Tamper-Proof Annotations By Construction

Technical Report 02-10

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

Michael Franz[1]
franz@uci.edu

Vivek Haldar
vhaldar@ics.uci.edu

Chandra Krintz
ckrintz@cs.ucsb.edu

Christian H. Stork
cstork@ics.uci.edu

March 2002

[1]Authors in alphabetical order.

Abstract

Dynamic compilation often comes at the price of reduced code quality, because there is not enough time available to perform expensive optimizations. One solution to this problem has been the addition of annotations by the code producer that enable a dynamic code generator on the code consumer's side to shortcut certain analysis and optimization steps. However, code annotation often creates a new problem, in that most such annotations are unsafe—if they become corrupted during transit, then the safety of the target system is in jeopardy.

In order to provide safety and to guard against potentially malicious actions, mobile programs are verified by the code recipient. Such verification is needed even when a mobile program originated in a "safe" language such as Java, because the transmission might have been corrupted by an adversary. The advanced optimizations expressed in annotations typically cannot be verified in this manner without repeating the expensive analysis that they were intended to circumvent in the first place.

In this paper, we describe a way of encoding mobile programs in a manner that makes it impossible to represent illegal programs in the first place. In such an inherently safe format, any given bit-sequence of sufficient length is guaranteed to map back to a legal program in the original encoding domain, which in our prototype is Java. Hence, any incoming program that meets Java's well-formedness criteria is guaranteed to be legal and no code verification is necessary.

Interestingly, our method also enables the tamper-proof transport of annotations along with the program. In our current implementation, we are able to perform escape analysis at the code producer's side and can encode the results of this analysis in a manner that cannot be falsified in transit. Moreover, adding these annotations increases encoding density since it reduces the number of valid choices that need to be represented, so that the addition of the annotations comes at almost no space cost.

1 Introduction

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 [33], and communicating analysis information that is too time-consuming to collect on-line [5, 30, 43, 24, 45].

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 [47, 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 [47]. 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 [47, 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, however, would introduce a runtime overhead that, could become as complex as performing the analysis itself.

In this paper, we describe a novel approach to transporting not only the annotations in a tamper-proof manner, but actually the whole program. Key to our approach is an encoding that can provably represent only valid programs and annotations and hence provide *security by construction*.

In the following, we first give an overview of mobile code security in Java. We then describe the basic concepts behind grammar-based encoding (Section 3), and how this leads to safety *by construction* (Section 4). We then explain how this technique can be used to transport the results of escape analysis in a tamper-proof manner, using an extended type system. In Section 6 we briefly describe our generic compression engine. We then present some measurements demonstrating how exceptionally dense our encoding is (Section 7). Finally, we present related (Section 8) and future (Section 9) work and then conclude the paper.

2 Mobile Code Security

Most mobile-code deployment scenarios involve a verifier at the code consumer's site that ascertains that the recipient system has not been corrupted by a potentially malicious incoming program. Such verification is necessary even when the mobile code originated in a type-safe source language such as Java, because prior to transport it has been translated into potentially insecure bytecode, which an adversary might have corrupted during transit.

The most common form of verification, as typified by the bytecode verifier for the Java Virtual Machine (JVM), essentially encompasses a symbolic execution of the program along all of its data paths. This is complex and time consuming, and hence ill-suited for scenarios involving interactive users or limited-resource environments such as handheld devices. Another problem is that the need for verification rules out many of the optimizations that one could otherwise apply at the code producer's site, but that the verifier would not be able to distinguish from malicious modifications [4].

An alternative to full-scale verification at the code consumer's site is the use of proof-carrying code (PCC) [40, 37, 38]. This approach shifts much of the workload of the verification task to the code producer, simultaneously reducing the size of the trusted code base needed at the code consumer's site, as well as the verification effort required there. For example, Sun's KVM virtual machine [34] uses a form of PCC to enable verification of JVM bytecode in linear time.

