RC23907 (W0603-067) March 8, 2006
Computer Science

IBM Research Report

INTELLECT: INTErmediate-Language LEvel C Translator

Sumit K. Jain
Intel Corporation
Jones Farm 4
2111 N.E. 25th Avenue
Hillsboro, OR 97124

Guillaume Marceau
Department of Computer Science
Brown University
Providence, RI 02912

Xiaolan Zhang, Larry Koved
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Trent Jaeger
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

INTELLECT: INTErmediate-Language LEvel C Translator

Sumit K. Jain*
Intel Corp.
Jones Farm 4, 2111 N.E. 25th Avenue
Hillsboro, OR 97124
sumit.jain@intel.com

Guillaume Marceau*
Department of Computer Science
Brown University
Providence, RI 02912
gmarceau@cs.brown.edu

Xiaolan Zhang Larry Koved
IBM T. J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10520
{cxzhang,koved}@us.ibm.com

Trent Jaeger
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802
tjaeger@cse.psu.edu

Abstract

Static analysis tools have proven to be valuable in detecting software bugs in early development stages. As Java emerges to be the language of choice for software developers, many Java analysis tools have been developed. Since many properties that are checked by these tools are also desirable in C programs, it makes sense to make these analyses available for software written in C. In this paper we present an intermediate language level C to Java translator called INTELLECT. INTELLECT preserves the precise control and data information needed by static analysis tools such that no information is lost due to the translation from the perspective of static analysis. Our experiments demonstrate that INTELLECT is sufficiently robust – it successfully translated a subsystem of a complex embedded kernel written in C. We were also able to apply analyses originally developed for Java on the translated code. Thus we believe that building such a C to Java translator is practical, and that such a tool can facilitate the reuse of existing and future Java analysis tools on programs written in C, thus greatly enhancing the return on the cost of developing such tools.

1. Introduction

Static analysis tools have proven to be valuable in detecting software bugs in early development stages [5, 20, 2, 3, 22, 8]. As Java emerges to be the language of choice for software developers, many Java analysis tools have been developed, ranging from general bug finding tools [1, 9], to sophisticated tools that can verify high-level security properties [12, 11]. Since many properties that are checked by these tools are also desirable in C programs, it makes sense to make these analyses available for software written in C. There are two solutions to this problem. We can port all these analyses to a C analysis backend. Alternatively, we can develop a translator that translates C code to Java code. We choose to go with the latter approach, because we believe it is more cost-effective – once the translator is developed, we can reuse all existing as well as future Java analyses tool on C program.

Developing such a language translator is the first step of a much larger endeavor [23] that aims to build a unified program analysis framework where we can plug and play analysis tools. By reusing existing analyses that are already developed for other languages, we save the cost of developing the same analyses for different languages, or on different analysis backends, thus significantly reducing the cost of tool building (and maximizing the return on the cost of developing these analyses).

Such a language-neutral analysis framework provides some additional benefits, one of which being that we can now analyze multi-language software, software that is written in more than one language. It is not uncommon for a complex software package to be written in multiple languages. For example, it is sometimes necessary to have mixed C code in a Java for improved performance. One would like to still be able to run static analysis tools against the mixed code. Unfortunately none of today's static analysis tools support multi-language software.

In this paper we present a tool called *INTELLECT*, standing for INTErmediate Language LEvel C Translator, that

*Work performed while being a summer intern at IBM T.J. Watson Research Center

translates programs written in C into Java code. The translation is performed at intermediate language level, because we believe that the languages are closest at such level, thus we can achieve the best accuracy. INTELLECT preserve the control and data information needed by static analysis tools such that no information is lost due to the translation from the perspective of static analysis. In other words, the analysis results would be the same if we had ported the analysis to a comparable backend for C.

Our experiments demonstrate that INTELLECT is sufficiently robust – it successfully translated a subsystem of a complex embedded kernel written in C. We were also able to apply analyses originally developed for Java on the translated code. Thus we believe that building such a C to Java translator is practical, and that such a tool can be beneficial to the static analysis community.

