The Case for Analysis Preserving Language Transformation

Xiaolan Zhang cxzhang@us.ibm.com
Larry Koved koved@us.ibm.com
Marco Pistoia pistoia@us.ibm.com
Sam Weber samweber@watson.ibm.com
Trent Jaeger* tjaeger@cse.psu.edu
Guillaume Marceau† gmarceau@cs.brown.edu
Liangzhao Zeng lzeng@us.ibm.com

IBM T. J. Watson Research Center
19 Skyline Drive, Hawthorne, NY 10532, USA

ABSTRACT

Static analysis has gained much attention over the past few years in applications such as bug finding and program verification. As software becomes more complex and componentized, it is common for software systems and applications to be implemented in multiple languages. There is thus a strong need for developing analysis tools for multi-language software. We introduce a technique called Analysis Preserving Language Transformation (APLT) that enables the analysis of multi-language software, and also allows analysis tools for one language to be applied to programs written in another. APLT preserves data and control flow information needed to perform static analyses, but allows the translation to deviate from the original program's semantics in ways that are not pertinent to the particular analysis. We discuss major technical difficulties in building such a translator, using a C-to-Java translator as an example. We demonstrate the feasibility and effectiveness of APLT using two usage cases: analysis of the Java runtime native methods and reuse of Java analysis tools for C. Our preliminary results show that a control- and data-flow equivalent model for native methods can eliminate unsoundness and produce reliable results, and that APLT enables seamless reuse of analysis tools for checking high-level program properties.

*Work done while at IBM T.J. Watson Research Center. Current address: Department of Computer Science and Engineering, The Pennsylvania State University, 346A Information Sciences and Technology Building, University Park, PA 16802, USA.

†Work done while summer intern at IBM T.J. Watson Research Center. Current address: Department of Computer Science, Brown University, Providence, RI 02912, USA.

1. INTRODUCTION

Static analysis is often used to build models of software for the purpose of extracting or verifying properties of code. This has proven to have many applications in software engineering, including refactoring [34], program comprehension [17], and maintenance [46]. In particular, static analysis has been successfully applied to the area of security, where the analyses aim to determine whether or not a given piece of software violates a set of security properties [6, 7, 12, 18, 23, 24, 30, 42, 49, 50]. Examples of security properties include complete mediation of mandatory access control mechanisms [49], proper input sanitization [29], absence of vulnerabilities such as Time of Check to Time of Use (TOCTTOU) [10], buffer overflow [31] and format string [42], permission computations [30], and placement of privileged calls [38].

There are a myriad of vendors who provide customized software engineering tools tailored to particular applications and languages. Although this has been sufficient in the past, software and systems are becoming more complex and componentized, and are now frequently implemented in multiple languages. For example, the Java™ runtime libraries include "native" methods written in other languages, usually for performance or compatibility reasons. The reference implementation of the Java 1.4.2 runtime library includes 1338 native methods (about 2% of all methods).

It is thus desirable to have a multi-lingual tool that can apply the same analysis across all the different languages used by a software system. In the case of security analysis, this is a necessity rather than a luxury, because making end-to-end security guarantees or assurance statements requires that the security properties are preserved across the entire software stack. In other words, the soundness of the security analysis depends on analyzing the entire code. For example, in the case of permission analysis [30], where the least privileges required to complete a task are computed for a given task, it is necessary that the native methods are included in the analysis for the results to be sound.

Another need for general multi-lingual analysis capabilities arises in analysis reuse. Many of the high-level properties we are interested in, such as the complete mediation property, are language-independent. It is thus desirable to apply an analysis developed for one language to software written in a different language. In addition, programmers

often implement novel analysis techniques for their favorite language. It typically takes a long time for similar features to be ported to tools for other languages. For example, the SAFE Java typestate checker developed at IBM Research employs advanced techniques, such as adaptive verification based on the nature of property being verified [20, 48], and integrated aliasing and verification, which are lacking in contemporary C checkers. On the other hand, the ESP [14] checker for C uses advanced path-sensitive analysis techniques which are not present in current Java checkers. Thus, techniques that enable the analysis of multiple languages can also be used to facilitate tool reuse by allowing analysis written for one language to be applied to software written in another language. Essentially, we wish to offer users the freedom to choose a checker most suitable for the properties to be checked, without worrying about the underlying implementation language of the target program.

One approach to multi-language analysis is to have multiple analyzers, one for each target language, implementing the same analysis. Results from each analyzer are then combined at the end. This approach does not entirely satisfy the completeness requirement, because each analysis is local to the specific component, and cross-component interactions are lost due to the localized analysis scope. As a result, each analysis is only partial, thus the soundness of the analysis can not be guaranteed. In addition, this approach is not cost effective: the learning curve is steep for porting an analysis to each different target language. Worse, the effort is quadratic on the number of analysis, since a new port is required for each new analysis/language pair.

Another approach to solving the multi-lingual analysis problem is to build analysis frameworks by translating multiple front-ends to a *Common Intermediate Representation* (CIR) and performing the analysis on the CIR [2, 35]. Such an effort, although desirable, is not feasible in our case for two reasons: Firstly, constructing an intermediate representation that can encompass all language semantics required for the analysis to be sound is a non-trivial task, and secondly, such an approach would require us to port all currently existing analyses to the new CIR. This is not practical. Instead, we prefer a solution that allows us to reuse off-the-shelf tools that are already available on the market.