We have invented a third alternative whose main idea is to prevent the *transport* of "illegal" programs, by making it impossible to encode them in the transport format[1]. The basis of our scheme is a mapping from "legal programs" (definition to follow) onto "bit sequences". In order to achieve maximum density, every bit sequence—if it is long enough—maps backs back to a legal program. Hence, **every** bit pattern of sufficient length, including your favorite GIF of the Mona Lisa [15], is guaranteed to correspond to **some** program that is legal in the original domain. Bit patterns that aren't long enough to represent a legal program can be rejected trivially[2].

Our method is based on adaptive grammar-based compression, in such a manner that "illegal" programming constructs cannot be expressed. Somewhat surprisingly, such a compression scheme need not necessarily be restricted to modeling only the underlying language grammar, but can also include static semantics, even beyond those defined at the source-language level. As described below, we have been able to successfully incorporate the results of escape analysis into our encoding, in a manner that cannot be falsified.

Our current implementation focuses on Java. In particular, a "legal" program in our definition is one that compiles without errors according to the Java Language Specification[23]. This definition is more restrictive than simply requiring syntactic correctness; in particular, we are able to provide *Java binary compatibility* semantics at the code receiver's site. As can be expected, our format is much denser than either source code or JVM bytecode, since the encoding domain of "legal Java programs" is much smaller than domains such as "all possible source texts" In fact, we know of no denser encoding for Java at this time.

[1]This is similar to the idea described in Orwell's classic *Nineteen Eighty-Four* [41] in which the language Newspeak is derived from Oldspeak (English) by deleting all the words that could be used to express heretical thought. The general question whether it is possible to "think the unthinkable" (existence of a Gödelization of thought) goes back to Wittgenstein's *Tractatus Logico-Philosophicus*[49].

[2]Bit patterns can also be "too long", i.e., only their prefix maps to a valid program. In this case, the remainder is simply ignored. Hence, any bit pattern can be classified as mapping to either (a) an incomplete program (the pattern is too short), or (b) a legal program (the pattern is valid), or (c) a legal program plus extra bits (the beginning of the pattern contains a valid prefix). In practice, there are a few prefixes that are not legal.

3 Grammar-Based Compression

A grammar-based compressor encodes sentences of a given grammar that are known to fully conform to the grammar (i.e., there are no syntax errors). It is fairly easy to construct such an encoder simply by numbering the available choices whenever there is an alternative in a production of the grammar, and transmitting the language sentence as a sequence of choice designators. Since the decoder has the same grammar specification available to it, it is able to reconstruct the sentence based only on these choices; in particular, sequences with no choices require no extra communication at all.

For example, consider the following excerpt from a very simple grammar:

```
statement ::=
  | ‘‘LET’’ identifier ‘‘:=’’ expression
  | ‘‘WHILE’’ expression ‘‘DO’’ { statement } ‘‘END’’
  | ‘‘IF’’ expression ‘‘THEN’’ { statement } ‘‘END’’
  | ‘‘REPEAT’’ { statement } ‘‘UNTIL’’ expression.
```

To encode a *statement* in this grammar, we need to communicate which of the four choices (assignment, while-statement, if-statement, repeat-statement) we are dealing with. However, once we have selected a choice, we do not need to send additional information until we arrive at another choice. Take the program fragment

```
IF ex1 THEN
   REPEAT S1 UNTIL ex2;
   LET i := j
 END
```

Assume that the encoding of *ex1* yields the bit sequence *bits-ex1*, that the encoding of *S1* yields the bit sequence *bits-S1*, etc., the choices in *statement* are numbered (1, 2, 3, 4), and that the *END* and *UNTIL* choices are numbered zero, then the above sentence can be encoded as:

```
3  bits-ex1
  4  bits-S1  0  bits-ex2
  1  bits-i  bits-j
  0
```

Hence, in particular, we do not need to explicitly encode the facts that there is a *THEN* in the if-statement, nor that there is an ":=" in the assignment. Note how the above encoding corresponds to the depth-first traversal of the program's abstract syntaxt tree.

4 Safety By Construction

Intuitively, one senses that compression and safety must be complementary to each other—if we design an encoding that can represent only a subset of all possible programs (the "legal" ones, according to some statically decidable analysis) then there are fewer alternatives to encode and hence the encoding should be denser. This merely takes the idea of grammar-based encoding one step further: A grammar-based compressor encodes only those character sequences that are valid sentences of its grammar; now we are further limiting ourselves to those sentences of the grammar that conform to some additional static semantic constraints.