The remaining of the paper is organized as follows. Section 2 describes basic differences between C and Java. Section 3 discusses related work. Section 4 presents the basic translation algorithm. Section 5 details the implementation on GCC and the challenges we encountered. Section 6 presents initial experiences and results of using INTELLECT. and Section 7 concludes.

2. Problem

2.1 C vs. Java

Most C language structs have their corresponding counterparts in Java and thus can be translated in a straightforward way. Examples include expressions, field references, statements, functional calls, etc. Some of them require slight modification. For example, data structures in C become public classes in Java.

That being said, C also differs from Java significantly in a few aspects, with pointers being the most noteworthy. In the C language, the address of a piece of data is explicit and can be manipulated using pointer arithmetic (integer operations). In Java, there is no way of explicitly representing an address. Moreover, one can cast any data type to any other data type in C, whereas in Java, the type of a given object is encoded in the data itself and type casting follows *type safety* rules. Thus C is a *type unsafe* language, while Java is a *type safe* language.

Another major difference is goto statements. In C, control of the program can be directed to any other statement in the same procedure using a pair of `goto` and label instructions. In Java the control flow of a program follows a more rigid rule and `goto` statement is no longer supported.

These differences require careful handling during translation.

2.2 Classifying Translators

Depending on whether the translation occurs at the source language level, or at the IL (intermediate language) level, there are typically two types of translators: source-to-source translators and IL-to-IL translators. Source-to-source translators produce more human-readable output, which facilitates debugging. In addition, the translation is more succinct, because it occurs at a higher, more abstract level. The main problem with source-to-source translation is goto elimination [6], which adds additional transformation overhead and may result in modified data/control flow in the translated code. IL-to-IL translators do not suffer from this problem, because all intermediate languages support the goto construct in one form or another. Furthermore, intermediate languages are semantically very close to each other, even if the original languages are very different. Thus, in most cases, the translation is a one on one mapping between two ILs, with just a few exceptions. This results in a potentially much more accurate translation in terms of preserving data/control flow, compared to a source-to-source translation.

INTELLECT is an IL level translator. Before we detail the approach taken by INTELLECT in Section 4, we first give an overview of currently available C-to-Java translators in Section 3.

3. Related Work

Translating C to Java is not an easy task, and there have been a few attempts of doing it. Here we compare a few representative translators to ours.

Jazillian [10] is a commercial C-to-Java translator that takes a C file as input and produces functionally equivalent Java files. With Jazillian, the resulting Java file is expected to *execute* the same way as the original C file. As such the translation occurs at a higher, semantic level, and the resulting Java file does not necessarily have the exact same data/control flow. For instance, C library functions are replaced with functionally equivalent Java libraries which have different names and invocation conventions. The following example shows that one printf call in C is replaced with two method calls in Java.

```
printf("%3.2f",     f);

...becomes...

DecimalFormat    myFormat   =
  new DecimalFormat("###.##");
System.out.println(myFormat.format(f))          ;
```

In addition, because Jazillian needs to do quite some guess work in order to produce functionally equivalent

code, there are cases that the heuristics employed in Jazillian does not cover, and as such Jazillian is not guaranteed to work 100% of the time. In contrast, our goal is slightly less ambitious – we expect the resulting Java file to produce the same analysis results as the C file. Thus our translation occurs at a lower level and covers a much larger set of cases. The translation preserves all data/control flow information accurately (with regard to the analysis). The resulting Java file, however, does not run unmodified in general.

Ephedra [13, 14, 15] is another C-to-Java translator that shares similar goals with Jazillian. As with Jazillian, Ephedra is a source-to-source translator. Thus it suffers the same goto elimination problem. As a matter of fact, Ephedra does not deal with gotos at the time of this writing.