In this paper, we propose a new approach. Instead of porting the analyses, we translate the program of interest from its original language to a target language for which the analysis is to be applied. At first glance, this looks like an intractable task. Traditionally, language translation is done for the purposes of compilation, for which it is crucial that the translated program execute identically to the original program. Implementing this is a difficult task. The crucial observation that makes our approach feasible is that, for many analyses, such as ours whose purposes are to detect bugs and security violations, preserving the execution semantics is not necessary. For example, typical security-related analyses are mainly concerned with determining whether certain "bad" code paths are executed, but are not concerned about properties like whether certain expressions are constant. Therefore, in our translation, sufficient data and control flow information needed to perform static analysis are preserved, while the translation is allowed to deviate from the original program's semantics in other aspects that are not pertinent to our analysis task. We call this technique *Analysis Preserving Language Transformation* (APLT), as opposed to *Execution Preserving Language Transformation* (EPLT), where the translated program executes the same as the original version.

Compared to the traditional approach of porting analysis algorithms, our approach offers two advantages. First, because the analysis can be performed on the entire code base, the analysis is complete. Secondly, the approach is more cost effective: once the translator is developed, we can reuse it with existing analyses as well as future ones. Compared with the CIR approach, our approach has much broader applicability, as the target language is a standard language, not an intermediate language that is specific to a particular analysis engine. What we give up for this generality is precision with regard to code generation. However, our translation is precise with respect to a large class of analyses (with a few exceptions, see Section 6). If the ultimate goal is code analysis, rather than code generation, we believe the tradeoff is worthwhile.

We demonstrate the feasibility of APLT through two usage cases. Based on the APLT technology we built a C to Java translator called FICtoJ. We perform security analyses [30, 38] on the Java runtime native methods after translating them to a Java equivalent form using FICtoJ. Our preliminary results show that a control- and data-flow equivalent model for native methods can eliminate unsoundness and produce reliable results. Using FICtoJ, we have also successfully applied an advanced analysis available to Java (the SAFE model checker) to programs written in C, for which no tool with equivalent capabilities is publicly available. These results demonstrate that analysis preserving language transformation is practical. Further, it serves as a powerful foundation for solving real world software engineering challenges, in particular, multi-lingual program analysis and analysis reuse.

In the remainder of this paper, we present in more detail our proposed approach. We have implemented a specific APLT translator from C into Java, which we describe in detail in Section 3. We provide details of experiences and experimental results in Section 4. We finish by describing related work and our future research goals.

2. ANALYSIS-PRESERVING LANGUAGE TRANSFORMATION

In general, faithful translation between languages is difficult, if not impossible. Different languages support different approaches to data representation. Consequently, mapping the data operations between languages often requires a significant transformation of the structure of the program. This difficulty is particularly visible when translating low-level data manipulation (such as C pointer arithmetic) to a high-level language (such as Java).

On the other hand, there is sufficient common ground between the control-flow and the data-flow structures of many languages. Many such constructs operate similarly across a broad family of imperative languages: basic blocks, functions, conditionals and loops, as well as lexical scopes and lexical bindings. Consequently, control and data-flow operations are easier to translate faithfully than operations on data representation.

Fortunately, the classes of security properties we are interested in verifying are high-level properties [11, 12, 30, 49]. Therefore, analyses of these high level properties are more

sensitive to control and data flow, and much less sensitive to data values [21]. Below we list a few representative examples of these properties and explain why this is the case.

- **Complete Mediation.** Complete mediation [41] requires that all security sensitive operations must be mediated through a reference monitor [49].
- **Tainted Variables Analysis.** Tainted variable analysis aims to find unsanitized variables that can reach security sensitive sections of the code space and thus affect security decisions [38, 42]. For example, in the case of the Linux kernel, we are interested in whether a variable coming from an un-authenticated source (e.g., a user supplied input without sanitation) can reach security sensitive sections of the code base (e.g., inside the kernel space).
- **Access Rights Analysis.** In the Java security model, applications must obtain permissions first before invoking privileged (security sensitive) operations. Least privilege permission calculation tries to statically determine the minimum set of permissions required by the application to carry out its functionality [30].
- **Privileged Code Placement.** Some languages, such as Java and Microsoft .NET Common Language Runtime (CLR), have adopted a form of access control based on stack inspection. When access to a restricted resource is attempted, all the callers on the stack need to be granted the necessary permission. If library code attempts a security-sensitive operation that its clients did not explicitly request (such as reading from a configuration file), that operation should be made privileged. Privileged code in libraries prevents the stack inspection mechanism to reach client code. A privileged-code analysis analyzes library code to detect which code portions of that library should be made privileged to prevent client code from requiring unnecessary permissions [38].
- **TOCTTOU Vulnerability Analysis.** TOCTTOU attacks refer to attacks that exploit the timing window between the time of check and the time of the actual operation. A TOCTTOU analysis aims to catch such vulnerabilities [10].

These properties concern whether certain "bad" code paths are executed, rather than whether a variable holds a certain value at a particular execution point. Verification of such properties thus requires analyses that are less accurate than those used for code generation. On the other hand, since the target software of these analyses are often large systems, such as the Linux kernel, comprising of thousands, if not millions, lines of code, the analyses have to be sufficiently scalable to handle such large code bases.

A common tradeoff of accuracy for scalability is to collapse all references to array elements to the 0th element, essentially modeling the array as a giant union of all its elements. Also, many C analysis engines assume type safety and ignore pointer arithmetic. For instance, the CQUAL analysis tool does not attempt to model unsafe features of C, such as type casts, variable-argument functions, and arbitrary pointer arithmetic [22]. Despite this limitation, CQUAL has proved to be a valuable tool [42, 49]. When using these analysis engines we therefore can safely skip translation of pointer arithmetic without sacrificing analysis accuracy. Similarly, we generally do not need to track the array index value during translation.

The APLT approach takes advantage of these observations and does not try to translate the data values faithfully in all cases. Instead, it tackles the generally easier problem of preserving sufficiently precise control and data flow information. Given an input program, the APLT translator conservatively generates an equivalent output program with regards to control and data flow required for the analysis. In general, the input program and the output program do not execute alike (in fact, the output program probably does not run at all). However, the difference is not consequential to a typical program analysis, including those in which we are interested.

