The Source is the Proof

Vivek Haldar Christian H. Stork Michael Franz
University of California, Irvine
CA 92612
USA

{vhaldar,cstork,franz}@ics.uci.edu

ABSTRACT

We challenge the apparent consensus for using bytecode verification and techniques related to proof-carrying code for mobile code security. We propose an alternative to these two techniques that transports programs at a much higher level of abstraction. Our high-level encoding can achieve safe end-to-end transport of program source semantics. Moreover, our encoding is safe by construction, in the sense that unsafe programs cannot even be expressed in it. We contrast our encoding with certifying compilation and bytecode-based approaches, and describe how it overcomes some of their deficienciesl

Keywords

Mobile code, language-based security, abstract syntax trees.

1. INTRODUCTION

Mobile code has become increasingly widespread in many forms – executable content on web pages, application plugins, scripts, and even entire applications. It is now commonly used on a large gamut of hardware, from smart cards (JavaCard) to eCommerce servers. It usually originates from myriad untrusted sources, so a major concern is its security. Malicious (or even buggy) mobile code could consume system resources, leak secret data, or attack a system in a variety of ways. We need to make some guarantees about its behaviour before executing it.

The problem of mobile code security has recently been the focus of much research. Broadly, the goal is to make secure mobile code representations, and accompanying mechanisms that can check the code for conformance with a security policy before execution. This checking could be static (at compile-time or load-time), dynamic (at run-time), or a combination of both.

Currently there are two major thrusts driving mobile code research. Industry and consumers mostly use the Java virtual machine and its bytecode representation. In the research community, the most interest has been generated by the technique of proof-carrying code (PCC), certifying compilation, and its variants.

In this paper, we describe an alternative to both these techniques. We challenge some of their underlying assumptions and examine some of their deficiencies. Our approach, called WELL (Wellformed Encoding at the Language-Level), is based on transporting compressed abstract syntax trees (CASTs). We compare WELL with certifying compilation and bytecode-based approaches, and also discuss its advantages and pitfalls.

The rest of the paper is structured as follows: section **2** presents our WELL encoding; section **3** presents an evaluation of our encoding, and compares it with PCC and bytecode-based approaches; we give the status of our work and outline future work in section **4**; section **5** concludes.

2. HIGH-LEVEL MOBILE CODE

In this section we explain WELL-encoding, and then go on to show how it also turns out to be amenable to safely carrying annotations.

2.1 Transporting Source-Level Semantics

By "safety" of code we mean *type safety*. Type-safe code respects the typing discipline of the language in which it was written. This allows us to make safety guarantees about the *runtime* behaviour of a program by *statically* checking it for type safety.

Consider a program (written in a high-level typed language such as Java) that successfully typechecks. Assuming the type system of the language is sound, this program can now be deemed safe with respect to the type system. In essence, the proof of safety is in the source code itself. This is what we mean when we refer to a source-level proof. The question now is how to *transport* this proof of safety in a manner such that it can be recovered at the consumer's end.

Our solution is to transport something very close to the source level – abstract syntax trees. An abstract syntax tree (AST) is the parse tree of a program stripped of superfluous syntactic elements, such as brackets, which are unnecessary because the structure of the program is given by the AST. Our encoding of abstract syntax trees can safely transport source level semantics. We enforce these semantic constraints as an intrinsic part of the encoding itself. As a result, programs that do not satisfy these constraints are inexpressible in our encoding. Thus, we guarantee safe transport of source-level semantics by construction. We call this *wellformedness by construction.*

We illustrate the principal ideas behind our encoding with the help of a simple example. Consider the following Java code snippet:

```
int i, j, k;
    char a, b, c;
    j = 10;
    i = j; //encoding this
```

Consider the encoding of the last statement, given that everything before it has been encoded. WELL encoding

proceeds by walking the AST in pre-order[1], and performing the following two steps at every node:

1. Given all the constraints applicable at this point, generate the possible legal successors to this node.
2. The successor to the current node is encoded simply as its index among the successors generated in the previous step (if it is not one of the legal successors, we reject the program).