C2J [19] is yet another C-to-Java translator that aims to achieve the same goal as the previous two translators. Judging from the limited documentation, it appears that C2J is also a source-to-source translator and it at least suffers the same limitations as the other two translators.

Demaine [4] talks about an automatic conversion of pointers into references. His theoretical considerations are well founded, but they were not accompanied by any implementation.

4. Approach

Our IL-to-IL translation approach has two advantages over traditional source-to-source translation approaches: accuracy with regard to preserving control and data flow, and no goto complication. In the following sections, we first show why goto elimination is a complicated issue, and why traditional approaches are not suitable for our purposes. We then describe the basic translation rules. Finally we discuss a few challenging issues, highlighting the differences between our approach and the previous ones.

4.1. Goto Elimination

Goto elimination is necessary for source-to-source translators since the Java source language does not support goto statements. Most translators incorporate the standard goto elimination algorithm [6] or its variations. The basic idea is to move each `goto` statement closer to the corresponding `label` instruction in a series of steps, until they are within the same basic block and then can be replaces with more common control constructs such as `if` statements.

Figure 1 shows a simple C program with goto statements, and the corresponding translated Java code using an *ideal* [1] implementation of the goto elimination algorithm described in [6].

[1]The reason we use the word 'ideal' here is because no source-to-source translators we have experimented so far that claim their algorithm was based on [6] were capable of translate this C example correctly.

It can be seen from this example that goto elimination is a rather complicated task, and thus is error prone. As a matter of fact, so far we have yet to see an implementation of the goto elimination algorithm [6] that's provably correct.

Even assuming such an implementation exist, it will not be suitable for our purpose – applying static analysis on the translated code. Because our goal is static analysis, we want the goto elimination to preserve the original data and control flow accurately, such that the translation does not alter analysis results in any un-intended way.

The goto elimination algorithm [6] does not satisfy the above requirement. As shown in Figure 1, the translation modified the original data and control flow by introducing three new variables and using them in new conditional statements. Therefore, such translation might alter the results from static analysis (despite the fact that the translated code might run correctly and produce the same execution results). For example, an analysis engine has to support *path sensitivity* in order to correctly reason that if line 18 is executed, then line 23 must also be executed. Most analysis engines do not support path sensitivity. And since one of the uses of our translator is to compare different static analysis engines, requiring them to support path sensitivity would defeat this original purpose.

In addition to introducing side effects that alter analysis results, the standard goto elimination algorithm **??** is sub-optimal in performance. For the simple example shown in Figure 1, the translated code added 3 new variables, essentially increasing the total number of variables by 150%, which could result in significant slowdown in analysis time.

An IL-to-IL translator bypasses the goto elimination phase, because gotos are universally available at IL level. Therefore, with an IL-to-IL translator, not only do we achieve better accuracy with regard to analysis results, the translated code is also much more succinct compared to the traditional approach of goto elimination at source level.

4.2. Basic Translation

At IL level, languages are surprisingly similar, so in most cases, there is a one to one mapping between the two language constructs and the highest level of accuracy can be achieved.

Table 2 in the Appendix shows the mapping between C and Java for basic language constructs [2]. Basic types such as `char` , `int` and `float` in C are mapped directly to the same types in Java. Structures are mapped to classes with all fields set to public. Functions become public methods of a global class representing the entire file being translated. Structures and unions become classes with the corresponding fields.