Based on the above discussion, we give a semi-formal definition of analysis-preserving language translation. Let L_1 and L_2 be two languages, A_{L_2} be an analysis for language L_2, and $\mathcal{I} : L_1 \rightarrow L_2$ be an ideal, perfect, translation function that converts all programs written in language L_1 to exactly equivalent programs in L_2. Further let $A_{L_2}(p)$ denote the result of executing analysis A_{L_2} on program p. Then we say that a translator $T : L_1 \rightarrow L_2$ that translates (possibly imperfectly) programs written in language L_1 to language L_2 is *analysis-preserving with respect to analysis* A if for all programs P written in language L_1, $A_{L_2}(T(P))$ is equivalent to $A_{L_2}(\mathcal{I}(P))$, where equivalence means that the results of the analyses are the same.

By definition, the soundness of our APLT translator depends on whether the target analysis falls into the category of analyses that the translator is designed to support. Our experiences show that the class of analyses our APLT translator supports is sufficiently general and encompassing that for almost all security analyses we are interested in, the translator is sound. In addition, the translator preserves precise control information (with a few exceptions discussed in Section 6) [1]). Thus, for analyses that concern only control flow [28, 12], the translator is sound. For more precise (and less scalable) analyses, such as those supporting path sensitivity, our translator is not sound. Even in those cases, we argue that the APLT translator is useful, because it makes analyses available for target languages that otherwise would not get analyzed at all (e.g., native code in a Java environment). In this regard, we share the same philosophy as Hallem, Chelf, Xie, and Engler on soundness vs. usefulness [24].

To illustrate the feasibility and complexity that arise in building a translator that sufficiently preserves the control and data flow information, we next describe the techniques we developed in tackling these issues, using a C-to-Java translator named FICtoJ as an example. Although the example is specific to C and Java, the techniques are sufficiently generic so they can be applied to building other translators.

3. FICTOJ: A C-TO-JAVA TRANSLATOR

The main novelty of our translator lies in two aspects: implementing function pointers using virtual methods and anonymous inner classes in Java; and implementing goto

[1]Although our translator does not handle certain exceptional cases, the translator can detect such usages and reject the program.

C	Java
`int i;`	`int[] i;`
`int *pi;`	`int[] pi;`
`int **ppi;`	`int[][] ppi;`
`*pi = 23;`	`pi[0] = 23;`
`*ppi = pi;`	`ppi[0] = pi;`
`scanf("%i", &i);`	`scanf("%i", i);`
`*i++;`	`i[0]++;`
`pi = &i;`	`pi = i;`
`(*pi)++;`	`pi[0]++;`

Table 1: C to Java Mappings for Pointer Related Language Constructs.

translation in a way that does not modify control and data flow of the original program. Although most of the translation is done at the source level, FICToJ is essentially a C to Java bytecode translator. Translation of basic data structures is done in a straightforward way (Section 3.1), whereas pointers require some special handling (Section 3.2). Section 3.3 and Section 3.4 detail our approaches to dealing with function pointers and goto statements. Due to space constraints, we only highlight the main technical challenges in this section. Our research report [26] contains a more complete treatment of this topic.

3.1 Basic Translation

Simple types such as `char`, `int` and `float` in C are mapped directly to the same types in Java. Structures and unions are mapped to classes with all fields declared as public. Functions become public methods of a global class representing the entire file being translated.

Sideway casts (casts of class types that do not have inheritance relationship) are statically illegal in Java, so they are first generalized as an `Object` instance, and when needed, cast to the appropriate type.

3.2 Pointers

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

3.3 Function Pointers

FICToJ maps C function pointers to Java virtual methods utilizing Java's anonymous inner classes. The mapping consists of 3 steps. All function pointer types are first mapped to a class `Fn`. This class is constructed with one method named `idrCall`, 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 2 shows how this step works. This approach does not modify the control flow of the program (it does introduce an additional level of indirection through the wrapper class).

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

3.4 Variable Argument Functions

Functions with variable numbers of arguments (`vararg` functions) are implemented with a method that has one argument of type `Object[]`. At each call to a `vararg` function, the arguments are packed into an array before being passed to the function.

In C, `vararg` functions implement their own argument unpacking. No attempt is made to translate the variety of unpacking protocols available. Rather unfortunately, this means the body of such functions cannot be processed and a warning is printed on standard error whenever one is omitted.

Translation of indirect calls to `vararg` functions requires another level of indirection. First, the default bodies of the indirect methods package their arguments into an array. They then invoke another method `varargCall`, which is overridden to invoke the destination function when the address of a `vararg` function is taken, as described in Section 3.3. The overriding method is guaranteed to receive its arguments in an array with the same number of arguments as provided by the call site.

Our original design point was to use Java 1.3 and 1.4 as the target environment for the translation. At that time, the Java language, compilers and runtime did not support `vararg` functions. With Java 5, it is possible to use its support for variable argument functions. In Java 5, the `vararg` capability is implemented by autoboxing the argument parameters into an `Object` array. The net effect is identical to our translation process.

3.5 Goto Translation

Although Java does not support goto statements, Java bytecode does have have a goto instruction. FICToJ performs the goto translation in two steps. First, it hides the goto statements in the Java source code and then reinserts them in the bytecode. In contrast to the standard goto elimination algorithm [19], which introduces new variables and modifies the control and data flow of the original code, our goto translation algorithm takes great care to avoid introducing spurious control or data paths so that the translation does not alter analysis results in any unintended way.