As an example of such semantics, we could design an encoding that implicitly enforces some of the typing rules of a programming language. Take the following Java program fragment:

```
class Basic {...};
class Extended extends Basic {...};
...
    static void sample() {
        Basic b1, b2; Extended x1 ,x2;
        ...
    }
...
```

Now consider which assignments can be written down inside method *sample*: some of these assignments are illegal under Java's type system, while certain others, although legal, are pointless because they assign a variable to itself. In this particular example, only half of all possible assignments are actually simultaneously legal and useful.

In a type-unaware encoding, each assignment between two variables represents one choice out of 16 (four possible left sides and four possible right sides), which might be encoded by using two bits each for each variable, for a total of 4 bits. Conversely, an encoding that incorporates static semantics might enumerate all eight useful assignments and simply use the index of the appropriate assignment in this enumeration to communicate the choice—this would require only 3 bits. Hence, incorporating the type semantics into the encoding results in a greater encoding density since there are only half as

Assignments		
useful	illegal	pointless
b1 := b2	x1 := b1	b1 := b1
b1 := x1	x1 := b2	b2 := b2
b1 := x2	x2 := b1	x1 := x1
b2 := b1	x2 := b2	x2 := x2
b2 := x1		
b2 := x2		
x1 := x2		
x2 := x1		

many assignments to choose from[3].

More importantly, the type-aware encoding is **inherently immune to malicious modifications that would undermine type safety**, since programs that violate the assignment compatibility rule cannot be represented in the first place. Hence, unlike programs expressed in JVM language, which need to be *verified* upon arrival at the target machine, assignment compatibility in the format just described need never be verified at all.

In practice, of course, the set of valid choices in an assignment might be infinite, since it may contain expressions other than simple variables. For example, when working with dynamically linked data structures we might need to encode the statement

```
l.next = l.next.next
```

However, as we explained in Section 3 above, our type-aware encoding works *in conjunction with* a grammar-based encoding that already possesses the capability of encoding these choices.

A further requirement is that the decompressor needs to be able to reconstruct the valid choices using the information available to it; i.e., based only on some static rules and the information already transmitted. In the example above, this means that the variables used in the program, as well as their types, would need to be transmitted before the actual statements. This

[3]Effectively, adding the type rules "squeezes some entropy out of the encoding domain". Hence, the resulting coding density should always be greater or equal, even if one uses a more intelligent encoding for communicating the actual choice than simply using *log(#choices)* bits. Examples of "better" encodings include Huffman[26] coding and arithmetic coding[48]. The latter is used in our actual bit-level encoding of choices, as mentioned below.

is usually easy to arrange, but as we shall see below, might in some cases limit the static properties that can be encoded.

5 Extending the Type System to Transport Unsafe Annotations Safely

In Section 4, we explained how one can design an encoding that provides *safety by construction*, by restricting the domain of "what can be encoded" to apply only to "legal" programs in the first place. In this section, we will explain how the static semantics that are enforceable in this manner need not even be part of the original programming language definition, but can correspond to compiler-inferred properties. Hence, one can use such an intrinsically safe encoding not only to transport semantically relevant information, but also performance-relevant attributes.

In particular, we have designed an encoding that can transport the results of escape-analysis in a tamper-proof manner. By this we mean that if a program can be encoded in our format at all, then its escape-analysis annotations are guaranteed to be correct. Stated the other way around, it is impossible to even hand-craft a program in our representation that contains references marked as "non-escaping" that do escape. **We know of no other technique for communicating escape-analysis results safely**. All published annotation-based solutions [5, 30, 43] are unsafe, i.e., they are never verified at the target machine.

