Making Mobile Code Both Safe And Efficient*

Michael Franz
Peter H. Fröhlich
Fermín Reig

Wolfram Amme
Vivek Haldar
Jeffery von Ronne

Matthew Beers
Andreas Hartmann
Christian H. Stork

Niall Dalton
Peter S. Housel
Sergiy Zhenochin

School of Information and Computer Science
University of California
Irvine, CA 92697

Abstract

Mobile programs can potentially be malicious. To protect itself, a host that receives such mobile programs from an untrusted party or via an untrusted network connection will want some kind of guarantee that the mobile code is not about to cause any damage. The traditional solution to this problem has been verification, by which the receiving host examines the mobile program to discover all its actions even before starting execution. Unfortunately, aside from consuming computing resources in itself, verification inhibits traditional compiler optimizations, making such verifiable mobile code much less efficient than native code.

We have found an alternative solution by identifying a class of mobile-code representations in which malicious programs can simply not be encoded to begin with. In such an encoding, verification turns into an integral part of the decoding routine. Moreover, support for high-quality just-in-time code generation can be provided. We present two such encodings, one based on highly effective compression of abstract syntax trees, and another based on a reference-safe and type-safe variant of Static Single Assignment form.

*Parts of this material were previously published under the title "Project transPROse: Reconciling Mobile-Code Security With Execution Efficiency" in *The Second DARPA Information Survivability Conference and Exhibition (DISCEX II)*, Anaheim, California, June 2001, IEEE Computer Society Press, ISBN 0-7695-1212-7, pp. II.196–II.210. This research effort was 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 transport mobile code between machines is one of the most important enabling technologies of the Internet age. Platform-independent mobile code greatly alleviates many 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 adversaries deceive us into executing malicious programs supplied by them, 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.

In the past four years, we have concerned ourselves with a *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 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 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. To avoid this trap, we have from the outset deviated from the common approach of studying security in isolation, and instead have focused on satisfying multiple goals of mobile-code quality simultaneously. Some such additional qualities 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 our research, starting with definitions and scope of the project and then presenting its major result, two *inherently safe mobile code representations:* one based on compression of abstract syntax trees (that is also applicable to XML transport), and another based on a referentially safe and type-safe variant of Static Single Assignment form.

2 Definitions and Scope

We define the term *mobile code* to denote any *intermediate representation* that fulfills the following criteria:

- **Completeness:** The intermediate representation preserves 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:

- **Generality:** The intermediate representation should be general enough to allow multiple source languages to be transported effectively.
- **Pipelineability:** The intermediate representation should enable a partial overlap of decoding and code-generation activities, so that code generation (and potentially even execution) can commence before the mobile program has been fully downloaded.

The architecture of a mobile-code 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 "backends" of the architecture, to generate native object code suitable for execution on their machines.

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

Within the design space bounded by these requirements, we 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. Instead of actually protecting the execution platform from being compromised, they merely establish, sometimes wrongly, that a reliable authority has certified the code's integrity.
- 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 safely.
- 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 is often a low-bandwidth wireless one.

Given these definitions and scope, one immediately wonders how this compares to the currently dominant industry solution for mobile code, 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 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 producer. Java bytecode 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 bytecode 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 our research we identified certain mobile-program representations that not only remove the need for verification altogether, but that can also transport safely the results of program analyses benefiting code generation on the eventual target machine. As we suggested 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 originate in a type-safe programming 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 security scheme founded on type safety 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 inside the resident library 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, however, 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.

3.1 Bytecode Considered Harmful

A direct consequence of using a representation other than source code for transporting mobile programs from code producer to code consumer is that there can potentially

be *subtle semantic mismatches* between the original source language and the actual transportation language. These mismatches can lead to unexpected results. For example, there are programs that can be expressed in the Java Virtual Machine's bytecode language but that have no equivalent at the Java source language level, and there are also Java source programs that are legal according to the language definition but that cannot be transported using Java bytecode [50]—a reflection, perhaps, of the fact that Java was rushed to market without extensive prior testing. Situations like these should be avoided.

It is precisely the large semantic gap between the Java source language and the JVM bytecode that makes Java verification necessary. For example, type-unsafe accesses can be made in JVM bytecode, but not in Java source language. Similarly, arbitrary jumps can be made in JVM bytecode, but not in source Java. Fundamentally, all the effort expended for bytecode verification is to make sure that this semantic gap is not maliciously exploited.

By using Abstract Syntax Trees (ASTs) as a mobile code format, what is transported is much closer to the semantics of the original source language. As a direct consequence of this, one can reduce the verification overhead versus what is required with bytecode-based formats. For example, type safety is evident at the source language level. A proof of safety exists at the source level and that should be preserved through the mobile code pipeline. In essence, transporting bytecode throws away this source-level *proof*, requiring verification effort that in some sense tries to reconstruct guarantees that existed at the source level originally. High-level ASTs preserve this source-level proof throughout.

From the viewpoint of programmers, it is the semantics of the source language that they understand, and indeed it is those semantics that should be *transported* safely. High-level encoding of programs protects the code consumer against attacks based on low-level instructions, which are hard to control and verify. Even if tampered with, a file in such a high-level format that passes a rudimentary well-formedness check guarantees adherence to the semantic constraints that are enforced, thereby providing safety by construction.

Another major disadvantage of bytecode-based formats is the great amount of effort that is required to optimize them to efficient native code. This is again due to the semantic gap between the source language and the low-level bytecode format. In general, most backend code-generating optimizations are greatly helped by high-level information—the kind that is available in source code. But it is precisely this kind of information that is lost when source is compiled to bytecode. As a result, the backend optimizer has to expend great effort to recover some of this high-level structure.

For example, the first thing an optimizer does with bytecode is to construct a control flow graph. Transporting a high-level format that is very close to source code trivially solves this problem. The backend optimizer now has all the information about the high-level structure of the program. Since an AST-based mobile code format contains all the information provided by the programmer at the source language level, the runtime system at the code consumer site can readily use this information to provide optimizations and services based on source language guarantees. Kistler and Franz [36, 35, 34] use the availability of the AST to make dynamic re-compilation at runtime feasible.

It is sometimes feared that using a high-level representation such as abstract syntax trees would easily give away intellectual property in the encoded program. This claim is not well-founded. Local variable names are the major factor in understanding code, and they can be removed from AST-based representations to the same extent as they are absent from JVM bytecode. At closer sight, low level representations such as JVM bytecode offer only illusory protection of intellectual property. This is readily demonstrated by freely available bytecode decompilers such as the Java Optimize and Decompile Environment (JODE) [28]. Moreover, recent theoretical results question the very possibility of obfuscation [7].

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 can be encoded in the first place. We use the term *legal* to describe programs that are syntactically well-formed and adhere to specific static semantic constraints, such as fulfilling the type compatibility rules of a certain source language.

This kind of encoding is essentially achieved by constantly adjusting the "language" used in the encoding to reflect the semantic entities that are legally available under the above 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

As a concrete example of an encoding that has the property of being able to encode only legal programs, consider an encoding mechanism that parses the intermediate abstract syntax tree representation of a program

and encodes it based on the intermediate representation's grammar and continuously adapted knowledge about the program semantics.