With the standard goto elimination algorithm, even if the translated code runs correctly and produces the same results, the results from static analysis might still be different due to the side effects of the translation. Figure 1 shows a simple C program and the resulting translation according to the goto elimination algorithm described in Erosa et. al. [19]. The introduction of the `goto_L1` variable and the additional `if (!goto_L1)` statement in line 10 create an infeasible path that goes from statement 5 (the true block of the `if (goto_L1)` statement) to statement 11 (the true block of the `if (!goto_L1)` statement), and finally to statement 14. Although a precise static analysis tool (e.g., one

	C	Java

Step 1.	`int (*fa)(int);`	`public class Fn {`
		`  public int idrCall(int p1) { ... }`
		`}`

Step 2.	`int a(int p1) { ... }`
	
	`fa = &a;`
		`public int a(int p1) { ... }`
		
		`Fn fa = new Fn() {`
		`  int idrCall(int p1) { return a(p1); }`
		`}`

Step 3.	`x = (*fa)(23);`	`x = fa.idrCall(23);`

Table 2: C to Java Mappings for Function Pointers.

```
 switch (i) {              1.  switch (i) {
   case 1:                 2.    case 1:
     if (cond)             3.      boolean goto_L1 = cond;
       goto L1;            4.      if (goto_L1)
     break;          ->    5.        break;
   case 2:                 6.      break;
     ...                   7.    case 2:
}                          8.      ...
...                        9.  }
L1:                        10. if (!goto_L1) {
  do_sth();                11.   ...
                           12. }
                           13. L1:
                           14.   do_sth();
```

Figure 1: A Goto Elimination Example.

```
 switch (i) {              1.  switch (i) {
   case 1:                 2.    case 1:
     if (cond)             3.      if (cond)
       goto L1;            4.        Dummy.go_to("L1");
     break;          ->    5.      break;
   case 2:                 6.    case 2:
     ...                   7.      ...
}                          8.  }
...                        9.  ...
L1:                        10. Dummy.label("L1");
  do_sth();                11.   do_sth();
```

Figure 2: Goto Elimination in FICtoJ.

that supports path sensitivity) might detect that the two true blocks cannot both be executed (because the condition expressions in the two `if` statements are negations of each other), most scalable static analysis tools, including the ones we use, do not support path sensitivity. Our goto translation algorithm, in contrast, does not introduce any new variables and infeasible paths.

Our goto translation algorithm is based on the JLAPACK tool [16] and uses a combination of simple control flow analysis and binary rewriting techniques to achieve the goal of preserving precise flow information. The algorithm consists of three stages. In the first stage, goto statements and their destination labels are converted into a dummy call pair `Dummy.go_to("dst")` and `Dummy.label("dst")`, as shown in Figure 2. In the second stage, we compile the converted Java source into Java object code. In the final stage, we use a binary rewriting tool called *shrike* [37] to replace the dummy calls with goto statements.

The rewriting of goto statements with dummy calls can potentially make certain parts of the code unreachable, which is not allowed in Java. For example, if line 9 in Figure 2 is a `return` statement, then the Java compiler will complain that line 10 is unreachable (not surprisingly, a C compiler would allow such cases).

To solve this problem, in the first stage we perform some simple control flow analysis to determine if the goto destination statement is reachable after the translation. If it is unreachable, we replace any statement before the destination statement that diverts the control flow (e.g. `break`, `continue`, and `return`) into corresponding dummy forms (e.g., `Dummy.Return()`) that will pass the Java compiler. Note that a `break` statement is replaced with a pair of `Dummy.go_to()` and `Dummy.label()` statements, where the dummy label statement is inserted at the end of scope enclosing the `break` statement. If the break statement takes a label argument, then no additional dummy label statement is created. Similarly, a `continue` statement is replaced with a pair of `Dummy.go_to()` and `Dummy.label()` statements with the dummy label statement inserted at the beginning of the inner-most loop scope.

4. RESULTS

In this section we evaluate the feasibility and effectiveness of analysis preserving language transformation, using FICtoJ as an example. To validate the correctness of our translator implementation, we selected an existing C analysis, reimplemented the same analysis on Java and compared the two results. We then demonstrated the effectiveness of APLT by showing how it can help (1) analyze multi-language software such as the Java runtime; and (2) make state-of-the-art analysis techniques readily available to languages other than the original target language.

4.1 Translator Validation

In this experiment, we test FICtoJ by comparing an analysis written for C with the same analysis ported for Java target programs. We select a target program, translate it to a Java equivalent with FICtoJ and then apply the ported Java analysis on the translated code. The comparison allows us to determine the correctness and robustness of the translator implementation.

The analysis concerns the verification of the *complete mediation* property of reference monitor interfaces, which states that all security-sensitive operations must go through the reference monitor interface. An example reference monitor is the Linux Security Modules (LSM) interface [47]. The

```
1:  /* Code from fs/read_write.c */
2:  sys_lseek(unsigned int fd, ...) {
3:    struct file * file = fget(fd);
4:    ...
5:    retval = security_ops->file_ops
                 ->llseek(file);
6:    if (retval) {
7:      // failed check, exit
8:      goto bad;
9:    }
10:   // passed check, perform operation
11:   retval = llseek(file, ...);
12:  }
```

Figure 3: An example of LSM hook.

Linux Security Module provides a Mandatory Access Control (MAC) architecture inside the Linux kernel. Before each security-sensitive operation, a hook invokes a reference monitor which determines if the current process has sufficient authority to perform the operation. In this architecture, the security of the kernel depends on verifying that each security-sensitive operation is collectively dominated by its checks. That is, all paths through the kernel leading to a security-sensitive operation include at least one corresponding check.

The code segment in Figure 3 shows an example of what LSM hooks look like. The function `sys_lseek()` implements the system call `lseek`. The security hook, `security_ops->file_ops->llseek(file)` (line 5), is inserted before the actual work (the call `llseek()` at line 11) takes place. The goal is to check that all security-sensitive operations (e.g., `llseek()`) are dominated by a check to the reference monitor (e.g., the security hook at line 5).

