics of the source language is also transmitted, or obtained from a library of mappings for common languages. The code consumer decodes the original abstract syntax tree and the mapping, then applies the mapping to obtain the original program represented in the core calculus. The consumer then compiles the program from this representation to machine code on the target architecture.

To ensure the safety of this scheme, it is necessary for the core calculus to be a type-preserving one. One possibility would be to use the SafeTSA described later in this paper, though language-specific aspects of SafeTSA might result in some of the same problems described earlier for the Java Virtual Machine.

The representation we have chosen in this line of research is to use the typed lambda calculus, System F_ω . This representation has been studied extensively by the programming language community, and is amenable to compiler analysis using well-known techniques.

Advantages of this scheme include:

- The high-level representation makes it possible to compress the code efficiently.
- Source languages can be added *ad hoc* without user intervention or increasing the size of the trusted computing base.
- Because it is easier to write a mapping file than to develop a full compiler back-end, the system can serve as part of a “language design workbench.”
- Because programs are transmitted at a high level, and almost all of the compilation machinery is present at the code consumer, it becomes possible to provide a convenient API for code-writing programs.
- Because the source program is transmitted using a high-level representation, it remains possible to use a

(potentially more efficient) language-specific compiler backend for more commonly used languages.

The system as described so far places the burden of work on the code consumer. If the mapping is changed to map a lower-level code to the core calculus, however, it becomes possible to do variable amounts of optimization before the code is transmitted. This allows the flexibility to choose the best place in the continuum of representation levels for the application at hand.

6 Compression of Abstract Syntax Trees

Among the major approaches to mobile code compression are (a) schemes that use code factoring compiler optimizations to reduce code size while leaving the code directly executable [12], (b) schemes that compress object code by exploiting certain statistical properties of the underlying instruction format [14, 18, 25, 31], and (c) schemes that compress the abstract syntax tree (AST) of a program by using either statistical [7, 13] or dictionary-based approaches [17].

Our approach falls into the last category. The source code (modulo comments, layout, and names of internal identifiers) can easily be regenerated from an AST. Since the AST is composed according to a given abstract grammar (AG), we are using domain knowledge about the underlying language to achieve a more compact encoding than a general-purpose compressor could achieve.

Our compression framework applies to different kinds of code. It is convenient to think of our compression algorithm as being applied to some source language, which—after decompression at the code receiver site—is compiled into native code. But generally, our scheme applies to all code formats, which can be expressed in form of a grammar. Theoretically, this includes all forms of code: source code, intermediate representations (e.g., byte code), and object code. Our prototype implementation compresses Java source programs, which can then be compiled to native code, thereby circumventing compilation into byte code and execution on the JVM.

We chose the compression of Java programs (as opposed to other languages) as a proof-of-concept because a sizeable body of work on the compression of Java programs exists already, especially Pugh’s work on jar file compression [31]. This gives us a viable yardstick to gauge our results against.

Our compression scheme does not assume that source code will be re-generated at the code consumer’s site. In fact, in our current implementation the decompressor interfaces directly to the GCC backend.

In our framework, source code is required in order to generate a compressed AST and, inversely, a compressed

AST possesses the intrinsic capability to regenerate the source code (deprived of comments and internal identifier names). These two facts position our encoding as the ideal distribution format¹ for Open Source Software. Files in our format are more compact and span several architectures, thereby reducing the maintenance effort for packaging.

Since our compression format contains all the machine-readable information provided by the programmer at source language level, the runtime system at the code consumer site can handily use this information to provide optimizations and services based on source language guarantees.² Kistler [23] uses the availability of the AST to make dynamic re-compilation at runtime feasible. Furthermore, distributing code in source language-equivalent form provides the runtime system with the choice of a platform-tailored intermediate representation. The success of Transmeta’s code morphing technology shows that this is a viable approach, even when starting with an unsuitable intermediate representation at a much lower abstraction level.