At each step in the encoding, the alternatives that can possibly occur at this point are enumerated. We encode the actual choice as its index into this enumeration. In order to encode a whole program, the encoding proceeds like this in a depth-first left-to-right fashion, starting with the root of the syntax tree.

For example, let's say we are at the point where we have to encode a program statement. Such a statement could be either an assignment, a while-statement, an if-statement, or some other kind of statement. If our program contains a while-statement at this point, then we can encode it as item number two of the above enumeration. Furthermore, a while-statement is made up of a condition, which must be a boolean expression, and a body, which must be a statement again. Next we encode the boolean expression. Expressions can include variable accesses, multiplications, comparisons, or something else defined by the language definition. At this point, multiplication is not an option since it does not result in something of boolean type. The variable access is allowed but only if it is a visible variable of boolean type that has been properly initialized. (The latter restrictions are examples of static semantic constraints.) All kinds of comparisons are potential while-conditions.

If an encoded program has been tampered with, it either still encodes a legal program (which cannot be detected by our technique) or it yields an index that lies outside of the enumeration bounds, resulting in an error. (In Section 5.1 we will go one step further and show how to use a compression techique called *arithmetic coding* in order to make even the encoding of out-of-bounds indices impossible. This reveals an interesting synergy of compact and legal encodings.)

The kind of programs that can be encoded depends on the algorithm that provides the alternatives for enumeration. For example, if this algorithm enumerates only "type-safe choices" then the encoded program will be type safe. This algorithm is shared knowledge between the encoder and decoder. Assuming the correctness of the implementation, we can guarantee that only legal programs will be decoded because the decoder is part of the trusted code base.

4.2 Technical Approach

Our technique is an improvement on the earlier "Slim Binary" method [23, 33], 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 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 discussed in Section 5.

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 [2, 47], 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 Section 6.

5 Compression of Abstract Syntax Trees

With the advent of mobile code, there has been a resurgent interest in code compression. Compactness is an issue when code is transferred over networks limited in bandwidth, particularly wireless ones. It is also becoming increasingly important with respect to storage requirements, especially when code needs to be stored on consumer devices. Considering the development of the classic PC over the last decade, CPU performance increases exponentially over the storage access time. Therefore it makes sense to investigate compression as a means to use additional CPU cycles to decrease the demand on storage access [23]. Note also that, since compression is a one-time effort at the the code producer site, we can use very expensive computational resources in order to achieve a high compression ratio.

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 [18], (b) schemes that compress object code by exploiting certain statistical properties of the underlying instruction format [20, 25, 41, 45], and (c) schemes that compress the abstract syntax tree (AST) of a program by using either statistical [10, 19] or dictionary-based approaches [23].

Our approach falls into the last category: after extracting the AST from the source text we compress it using a probabilistic encoding. 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 that can be expressed as an abstract grammar. Theoretically, this includes all forms of code: source code, intermediate representations (e.g., bytecode), and object code. Our prototype implementation compresses Java programs, which can then be compiled to native code, thereby circumventing compilation into bytecode 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 [45]. This gives us a viable yardstick to compare our results against.

In our framework, source code is the starting point for the generation of the compressed AST and, inversely, a compressed AST contains the information to regenerate the source code deprived of comments, layout, and internal identifier names. Of course, 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 to the GCC backend. This intrinsic relationship to source code and the fact that our scheme knows how to make use of the added information in the source code makes our encoding the ideal distribution format for Open Source Software.[1] Files in our format are more compact and could potentially 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 readily use this information to provide optimizations and services based on source language guarantees.[2] Further, 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 very promising approach, even when starting with an unsuitable intermediate representation like x86 machine code.

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 bytecode programs, which have to go through an expensive verification process prior to execution.

[1] Of course, our format is only meant as replacement for the binary distribution of Open Source Software, and not for the fully commented source text.

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

5.1 Compression Techniques for ASTs

Computer program sources are phrases of formal languages represented as character strings. But programs proper are not 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 manifest in the AST's tree structure.[3]

5.1.1 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 syntactically[4] correct programs by eliminating superfluous details of the source program.

AGs consist of *rules* defining symbols much like concrete grammars consist of productions defining terminals and nonterminals [42]. 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 say, defines its *kind*. For simplicity, we will discuss only three forms of *rules*, which suffice to specify sensible AGs.

The first two forms of rules are compound rules defining nodes corresponding to nonterminals in concrete grammars. *Aggregate rules* define AST nodes (*aggregate nodes*) with a fixed number of children. For example, the rule for the while-loop statement in Figure 1 is an aggregate rule. It defines a $While$ node with two children of kind $Expr$ and $Stmt$.

Choice rules are rules that define AST nodes (*choice nodes*) with exactly one child. The kind of child node can be chosen from a fixed number of alternatives. Figure 1 shows

[3] Due to their equivalence, we often use the terms AST and source program interchangeably.

[4] Here the term "syntactically" refers by convention to the context-free nature of the grammar.

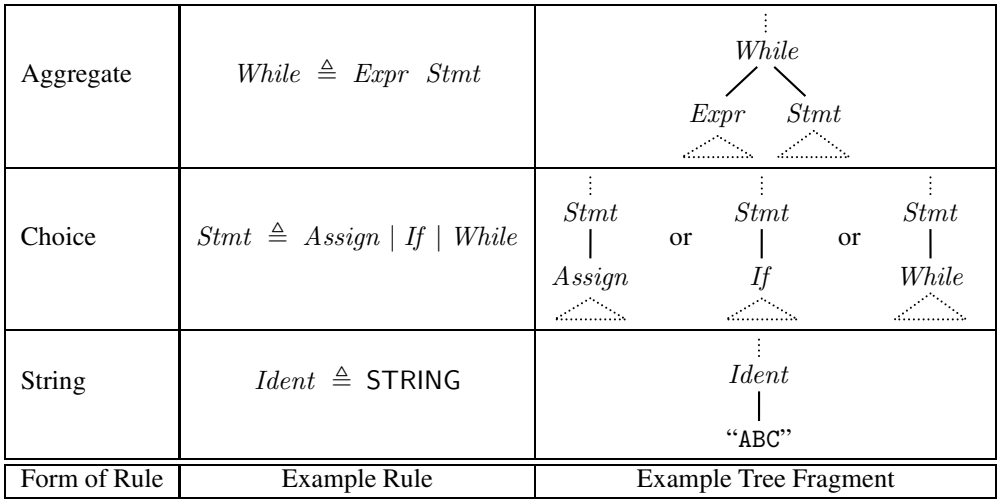


Figure 1. Abstract rules and corresponding abstract tree fragments

a (simplified) choice rule representing *Stmt* nodes with either an *Assign*, an *If*, or a *While* child, i.e., a statement is either one of these three.

The last form of rule defines *string rules*. The right hand side of a string rule is the predefined STRING symbol. String rules represent AST leaf nodes (*string nodes*)and in this sense they are the equivalent of terminals in concrete grammars. (But note that—in contrast to the parse tree for a concrete grammar—the inner nodes of ASTs carry non-redundant information.) String nodes have exactly one string as a child. See the example definition of an *Ident* node in Figure 1.