Note how the combination of these two steps – generating valid successors and then choosing *only* from among them – makes our encoding *safe by construction*. A program that does not satisfy the constraints being imposed by our encoding *cannot be expressed* in it at all. This is the central idea behind our encoding.

Also note that often there is only one legal successor generated by step 1. In this case, nothing needs to be encoded, as the next choice is obvious.

The first constraint WELL encoding imposes is conformance to the abstract grammar of the language being encoded. The relevant grammar rules (in standard EBNF notation) are:

```
Stmt = If   | While | Assign | ….
Assign = Lvalue Expr
Lvalue = Field |VarAccess | …
Expr = Unary | Binary | …
Unary = VarAccess | FieldAccess | …
```

Each node in the AST corresponds to one such rule. The first rule (Stmt) is an example of a *choice rule*, where the next valid construct is one of a number of choices. The second rule is an example of an *aggregate rule*, where the next valid construct is a *fixed* sequence of other constructs.

We are encoding a statement (`Stmt`) that is an assignment (`Assign`). The index of `Assign` is encoded from among the possible choices (`If`, `While` etc.) for the `Stmt` rule. The left hand side of the assignment (`Lvalue`) is a variable access (`VarAccess`). Since Assign is an aggregate rule, its successors are fixed, so nothing needs to be encoded. Now we need to generate possible choices for the `VarAccess` rule. At this point, we impose another constraint – lexical scoping. There are six variables in scope, and we encode `i` as the successor. Now we impose yet another constraint – the typing rules of the language. Given that the left hand side is an integer, the right hand side must be of a compatible type. Even though there are six variables in the current scope, only three are of a compatible type (`int`). The next choice must be made from only among these three. Note how the encoding rules out any possibility of expressing an illegal assignment.

The decoder runs conceptually in lockstep with the encoder. It knows the same set of semantic constraints as the encoder. It also generates possible legal successor nodes at every step. However, now the decoder needs to know which node among the successors to choose from. For this, it looks up the index of the successor from the encoded file. If the file is tampered with during transit, the possibilities are:

1. The index looked up from the file refers to one of the valid choices. In this case, even though the program was tampered with, it conforms to the safety constraints of the decoder, and decoding proceeds. The decoded program may do something different than intended or nonsensical, but it is guaranteed to be safe[2].
2. The index looked up from the file does not refer to one of the valid choices. This is not possible since we arithmetic encoding and any bit pattern will yield one of the indices.
3. The file ends prematurely. The program is rejected.

In all three cases, unsafe programs are rejected. Our encoding is essentially a mapping from valid programs to bit-sequences. Every bit-sequence of sufficient length is guaranteed to map back to a valid program conforming to the semantic constraints enforced by the encoding.

Here we have explained our encoding in broad strokes. The encoding of choices in step 2 is much more complicated than explained here. We do a probabilistic modelling of the possible choices, and use it to drive an arithmetic encoder to generate the actual bits that are transmitted. This allows us to achieve an excellent compression factor that is better than the best published scheme for Java bytecode. This is expected, because our semantic constraints greatly reduce the encoding space. For more details, see [13] and [15].

2.2 Transporting Annotations Safely

One of the requirements of mobile code is quick start-up time. Since mobile code is in some machine-independent intermediate representation, we need to generate native code from it prior to execution[3]. However, this usually happens at run-time while the user is waiting, and must be fast. It is also very common to use mobile code in a framework that uses dynamic optimization to incrementally improve code quality[5].

We can improve both start-up time and quality of generated native code by shipping relevant annotations along with the mobile code. Typically, these annotations are the results of some static analysis, and help the consumer to generate better code faster. The goal is to shift the load of compute-intensive analyses to the producer.

Annotations can be broadly classified into two types – those that do not affect program semantics, but are only helpful hints to the compiler[4]; and those that do affect program semantics (such as array bounds-check elimination, or escape analysis). Annotations of the first

[1] or any other traversal order chosen beforehand

[2] Integrity of the transmitted file can be easily verified by using a cryptographic digest function. This is completely independent of our encoding.

[3] The other option, interpretation, is too slow and rarely used.