The main idea is surprisingly similar to the encoding described in Section 4: 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[4]. The task of the encoding then becomes to disallow all assignments between variables that could possibly allow a captured reference to escape. For example, consider the following Java program fragment:

[4]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.

```
static void sample2 () {
   captured Object cap1, cap2; Object o1, o2;
   ...
}
```

The annotation "captured" in the declaration of variables *cap1* and *cap2* indicates that an analysis in the compiler front-end has determined that these variables will never be involved in an assignment that would let the referenced objects escape. As a consequence, assignments from captured to other variables must not be representable in the encoding. The following table summarizes the assignments that would be allowed or prohibited in method *sample2* under the capturedness type rules:

Assignments		
legal	illegal	pointless
cap1 := cap2	o1 := cap1	cap1 := cap1
cap1 := o1	o1 := cap2	cap2 := cap2
cap1 := o2	o2 := cap1	o1 := o1
cap2 := cap1	o2 := cap2	o2 := o2
cap2 := o1		
cap2 := o2		
o1:= o2		
o2 := o1		

Using the method described in Section 4, 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[5].

[5]Note that the additional "capturedness" type dimension needs to be considered during linking. Hence, if an adversary modifies a class in transit, changing the annotation of an *imported* reference in a method signature from captured to escaping, then that would be detected during link-time signature matching. In our current solution, if one changes the implementation of a library method in such a manner that it effects the capturedness of any parameter in its signature, then all clients of the library should be recompiled. Section 5.1 revisits this issue.

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 [5, 30, 43], 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.

5.1 Capabilities and Limitations

Our current implementation enforces the *capturedness* property of local variables, method parameters, and return references. Additionally, we register the capturedness property of newly created objects—if these are captured, then they can safely be allocated on the stack.

It is important to note that the decoder in our approach only has available to it the information that was previously transmitted by the encoder—it does not have a "global view", and in particular, only hindsight.

A limitation of our escape analysis annotation is our assumption that the entire program is analyzed and annotated. We analyze the application and all of the library routines that can be called by it during construction of the points-to escape graphs that determines the captured state of objects. However, we encode, compress, and transport only the application (not the libraries). As a result, our technique assumes that the libraries at the destination have been analyzed and annotated[6]. We believe this is a fair assumption since this analysis benefits the user by improving program performance. It is a general belief that libraries at the destination will be fully optimized. As part of library optimization, escape and similar analysis annotation will be included with the method signatures of the library metadata. At dynamic link time, we verify that the client program was compiled against a version of the library that has compatible capturedness annotations in method signatures as the one currently available on the target machine.

[6]In particular, for optimizing performance we need access to the initialization code for newly created objects.

5.2 Realization

In our current implementation, the tamper-proof transport of annotated Java programs is implemented as a source-to-source transformation; that is, our decoder reconstructs a Java source program containing **verified annotations** rather than directly generating native code in the instruction set of the target machine. This is merely an implementation convenience; we already have a working compiler that compiles directly from our safe intermediate representation into native code, but this back-end does not yet support annotations. Note that the source code being reconstructed at the code receiver's side is guaranteed not only to be syntactically correct, but also to conform to all the static semantic rules that were part of the encoding; this includes the correctness of the annotations.

At the code producer's side, our annotation insertion system is an extension of our prior annotation work [33]. The system is invoked during program encoding and enables both static analysis and execution characteristics about the program to be collected. The annotation insertion process consists of three steps: program compilation, static bytecode analysis, and decompilation.

We first compile the program to bytecode using the Java HotSpot Client JDK, version 1.3.1, available from Blackdown Corporation [6]. Following this, we perform escape analysis on the bytecode of the program as well as on that of the library methods used by the program. This analysis is enabled through the use of the Bytecode Instrumentation Tool (BIT) [35]. The BIT interface enables elements of the bytecode class files, such as bytecode instructions, basic blocks and methods, to be queried and manipulated. We then annotate the bytecode with the results of the analysis information.

The last step of the process is to convert the bytecode back to Java source code. To enable this, we use the freely available (via GPL) decompiler called jode [29]. We extended jode to convert bytecode annotations to source code annotations during decompilation. Since we start this process with source code, we are able to add local variable information (via the `-g:vars` flag to java) to the bytecode. As a result, the decompiled source is very similar to the original programs in form and control (and equivalent to it in function). The end result is an annotated Java source program that is then encoded using our inherently safe encoding method.