Lastly, high-level encoding of programs protects the code consumer against all kinds of attacks based on low-level instructions, which are hard to control and verify. Our encoding also has the desirable characteristic that even after malicious manipulation it can only generate ASTs which adhere to the abstract grammar (and additional semantic constraints), thereby providing some degree of safety by construction. This is in contrast to byte code programs, which have to go through an expensive verification process prior to execution.

6.1 Compression Techniques for ASTs

Computer program sources are phrases of formal languages represented as character strings. But programs proper are not really character strings, in much the sense that natural numbers are not digit strings but abstract entities. Conventional context-free grammars, i.e., *concrete grammars*, mix necessary information about the nature of programs with irrelevant information catering to human (and machine) readability. An AST is a tree representing a source program abstracting away concrete details, e.g., which symbols are used to open/close a block of statements. Therefore it constitutes the ideal starting point for compressing a program. Note also that properties like precedence and different forms of nesting are already implicit in the AST’s tree structure.

¹Of course, our format is only meant as replacement for the binary distribution of Open Source Software. Since the right to modify the source and documentation is integral part of the Open Source philosophy our format is no alternative to the fully commented source text.

²As an example, note that the Java language provides much more restrictive control flow than Java byte code, which allows arbitrary gotos.

6.1.1 Abstract Grammars

Every AST complies with an *abstract grammar* (AG) just as every source program complies with a concrete grammar. AGs give a succinct description of syntactically³ correct programs by eliminating semantically superfluous details of the source program.

AGs consist of *rules* (also called *productions*) defining symbols much like concrete grammars define terminals and nonterminals [26]. Whereas phrases of languages defined by concrete grammars are character strings, phrases of languages defined by AGs are ASTs. Each AST node corresponds to a rule, which we will often refer to as the *kind* of node. For the purpose of a simple presentation, we will discuss only three forms of *rules*, which suffice to specify sensible AGs. (These three rules are a subset of the rules used in our framework.)

The first two rules are compound rules defining symbols corresponding to the well-known non-terminals of concrete grammars. An *aggregate rule* defines AST nodes (*aggregate nodes*) with a fixed number of children. For example, the rule for the while-loop statement defines a *WhileStmt* node with two children of kind *Expression* and *Statement*.

$$\textit{WhileStmt} \triangleq \textit{Expression}; \textit{Statement}.$$

The second form of compound rule is the *choice rule*, which defines AST nodes (*choice nodes*) with exactly one child. The kind of child node can be chosen from a fixed number of alternatives. The following (simplified) rule says that a *Statement* node has either an *Assignment*, *IfStmt*, or *WhileStmt* child, i.e., a statement is either one of these three.

$$\textit{Statement} \triangleq \textit{Assignment} \mid \textit{IfStmt} \mid \textit{WhileStmt}.$$

The last kind of rule is the *string rule*, which defines *string nodes*. The right hand side of a string rule is the predefined STRING symbol. String rules define the equivalent of terminals in concrete grammars. String nodes contain an arbitrary string and they are the leaf nodes of the AST. To define the *Ident* node to be a string node we write the following.

$$\textit{Ident} \triangleq \text{STRING}.$$

User-defined symbols of AGs must be defined by exactly one rule with the exception of the predefined STRING symbol. As usual, one symbol is marked as the *start symbol* of the AG.

6.1.2 Encoding ASTs

In order to encode (i.e., store or transport) ASTs they need to be serialized. ASTs can be serialized by writing out well-defined traversals. We serialize an AST by generating its

³Here the term “syntactically” refers by convention to the context-free nature of the grammar.

preorder representation. Such a traversal provides a linearization of the tree structure only. In order to encode the information stored at the nodes several mechanisms exist. The most common technique pre-scans the tree for node attributes, stores them in separately maintained lists, and augment the tree representation with indices into these lists. For now, we ignore the problem of efficiently compressing strings (our only node attributes) for the sake of simplicity and assume that strings are directly encoded whenever they appear.