[4] An example of this is annotations that simply point out methods where greater optimization effort should be spent.

kind do not need to be verified, because even if they are tampered with correct code is generated. Annotations of the latter kind, however, do need to be verified. Maliciously tampered annotations could lead to unsafe code being generated an executed. As a simple example, suppose we are shipping the results of an array bounds check analysis by indicating which array accesses do not need to be bound-checked. If a malicious annotation is inserted to indicate that a bound-check is not required for an array access where is a bound-check is indeed required, this opens a security hole where arbitrary memory could be accessed by indexing this array.

It turns out that WELL encoding is very suitable for carrying a certain class of annotations safely. As proof-of-concept, we have augmented WELL to transport the results of escape analysis. Escape analysis indicates when an object allocated dynamically on the heap does not escape its enclosing scope (i.e. no pointers outside its scope refer to it). Objects that do not escape (or are captured) can then be allocated on the stack and be automatically freed on exit from the scope. This reduces garbage collection overhead, and has been reported to give significant performance gains[2].

We transport escape analysis information by essentially adding type modifiers for escaped and captured objects, and then enforcing some semantic source-level rules to make sure that objects with these annotations are used consistently. For example, we do not allow a captured object to be assigned to an escaping reference. The consumer does not have to repeat the compute-intensive analysis, but can quickly verify the results. Full details are beyond the scope of this paper – we refer the reader to our technical report[14].

3. COMPARISON AND DISCUSSION

Our goal is to explore the design space for mobile code intermediate representations. Currently, the most prevalent mobile code format is Java bytecode[6]. In this paper, we present an alternative mobile code representation that is at a much higher level than bytecode – compressed abstract syntax trees. We have a prototype implementation of an end-to-end system that is able to replace the Java classfile format as a format for mobile code transportation. At the same time, our high level encoding has a number of advantages over traditional bytecode-based approaches.

3.1 Bytecode considered harmful

Our work was partly motivated by a number of shortcomings of bytecode-based approaches to mobile code. The first and most important among these is the large *semantic gap* between the language being encoded and bytecode that is actually transported. In the Java language and its corresponding virtual machine for example, type-unsafe accesses can be made in bytecode, but not in Java; arbitrary jumps can be made in bytecode, but not in Java. Fundamentally, all the effort expended for bytecode verification is to make sure that this semantic gap is not maliciously exploited. This semantic gap is further highlighted by the existence of legal Java programs that are rejected by all bytecode verifiers[12].

By using 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, we 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 point of view of the programmer, it is the semantics of the source language that she understands, 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 our format guarantees adherence to the semantic constraints that we enforce, or is invalidated, 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 its 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 our 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[5] uses 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, a major factor in understanding code, are simply indices in our encoding. Low level representations such as bytecode offer only illusory protection of intellectual property. This is readily demonstrated by freely available bytecode decompilers such as JODE[17]. Moreover, recent theoretical results question the very possibility of obfuscating well[16].

Furthermore, distributing code in source language-equivalent form provides the runtime system with the choice of a platform-tailored intermediate representation. For example, it is possible to use an existing target-specific backend. As proof of concept, we have implemented a GCC-based backend.

3.2 Proof-Carrying Code and Certifying Compilation

Proof-carrying code (PCC)[7] is a technique in which the code is accompanied with a formal proof of its safety with respect to a fixed safety policy. This proof may be hard to generate, but it is easy to check. The code consumer mechanically checks the proof before executing it. For PCC to be practical, the proof must be automatically generated. PCC at the assembly-language level utilizes theorem-proving to generate proofs. Certifying compilation tries to generate PCC-annotated binaries from a high-level language.

PCC so far has been demonstrated with DEC Alpha assembly language[7], a compiler for Java bytecode to x86 assembly language[3], and a compiler from a type-safe subset of C to Alpha assembly language[5] [8]. However, it has yet to be demonstrated with a full, high-level *source* language. The fundamental reason for this is the difficulty of preserving a proof all the way from the source level to low-level assembly. It is also an open question how proofs can be generated for properties other than type-safety.

All the applications of PCC so far have been on low-level code. As we explained in section 3.1, this suffers from the same problems of having a large semantic gap from the source. Our philosophy, on the other hand, is to preserve the source-level proof, and transport that.