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

5.1.2 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 chose to serialize an AST as its preorder representation. 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. The left side of figure 3 shows how the AST is traversed and which nodes need to be encoded. In our example, *Expr*, *Stmt*, *Ident*, *BinOp*, and *AssignOp*nodes need not be encoded. Also note that there is no need to specify when to ascend from a subtree since the abstract grammar provides this kind of information too.

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 encode them as indices into a table of strings. In the example of figure 3 the list of *Ident*s contains some other identifiers, presumably defined earlier in the program.

5.1.3 Arithmetic Coding

So far we reduced the serialization of compound rules to encoding the decision made at each choice or string node as an integer $c \in \{1, 2, \ldots, n\}$, where n depends on the kind of 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 [51] is fast, but is much less flexible compared to arithmetic coding. Cameron [10] shows that arithmetic coding is more appropriate for good compression results.

An arithmetic coder is the best means to encode a number of choices. Figure 3 shows how our example AST is encoded. Arithmetic Coding starts out by considering the right-open unit interval $[0, 1)$ and successively narrowing it down. We use binary numbers for the interval bounds. In order to encode the next choice the probabilities of the alternatives are mapped onto the interval. In the example, we have to chose the child of a *Stmt* node first. The alternatives are *Assign*, *If*, *While*, and *Do*. We assume that all alternatives are equally likely, which is represented by the equal length of all for subintervals. In order to encode the *While* node, we chose its corresponding subinterval, $[0.1, 0.11)$. The next choice will be encoded as a subinterval of the current subinterval and so on. In our example, it is the

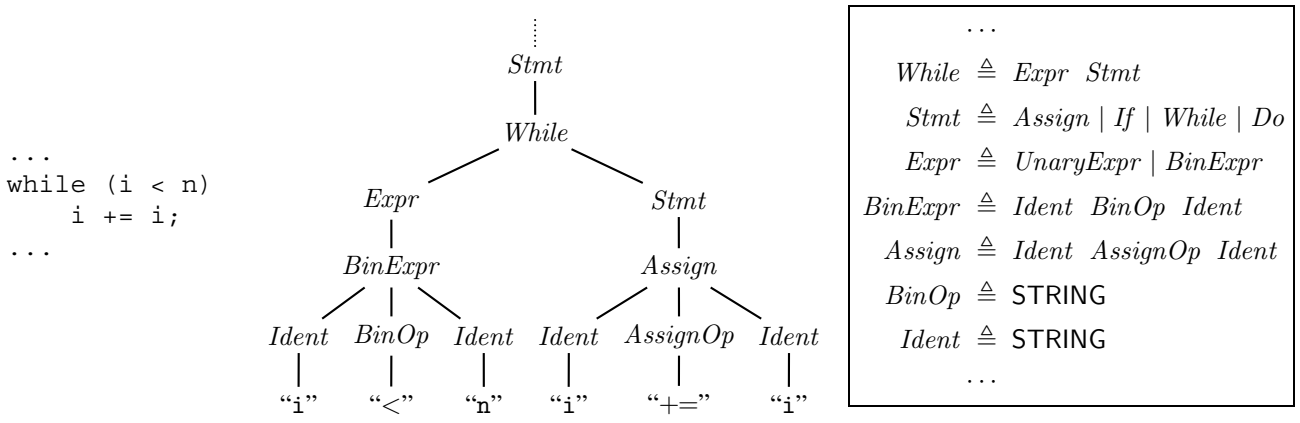


Figure 2. Source code snippet, its AST fragement, and some relevant AG rules

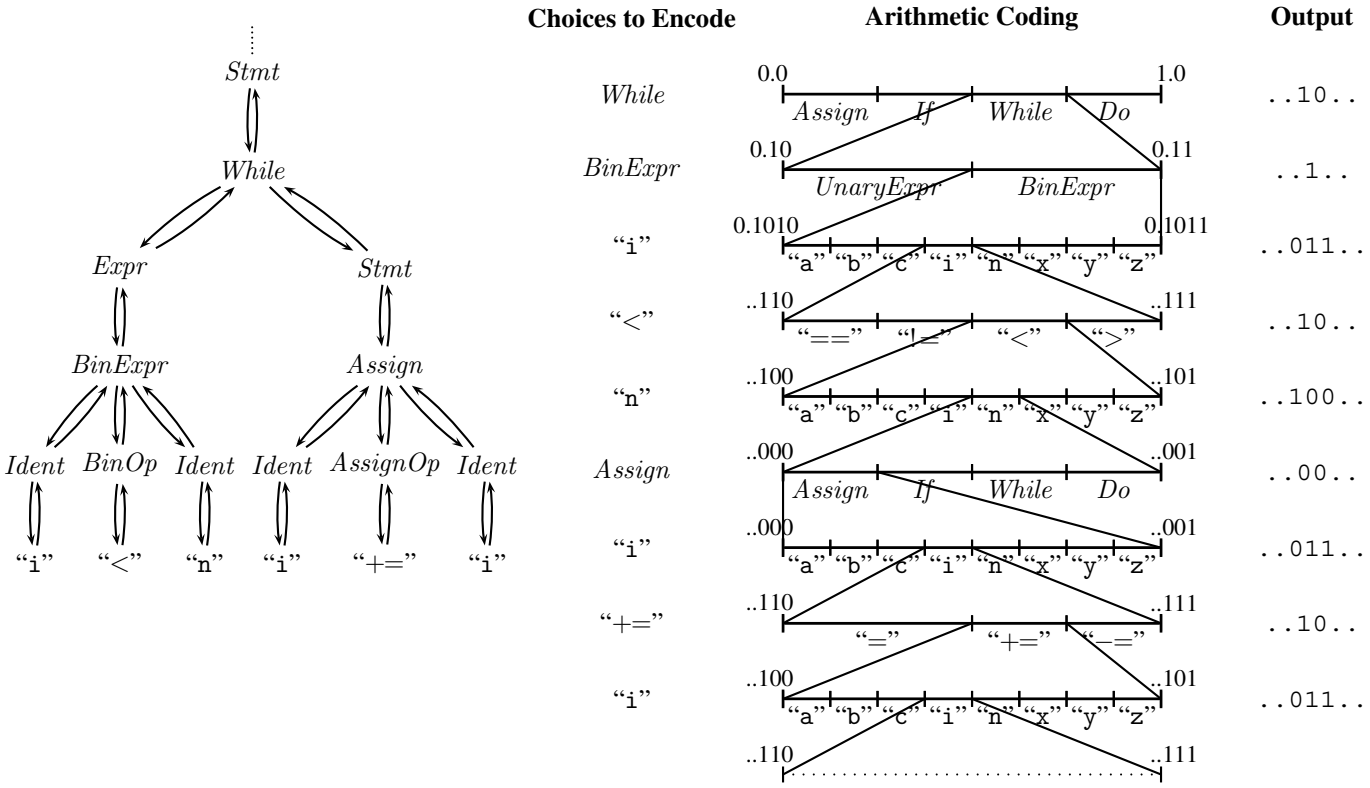


Figure 3. Encoding and Compression of sample AST via Arithmetic Coding

choice between a $UnaryExpr$ and $BinExpr$. Again both are equally likely. We chose the subinterval $[0.101, 0.1011)$. Note that this process approximates a real number and we learn this number bit by bit with each additional digit that matches between the lower and upper bound. These digits (excluding the redundant leading 0) are the compressed file. Essentially we are encoding the real number that "makes all the subinterval choices" necessary to encode the AST.