[2]Note that these examples illustrate the mappings at the conceptual level for ease of understanding. The actual translation occurs at IL level

```
int main() {
    int x;
    int y = 0;
    x = 5;
    if (x) {
        x++;
        goto L1;
    }
    L2:
    x--;
    if (x) goto end;
    L1:
    y++;                    ⇒
    goto L2;
    end:
}
```

```
1.  public class Main {
2.      public static void main(String[] args) {
3.          boolean goto_end = false;
4.          boolean goto_L2 = false;
5.          boolean goto_L1 = false;
6.          int x;
7.          int y = 0;
8.          x = 5;
9.          if (x != 0) {
10.             x++;
11.             goto_L1 = true;
12.         }
13.         do {
14.             goto_L2 = goto_L2;
15.             if (goto_L2 || !(goto_L1)) {
16.                 x--;
17.                 if (x>0)
18.                     goto_end = true;
19.                 else
20.                     goto_end = false;
21.             }
22.             if (goto_end) {
23.                 break;
24.             }
25.             y++;
26.             goto_L2 = true;
27.         }
28.         while (goto_L2);
29.     }
30. }
```

Figure 1. A Goto Elimination Example.

Sideway casts (casts of class types that do not have inheritance relationship) are statically illegal in Java, so they are hidden away from the compiler with a cast to `Object` first.

4.3. Challenging Issues

4.3.1 Pointers

One of the challenges in translating C to Java is of course dealing with pointers n C. Our approach maps pointers in C to arrays of length one in Java. Dereferencing a pointer thus becomes a referencing of the 0th element of the array. Similarly, variable and fields whose address is being taken are given an extra level of dereference via arrays of size 1. Accesses to these variables in C thus need to be mapped to accesses to the 0th element of the translated array variables. Table 3 in the Appendix shows translations for pointer related data structures.

4.3.2 Function Pointers

Previous approaches deal with function pointers using reflection, which changes the data/control flow of the original program in a significant way. INTELLECT takes a more elegant approach that maps function pointers in C to virtual methods in Java, making use of Java's support for anonymous inner classes. The mapping consists of 3 steps. All function pointers types are first mapped to `Fn`. The `Fn` class is constructed with one method named `idrCall`, but it is overloaded multiple times. For each signature of different length used at an indirect call site anywhere in the program, `idrCall` is overloaded one more time. Step 1 in Table 4 in the Appendix shows how this step works.

Whenever the address of a function is being taken, the class `Fn` is extended anonymously. The member method with matching signature is overridden to now branch to the destination function, as shown in step 2 of Table 4. Finally, indirect call sites are then relinked to transit via the virtual function, as shown in Step 3.

4.3.3 Variable Argument Functions

Functions with variable arguments are implemented with a method that has one argument of an Object array type. At each call to a variable argument function, the arguments are packed into an array before being passed to the variable argument function, as shown in Table 5.

In C, variable argument functions implement their own unpacking of the arguments. No attempts is made to translate the variety of unpacking protocols available. Rather

unfortunately, this means the body of such function cannot be not processed and a warning is printed on standard error whenever one is omitted.

Indirect calls to function with a variable number of arguments will hop yet one more time. First, the default bodies of the idrCall methods package their argument into an array. They then invoke another method of `Fh`: the `varargCall` method. The `varargCall` method can then be overridden when the address of a variable argument function is being taken. The overriding method is guaranteed to receive its arguments in an array that contain the arguments passed at the call site.

4.4 Limitations

Our translation algorithm has a few limitations. First we do not deal with pointer arithmetics. Secondly, array indices are ignored and all references to array elements are mapped to references to the 0th element. We believe that these limitations do not compromise our goals of static analysis because these limitations exist in most analysis tools (C or Java). Thus no accuracy of analysis is lost due to the translation. As we move to more powerful analysis tools, however, we will refine our translation algorithm to translate array indices faithfully, and to handle simple and legal cases of pointer arithmetics.

5. Implementation

INTELLECT uses two intermediate languages, *GIMPLE*, an intermediate representation provided by the GNU C [7] compiler, and *Jimple*, an intermediate representation for Java provided in the SOOT framework developed at McGill University [21].

5.1. GIMPLE

Traditionally, GCC always compiles the source code to RTL (Register Transfer Language), a low-level intermediate language, before applying optimizations. Higher level semantic information, such as data types, structures and fields, are lost at RTL level. Therefore, it is not possible to perform higher level optimization using RTL. From security analysis point of view, this excludes a large set of security analyses, such as complete mediation and information flow analysis, because they need to be reasoned at data type level.