An issue with PCC was the size of its proofs. Recent work[9] has shown that this can be compressed down to a small overhead. Though done independently of our work, their encoding of proofs is similar in spirit to our encoding of programs[13]. The proof is essentially viewed as a tree. At a certain point in the proof tree, only a certain number of rules apply, and the checker must decide which rule to use. The job of the prover (at the code producer's end) is to point out to the checker which rule to use next. This is simply encoded as an index into the list of the rules currently applicable. This oracle-like encoding that works by "narrowing of possibilities" is very similar to ours. In our WELL encoding, the proof and the program are intrinsically bound together. The oracle-like encoding of PCC proofs approaches this idea.

One advantage of PCC is that it has a very small trusted computing base (TCB) – the proof checker. One generic checker can check a large number of properties. The trusted checker can be made even smaller by using a more basic logic and expressing other properties in it. This is the approach taken by foundational PCC (FPCC)[1]. However, FPCC is even harder to scale to real-world, high-level languages. There is clearly a trade-off here between size of the TCB and its scalability to high-level languages. Our approach is at the other end of the spectrum. We easily capture and safely transport the source-level semantics of a large language (Java). However, at the consumer end, the entire backend, which is essentially a Java compiler, is trusted.

Another problem with the PCC approach is that the security policy is fixed and must be known at the time of proof-generation. This is a limitation because security needs are varied, and it is unrealistic for the producer to know all the policies of interest to a consumer. This has motivated approaches such as model-carrying code (MCC)[11]. MCC needs to extract a model of program behaviour from code. One way to do this is to abstract a program to retain only parts of interest. Our high-level WELL encoding is particularly well-suited for this purpose, and could possibly be used as input to an MCC system.

[5] Note that PCC done with assembly language is not portable.

3.3 An end-to-end mobile code system

We envision a mobile code infrastructure where code producers distribute programs as compressed abstract syntax trees (ASTs) and code consumers deploy these programs after decompression by compiling them for their specific platform. Compressed ASTs provide the following prerequisites for mobile code: platform independence, safety, compactness, and suitability for target-specific optimization.

The code producer distributes software as compressed ASTs, which constitutes a platform-independent format at a high level of abstraction. Compression of ASTs is allowed to be computationally expensive because it is only a one-time effort performed by the code producer. The general philosophy is to shift the computational load from the consumer to the producer. Augmenting the AST encoding with hard-to-compute but easy-to-verify annotations is a step in this direction.

As proof-of-concept, we have annotated our encoding with escape analysis information, and transported it in a safe manner such that it can be quickly verified at the consumer's site[14]. We believe that the same concept can be applied, with relatively little modification, to a number of other static analysis techniques that are computationally intensive but whose results are easy to verify. The major challenge was to demonstrate an *effective technique for transporting these results safely*, and we have shown how to do that.

Compactness is an issue when code is transferred over networks limited in bandwidth, such as wireless networks. It is also becoming increasingly important with respect to storage requirements, especially when code needs to be stored on embedded devices. Processor performance has increased exponentially over storage access time in the last decade. It is therefore reasonable to investigate compression as a means of using additional processor cycles to decrease the demand on storage access[4], leading to a net gain in performance.

Our prototype implementation compresses Java ASTs, which are then compiled to native code, thereby circumventing compilation into bytecode and execution on the JVM. We chose Java as the language to compress because there have already been considerable efforts[10] to compress Java programs. This gives us a viable yardstick to gauge our results against.

4. STATUS AND FUTURE WORK

We currently have a end-to-end system based on WELL encoding. At the producer's site, Java source is encoded to WELL, which is transported. At the consumer's site, the WELL file is decoded and fed into a GCC-based backend, which generates native code.

The performance of our backend is not yet competitive. Our primary focus so far was to have a complete system rather than optimize for performance. The main thrust for future work now is to improve the performance of the backend, both in terms of start-up latency and code quality. The first step towards this is to make the backend utilize the escape analysis annotations that we transport. We would also like to explore transporting other annotations with our encoding.