Our source code annotations take the form of Java comments:

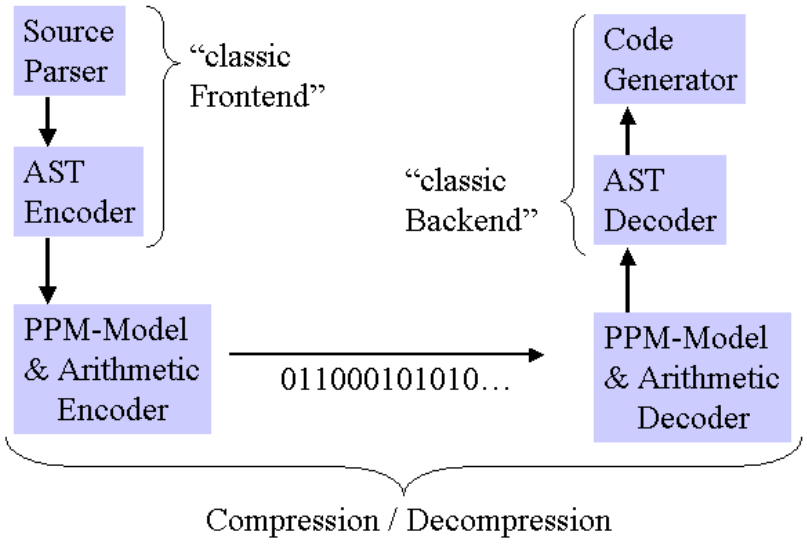
```
/*#captured*/RefType var
```

The comment immediately precedes the name of the syntactic entity for which the annotation is meant. Within the comment, the annotation begins with a `#` symbol to indicate that it is an annotation. Although we only describe our escape analysis here, we intend to use this same framework to support other annotations as well in the future.

6 Compression Using Prediction By Partial Match (PPM)

As has already been explained, our encoding mechanism uses grammar-based compression. Rather than compressing the source text directly, or rather, its parse tree, our compressor builds a canonical *abstract syntax tree* first. This abstract syntax tree (AST) corresponds to the concrete syntax stripped of all syntactic sugar; the transformation between the two is isomorphic. The abstract syntax is known to both the encoder and the decoder and obeys its so-called *abstract tree grammar*.

The AST is then encoded using a novel statistical approach, using a variant of a text compression technique called *prediction by partial match* (PPM) that we have adapted to work on trees. PPM models the statistical properties of ASTs by maintaining a set of contexts (i.e., paths form the root to a node). Each context maintains the counts of symbols that followed that context. Based on these counts, predictions can be made, and the actual choice can be encoded more space-efficiently (using arithmetic encoding) the next time this context is encountered.



Our compression framework has the following conceptual stages:

- *Parsing*: Produce an AST from the source code (i.e. a canonicalization of the parse tree).
- *Serialization*: The AST needs to be converted into a linear sequence of symbols before being encoded. We serialize the tree using a depth-first traversal.
- *Modeling*: The symbols of the serialized AST are encoded in a predictive manner, i.e. a probability of occurrence is assigned to every symbol which may occur next. The grammar and additional semantic constraints play an important role in this step, because they are used to limit the number of symbols which may possibly occur next.
- *Arithmetic Coder*: The arithmetic encoder [48] uses the model built by the previous step to output an actual stream of bits.

At the code consumer's end, the AST is incrementally rebuilt in the order it was traversed during encoding. The information being used by the modeler at the producer's end is limited to the symbols seen in the traversal up to the one currently being encoded. Hence, the coder and modeler at the consumer's end operate conceptually in lock-step with those at the producer's end.

6.1 Compressing Constants

A sizeable part of an average program consists of constants such as integers, floating-point numbers, and, most importantly, string constants and identifier names. An efficient way of encoding constants is to specify an index into a table which contains all constants. The number of bits of this index can be further reduced by maintaining different tables for different kinds of constants, such as strings, type names, identifiers and so on. Our prototype stores constants as references into various compressed tables of constants.