Fortunately, two new intermediate representations called GENERIC and GIMPLE [17] are introduced to the GCC compiler [3], opening the door for optimization and program transformation at a level higher than RTL, yet much simpler than source trees. GENERIC is a language-independent tree representation, and GIMPLE is a simplified subset of GENERIC. Due to lack of space we will not cover the details of GIMPLE, and encourage readers to consult the original paper on this topic [17]. It suffices to mention that GIMPLE resembles a parse tree, except that complex expressions are reduced to a 3-address form. GIMPLE also lowers all high-level control flow structures to conditional gotos.

On top of the two intermediate languages, a new optimization infrastructure called Tree SSA is provided in GCC [18]. The optimization framework provides three main modules: the gimplifier, Control Flow Graph(CFG) and the Static Single Assignment (SSA) module. The gimplifier converts a normal parse tree to GIMPLE format. The CFG is a directed graph that represents the execution (control flow) of a program. The SSA module transforms the GIMPLE representation to SSA form. INTELLECT uses only the first two modules.

5.2. Jimple

Jimple is part of the SOOT framework developed at McGill University [21]. Similar to GIMPLE, Jimple was developed to simplify analysis and transformation of Java bytecode. In addition, Jimple also uses 3-address form. It is thus not surprising that the grammar of Jimple is fairly close to that for GIMPLE. The similarity between the two intermediate language makes them most suitable for translation.

5.3. Translation Process

The translation process consists of three stages. In the first stage, we merge all C source files into one C file using the merger from the CIL tool kit developed at University of California at Berkeley [16]. In the next stage, the C file is processed by a modified GCC compiler that performs the GIMPLE to Jimple translation and produces the resulting Jimple files. In the final stage, we use the soot toolkit to generate Java class files from the Jimple files. In section 5.4 we detail our modification to the GCC Tree SSA infrastructure to implement the translation.

5.4. Implementation Architecture

Figure 2 shows the architecture of our translator implemented on top of the GCC Tree SSA infrastructure. We implements our translator as an additional pass after the C code is gimplified. The translator itself consists of two passes. In the first pass, we collect variables whose addresses have been taken and store them in a hash table. In the second pass, we translate GIMPLE code to Jimple format statement by statement. Variables in the hash table are

[3] Available in the new GCC release version 3.5

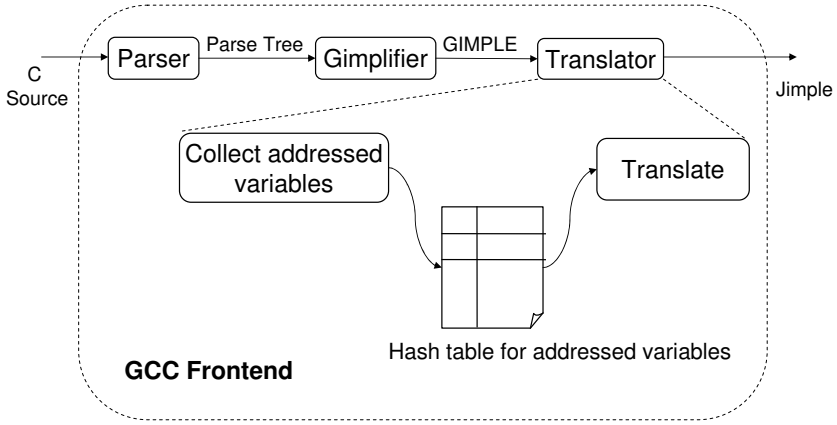

Figure 2. Implementation Architecture in GCC.

translated to corresponding array objects, and references to these variables become array references of the 0th element, as described in Section 4.3.1. For both passes we traverse the GIMPLE trees (basic blocks) in a linear fashion through the interface provided by Tree SSA framework. Whenever we need some information about any expression/declaration during the translation, we can access it from the corresponding GIMPLE tree, again by using the API provided by the framework.