Verifying the complete mediation property is challenging, because there is often a long path between the location of the security check (e.g., *check that the user has delete permission*), and the location of the security-sensitive operation (e.g., *actually remove the file*). The paths are often inter-procedural, and involve multiple operations on security-sensitive data structures. It is difficult, for example, to verify that the file the kernel is about to delete is the same file whose permissions were checked against those of the process, many function calls ago.

We had previously verified the correctness of the placement of the LSM hooks [49], using the type-based program analysis tool called CQUAL [21]. Since CQUAL targets C, we were able to analyze the Linux kernel directly. Although CQUAL was able to find bugs, it also generated many false positives, which required tedious sorting by hand.

We ported the complete mediation analysis to a Java analysis engine called JaBA [30, 38], a tool developed internally in IBM. JaBA is a highly-scalable interprocedural Java bytecode analysis engine with the following characteristics: it is context-sensitive, intraprocedural flow sensitive, interprocedural flow insensitive, path insensitive, and field sensitive [40]. By using a more powerful analysis engine, we expected the analysis to be more precise.

We translated Linux kernel version 2.4.9, containing 207,670 lines of complex C code, 4,836 functions and 1,121 structure types. The resulting Java object file had a size of 1.2 megabytes. We then applied the complete mediation analysis based on JaBA.

The analysis on JaBA found all true positives that were captured using the CQUAL tool. This served as empirical evidence that our translator implementation from C to Java was correct with respect to the analysis. In addition, the analysis based on JaBA generated significantly fewer false positives (172 vs. 524 with CQUAL), indicating that the techniques employed in the dominance checker improved its accuracy compared to the previous approach.

We are encouraged by this result — it demonstrates that analysis preserving transformation is not only practical, but that it is also effective for very low-level programs such as the Linux kernel.

4.2 APLT Usage Scenarios

This section describes two interesting experimental results that we obtained with our APLT.

4.2.1 Analysis of Multi-language Software

Traditionally, Java static analyzers have been unsound due to their inability to model the execution of native methods. Even programs that do not explicitly contain native methods can end up causing the execution of native methods through the underlying libraries. Some static analyzers include stubs that represent the execution of a few native methods. However, those stubs are often manually constructed, increasing the likelihood that the control and data flow of the modeled native code is not faithfully represented. In addition, potentially, the process of creating the stubs must be repeated every time a new version of the software is released since the control and data flow may change across different releases.

The APLT presented in this paper has allowed us to *automatically* model native methods written in C with control- and data-flow equivalent Java methods, and to analyze multi-language programs that consist of Java code triggering the execution of native methods written in C. For the purposes of this paper, our technique has been applied only to some security-sensitive native methods, such as the four forms of method `AccessController.doPrivileged()`. This method is invoked by trusted library code to perform security-sensitive operations without requiring client code to be authorized [38]. Without a sound model for `doPrivileged()`, any authorization analysis of a Java program would be severely incomplete since the security-sensitive operation would not be represented in the static-analysis model.

The authorization analyses we performed were the access-rights analysis [30] and privileged-code analysis [38], both based on JaBA. To demonstrate the validity of our technique, this section shows the analyses performed on large code bases. Specifically, we analyzed Eclipse V3.1 to identify (1) the permissions required by each Eclipse plug-in and (2) which portions of plug-in code should be made privileged in order to allow Eclipse to run with a Java 2 `SecurityManager` enabled without forcing the client to be granted unnecessary permissions, which would constitute a violation of the Principle of Least Privilege [41]. The results reported are from executing the analyses on an IBM Personal Computer with an Intel 1.6 GHz Pentium M processor and 1 GB of RAM, running Microsoft Windows XP SP2. The security analyzer ran on top of a Sun Microsystems Java Runtime Environment (JRE) V1.4.2_04. The JRE functionality was made part of the analysis scope by including the JRE V1.4.2_04 system and extension libraries. For each plug-in, all its public and protected methods were considered as entry points.

Eclipse	*Classes*		*Methods*		*Nodes*		*Edges*		*Instr. (bytes)*		**doPriv.**		**Perms**	
Plugin	*w/o*	*w*	*w/o*	*w*	*w/o*	*w*	*w/o*	*w*	*w/o*	*w*	*w/o*	*w*	*w/o*	*w*
`ant.core`	394	765	1246	2679	5604	14113	13176	27648	67668	164060	52	55	28	79
`core.runtime`	728	1102	3570	5195	18023	27966	37286	58743	179187	289052	110	116	17	75
`osgi`	926	1289	4543	6296	27353	41832	65373	110608	258909	384219	466	492	18	142
`core.resources`	751	1087	3679	5063	20556	29182	56830	74505	211350	304258	204	214	21	47

Table 3: Analysis Results without (*w/o*) and with (*w*) Models for Native Methods

Table 3 shows, for some Eclipse plug-ins, the number of classes and methods actually analyzed, the number of nodes and edges in the call graph, the instruction bytes, the number of `doPrivileged()` locations suggested by the analyzer, and the number of permission requirements reported. For each of these attributes, two values are reported: those obtained *without* the models for native methods, and those obtained *with* the models for native methods. In each case, higher numbers represent higher precision. In particular, the number of `doPrivileged()` locations and permission requirements reported in Table 3 are important because they strictly represent the security problems detected by the static analyzer and quantify the precision of the security analysis. Specifically,

- Missing a `doPrivileged()` suggested location would cause client code to require unnecessary permissions at run time. If client code is not sufficiently authorized, run-time authorization failures will occur, causing the entire system to fail. Solving this problem by granting extra permissions to client code would not be a safe solution since it would constitute a violation of the Principle of Least Privilege. Therefore, an analysis that correctly reports all the suggested locations where `doPrivileged()` should be called can help preventing authorization failures and security problems.