There is no need to keep track of the leading bits of the bounds as soon as we know that they coincide. We can output them and rescale the interval. Also note that the subinterval bounds need not be digital numbers of limited precision. Subintervals can be arbitrary. For example, very likely choices could have close to probability one and consequently contribute less than one bit to the resulting output.

The probabilty distribution of the alternatives (also called *model*) 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. In the next section, we show how to constantly adjust the probabilities in order to make encoding reoccuring patterns cheaper.

Interestingly, given the above technique any given bit-pattern maps back onto a legal program according to the original rules, because the corresponding real number falls into some interval. Although it is still unlikely that the end-of-file coincides with the end-of-encoding.[5]

5.1.4 Prediction by Partial Match

Prediction by Partial Match (PPM) [13] is a statistical, predictive text compression algorithm. Essentially, PPM provides a mechanism for an arithmetic coder with models. 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*. Considering, for example, character string `ababc` all substrings are contexts. For each such context PPM keeps track of the number of times that each character follows this context. We refer to this as the *count* of a character in a certain context. The length of a context is also called its *order*. For example, `aba` is called an order 3 context and `b` has count 1 in this context. When we speak of the order 0 context we refer to the empty context, i.e. we count the occurences of any character. Taking this idea of generalisation further, we refer to the order -1 context as the context that returns a count of 1 for all available characters. Table 1 shows contexts and context counts after processing the string `ababc`. Based on these context counts PPM can assign probabilities to potentially subsequent characters. We will not go into detail about how the different kinds of PPM arrive at their predictions. See Moffat and Turpin [43] to explore this further.

Order	Contexts	Occurences	Counts
0	*empty context*	|a|b|a|b|c	a:2, a:2, c:1
1	a b c	a|ba|bc a|b|a|b|c ababc|	b:2 a:1, c:1
2	ab ba bc	ab|ab|c a|ba|bc ababc|	a:1, c:1 b:1
3	aba bab abc	aba|bc a|bab|c ababc|	b:1 c:1
4	abab babc	abab|c ababc|	c:1
5	ababc	ababc|	

Table 1. Contexts as maintained by PPM after processing `ababc`

Normally, efficient implementations of PPM maintain contexts dynamically in a data structure called *context trie* [12]. A context trie is a tree with characters as nodes and where for any node the path to the root represents the context in which its character appeared. We also store the count at each node. The root node, denoted as $*$, does not contain any character and corresponds to the empty context. So in a context trie, children of a node constitute all characters that have been seen in the context starting from this node up. See Figure 4 for the context tries of `a`, `ab`, up to `ababc`. Note how the levels of the context trie correspond to the context orders.

Nodes are typeset in bold face if they mark the end of a current context. These nodes are called *active nodes*. The root of the trie, representing the empty prefix, is always active. Note that there is exactly one active node at every level.

Adapting PPM for ASTs We have adapted several variants of PPM algorithms to work on ASTs instead of text. In general, the AST nodes should be treated like the original PPM algorithm treats characters. Our alphabet corresponds

[5] This also assumes that the enumerating algorithm does not run into a "dead end", i.e., it checks all syntactically correct alternatives for possible follow-ups.

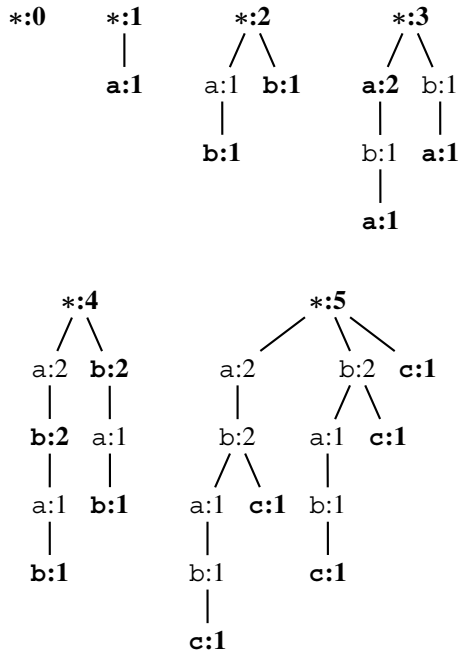


Figure 4. Evolution of `ababc`'s Context Trie (active nodes are bold)

therefore to the symbols/rules of the AG.[6] When applying PPM to trees, one must first define contexts for ASTs. We define the *context* of an AST node N as the concatenation of nodes from the root to N, exclusively. Other definitions are possible, but this one performed best in our initial trials and it has the nice property that the context trie corresponds closely to the original AST. So, while we traverse the AST in depth-first order the PPM algorithm is applied to the sequences of nodes as they appear.

However, the above changes 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 traversal ascends from subtrees thereby annihilating the current context? This necessitates some enhanced context trie functionality.

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 bound by 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.[7] We tried various weighing strategies, and our experiments indicate that a combination of the "classical" PPMC with some added constraints, which use the information of the abstract grammar, works best.

5.1.5 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 more. In our simplified definition of AGs, we used the predefined 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 as indices into string tables. 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

[6] Note that if an aggregate node has several children of the same kind then their position is relevant for the context.

[7] AST 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.

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.[8]

5.2 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 [8] 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 2) 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 [19]. 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).

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, `gzip`ed, and `bzip2`ed 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 [49]. 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.

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

Table 2 shows that our compression scheme improves compression by 15–60% 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.

5.3 Safe Annotations

Many common compiler optimizations are time consuming. This becomes a problem when code is generated on the fly while an interactive user is waiting for execution to commence, for example, in the context of Java dynamic compilation. A partial solution has been provided by *annotation-based techniques*, enabling optimized execution speeds for dynamically compiled JVM-code programs with reduced just-in-time compilation overhead [38], and communicating analysis information that is too time-consuming to collect on-line [6, 30, 44, 26, 46].

Annotations make costly optimizations feasible in dynamic optimization environments. Analysis results that are time- and space-consuming to generate can then be employed since the analyses can be performed off-line and the results compactly communicated via annotation. An example of one such analysis is escape analysis [56, 11], a technique that identifies objects that can be allocated on the stack as opposed to on the heap. Escape analysis can also reveal when objects are accessed by a single thread. This information can then be used to eliminate unnecessary synchronization overhead.

Escape analysis is both time- and space-consuming since it requires interprocedural analysis, fixed-point convergence, and a points-to escape graph for each method in the program [56]. Fixed point convergence is required for loops (within the method and within the call chain) to ensure that the points-to escape graph considers all possible paths that affect object assignment. However, existing escape analysis implementations indicate that its use offers substantial execution performance potential [56, 11].