The actual tree representation can make effective use of the AG. Given the AG, much information in the preorder encoding is redundant. In particular, the order and the kind of children of aggregate nodes is already known. Therefore the encoding boils down to noting the choices made at each choice node. Since the order of alternatives in choice nodes is fixed, it suffices to encode only the position (1, 2, 3, ...) of the chosen alternative. Of course, if only one alternative is given there is “no choice” and therefore nothing needs to be encoded.

6.1.3 Arithmetic Coding

So far we reduced the serialization of compound rules to encoding the choice made at each choice node as an integer $c \in \{1, 2, \dots, n\}$, where n depends on the kind of choice node and is equal to the number of given alternatives. We want to use as few bits as possible for encoding the choice c . The two options are to use Huffman coding or arithmetic coding. Using Huffman code as discussed in Stone [36] is very fast, but is much less flexible compared to arithmetic coding. Cameron [7] shows that arithmetic coding is more appropriate for good compression results.

An arithmetic coder is the best means to encode a number of choices if each alternative $i \in \{1, 2, \dots, n\}$ has a certain probability p_i , where $\sum_{i=1}^n p_i = 1$ and n is given by the kind of choice node. The tuple $M = (p_1, p_2, \dots, p_n)$ is called the *model* M for the arithmetic coder. When encoding, an arithmetic coder takes a sequence of choices c_j along with their respective models M_j as argument and outputs a sequence of bits B . From this information, the arithmetic coder produces the most compact encoding of the sequence of choices c_1, c_2, \dots . When decoding, an arithmetic coder takes the sequence of bits B and the above sequence of models M_1, M_2, \dots as arguments. For each given model M_j it then reproduces the next choice c_j . It is important to note that the model M_j can depend on all previous choices c_1, c_2, \dots, c_{j-1} . The choice of models determines the compression ratio. If the probabilities are picked in an “optimal” fashion (i.e., taking “all” available information into account and adapting the probabilities appropriately) then the encoding has maximal entropy.

A simple and fast way to chose the models is to fix the

probability distributions for each kind of node. Good static models can be determined based on statistics over a representative set of programs.

6.1.4 Prediction by Partial Match

Prediction by Partial Match (PPM) [9] is a statistical, predictive text compression algorithm. PPM and its variations have consistently outperformed dictionary-based methods as well as other statistical methods for text compression. PPM maintains a list of already seen string prefixes, conventionally called *contexts*. For each such context it keeps track of the occurrences of the following characters. For example, after processing the string *ababc*, the contexts are the empty context, *a, b, c, ab, ba, bc, aba, bab, abc, abab, babc*, and *ababc*. The length of a context is also called its *order*. The counted subsequent characters for, say, *ab* are *a* and *c* both with one occurrence. Normally, efficient implementations of PPM maintain contexts dynamically in a *context trie* [8]. A context trie is a tree with characters as nodes and where any path from the root to a node represents the context formed by concatenating the characters along this path. The root node does not contain any character and represents the empty context (i.e., no prefix). In a context trie, children of a node constitute all characters that have been seen after its context. In order to keep track of the number of times that a certain character followed a given context, the number of its occurrences is noted along each edge. Based on this information PPM can assign probabilities to potentially subsequent characters.

Adapting PPM for ASTs We have adapted the unbounded variant of the PPM algorithm (PPM*) [8] to work on ASTs instead of text. When applying PPM to trees the first problem to solve is the definition of contexts for ASTs. The *context* of an AST node N is defined as the concatenation of nodes from the root to N , exclusively. This means our modified PPM algorithm treats AST nodes like the original PPM algorithm treats characters. Our alphabet corresponds therefore to the symbols/rules of the AG.⁴ The PPM* algorithm is applied to the sequences of nodes as they appear while traversing the AST in depth-first order.

However, above changes by itself are not sufficient to adapt PPM to ASTs. The maintenance procedure of the context trie needs to be augmented too, since the input seen by the modified PPM does not consist of contiguous characters anymore. No change is needed when the tree traversal descends to a child node. This corresponds to the familiar addition to the current context. But what happens if the

⁴Note that if an aggregate node has several children of the same kind then their position is relevant for the context. Since this does not happen that often, we have not implemented this refinement yet.

traversal ascends from subtrees thereby annihilating the current context? This necessitates some enhanced context trie functionality. PPM* maintains a set of nodes in the context trie called *active nodes*. Active nodes mark the positions where the current contexts end. The root of the trie, representing the empty prefix, is always active.