- In Java (and other stack-inspection-based authorization systems, such as Microsoft .NET CLR), when security is enabled, every time an authorization check is performed all the callers on the stack will need to be granted the permission being checked. To model the stack inspection mechanism without missing a permission requirement, a static analyzer must over-approximate all the possible stacks of execution. To achieve this result, it is necessary to have a sound representation of the native methods encountered during the execution of a program and its underlying libraries. Otherwise, the analysis will potentially miss permission requirements.

The differences in the results obtained without and with the models for native methods, as reported in Table 3, quantify what we have just observed. Specifically, the results with the models for native methods are more precise (more `doPrivileged()` and permission requirements are detected and reported) because the automatically generated models for the four forms of `doPrivileged()` are included in the analysis scope. In particular, with the translated methods, the privilege-code analysis discovers many additional opportunities for inserting `doPrivileged()` calls: `ant.core` has 3 more, `core.runtime` has 6, `osgi` has 26, and `core.resources` has 10; and the access-right analysis detects, respectively, 51, 58, 124, and 26 new permission requirements that are missing when the translated methods are not included.

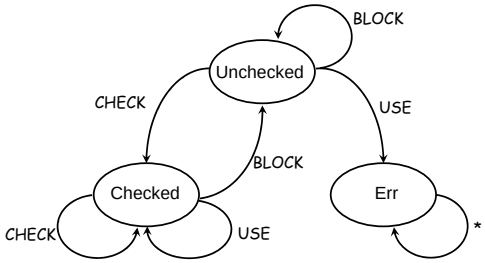

Figure 4: Typestate Model for TOCTTOU Vulnerability

4.2.2 Analysis Reuse

Our analysis of interest in this case is *typestate* verification [44, 43] a technique that uses data flow to track the typestates of variables at each program points and detect operations that violate the typestate rules of the language. Typically, the user specifies the properties to be verified using a finite state automaton. The verifier then checks if the error state in the finite automaton can be reached.

We are interested in typestate analysis for two main reasons. First, since its introduction, it has gained much attention in the space of software verification. In particular, we have found it to be suitable for verifying security properties. Secondly, the properties to be verified using typestate analysis are typically high level properties, and thus lend themselves to analysis reuse.

We next show an example of security property that can be checked using typestate analysis, namely, the (TOCTTOU) vulnerability [10]. TOCTTOU is a special type of race condition that allows an attacker to perform an unauthorized operation due to the time difference between the authorization check and the actual operation.

Figure 4 shows a simplified automaton model for checking data race induced TOCTTOU attacks in the Linux kernel. In the kernel space, TOCTTOU are generally caused by a thread's control data being modified while the thread is in a `blocked` state. Modeling the intricate interactions between threads is infeasible and not necessary for TOCTTOU detection, thus we make a conservative assumption that any blocking event might modify the states of the thread in arbitrary way, and thus any authorized checks should be invalidated. Based on this assumption, the TOCTTOU detection problem becomes checking for the simple property that there should not be a blocking event between a check and a use event.

The specific typestate checker we use for the experiment is called SAFE. SAFE employs several novel techniques [20, 48] that tailor the verification to the specific property being checked, leading to improved precision and scalability.

Using FICTOJ, we translated the Linux kernel to Java and then applied the SAFE tool on the translated code. We used Linux kernel version 2.2.14, because this version contains a known TOCTTOU bug. We applied the analysis on two subsystems, the file system and the kernel system. The two subsystems cover 127 system calls, which is about half of the total number of system calls, and 683 methods. The verification processing time was about 30 seconds, which demonstrates the efficiency of the SAFE tool. We found 10 warnings, one of which is the famous `ptrace/execve` TOCTTOU vulnerability [1] [2].

Because SAFE provides a pattern specification language for specifying high-level properties, applying SAFE to the Linux code base does not take much effort at all after the translation is done. Thus, this exercise serves as a convincing example of how APLT enables us to take immediate advantage of advanced features of tools that were not originally designed for the specific language of interest to us.

5. RELATED WORK

This section describes previous work in the areas of multilingual analysis frameworks and program translation.

5.1 Multilingual Analysis Frameworks

There are now a number of compilers and compiler frameworks that support multiple source languages [2, 25, 3]. The Vortex compiler, for example, has front-ends for several Object-Oriented languages including Java and C++. These Vortex front-ends translate code in each language into the common Vortex RTL intermediate representation, which can then be further processed.

As these compilers are able to process multiple languages, and implement the data and control flow analyses that are expected of modern compilers, it is natural to ask whether it is possible for powerful multilingual analysis tools to be built using this same approach. In other words, could code written in multiple languages be represented using the same intermediate representation, to be used by deep analysis tools?

Indeed, many such compilers provide support for such analysis tools: Vortex, and Suif among them. Other analysis frameworks that use this technique but are not primarily compilers include BANE [4] and LLVM [32].

Unfortunately, when attempting to use these systems for deep analysis of multiple languages (which, to be fair, is beyond what they were designed for), the same issue seems to re-occur.

As an example, in Fortran 77 [5], if two formal parameters alias each other, then neither parameter can be written to in a legal program. Clearly, this restriction makes verifying many properties much easier. The problem with common multi-lingual program representations is that in order to encode multiple languages they essentially represent a super-language encompassing all the features of all the supported languages, and thus lose language-specific restrictions. We know of no multi-lingual framework which is able to represent Fortran 77's alias restriction.

[2]A more refined automaton model will potentially reduce the false positive rate. This is ongoing work.

Sadly, any non-trivial program property in any Turing-complete language is incomputable. Furthermore, in practice only computations that can be performed in low-order polynomial time and space are feasible. Useful tools can still be built, despite these facts, because actual programs often are well-behaved, and because languages are often designed with restrictions, like Fortran 77's, that can help analyses. Losing language-specific restrictions makes the analysis problem more difficult, which can tip the balance towards infeasibility. For example, even though BANE is multi-lingual, Rountev, Milanova and Ryder's points-to analysis [39] using it is Java-specific, because it relies on Java method and field semantics in order to get useful results.