7 Measurements

The benchmarks we have used to empirically evaluate our results were taken from a set of applications developed by the JavaGrande Forum [19]. The following is a list of these benchmarks and a brief description of their fuctionality:

- Euler: Computational fluid dynamics
- MolDyn: Molecular dynamics simulation
- Montecarlo: Monte Carlo simulation
- Raytracer: 3-dimensional ray tracer
- Search: Alpha-beta pruned search

We first present the results of our escape analysis. Table 1 shows the number of static allocation sites that allocate captured objects (over the total number of allocation sites). Objects allocated at these sites can be stack-allocated, and synchronization performed on these objects can be removed. The type declarations of each stack-allocatable object as well as occurrences in method signatures of method parameters that do not escape inside are annotated with `/*#captured*/`. On average, 34% of the static allocation sites are such sites. These results are in line with those presented in prior work [47].

We next present our encoding relative to four different compression techniques: Jar, Pack, Gzip, and Bzip. The Java archive (jar) format is the

Program	Captured / Total	Percent Captured
Euler	16 / 48	33%
MolDyn	5 / 9	55%
Montecarlo	34 / 105	32%
Raytracer	17 / 57	29%
Search	13 / 29	44%
Avg	17 / 50	34%

Table 1: Escape Analysis Results

most common tool for collecting (archiving) and compressing Java application files [28]. The format is based on the standardized PKWare zip format [42] and enables archiving of various components of Java applications (class, image, and sound files).

Pack [44] is a jar file compression tool from the University of Maryland. This utility defines a compact representation of class file information and substantially reduces redundancy by exploiting the Java class file representation, and by sharing information between class files. The compression ratios achieved by this tool are far greater than any other compression utility for Java applications.

Gzip and bzip are both standard compression utilities, commonly used on UNIX operating system platforms. Gzip does not consider domain specific information and uses a simple, byte-oriented algorithm to compress files. As such, gzip has very fast decompression times but does not achieve the compression ratios of pack. Bzip is a freely available, high-quality data compression utility [8] that makes use of the Burrows-Wheeler method for compression.

Table 2 shows the size in bytes of bytecode programs encoded using jar, pack, gzip, and bzip compression (columns 2-5, respectively). The last two columns show the sizes (in bytes) of the programs when we use our encoding, which we call Compact Abstract Syntax Trees (CAST), on the Java source files. The encoding results for our format are given both for the case where escape-analysis annotations are included (ACAST) or omitted (CAST). No escape-analysis annotations are included in the results presented for the non-CAST encodings.

Table 3 quantifies the amount of annotation information that is actually transported in our encoding. All numbers are given in bytes. The first column

	Source			ByteCode				Compressed AST	
Program	Text	Gzip	Bzip2	Jar	Pack	Gzip	Bzip	ACAST	CAST
euler	31689	5251	5015	10250	4108	9978	10460	4171	4111
moldyn	10732	2920	2910	6533	2346	6305	6804	2290	2260
montecarlo	37210	6465	5992	20496	5340	19191	19718	5241	5065
raytracer	15690	4141	3922	12038	3079	11008	11493	2881	2773
search	11011	3247	3234	7146	2833	6797	7296	2754	2647
section3	216707	27786	23997	62243	15561	57436	58008	15944	15504

Table 2: CAST compared to compressed source and bytecode formats. For each benchmark, we show the size in bytes. The source columns show the sizes of the Java source files without compression, and when compressed with gzip or bzip2. The bytecode columns show the sizes when compressed using jar, pack, gzip, and bzip compression. The final two columns show the sizes of the programs when we use our encoding, both including annotations communicating the results of escape analysis (ACAST) as well as without (CAST). The section3 benchmark is the combination of all of the preceding benchmarks.

shows the size of the actual Java class file, the second shows the amount of annotation information one would need to add in order to transport the results of escape analysis using the annotation format of the annotation-aware Open Runtime Platform (ORP) [12] described in [33]. The third column gives the size of our CAST format *without* annotation information encoded, and the last column gives the delta between our ACAST format (including annotations) and CAST.

The size overhead for incorporating annotations into ASTs is lesser than that for doing the same for Java class files in all but one case. Evidently, beyond the advantage of being tamper-proof, our method is also quite space effective. This is important: annotation transport size should be small so as not to negate the benefit in execution time with transfer delay—especially in a wireless networking environment. Past annotation frameworks have reported file size increases ranging from 7% to 97% [43, 30, 27].