New nodes in the context trie are always created as children of active nodes. However, in our adaptation of PPM, unlike regular PPM, whenever we reach a leaf of the abstract syntax tree, we *pop* the context, i.e., all nodes marked as active (except the root) in the context trie are moved up one node to their parents. This ensures that all children of a node N in the AST appear as children of N in the context trie too. This works because we traverse the AST in depth first order while building up contexts. A desirable consequence of this technique is that the depth of the context tree is at most the depth of the abstract syntax tree which we are compressing.

Weighing Strategies In order to generate the model for the next encoding/decoding step, we look up the counts of symbols seen after the current context in the context trie. Since the active nodes, to which we have direct pointers, correspond to the last seen symbol, this is a fast lookup and does not involve traversing the trie. These counts can be used in several ways to build the model. Normally the context trie contains counts for contexts of various orders. We have to decide how to weigh these to get a suitable model. There is a trade-off here : shorter contexts occur more often, but fail to capture the specificity and sureness of longer contexts (if the same symbol occurs many times after a very long context, then the chance of it occurring again after that same long context is very high), and longer contexts do not occur often enough for all symbols to give good predictions. Note that the characteristics of AST contexts differ from text contexts.⁵ We tried various weighing strategies, and our experiments indicate that ignoring predictions made by order 0 contexts (which are simply occurrence counts of symbols, and form the first level of the context trie) and weighing all other predictions equally yields the best compression.

6.1.5 Compressing Constants

A sizable part of an average program consists of constants like integers, floating-point numbers, and, most of all, string constants. String constants in this sense encompass not only the usual string literals like "Hello World!" but also type names (e.g., `java.lang.Object`), field names and

⁵AST contexts are bound by the depth of the AST and tend towards more repetitions since the prefixes of nodes for a given subtree are always the same.

more. In our simplified definition of AGs, we used the pre-defined `STRING` symbol to embody the case of constants within ASTs.

Each string node is attributed with an arbitrary string. However, when observing the use of strings in ASTs of typical programs, it is apparent that many strings are used multiple times. Therefore it saves space to encode the different strings once and refer to them at later occurrences. Such a reference is an index into a list of strings. The higher the number of strings is, the more bits are needed to encode the corresponding index. By distinguishing different kinds of strings (e.g., type names, field names, and method names) different lists of strings can be created. The split lists are each smaller than a global list. Given that the context determines which list to access, references to strings in split lists require less space to encode. As these considerations show, context-sensitive (as opposed to context-free) information such as symbol tables can be encoded and compressed at varying degrees of sophistication.⁶

6.2 Preliminary Results

Our current implementation is a prototype written in Python consisting of roughly 40 modules handling grammars, syntax trees, and their encoding/decoding. Our frontend is written in Java and uses Barat [5] for parsing Java programs. All information necessary to specify the AST's encoding and compression is condensed into one configuration file. The configuration file contains the AG augmented with additional information, e.g., on how to compress the symbol table. Given the availability of our framework at the code producer and consumer sites, the only additional requirement for each supported language is that identical copies of the configuration file are present at both sites.

We chose primarily Pugh's compression scheme for comparison (see Table 1) because, to our knowledge, it provides the best compression ratio for Java class files and it is freely available for educational purposes. The other comparable compression scheme is syntax-oriented coding [13]. But for this scheme there are no detailed compression numbers available, only an indication that the average compression ration is 1 : 6.5 between their format and regular class files.

It should be noted that Pugh actually designed his compression scheme for `jar` files, which are collections of (mostly) class files. His algorithm therefore does not perform as well on small files as it does on bigger ones. We use the evaluation version 0.8.0 of Pugh's Java Packing tool and feed it with `jar`-files generated with the `-M` option (no manifest).