Our approach reduces the impact of this problem by isolating the impact in the language-specific translator. For example, to handle Fortran's alias reastriction, the translator can make explicit copies of function parameters, ensuring the non-existence of aliases. Since we are not executing the resulting program, the obvious inefficiency of this translation is irrelevant.

Another possible way to surmount the common-representation problem is to allow language-specific representations to be used by a common infrastructure. GENOA [15] was early work that took this approach, by supporting a query language which can be executed on arbitrary parse trees. Although useful, the information which can be thus extracted is much less than needed by modern analyses. More recently, the OpenAnalysis toolkit [45] can use multiple representations by use of an abstraction layer between the representations and the analysis engines. However, this work is currently limited to imperative languages (not object-oriented), and, as they note, cannot utilize constraints such as the above-mentioned Fortran alias restriction.

Cousot [13] observed that abstract interpretation could be defined as taking a programming language semantics as a parameter. In this way multi-language analysis tools could be built by supplying both the program to be inspected and the semantics of the language it is written in. However, to our knowledge no implementations of this have been attempted, likely because formalizing programming language semantics is usually considered to be prohibitively difficult.

5.2 Program Translation

Source-to-Source programming transformation is a standard technology for software maintenance and evolution [8, 9]. The goal is to automatically generate source code in the language of choice from a higher-level design specification, such that by modifying the specification one can evolve and/or port the software to a different language in an automatic way. The transformation is thus between a high-level specification language and a low-level implementation language.

In contrast, our goal is to transform the code in a way that preserves sufficiently precise data and control flow information (with regard to the analysis), so that results from different analyses are comparable. We thus perform direct source to source translation between two languages of comparable levels.

A large number of direct C-to-Java translators exist, including Jazillian [27], Ephedra [33], and C2J [36]. As with most source-to-source transformation systems, these translators aim to produce target code that *execute* the same

way as the source code. As such the translation occurs at a higher semantic level, but the resulting Java file does not necessarily have the exact same data/control flow, which makes them unsuitable for our purpose.

In addition, because these tools lack a high-level design specification, they need to do some guess work in order to produce functionally equivalent code. As a result there are cases that the heuristics employed in these tools might not cover, and consequently they are not guaranteed to work all the time. For example, none of these tools can deal with programs the scale of the Linux kernel. In contrast, since our goal is slightly less ambitious, our translation covers a much larger set of cases, and scales to large programs with hundreds of thousands of lines of code. The resulting Java file, however, is, in general, not intended to run unmodified.

Another major difference between these mainstream C-to-Java translators and our translator lies in the goto statement elimination algorithm, which is described in detail in Section 3.5. To summarize, conventional execution-preserving translators either use goto removal algorithms that alter the control and data flow of the source program, or they do not support goto statements at all.

6. DISCUSSION

In this section, we describe the limitations of our translator and discuss the generality of APLT.

6.1 Limitations

Our translator has a few limitations, as summarized below.

- **Pointer Arithmetic.** We do not support arbitrary pointer arithmetic.
- **Collapsing of array references.** We collapse array references to the reference of the 0th element.
- **Body of variable argument functions.** As we mentioned, we do not currently translate the body of function with a variable number of arguments.
- `setjmp/longjmp`. We do not handle `setjmp/longjmp`.
- **In-line assembly code.** We do not translate the content of embedded assembly code. Each such instance becomes calls to abstract methods, which can then be overridden.
- **Cast between pointers and integers.** The Java bytecode specification does not allow cast between integer and floating point types, and object types. As opposed to the first three limitations, which are implementation issues, this problem is harder to avoid and remains an open issue.

These limitations can potentially affect the precision of the translation and restrict the range of analyses for which the translation is sound. Fortunately, our experiences indicate that occurrences of most of these special cases are rare in production-quality code. For cases that occur relatively more frequently, such as pointer arithmetic, array references and assembly code, the types of analyses that depend on precise translation of these cases are rare. Thus, these limitations are not prohibitively restrictive. In the case where soundness must be guaranteed, our translator can detect occurrences of these special cases, and either reject the program or prompt for a manual fix.

For system-level applications, in particular, device drivers, assembly code can comprise a large portion of the code. Thus, it is important to be able to translate assembly code to data and flow equivalent high-level language. Theoretically, translating assembly code using our technique is no different from translating Java native methods: in both cases one has to model the interaction between the two languages, so as to capture the data flow correctly, and then model the control and data flow of the lower-level language. It would be interesting future work to attempt this. We speculate that this work would encounter no intrinsic limitations, but rather pragmatic ones: a translator's implementation difficulty increases as the source language becomes lower-level and the control flow becomes less structured, while the number of code bases that require such constructs decreases. There will be a point of diminishing returns reached, at which it is no longer cost effective to support additional constructs. This is the same reason that caused us not to support `setjmp/longjmp`: we actually had no examples of these statements in any of our target code.

6.2 Generality of APLT

We set out to investigate security-related technology, hence this paper's focus on applications in this domain. However, the techniques we describe are generally applicable to generic bug finding and performance diagnosis which, like our applications, base the analyses on modeling and traversing control and data flow of the program. Generic bug finding and performance diagnosis cover a large range of software engineering applications, such as analysis of uninitialized pointers and variables, deadlock detection, matching function call pairs such as `open` and `close`, race conditions, and analysis of unwanted synchronizations.

7. CONCLUSION

In this paper we propose a new program analysis methodology based upon analysis preserving language transformation. We claim that by making use of such translations we can correctly handle applications written in multiple languages, such as Java programs, which use native methods. Furthermore, this approach allows the reuse of existing analysis tools on code bases written in different languages.