	Java With Annotations		Compressed AST	
Program	byte code	Annotation	ACAST	ACAST–CAST
euler	10250	85	4171	60
moldyn	6533	51	2290	30
montecarlo	20496	193	5241	176
raytracer	12038	125	2881	108
search	7146	71	2754	80
section3	62243	**525**	15944	**454**

Table 3: Amount of annotation information implicit in ACAST. The bytecode column shows the size of Java class file and the annotation column shows the size of additional annotation information had it been encoded as annotations in the Java classfile format. The third column shows the sizes of the programs when we use our encoding without annotation, and the final column the delta that is added (in bytes) when annotations are incorporated into our encoding.

8 Related Work

The initial research on syntax-directed compression was conducted in the 1980s primarily to reduce the storage requirements for source text files. Contla [13, 14] describes a coding technique essentially equivalent to the technique described in section 6. 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 [3] implemented another compression system based on probabilistic grammars and LR parsing. Cameron [9] introduces a combination of arithmetic coding with the encoding scheme from section 6. He assigns fixed probabilities to alternatives appearing in the grammar and uses these probabilities to arithmetically encode the pre-order 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 (including comments) to 10–17% of their original size.

Katajainen and Mäkinen [31] present a general survey of tree compression mentioning the above methods. It seems that before this survey all of the above four efforts were pursued independently of each other. Tarhio [46]

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, i.e., pools of strings, etc.) by 20% compared to a technique close to Cameron's. [7] Cheney [10] suggests applying PPM in the context of term compression.

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 (Katjainen [32] even reflect incorrect semantics in their tree). Franz [20, 21] was the first to use a tree encoding for transporting (executable) mobile code. He uses a dictionary-based encoding to compress the abstract syntax tree of Oberon programs.

Necula [39] uses a technique very similar to tree compression in order to compress PCC proofs. Rather than transmitting the entire proof, only those points in the proof are transmitted where a choice must be made among alternative paths.

Even though seemingly placed in the same application domain, research on "code compression" [18, 22, 36, 16] is generally not comparable to the above line of work on source text and AST compression. The reason is that code compression focuses much more on the specifics of machine code such as choice of opcodes, operand formats, lack of apparent high-level structure, and so on. Nevertheless, will we try to identify potential overlap between our work and other work on code compression.

Java, currently the most prominent mobile code platform, has attracted much attention with respect to compression. Horspool and Corless [25] 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 [7] further improve the compression ratio of their scheme and extend its applicability to Java archives (`jar`-files). An even better compression scheme for `jar`-files was proposed by Pugh [44]. 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). Pugh offers his tool for free evaluation.

All of the above Java compression schemes start out with the byte code

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

of Java class files. Eck, Changsong, and Matzner [17] employ a compression scheme similar to Cameron's and apply it to Java source programs. They report compression down to around 15% of the original source file, although more detailed information is needed to assess their approach.

Amme et al. [4] describe an alternative method of transporting Java programs a tamper-proof manner. Their approach tackles the problem at a different semantic level, using a form of SSA as the basis of their intermediate representation. Hence, their representation is much further removed from the original program semantics than ours, and it is not intrinsically well-formed (i.e, not every bit pattern corresponds to a valid program and some checking is necessary with their format). They do not address the problem of transporting annotations and their code sizes are far larger than ours.

9 Future Work

We are working on encoding the full and exact Java type semantics in using the technique described in Section 4, which would give us a tamper-proof mechanism for transporting Java programs that would not require any *code verification* at all (link-time matching of interfaces across class boundaries is still necessary). Unfortunately, this turned out to be far trickier than we expected, and we weren't able to finish debugging by the paper submission deadline.

We are currently also investigating other annotation-based optimizations, including inlining, optimization filtering [33], and register allocation. For the latter, complex algorithms are currently required to effectively allocate registers for Java programs. The use of such techniques in a dynamic compilation setting is infeasible due to the compilation delay imposed. However, register allocation, like escape analysis and other compilation techniques become viable if performed off-line and communicated via annotation. Since our encoding methodology eliminates all of the drawbacks previously associated with mobile code annotation (transfer size and security) we hope to enable highly optimized as well as highly-compact encoding and safe execution of mobile programs using it.