⁶Note that conventional symbol tables can conveniently be expressed as some kind of AST with the appropriate string nodes.

Name	Class File	Gzip	Bzip2	Jar	Pugh	PPM	PPM/Pugh
ErrorMessage	388	256	270	409	209	105	0.50
CompilerMember	1321	637	641	792	396	230	0.58
BatchParser	5437	2037	2130	2189	1226	1069	0.87
Main	13474	5482	5607	5627	3452	3295	0.95
SourceMember	15764	5805	5705	5920	3601	2988	0.83
SourceClass	37884	13663	13157	13975	8863	7849	0.89
javac (package)	92160	32615	30403	36421	18021	14070	0.78

Table 1. File sizes of compressed files for some classes from `javac` (all numbers in bytes).

With a simple addition, our framework can be applied to collections of classes as present in packages or `jar`-files. These arbitrary collections of classes share the same lists of strings, thereby reducing redundancy caused by entries that appear in several classes. We can use our framework with the extended AG to compress the classes contained in `jar`-files. This gives us the basis for a good comparison with Pugh’s work. For reference purposes we also include the size of the original, `gzipped`, and `bzip2ed` class files. In case of the `javac` package we applied `tar` first.

Our choice of single classes tries to be representative of the sizes of classes in the `jvm98` suite [35]. We use the official Java compiler package `javac` (from JDK 1.2.2 for Linux) to compare our results to others since it is available in source form.

Table 1 shows that our compression scheme improves compression by 5–50% over Pugh’s results. Our experience shows that PPM adapts fast enough to each program’s peculiarities that efforts to improve compression by initially using statistically determined probabilities did not yield any significant gains in compression.

6.3 Related Work on Compression

The initial research on syntax-directed compression was conducted in the 1980’s primarily in order to reduce the storage requirements for source text files. Contla [10, 11] describes a coding technique essentially equivalent to the technique described in section 6.1.2. This reduces the size of Pascal source to at least 44% of its original size. Katajainen et. al. [22] achieve similar results with automatically generated encoders and decoders. Al-Hussaini [1] implemented another compression system based on probabilistic grammars and LR parsing. Cameron [7] introduces a combination of arithmetic coding with the encoding scheme from section 6.1.2. He statically assigns probabilities to alternatives appearing in the grammar and uses these probabilities to arithmetically encode the preorder representation of ASTs. Furthermore, he uses different pools of strings to encode symbol tables for variable, function, procedure, and type names. Deploying all these (even non-context-free) techniques he achieves a compression of Pascal sources to 10–17% of their original size. Katajainen and Mäkinen

[21] present a survey of tree compression in general and the above methods in particular. Tarhio [38] suggests the application of PPM to drive the arithmetic coder in a fashion similar to ours. He reports increases in compression of Pascal ASTs (excluding constants) by 20% compared to a technique close to Cameron’s.⁷

All of these techniques are concerned only with compressing and preserving the source text of a program in a compact form and do not attempt to represent the program’s semantic content in a way that is well-suited for further processing such as dynamic code generation or interpretation ([22] even reflects incorrect semantics in their tree). Franz [16, 17] was the first to use a tree encoding for (executable) mobile code.

Java, currently the most prominent mobile code platform, attracted much attention with respect to compression. Horspool and Corless [19] compress Java class files to roughly 36% of their original size using a compression scheme specifically tailored towards Java class files. In a follow-up paper Bradley, Horspool, and Vitek [6] further improve the compression ratio of their scheme and extend its applicability to Java packages (`jar` files). A better compression scheme for `jar` files was proposed by Pugh [31]. His format is typically 1/2 to 1/5 of the size of the corresponding compressed `jar`-file (1/4 to 1/10 the size of the original class files). All of the above Java compression schemes start out with the byte code of Java class files, in contrast to the source program written in the Java programming language. Eck, Changsong, and Matzner [13] employ a compression scheme similar to Cameron’s and apply it to Java sources. They report compression to around 15% of the original source file, although more detailed information is needed to assess their approach. In contrast to Pugh, they make no evaluation tools available.