Ideally, we would like to encode escape analysis as an annotation that is transported with the program. At the destination, the dynamic compilation system can then allocate annotated objects on the stack to improve program performance without the overhead of on-line escape analysis. However, escape analysis annotations that have been described in the literature so far, just as those for other important analyses (register allocation, array bounds and null pointer check identification) are *unsafe*. That is, their accidental or malicious modification can cause security violations (as defined by the language and runtime system) that may result in system exploitation or crashes.

One way to enable the safe use of such annotated analyses would be to verify them at the destination. This,

Name	Class File	Gzip	Bzip2	Jar	Pugh	PPM	PPM/Pugh
`ErrorMessage`	388	256	270	409	209	84	0.40
`CompilerMember`	1321	637	641	792	396	227	0.58
`BatchParser`	5437	2037	2130	2189	1226	943	0.77
`Main`	13474	5482	5607	5627	3452	2909	0.85
`SourceMember`	15764	5805	5705	5920	3601	2477	0.69
`SourceClass`	37884	13663	13157	13975	8863	6428	0.73
`javac` (package)	92160	32615	30403	36421	18021	13948	0.78

Table 2. File sizes of compressed files for some classes from `javac` (all numbers in bytes).

however, would introduce a runtime overhead that could become as complex as performing the analysis itself.

We can extend our encoding mechanism to safely encode annotations as part of the ASTs. We extend the underlying type system by an additional dimension representing "capturedness". The choices here are *captured*, which means that the reference in question never occurs in an assignment that would make it escape its defining scope, and *other*, which means that the reference either escapes or that we cannot prove that it doesn't escape.[9] The task of the encoding then becomes to disallow all assignments between variables that could possibly allow a captured reference to escape.

The capturedness property can therefore be transported in a fully tamper-proof manner. If an adversary were to change the annotation of an escaping variable to erroneously claim that it was captured, then our encoding would not be able to encode any assignment that would let the variable escape. Conversely, if one were to change the annotation of a captured variable to escaping, then that would simply mean that a potential for optimization had been lost, without making the program any less safe.

Hence, our method overcomes a major drawback of existing approaches to using annotations with mobile code, namely that corrupted annotation information could undermine the security of the system. In previous approaches [6, 30, 44], annotations were generally unsafe because there would have been no way of verifying their correctness at the code consumer's site other than by repeating the analysis they were targeting to avoid. Using our method, any object that at its creation time is marked "captured" is guaranteed to be stack-allocatable. No verification is required at the destination to ensure that the annotations we encode are safe to use.

[9]Note that the "capturedness" property applies to *references* (pointer variables) and not to the *objects* that are attached to them. A captured reference may at times point to an object that does escape; we are merely guaranteeing that an escape *will not be caused* by assignments involving the captured reference.

5.4 Compressing Valid XML Documents

In this paper we concern ourselves mainly with enabling the safe transport of code, but we also provide a solution for the safe and efficient transport of data. We chose the eXtensibe Markup Language (XML) data format to demonstrate our approach since it is quickly becoming the universal format for structured documents and electronic data exchange. XML dialects are specified by a DTD (Document Type Definition), which restricts the structure and values of an XML document to the application at hand. Given the verbosity of XML, compression is crucial for its successful deployment. In our context, an XML document is like an AST with many string nodes for the unstructured text. The DTD is essentially a form of abstract grammar that puts restrictions on the tree structure of XML elements and attributes.

Given our success with compression of programs we applied the same techniques to XML documents. Our compression scheme has the advantage of providing the validation of XML documents as an integral part of the decompression process. (An XML document is called *valid* if it adheres to its DTD.) On average, we can compress XML documents to 10% of their original size. For now this includes only documents for which a DTD is available.[10]

There are XML-specific compressors that achieve better compression. This is due to the fact that most XML documents have a different structure than programs. Programs are more deeply nested than XML documents and thus provide more context to the compressor. Also, XML has some idiosyncrasies that are not yet well covered in our implementation. It seems, for example, desirable to treat attribute and element nodes differently, but we are not yet sure how to do this.

Note that our basic approach via abstract grammars and context tree manipulation provides a very general framework for tree compression. Also note that our compression scheme has the advantage of providing the validation of XML documents as an integral part of the decompression process.

[10]An alternative is to learn a simple abstract grammar and transport it at the beginning of the document. We plan to evaluate this idea in the future.

5.5 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 [16, 17] describes a coding technique essentially equivalent to the technique described in Section 5.1.2. This reduces the size of Pascal source to at least 44% of its original size. Katajainen et. al. [32] achieve similar results with automatically generated encoders and decoders. Al-Hussaini [1] implemented another compression system based on probabilistic grammars and LR parsing. Cameron [10] introduces a combination of arithmetic coding with the encoding scheme from Section 5.1.2. 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. Note that Cameron retains comments, like most of the others, too. All of the above four efforts were pursued independently. Katajainen and Mäkinen [31] present a survey of tree compression in general and the above methods in particular. Tarhio [52] 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.[11]

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 (Katajainen et. al. [32] even reflect incorrect semantics in their tree). Franz [22, 23] 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 [29] 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 [9] 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 [45]. 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 bytecode of Java class files, in contrast to the source program written in the Java programming language. Eck, Changsong, and Matzner [19] 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.

[11] Unfortunately, we learned of Cameron's and Tarhio's work only after we developed our solution independently of both.

6 A Safe and Efficient Mobile-Code Format Based On Static Single Assignment Form

The Java Virtual Machine's bytecode format (JVML) has become the de facto standard for mobile code distribution. However, it is generally acknowledged that JVML is far from being an ideal mobile code representation—a considerable amount of preprocessing is required to convert JVML into a representation more amenable to optimization. This is particularly important in the context of just-in-time compilation, when this preprocessing slows down program execution and increases preceived latency. In addition, both the semantic gap between JVM's stack architecture and the register architecture usually found in general purpose processors as well as the requirements of JVM's type safety verifcation procedure hinder ahead-of-time optimization.

For example, the redundancy of a type check is often known by the front-end, Java to JVML compiler,[12] but JVML has no mechanism available to convey this information to the just-in-time compiler. If, however, a means existed to safely communicate this fact, the backend would not need to expend effort to rediscover it at runtime. Another example is common subexpression elimination: a compiler generating JVML could in principle perform CSE and store the resulting expressions in additional, compiler-created local variables, but this approach is clumsy.

In contrast, the SafeTSA representation that we introduced in 2001 [3] departs from the standard stack architecture and instead draws upon a decade of optimizing compiler research and is based on in Static Single Assignment (SSA) Form. The SafeTSA representation is a genuine static single assignment variant that differentiates not between variables of the original program, but only between unique definitions of these variables, each of which is assigned its own name and is referrenced specifically in instruction operands. Being in SSA Form, SafeTSA contains no assignments or register moves, but encodes the equivalent information in ϕ-functions that model dataflow. Unlike straightforward SSA representations, however, SafeTSA provides type safety through type separation and referential integrity as a inherent property of its encoding.