10 Summary and Conclusion

Previously, escape analysis in mobile code contexts had to be performed on the code consumer side, or be transported as *unsafe* annotations. We have solved the problem of how such annotations can be transported in a tamper-proof manner, making them safe. Our method can be extended to transport other statically verifiable properties of programs.

Our technique builds on grammar-based compression. It is generic and has been applied to multiple programming languages. Yet in spite of this genericity, we have achieved encoding densities that surpass those of other published compression schemes, including dedicated ones.

References

[1] *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI)*, Vancouver, British Columbia, 18–21 June 2000. *SIGPLAN Notices* 35(5), May 2000.

[2] *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 20–22, 2001. *SIGPLAN Notices,* 36(5), May 2001.

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

[4] 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* [2], pages 137–147. *SIGPLAN Notices,* 36(5), May 2001.

[5] 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.

[6] Blackdown. Java Linux. `http://www.blackdown.org/`.

[7] 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.

[8] Bzip2 compression utility.

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

[10] J. Cheney. Statistical models for term compression. In *Data Compression Conference*, page 550, 2000.

[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] M. Cierniak, G.-Y. Lueh, and J. N. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI)* [1], pages 13–26. *SIGPLAN Notices* 35(5), May 2000.

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

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

[15] L. da Vinci. *Portrait of Mona Lisa (1479-1528), also known as La Gioconda, the wife of Francesco del Giocondo; Oil on wood, 77 x 53 cm (30 x 20 7/8 inches)*. Musee du Louvre, Paris, 1503-06.

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

[17] 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.

[18] 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.

[19] J. G. Forum. The Java Grande Forum benchmark suite.

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

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

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

[23] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition.* Addison Wesley, 2000.

[24] 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.

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

[26] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the Institute of Electronics and Radio Engineers*, 40:1098–1101, 1952.

[27] J. Hummel, A. Azevedo, D. Kolson, and A. Nicolau. Annotating the Java Bytecodes in Support of Optimization. In *Journal Concurrency:Practice and Experience, Vol. 9(11)*, Nov. 1997.

[28] S. M. Inc. The Java ARchive utility. http://java.sun.com/products/jdk/1.1/ docs/tooldocs/solaris/jar.html.

[29] Jode. Java optimize and decompile environment. `http://jode.sourceforge.net/`.

[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] 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* [2], pages 156–167. *SIGPLAN Notices,* 36(5), May 2001.

[34] Connected limited device configuration (cldc) and the k virtual machine. See online at http://java.sun.com/products/cldc/ for more information.

[35] H. Lee and B. Zorn. BIT: A tool for instrumenting Java bytecodes. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems (USITS97)*, pages 73–82, Monterey, CA, Dec. 1997. USENIX Association.

[36] S. Lucco. Split-stream dictionary program compression. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI)* [1], pages 27–34. *SIGPLAN Notices* 35(5), May 2000.

[37] G. C. Necula. Proof-carrying code. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 106–119, Paris, France, Jan. 1997.

[38] G. C. Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Sept. 1998.

[39] G. C. Necula. A scalable architecture for proof-carrying code. In *Fifth International Symposium on Functional and Logic Programming*, Waseda University, Tokyo, Japan, 7–9 Mar. 2001.

[40] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–243, Seattle, WA, Oct. 1996.

[41] G. Orwell. *Nineteen Eighty-Four*. Martin Secker & Warburg, London, 1949.

[42] PKWare Inc. `http://www.pkware.com/`. PKZip format discription: `ftp://ftp.pkware.com/appnote.zip`.

[43] 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.

[44] W. Pugh. Compressing Java class files. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI)*, pages 247–258, Atlanta, Georgia, 1–4 May 1999. *SIGPLAN Notices* 34(5), May 1999.

[45] 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.

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

[47] 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.

[48] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.

[49] L. Wittgenstein. *Tractatus Logico-Philosophicus*. 1921.