7 The SafeTSA Representation

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 ac-

⁷Unfortunately, we learned of Cameron’s and Tarhio’s work only after we developed our solution independently of both.

knowledge 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.

The approach taken with SafeTSA developed under the TRANSPROSE project 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.

7.1 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 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 in Java it is not possible to reference a local variable prior to assigning a value to it, 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 n th argument of the phi function corresponding to the n th 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

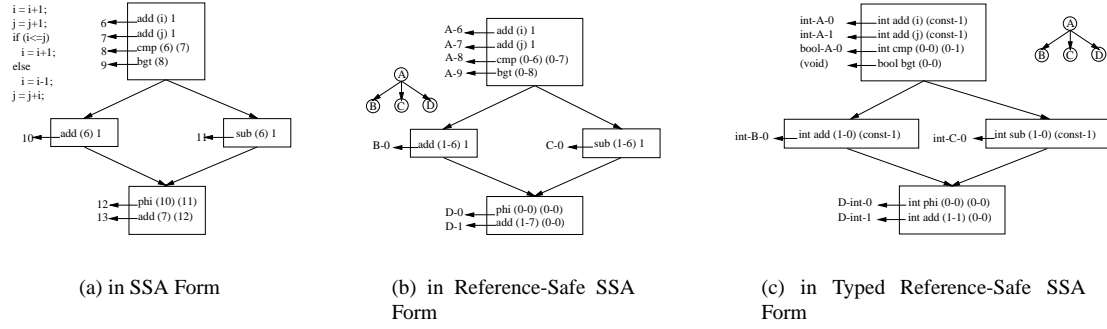


Figure 1. An Example Program

r does not exceed the permissible maximum. The latter fact can actually be exploited when encoding the $(l-r)$ pair space-efficiently.

7.2 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).

7.3 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.⁸

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

⁸Because of the need to support dynamically-sized arrays, *safe-index* types are actually bound to array *reference values* rather than to their static types.

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 (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. For example, the primitive for data member write access is

setfield *ref*-type object field 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***.

The *setfield* operation and the corresponding *setelt* for arrays 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

7.4 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.

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 sizes of SafeTSA files are usually smaller than the equivalent Java byte-code files, and sometimes substantially so.

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 to 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.

Class Name	Instruction Count			Phi Instruction			Null-Checks			Array-Checks		
	JVM	STSA	Opt.	with	w/o	$\Delta\%$	with	w/o	$\Delta\%$	with	w/o	$\Delta\%$
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.

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.

7.5 Related Work on Typed IRs

It is difficult to generate quality native code from JVM-code[20]. 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[30] 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 [33], Marmot [15], and the HotSpot Server compiler [37]. Jalapeno [2] 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 guarantee 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)[29] 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) [27] 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.

8 Conclusions

The TRANSPROSE project has made contributions to the areas of syntax tree compression and mobile-code representations. We are continuing to explore trade-offs between security, flexibility, compactness, and efficiency.

Acknowledgements: The authors would like to thank Thomas Kistler, Bratan Kostov, Ziemowit Laski, and Sumit Mohanty for their contributions in the early stages of this project. Christian Rattei designed and implemented a reusable library of general compression algorithms.