To facilitate lookup, the hash table contains information about the scope in which the addressed variables have been declared. If a field of structure is being addressed then its scope is name of the structure. For a local variable inside a function, the scope is the name of the function. For globals the scope is the file name. GCC removes all the local scopes within a function and just keep one scope for that function by renaming the variables. This helps us greatly as we do not have to deal with multiple levels of scopes within a function.

5.4.1 Challenges

As close as the two intermediate languages are, there exist some disparities between them which give rise to a few challenges during translation. The disparities are largely due to two reasons: over-simplification and under-simplification of the GIMPLE intermediate language.

Over-simplification When the addresses of a structure field is taken, GCC translates expression directly to the address of the base structure plus the field offset, as shown by the example below, thus losing information on which field is accessed. We call this *over-simplification*. Since Java does not have the notion of pointers, all references are done to named objects. Thus it is necessary to keep the field information for translating the code to Jimple.

C		GIMPLE
`int * a;`		`int * a;`
	⇒	`int *t = &st;`
`a = &st.b`		`a = t+4B;`

Ideally, we would like to change the GIMPLE grammar so that the field information is kept. Unfortunately, over-simplification happens in one of the parsing phases before gimplification. That means we need to modify the AST tree grammar as well. Modifying these two grammars will break many parts of GCC, which depend on them.

We decide to use the following approach. During the parsing phase, we replace the original addressOf instruction with a dummy instruction, and hide the original addressOf instruction (e.g., `&st.b`) in a special field of the dummy instruction. We call the hidden instruction *shadow instruction*. We add two fields to the data structure that represents GIMPLE instructions, one for indicating whether there is a shadow instruction, and the other for storing the shadow instruction itself.

During the gimplification phase, we recognize these special dummy instructions and perform gimplification on the sub-components of the shadow instructions. Note that we cannot directly gimplify the shadow instructions because they are not in a form expected by the gimplification phase.

During the translation phase, we test if the shadow bit is set for every instruction, and if so, we translate the shadow instruction instead of the dummy instruction.

Under-simplification Another problem is due to the fact that GIMPLE instructions are not truly in 3-address forms.

GIMPLE allows complex expressions such as component references (e.g., `a.b.c`) and addresses of component references (e.g., `&a.b`) to be passed as parameters or in the right hand side of an assignment instruction. In these cases, GIMPLE *under-simplifies* the C language. Jimple, on the other hand, uses true 3-address forms, so such expressions are not allowed in Jimple.

The solution is to simplify these complex expressions during the translation by introducing temporary variables. For example, `f(a.b)` is translated into two statements in Jimple: `t = a.b`, and `f(t)` .

We introduce temporary variables in cases where global variables are referenced. Because global variables are translated to member fields of the enclosing `C_method` class, their references become accesses to the corresponding fields of the `this` variable. Being a strict 3-address form language, Jimple does not allow such access to occur in the parameter of a function call. Therefore, we create a temporary variable for each global variable referenced in the function, and initialize the temporary variable with the corresponding global variable. References to the global variables are thus translated to references to the corresponding temporary variables.

6. Results

Figure 3 shows the translated Jimple code for the example in Figure **??**. As expected, the translated code has the exact same control and data flow as the original C code.

We applied the tool on a kernel that implements a high-availability secure operating system. Table 1 shows the file sizes in different stages of translation.

	C	Jimple	Class File
Size(bytes)	133,646	252,210	27,418

Table 1. File Sizes in Different Translation Phases.

The translated class file is then fed into a static analysis tool called Domo **??**, which is an inter-procedural analysis framework for Java. The fact that Domo was able to analyze the resulting class file shows that the resulting class file is well-formed. The details of the analysis are beyond the scope of this paper and will be covered in a forthcoming paper [4].