5. SUMMARY

Security guarantees for mobile code are easier to reason about at the source-language level. However, the two major mobile code techniques, bytecode, and proof-carrying code and its variants, take a low-level view of mobile code. We argue that the large semantic gap between high-level source and low-level mobile code creates inefficiencies both in reasoning about security properties of the code, as well as its performance.

Our alternative mobile code representation encodes programs at a level much closer to source. It is much easier to transport source-level semantics in our encoding than in the prevalent low-level approaches. Our encoding also provides safety by construction, as illegal programs cannot even be expressed in it. Other advantages of our encoding are an excellent compression factor, and the ability to safely transport performance-enhancing annotations.

As proof-of-concept, we have an end-to-end prototype implementation that can serve as a complete replacement of the Java virtual machine pipeline.

ACKNOWLEDGEMENTS

This research effort is partially funded by the U.S. Department of Defense, Critical Infrastructure Protection and High Confidence, Adaptable Software (CIP/SW) Research Program of the University Research Initiative administered by the Office of Naval Research under agreement N00014-01-1-0854, and by the National Science Foundation, Program in Operating Systems and Compilers, under grant CCR-9901689.

REFERENCES

[1] A. Appel; "Foundational proof-carrying code"; In Proceedings of the 16th Annual Symposium on Logic in Computer Science, pages 247-256. IEEE Computer Society Press, 2001.

[2] J.D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff; "Escape Analysis for Java"; In Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99), Nov. 1999.

[3] C. Colby, P. Lee, G. C. Necula, F. Blau, K. Cline and M. Plesko; "A Certifying Compiler for Java"; In Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI00), Vancouver, British Columbia, Canada, June 18-21, 2000.

[4] M. Franz and T. Kistler; "Slim Binaries"; Communications of the ACM, 40:12, pp. 87-94; 1997.

[5] T. Kistler and M. Franz; "Automated data-member layout of heap objects to improve memory-hierarchy performance"; in ACM Transactions on Programming Languages and Systems, May 2000.

[6] T. Lindholm and F. Yellin; "The Java™ Virtual Machine Specification"; Addison-Wesley, 1999.

[7] G. C. Necula; "Proof-Carrying Code"; in Proceedings of the Conference on Principles of Programming Languages, January 1997.

[8] G. C. Necula, P. Lee; "The Design and Implementation of a Certifying Compiler"; in Proceedings of the '98 Conference on Programming Language Design and Implementation, Montreal, 1998

[9] G. Necula; "A Scalable Architecture for Proof-Carrying Code"; in The Fifth International Symposium on Functional and Logic Programming, Tokyo, March 2001.

[10] W. Pugh; "Compressing Java class files"; In Proceedings of ACM/SIGPLAN Conference on Programming Language Design and Implementation (PLDI) '99, May 1999.

[11] R. Sekar, C.R. Ramakrishnan, I.V. Ramakrishnan and S. Smolka; "Model-Carrying Code: A New Paradigm for Mobile Code Security"; in New Security Paradigms Workshop, 2001.

[12] R. F. Stärk and J. Schmid; "The Problem of Bytecode Verification in Current Implementations of the JVM"; Technical Report, Department of Computer Science, ETH Zürich, 2000.

[13] C. Stork, V. Haldar, and M. Franz; "Generic Adaptive Syntax-Directed Compression for Mobile Code"; Technical Report No. 00-42, Department of Information and Computer Science, University of California, Irvine, November 2000.

[14] C. Stork, V. Haldar, M. Beers and M. Franz; "Tamper-proof Annotations – by Construction"; Technical Report 02-10, Department of Information and Computer Science, University of California, Irvine, Mar 2002.

[15] C. Stork and V. Haldar; "Compressed Abstract Syntax Trees for Mobile Code"; in Workshop on Intermediate Representation Engineering, July 2001.

[16] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang; "On the (Im)possibility of Obfuscating Programs (Extended Abstract)"; in Advances in Cryptology (CRYPTO), 2001

[17] Java Optimize and Decompile Environment (JODE); http://jode.sourceforge.net

New Security Paradigms Workshop '02, September 23-26, 2002, Virginia Beach, Virginia.