As a safe variant of SSA, SafeTSA allows virtual machines utilizing just-in-time compilation technology to shift

[12] Because the compiler's existing analysis can show that the value in question is of the correct type on every path leading to the check.

analysis and optimization effort from the end user to the code producer without compromising safety or machine-independence.[13] Because SSA Form can be viewed as a factored use-def chain, much of the effort needed for dataflow analysis is already done. Another advantage is gained leveraging the extensive library of standard optimizations using SSA (e.g. common subexpression elimination), which can be performed on SafeTSA. In addition, we present mechanisms for integrating the results of null-check and bounds-check eliminations and escape analysis results into the SafeTSA type system.

6.1 The SafeTSA Representation

6.1.1 Inherent 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 ϕ-function that models the merging of values from multiple definitions into a single variable based on the control flow. A naive conception of SSA, however, would view it as unsuitable for application domains with untrusted code supplier requiring verification of referential integrity. This is because SSA contains an unusually large amount of variables, one for each definition in addition to ϕ-functions, which would seem to necessitate expensive expensive verification.

As an example, consider the program in Figure 5(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 initial definitions of i and j. Since in Java it is not possible to reference a local variable prior to assigning a value to it, these initial definitions must always exist—in most cases, they would be 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 derived from 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, 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 5(a) is given in Figure 5(b).

The resulting representation using such (l-r) value-references provides inherent referential integrity without requiring any additional verification besides the trivial one of ensuring that l is bound by the references depth in the dominator tree and that each relative instruction number r does not exceed the permissible maximum. These bounds can actually be exploited when encoding the (l-r) pair space-efficiently.

6.1.2 Type Separation

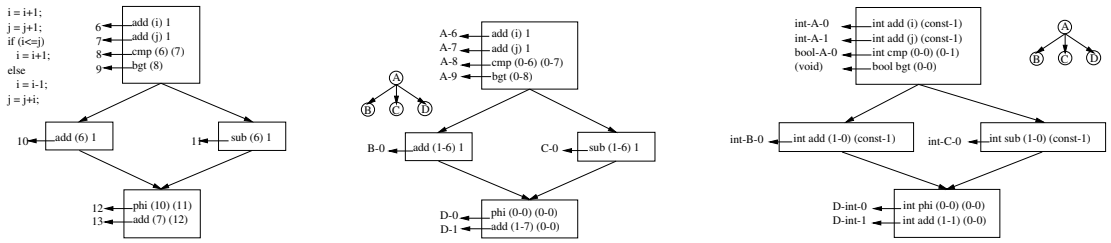
The second major idea of our representation is *type separation*. While the "implied machine model" of ordinary SSA is one with an unlimited number of registers (actually, one for each definition), SafeTSA uses a model in which there is a separate *register plane* for every type.[14] The register planes 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

[13] It has been suggested that the cost of an SSA-based mobile code format is in the loss of interpretability [37]; this is not the case, and a prototype SSA interpreter has been developed [55].

[14] Disregard, for a moment, the added complication of using a two-part (l-r) naming for the individual registers, and also temporarily disregard type polymorphism in the Java language—both of these are supported by our format, as explained below.



(a) in SSA Form

(b) in Reference-Safe SSA Form

(c) in Typed Reference-Safe SSA Form

Figure 5. An Example Program

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.

This type separation prohibits implicit type coercion. Instead, SafeTSA provides explicit cast instructions that read a register from one type's plane, convert it to the new type, and write it to a register of the new type's plane. In place of the implicit type coercions present in the Java source program, the compiler must insert explicit casts to convert from one type to another. These *downcast*s can be verified by the virtual machine at load time and for reference types do not require any implementation at runtime. Many of the explicit casts of reference variables in Java source programs, however, require dynamic checks to verify that the instance referenced does in fact belong to the appropriate class or implement the correct interface. In theory, the SafeTSA *upcast* instruction, writes to the register of the new type plane if and only if this dynamic check succeeds otherwise it transfers execution to the exception handler. In the implementation, however, the input and output registers will normally be coallesced into a single machine register and no register transfer occurs in either case; rather, only the check and, when necessary, the exection will be performed.

ϕ-functions are strictly type-separated: all operands of a ϕ-function, as well as its result, always reside on the same register plane. If two definitions, which are of compatible types at the source level, need to be merged together as inputs to a single ϕ-function, they must first be downcast to an appropriate common type.

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 result of applying both type separation and reference safe numbering to the program fragment of Figure 5(a) is shown in Figure 5(c).

6.1.3 Type and Memory Safety

The *newObject*, *newArray*, *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, these types, including language primitives and those of system libraries, are always verified against the actual implementation at load time, maintaining the type safety of the entire vritual machine.

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 same safety semantics as if Java source code were being transported to the target machine and compiled and linked locally.

6.2 Optimizations Supported by SafeTSA

6.2.1 Null-Check Elimination

For every reference type *ref*, our "machine model" provides a matching type *safe-ref* that implies that the corresponding value has been null-checked. The null-checking operation

can be seen as a specialized form of the *upcast* operator discussed above; it takes an explicit *ref* source type and an explicit register on that type's register plane. If it does not succeed, an exception will be generated, otherwise the virtual machine behaves as if the *ref* value is copied to an implicitly selected register (the next available) on the the corresponding *safe-ref* type's plane.

All memory operations, which in Java bytecode require implicit null-checks, 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, but not from the unsafe types. For example, the primitive for data member write access is

setfield ref-type object field value

where *ref-type* denotes a java class, *object* designates a register (through the indirect numbering scheme described above) 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 beauty of this approach is that it enables the null-check information to be propogated through ϕ-functions across basic-blocks.

6.2.2 Bounds-Check Elimination

For every array type, *arr* we provide a matching type *elemnt-arr* whose instances directly reference a an element of an array of that type. Conceptually, the *element-arr* is produced by the *validateElement* instruction:

validateElement array-type array index

where *array-type* denotes a Java Array type, *array* denotes a safe-reference to an array, and *index* denotes an integer to be used as the index. If the *index* is within the bounds of the array, *validateElement* outputs a validated reference (of type *element-arr* to the appropriate array element. In actual implementations, the *element-arr* registers could contain the computed address of that array element. If the index is, out of bounds, the *validateElement* instruction throws an exception.

For example, the Java code fragment:

```
int x[] = new int[5];
x[3] = 7;
```

could be translated into:

```
x <- newArray int 5
y <- boundsCheck [I x 3
z <- setElement [I y 7
```

where the operands refering to x and y would be replaced with the appropriate dynamic referentially-secure type-separated references described above.

6.2.3 Escape Annotations

One source of performance problems for Java programs is the excessive amount of heap memory allocation. This can be alleviated by the use of escape analysis to identify allocations that can be performed on the stack.

A method for supporting escape analysis by extending the SafeTSA type system with additional 'bound' reference types, which are treated similarly to the null-checked 'safe' types is proposed by von Ronne et al. [54]. Offline escape analysis is used to specialize regular reference types into corresponding bound reference types, which may be able to be allocated on the stack. The SafeTSA type system is then used to restrict the uses of these bound reference types. Call-stack-bound references can be used within a method or passed as arguments of the appropriate call-stack-bound reference type, but they may never be written to any field or returned from a method. This additional type is coupled with special allocation instructions, *stackallocc*. The *stackallock* instruction results in a variable of the null-checked call-stack-bound reference type. Thus, the type system enforces the invariants that *stackalloc*ed objects do not live longer than the creating method's call-frame. The JIT compiler may then take advantage of this knowledge, to safely allocate the objects produced with *stackalloc* on the stack. If it is necessary, however, to comply with the Java binary compatibility standard, it will be necessary to back out stack-allocations to perform link time verification and back out stack-allocations that are no longer valid. For more details refer to the work of Hartmann et al. [27].