[4]We are currently working on translating the Linux kernel and expect to have the results when the final version of this paper is due

```
public class C_method extends
  java.lang.Object
{
  public int main()
  {
    C_method this;
    int y;
    int x;

    this := @this: C_method;

    y = 0;
    x = 5;
    if x != 0 goto L0;
    if x == 0 goto L2;
    L0:
    x = x + 1;
    goto L1;
    L2:
    x = x - 1;
    if x != 0 goto end;
    if x == 0 goto L1;
    L1:
    y = y + 1;
    goto L2;
    end:
    return 0;
  }
}
```

Figure 3. Example of Translated C Code in Jimple Format.

7. Conclusion

In this paper we present an intermediate language level C to Java translator called INTELLECT. INTELLECT preserves the precise control and data information needed by static analysis tools such that no information is lost due to the translation from the perspective of static analysis. Our experiments demonstrate that INTELLECT is sufficiently robust – it successfully translated a subsystem of a complex embedded kernel written in C. We were also able to apply analyses originally developed for Java on the translated code. Thus we believe that building such a C to Java translator is practical, and that such a tool can facilitate the reuse of existing and future Java analysis tools on programs written in C, thus greatly enhancing the return on the cost of developing such tools.

References

[1] B. Alpern, D. W. Coleman, R. D. Johnson, A. Kershenbaum, L. Koved, G. Leeman, D. Rimer, K. Srinivas, and H. Srini-

vasan. SABER: Smart Analysis Based Error Reduction, 2003. In Submission.

[2] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001 Workshop on Model Checking of Software*, May 2001.

[3] H. Chen and D. Wagner. MOPS: An Infrastructure for Examining Security Properties of Software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244, 2002.

[4] E. D. Demaine. C to java: Converting pointers into references. *Concurrency: Practice and Experience*, 10(11–13):851–861, 1998.

[5] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operation System Design and Implementation (OSDI)*, October 2000.

[6] A. M. Erosa and L. J. Hendren. Taming Control Flow: A Structured Approach to Eliminating Goto Statements. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 229–240, May 16-19, 1994. Toulouse, France.

[7] Free Software Foundation, Inc. GNU Compiler Collection. http://gcc.gnu.org/.

[8] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 345–354, Washington, DC, 2003.

[9] IBM. DOMO. IBM internal tool for static analysis of Java code.

[10] Jazillian, Inc. How to convert c to java. Available at http://jazillian.com/how.html.

[11] T. Jensen, D. Metayer, and T. Thorn. Verification of control flow based security properties. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, May 1999.

[12] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for java. In *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002)*, pages 359–372, November 2002.

[13] J. Martin. Ephedra - a c to java migration environment, April 2002. Ph.D. Dissertation, University of Victoria, Kanada. Available at http://ovid.tigris.org/Ephedra/.

[14] J. Martin and H. A. Muller. Strategies for migration from c to java. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, pages 200–209. IEEE Computer Society, 2001.

[15] J. Martin and H. A. Muller. C to java migration experiences. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering*, pages 143–153. IEEE Computer Society, 2002.

[16] S. McPeak, G. C. Necula, S. P. Rahul, and W. Weimer. Intermediate Language and Tools for C Program Analysis and Transformation. In *Proceedings of Conference on Compiler Construction (CC'02)*, March 2002.

[17] J. Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *Proceedings of the GCC Developers Summit3*, pages 171–180, May 25-27, 2003.

[18] D. Novillo. Tree SSA: A New Optimization Infrastructure for GCC. In *Proceedings of the GCC Developers Summit3*, pages 181–193, May 25-27, 2003.

[19] Novosoft. C to java converter. Available at http://in.tech.yahoo.com/020513/94/1nxuw.html.

[20] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the Tenth USENIX Security Symposium*, pages 201–216, 2001.