We have validated this methodology by first implementing an analysis-preserving translator from C to Java. We then conducted three extensive experiments:

Linux LSM validation The Linux kernel mandates that LSM hooks mediate all sensitive operations. We used a C analysis engine to solve this problem and compared the results thus obtained with those produced by a state-of-the-art Java analysis engine on our translated code. The two analyses found the same set of violations, but the Java analysis resulted in fewer false positives.

Java native code analysis In order to correctly determine whether a Java application correctly checks permissions before executing sensitive operations, both the application's Java code and native code must be inspected. Using our technique we were able to correctly inspect Eclipse, a large, widely used application. To our knowledge, this is a novel result.

TOCTTOU detection An existing Java analysis engine, SAFE, is able to validate properties specified in typestate format for Java code. Using our technique, we were able to apply SAFE to detect a TOCTTOU violation in the Linux kernel, despite the fact that the kernel is written in C, not Java.

For these reasons, we believe that our APLT technique will prove to be valuable both as a testbed for examining different language analysis techniques, and as a unified toolset for broad program analysis.

8. ACKNOWLEDGMENTS

We wish to thank Vugranam Sreedhar for his many comments and observations during this work. We also wish to thank the anonymous reviewers of ISSTA 2006 for their insightful comments.

9. REFERENCES

[1] Security Focus Bugtraq Posting. http://cert.uni-stuttgart.de/archive/bugtraq/2001/03/msg00420.html.

[2] The SUIF 2 Compiler System. Available at http://suif.stanford.edu/suif/suif2/index.html.

[3] The vortex project. Available at http://www.cs.washington.edu/research/projects/cecil/www/vortex.html.

[4] A. Aiken, M. Fahndrich, J. S. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Types in Compilation*, pages 78–96, 1998.

[5] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *ANSI Fortran X3.9–1978*, 1978. Approved April 3, 1978 (also known as Fortran 77).

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

[7] T. Ball and S. Rajamani. The slam project: Debugging system software via static analysis. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL02)*, January 2002.

[8] R. M. Balzer, N. M. Goldman, and D. S. Wile. On the Transformational Implementation Approach to Programming. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 337–344, Oct. 1986.

[9] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution. In *Proceedings of the 26th International Conference on Software Engineering*, pages 625–634, May 2004.

[10] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, 1996.

[11] H. Chen, D. Dean, and D. Wagner. Model Checking One Million Lines of C Code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb. 4–6, 2004.

[12] 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.

[13] P. Cousot. Abstract interpretation based static analysis parameterized by semantics. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 388–394, London, UK, 1997. Springer-Verlag.

[14] M. Das, S. Lerner, and M. Seigle. ESP: Pathsensitive program verification in polynomial time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*, June 2002.

[15] P. T. Devanbu. GENOA: A customizable language- and front-end independent code analyzer. In *International Conference on Software Engineering*, pages 307–317, 1992.

[16] D. M. Doolin, J. Dongarra, and K. Seymour. JLAPACK - Compiling LAPACK Fortran to Java. Technical report, University of Tennessy, June 1998. Available at http://www.cs.utk.edu/ library/TechReports/1998/ut-cs-98-390.ps.Z.

[17] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *ICSM*, pages 602–611, 2001.

[18] 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.

[19] 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.

[20] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results, 2003.

[21] J. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*, pages 192–203, May 1999.

[22] J. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*, 2002.

[23] 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.

[24] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific static analyses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*, June 2002.

[25] IBM. The toronto portable optimizer.

[26] S. K. Jain, G. Marceau, X. Zhang, L. Koved, and T. Jaeger. INTELLECT: INTErmediate-Language LEvel C Translator. Technical Report 23907, IBM, 2006. Available at http://domino.research.ibm.com/library/cyberdig.nsf/index.html.

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

[28] 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.

[29] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the USENIX Security Symposium*, Aug. 2004.

[30] 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.

[31] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the Tenth USENIX Security Symposium*, pages 177–190, 2001.

[32] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[33] 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/.

[34] J. Morgenthaler. *Static Analysis for a Software Transformation Tool*. PhD thesis, 1997.

[35] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA, July 2003.

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

[37] R. W. O'Callahan. The shrike toolkit. Available at http://org.eclipse.cme/contributions/shrike/.

[38] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar. Interprocedural analysis for privileged code placement and tainted variable detection. In *ECOOP 2005 - Object-Oriented Programming: 19th European Conference, Glasgow, UK, July 25-29, 2005. Proceedings*, volume 3586 of *Lecture Notes in Computer Science*, pages 362–386. Springer, Aug. 2005.

[39] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for java using annotated constraints. In *Conference on Object-Oriented*, pages 43–55, 2001.

[40] B. G. Ryder. Dimensions of Precision in Reference Analysis of Object-Oriented Languages. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 126–137, Warsaw, Poland, April 2003. Invited Paper.

[41] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, Sept. 1975.

[42] 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.

[43] R. E. Strom and D. M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Transactions on Software Engineering*, 19(5):478–485, May 1993.

[44] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, Jan. 1986.

[45] M. M. Strout, J. Mellor-Crummey, and P. Hovland. Representation-independent program analysis. In *Proceedings of the The sixth ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, September 5-6 2005.

[46] A. von Mayrhauser and S. Lang. On the role of static analysis during software maintenance. In *Proceedings of International Workshop on Program Comprehension*, pages 170–177, 1999.

[47] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

[48] E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 25–34, New York, NY, USA, 2004. ACM Press.

[49] 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.

[50] X. Zhang, T. Jaegert, and L. Koved. Applying Static Analysis to Verifying Security Properties. In *Proceedings of the 2004 Grace Hopper Celebration of Women in Computing Conference*, Chicago, IL, 2004. Also available as IBM technical report RC23246 at http://domino.research.ibm.com/library/cyberdig.nsf/index.html.