6.3 Prototype Implementation

We have built a prototype system consisting of a "front-end" compiler that takes Java source files and translates them to the SafeTSA representation, and a version of Jikes RVM, which is a virtual machine developed by IBM Research, modified to support SafeTSA as well as Java bytecode.[15]

SafeTSA provides a safe mechanism for the transportation of optimized code. The "front-end" is thus able 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. We also utilize SafeTSA's type system to automatically elminate redundant null- and bounds-checks during common subexpression elimination [53].

The phases of the Jikes RVM optimizing compiler communicate through a series of intermediate representations: a high-level intermediate representation (HIR), a low-level

[15] Versions of Jikes RVM prior to its open-source release were known as Jalapeño and the performance results reported herin are from are modified version of Jalapeño's 1.1b University Release.

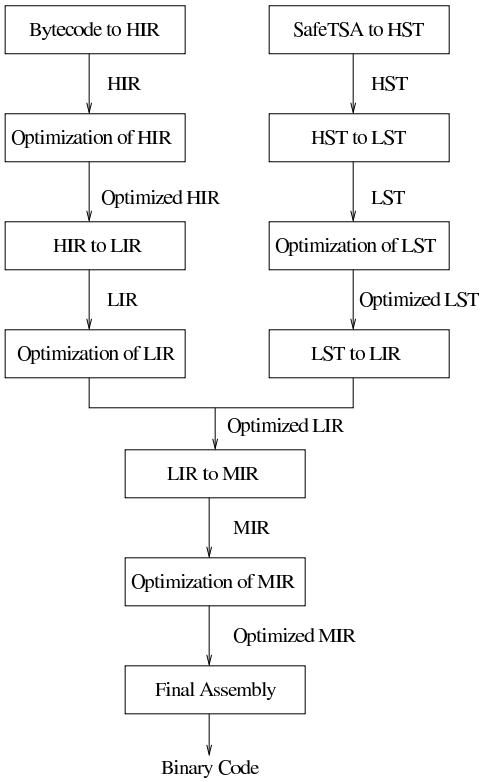


Figure 6. Compiling JVML and SafeTSA methods in Jalapeno

intermediate representation (LIR), and a machine-specific intermediate representation (MIR), as can be seen in Figure 6. A JVML method is initially translated into HIR, which can be thought of as a register-oriented transliteration of the stack-oriented JVML. The LIR differs from the HIR in that certain JVML instructions are replaced with Jikes RVM-specific implementations (that is, an HIR instruction to read a value from a field would be expanded to several LIR instructions that calculate the field address and then perform a load on that address). The lowering from LIR to MIR renders the program in the vocabulary of the target instruction set architecture. The final stage of compilation is to produce native machine code from the method's MIR. Depending on the configuration of the optimizing compiler, optimizations can be performed in each of these IRs.

Figure 6 also shows the internal structure of our SafeTSA compiler. In the first phase, the compiler transforms the method into its high-level SafeTSA representation (HST). The HST representation of the SafeTSA method is largely independent of the host runtime environment but differs from the original SafeTSA method in that there is some resolution of accessed fields and methods. Next the SafeTSA method is transformed from HST into the low level SafeTSA representation (LST). This process expands some HST instructions into a host-JVM specific LST operations, specializing the method for Jalapeno's object layout and parameter passing mechanisms. After this transformation the LST method is optimized and transformed into the same LIR used by Jalapeno's JVML optimizing compiler. The JVML compiler's LIR to MIR phase is used to perform instruction selection, scheduling, and register allocation.

The prototype compiler for SafeTSA is generally faster than the Jikes RVM Java Virtual Machine Language compiler, while producing executable code of comparable performance [4]. Considering that this is a prototype only, in which no extensive performance tuning has yet been invested, we are confident in claiming that SafeTSA is a capable substitute for JVML.

6.4 Related Work

λJVM [40] is, perhaps, the closest intermediate representation to SafeTSA. λJVM is an intermediate representation based on λ-calculus which has properties similar to SSA form [5]. It is designed as an intermediate step bridging the semantic gap between JVML (and Java) and FLINT in a type-preserving compiler [39]. Unlike SafeTSA, it has no defined serialization or file format, but only exists inside the compiler. By translating programs from JVML to λJVM and from λJVM to FLINT, Java classes are able to exist in the same type-safe environment sued by other languages, such as ML, but as Java and ML have significantly different

semantics, modules compiled from λJVM and other FLINT modules will not communicate transparently. Thus the result of this co-existence is quite differs from the commingling of JVML and SafeTSA classes within our Jikes RVM-based system, where the JVML and SafeTSA classes interact transparently. The work on λJVM also differs in that it focuses on type safety rather than performance.

In addition to Jikes RVM, there are many other JIT compilers implementing the Java Virtual Machine. The one most closely related to our work is Sun's HotSpot Server compiler [15], which uses an SSA-based internal representation similar to the one described in Click's dissertation [14]. It turns Java exceptions into explicit control flow in its IR and uses a full flow pass to discover types from the JVML. In contrast our representation explicitly marks all types.

There have also been several static JVML to machine code compilers utilizing intermediate representations based on SSA. None of these compilers, however, conserve their SSA based intermediate representation in any kind of file that could be used as a mobile code format with a just-in-time compiler.

The Swift Java Compiler [48] translates JVML 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 (such as, 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. Similar to SafeTSA, the type system of Swift can represent all of the types present in Java program. In contrast to the instruction set of SafeTSA, the instructions used by Swift are very specialized and adapted to its target architecture.

Marmot [21] is a research compiler from Microsoft that transforms JVML into Intel x86 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 (like in SafeTSA) all variables are typed, all coercion and conversions are explicit, and all overloading of operators is resolved. Marmot is tied to the x86 family of processors and does not support dynamic class loading.

7 Conclusions

We have found a new way of making mobile code simultaneously safe and efficient, by using encoding formats that cannot be used at all for transporting malicious programs. This eliminates the need for verification schemes that cost effort at the code consumer and that inhibit producer-side optimizations. The result is a genuine improvement over virtual-machine based solutions.

Acknowledgements

We had the great fortune of being under the auspices of a Program Manager at DARPA, Dr. Jaynarayan H. Lala, who both "got it" and "believed in it". Thank you, Jay, for shepherding this and all of the OASIS projects to a successful completion, and for your many wise words and insights. Thanks are also due to John Frank at Schafercorp, who made everything run like clockwork, and to Laurisa Graybill Goergen, who also had a hand in this. The authors would also like to thank Thomas Kistler, Bratan Kostov, Ziemowit Laski, Sumit Mohanty, and Christian Rattei for their research contributions in the early stages of this project, and to Vasanth Venkatachalam for proofreading the final version of this paper.