[21] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[22] X. Zhang, A. Edwards, and T. Jaeger. Using cqual for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

[23] X. Zhang, L. Koved, T. Jaeger, L. Zeng, G. Marceau, and S. K. Jain. Towards a Unified Program Analysis Framework. In Submission.

APPENDIX

C	Java

int x;
float y;
struct foo a;
 	
void f1() { ... }
int f2(int p1, ...)
{ ... }
	public class C_method
	{
	  int x;
	  float y;
	  foo a = new foo();
	
	  public void f1() { ... }
	  public int f2(int p1, ...)
	  { ... }
	}

typedef struct {
  char c;
  int i;
  struct bar b;
} foo;
	public class foo {
	  char c;
	  int i;
	  class bar b = new bar();
	}

struct foo *f; struct bar *b; f = (bar*)b;	foo f; bar b; f = (bar)(Object)b;

Table 2. C to Java Mappings for Basic Language Constructs.

C	Java

struct foo *pf;
int i;
int *pi;
int ai[3];
 	
pf = malloc(sizeof(
  struct foo));
pf→i = 23;
pi = i;
	foo[] pf;
	int i;
	int[] pi;
	int[] ai = new int[3];
	
	pf = new foo[1]
	  {new foo()};
	pf[0].i = 23;
	pi[0] = i;

int i;
i = 5;
scanf(”%i”, &i);
i++;
int *pi = &i;
(*pi)++;
f(pi);
	int[] i = new int[1];
	i[0] = 5;
	scanf(”%i”), i);
	i[0]++;
	int[] pi = i;
	pi[0]++;
	f(pi);

Table 3. C to Java Mappings for Pointer Related Language Constructs.

	C	Java

Step 1.	int x;
	
	x = (*fa)(23);
	x = (*fb)(23, 42);
	x = (*fc)(”boo”);
		int x;
		...
		public class Fn {
		  ...
		  public int idrCall(int p1) { ... }
		  public int idrCall(int p1, int p2) { ... }
		  public int idrCall(String p1) { ... }
		}

Step 2.	int a(int p1) { ... }
	int b(int p1, int p2) { ... }
	int c(char *p1) { ... }
	
	(int (*fa)(int)) = &a;
	
	(int (*fb)(int, int)) = &b;
	
	(int (*fc)(char*)) = &c;
		public int a(int p1) { ... }
		public int b(int p1, int p2) { ... }
		public int c(String p1) { ... }
		
		Fn fa = new Fn() {
		  int idrCall(int p1) { return a(p1); }
		}
		Fn fb = new Fn() {
		  int idrCall(int p1, int p2) { return b(p1, p2); }
		}
		Fn fc = new Fn() {
		  int idrCall(String p1) { return c(p1); }
		}

Step 3.	x = (*fa)(23);
	x = (*fb)(23, 42);
	x = (*fc)(”boo”);
		x = fa.idrCall(23);
		x = fb.idrCall(23, 42);
		x = fc.idrCall(”boo”);

Table 4. C to Java Mappings for Function Pointers.

C	Java

void vafunc(...) ;
int a1;
char a2;
vafunc(a1);
vafunc(a1, a2);
	public abstract void vafunc (Object[] args);
	
	vafunc(new Object[] {a1});
	vafunc(new Object[] {a1, a2});

int a1;
 	
char a2;
 	
fp = &vafunc;
 	
fp(a1);
fp(a1, a2);
	public class Fn {
	  public abstract void varfunc(Object[] args);
	  public Object varargCall(Object[] args) { return null; }
	  public Object idrCall(int a1)
	  { return varargCall(new Object[] {a1}); }
	  public Object idrCall(int a1, char a2)
	  { return varargCall(new Object[] {p1, p2}); }
	}
	Fn fp = new Fn() {
	  Object varargCall(Object[] args) { return vafunc(args); }
	};
	fp.idrCall(a1);
	fp.idrCall(a1,a2);

Table 5. C to Java Mappings for Variable Argument Functions.