References

- [1] A. M. M. Al-Hussaini. *File compression using probabilistic grammars and LR parsing*. PhD thesis, Loughborough University, 1983.
- [2] B. Alpern, C. R. Attanasio, et al. The Jalapeno virtual machine. *IBM System Journal*, 39(1), February 2000.
- [3] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Fifteenth Annual POPL Conference*, 1988.
- [4] M. E. Benitez and J. W. Davidson. The advantages of machine-dependent global optimization. In *PLSA '94: International Conference on Programming Languages and Architectures*, pages 105–123, Mar. 1994.
- [5] B. Bokowski and A. Spiegel. Barat – A front-end for Java. Technical Report B-98-09, Freie Universität Berlin, Dec. 1998.
- [6] Q. Bradley, R. N. Horspool, and J. Vitek. JAZZ: An efficient compressed format for Java archive files. In *Proceedings of CASCON'98*, pages 294–302, Toronto, Ontario, Nov. 1998.
- [7] R. D. Cameron. Source encoding using syntactic information source models. *IEEE Transactions on Information Theory*, 34(4):843–850, July 1988.
- [8] J. G. Cleary and W. J. Teahan. Unbounded length contexts for PPM. *Computer Jnl.*, 40(2/3):67–75, 1997.
- [9] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.
- [10] J. F. Contla. *Compact Coding Method for Syntax-Tables and Source Programs*. PhD thesis, Reading University, England, 1981.
- [11] J. F. Contla. Compact coding of syntactically correct source programs. *Software-Practice and Experience*, 15(7):625–636, 1985.
- [12] S. Debray, W. Evans, and R. Muth. Compiler techniques for code compression. In *Workshop on Compiler Support for System Software*, May 1999.
- [13] P. Eck, X. Changsong, and R. Matzner. A new compression scheme for syntactically structured messages (programs) and its applications to Java and the Internet. In *Data Compression Conference*, page 542, 1998.
- [14] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting. Code compression. In *Proceedings of the ACM Sigplan '97 Conference on Programming Language Design and Implementation*, pages 358–365, 1997. Published as Sigplan Notices, 32:5.
- [15] R. Fitzgerald, T. B. Knoblock, et al. Marmot: An optimizing compiler for Java. Microsoft Technical Report 3, March 2000.
- [16] M. Franz. *Code-Generation On-the-Fly: A Key to Portable Software*. PhD thesis, ETH Zurich, mar 1994.
- [17] M. Franz and T. Kistler. Slim Binaries. *Communications of the ACM*, 40(12):87–94, Dec. 1997.
- [18] C. W. Fraser. Automatic inference of models for statistical code compression. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1999.
- [19] R. N. Horspool and J. Corless. Tailored compression of Java class files. *Software-Practice and Experience*, 28(12):1253–1268, Oct. 1998.
- [20] 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.
- [21] J. Katajainen and E. Mäkinen. Tree compression and optimization with applications. *International Journal of Foundations of Computer Science*, 4(1):425–447, 1990.
- [22] J. Katajainen, M. Penttonen, and J. Teuhola. Syntax-directed compression of program files. *Software-Practice and Experience*, 16(3):269–276, 1986.
- [23] T. Kistler. *Continuous Program Optimization*. PhD thesis, University of California, Irvine, Nov. 1999.
- [24] T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems*, May 2000.
- [25] S. Lucco. Split stream dictionary program compression. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2000.
- [26] B. Meyer. *Introduction to the Theory of Programming Languages*. PHI Series in Computer Science. Prentice Hall, 1990.
- [27] 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.
- [28] A. C. Myers. Jflow: Practical mostly-static information flow. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL '99)*, San Antonio, Texas, USA, Jan. 1999.
- [29] G. C. Necula. Proof-Carrying Code. In *POPL '97*, Paris, France, Jan. 1997.
- [30] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Prog. Lang. and Sys.*, 21(5):895–913, September 1999.
- [31] W. Pugh. Compressing java classfiles. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 247–258, 1999.
- [32] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Fifteenth Annual POPL Conference*, 1988.
- [33] 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.
- [34] O. Shivers. Supporting dynamic languages on the Java virtual machine. In *Proceedings of the Dynamic Objects Workshop*, Boston, May 1996.
- [35] Standard Performance Evaluation Corporation. SPEC JVM98 benchmarks. See online at <http://www.spec.org/osg/jvm98> for more information.
- [36] R. G. Stone. On the choice of grammar and parser for the compact analytical encoding of programs. *Computer Journal*, 29(4):307–314, 1986.
- [37] Sun Microsystems. Hotspot compiler for Java. <http://java.sun.com/products/hotspot/>.
- [38] J. Tarhio. Context coding of parse trees. In *Proceedings of the Data Compression Conference*, page 442, 1995.