References

[1] A. M. M. Al-Hussaini. *File compression using probabilistic grammars and LR parsing*. PhD thesis, Loughborough University, 1983.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Fifteenth Annual POPL Conference*, 1988.

[3] W. Amme, N. Dalton, J. von Ronne, and M. Franz. SafeTSA: a type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 137–147. ACM Press, 2001.

[4] W. Amme, J. von Ronne, and M. Franz. Using the SafeTSA representation to boost the performance of an existing Java virtual machine. In *10th International Workshop on Compilers for Parallel Computers*, Jan. 2003.

[5] A. W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998.

[6] A. Azevedo, A. Nicolau, and J. Hummel. Java Annotation-Aware Just-In-Time Compilation System. In *ACM Java Grande Conference*, pages 142–151, June 1999.

[7] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *Lecture Notes in Computer Science*, 2139, 2001.

[8] B. Bokowski and A. Spiegel. Barat – A front-end for Java. Technical Report B-98-09, Freie Universität Berlin, Dec. 1998.

[9] 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.

[10] R. D. Cameron. Source encoding using syntactic information source models. *IEEE Transactions on Information Theory*, 34(4):843–850, July 1988.

[11] J. Choi, M. Gupta, M. Serrano, V. Shreedhar, and S. Midkiff. Escape analysis for Java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Nov. 1999.

[12] J. G. Cleary and W. J. Teahan. Unbounded length contexts for PPM. *Computer Jrnl.*, 40(2/3):67–75, 1997.

[13] 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.

[14] C. Click. *Combining Analyses, Combining Optimizations*. PhD thesis, Rice University, Houston, Texas, 1995.

[15] C. Click. High performance computing with hotspot server compiler. Talk at Sun's JavaOne, 2001.

[16] J. F. Contla. *Compact Coding Method for Syntax-Tables and Source Programs*. PhD thesis, Reading University, England, 1981.

[17] J. F. Contla. Compact coding of syntactically correct source programs. *Software-Practice and Experience*, 15(7):625–636, 1985.

[18] S. Debray, W. Evans, and R. Muth. Compiler techniques for code compression. In *Workshop on Compiler Support for System Software*, May 1999.

[19] 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.

[20] 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).

[21] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: an optimizing compiler for Java. *Software—Practice and Experience*, 30(3):199–232, Mar. 2000.

[22] M. Franz. *Code-Generation On-the-Fly: A Key to Portable Software*. PhD thesis, ETH Zurich, Mar. 1994.

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

[24] M. Franz, J. von Ronne, W. Amme, and N. Dalton. An alternative to stack-based mobile-code representations. In *Workshop on Intermediate Representation Engineering for the Java Virtual Machine*, July 2001.

[25] C. W. Fraser. Automatic inference of models for statistical code compression. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1999.

[26] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. Dyc: An expressive annotation–directed dynamic compiler for C. Technical Report Tech Report UW-CSE-97-03-03, University of Washington, 2000.

[27] A. Hartmann, W. Amme, J. von Ronne, and M. Franz. Code annotation for safe and efficient dynamic object resolution. In *2nd International Workshop on Compiler Optimization Meets Compiler Verification*, Apr. 2003. to appear.

[28] J. Hoenicke. Java optimize and decompile environment (JODE). http://jode.sourceforge.net.

[29] R. N. Horspool and J. Corless. Tailored compression of Java class files. *Software–Practice and Experience*, 28(12):1253–1268, Oct. 1998.

[30] J. Jones and S. Kamin. Annotating Java class files with virtual registers for performance. *Concurrency: Practice and Experience*, 12(6):389–406, May 2000.

[31] J. Katajainen and E. Mäkinen. Tree compression and optimization with applications. *International Journal of Foundations of Computer Science*, 4(1):425–447, 1990.

[32] J. Katajainen, M. Penttonen, and J. Teuhola. Syntax-directed compression of program files. *Software-Practice and Experience*, 16(3):269–276, 1986.

[33] T. Kistler and M. Franz. A tree-based alternative to Java byte-codes. *International Journal of Parallel Programming*, 27(1):21–34, Feb. 1994.

[34] T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems*, 22(3):490–505, 2000.

[35] T. Kistler and M. Franz. Continuous program optimization: Design and evaluation. *IEEE Transactions on Computers*, 50(6):549–566, June 2001.

[36] T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems*, 25(4):500–548, 2003.

[37] C. Krintz. Improving mobile program performance through the use of a hybrid intermediate representation. In *2nd Workshop on Intermediate Representation Engineering for Virtual Machines*, June 2002.

[38] C. Krintz and B. Calder. Using annotations to reduce dynamic optimization time. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 156–167, Snowbird, Utah, June 20–22, 2001. *SIGPLAN Notices,* 36(5), May 2001.

[39] C. League, Z. Shao, and V. Trifonov. Precision in practice: A type-preserving Java compiler. In *Compiler Construction: 12th International Conference, CC 2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 106–120, 2003.

[40] C. League, V. Trifonov, and Z. Shao. Functional Java bytecode. In *Proceedings of the 5th World Conf. on Systemics, Cybernetics, and Informatics*, July 2001. Workshop on Intermediate Representation Engineering for the Java Virtual Machine.

[41] S. Lucco. Split stream dictionary program compression. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2000.

[42] B. Meyer. *Introduction to the Theory of Programming Languages*. PHI Series in Computer Science. Prentice Hall, 1990.

[43] A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Kluwer Academic Publishers, 2002.

[44] P. Pominville, F. Qian, R. Vallee-Rai, L. Hendren, and C. Verbrugge. A Framework for Optimizing Java Using Attributes. In *Sable Technical Report No. 2000-2*, 2000.

[45] W. Pugh. Compressing Java classfiles. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 247–258, 1999.

[46] F. Reig. Annotations for portable intermediate languages. In N. Benton and A. Kennedy, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers, 2001.

[47] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Fifteenth Annual POPL Conference*, 1988.

[48] 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.

[49] Standard Performance Evaluation Corporation. SPEC JVM98 benchmarks. http://www.spec.org/osg/jvm98.

[50] R. F. Stärk and J. Schmid. The problem of bytecode verification in current implementations of the jvm. Technical report, Department of Computer Science, ETH Zurich, 2000.

[51] R. G. Stone. On the choice of grammar and parser for the compact analytical encoding of programs. *Computer Journal*, 29(4):307–314, 1986.

[52] J. Tarhio. Context coding of parse trees. In *Proceedings of the Data Compression Conference*, page 442, 1995.

[53] J. von Ronne, M. Franz, N. Dalton, and W. Amme. Compile time elimination of null- and bounds-checks. In *9th Workshop on Compilers for Parallel Computers*, June 2001.

[54] J. von Ronne, A. Hartmann, W. Amme, and M. Franz. Efficient online optimization by utilizing offline analysis and the SafeTSA representation. In *Recent Advances in Java Technology: Theory, Application, Implementation*, pages 233–241. Computer Science Press, Trinity College Dublin, 2002.

[55] J. von Ronne, N. Wang, and M. Franz. Interpreting programs in static single assignment form. Technical Report 03-12, Information and Computer Science, Univeristy of California, Irvine, Apr. 2003.

[56] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Nov. 1999.