Towards Cross-Platform Cross-Language Analysis with Soot

Steven Arzt
TU Darmstadt & Fraunhofer SIT, Germany
steven.arzt@cased.de

Tobias Kussmaul
TU Darmstadt, Germany
tobias.kussmaul@posteo.de

Eric Bodden
Paderborn University & Fraunhofer IEM, Germany
eric.bodden@upb.de

Abstract

To assess the security and quality of the growing number of programs on desktop computers, mobile devices, and servers, companies often rely on static analysis techniques. While static analysis has been applied successfully to various problems, the academic literature has largely focused on a subset of programming languages and frameworks, and often only on a single language at a time. Many tools have been created for Java and Android. In this paper, we present a first step toward re-using the existing Soot framework and its analyses for other platforms. We implement a front end for converting the CIL assembly code of the .net Framework into Soot's Jimple code and show that this is possible without modifying Jimple nor overly losing semantic information. The front end integrates Java/Android with CIL analysis and scales to large programs. A case study demonstrates the detection of real-world malware that uses CIL code inside an Android app to hide its behavior.

Categories and Subject Descriptors D.2.4 [*Software*]: Software Engineering; D.4.6 [*Software*]: Security and Protection; F.3.2 [*Semantics of Programming Languages*]: Program analysis

General Terms Verification, Security, Languages

Keywords Compiler, Soot, CIL, .net, Frontend, Static Analysis, Multi-Platform

1. Introduction

Current real-world programs have grown so large that, to ensure their quality and security, manual testing and code review is no longer sufficient. Customers expect the programs they depend upon to be stable and reliable, and to adequately protect the sensitive data they process. Consequently, many static-analysis techniques for various problems such as data-flow tracking (Arzt et al. 2014) have been proposed. Many of these techniques, have been designed for particular target platforms and programming languages. Popular targets of research include Java programs and Android apps. In reality, different programs are, however, often written in different programming languages. The choice of the language depends on the requirements of the deployment target, the language skills of the developer, the availability of powerful frameworks, and many other criteria. These criteria often differ significantly from the reasons why platforms or programming languages are chosen for academic research. As a consequence, the focus of research attention on individual platforms has lead to a large divergence in the availability of static-analysis tools for different platforms. This hinders the practical applicability of static-analysis tools.

SOAP'16, June 14, 2016, Santa Barbara, CA, USA
© 2016 ACM. 978-1-4503-4385-5/16/06...$15.00
http://dx.doi.org/10.1145/2931021.2931022

Some analyses (Gordon et al. 2015) are built directly on top of platfom-specific tools such as disassemblers. Migrating these tools to other platforms is a major undertaking. Other tools such as FlowDroid (Arzt et al. 2014) are based on more high-level frameworks such as Soot (Lam et al. 2011) or WALA (Watson). These tools work on the intermediate representation provided by the respective framework, such as Jimple (Vallee-Rai and Hendren 1998) in the case of Soot, or the WALA-IR in case of WALA. The intermediate representations provide a significant abstraction of assembly-level opcodes. In the case of Soot, this allows the framework to convert into Jimple not only Java bytecode, but also Android's Dalvik bytecode, through the Dexpler front-end (Bartel et al. 2012). For the analysis tool, there is little difference as to whether the client code originated from an Android or Java application. To be precise, while the tool needs to deal with the different peculiarities of the Android or Java libraries and frameworks, the tool can be agnostic to the origin of the Jimple code as such.

Supporting other languages in a shared IR, however, is more challenging. WALA offers *WALA CAst* (*WALA Common Abstract Syntax Tree*), a cross-language source front-end. This front-end currently supports Java and JavaScript as input. As the name implies, CAst, is, however, an AST-style data structure with support for multiple specialized IRs, and not a single, uniform, strongly typed IR. Therefore, one cannot seamlessly switch the framework to another input language and assume the same analyses to continue working as desired. Due to the nature of the highly dynamic JavaScript language, it remains an open question whether a truly shared IR for Java and JavaScript is actually possible or desirable.

In this paper, we make a first step towards turning Soot into a cross-language, cross-platform static analysis tool that produces the same IR irrespective of the input language. Specifically, we explain how we extended the Soot framework to convert into Jimple also the bytecode language of the Microsoft .net framework and the Mono open-source project. While this bytecode language CIL (Common Intermediate Language) is a fully managed-code language just as Java, we show that CIL code nevertheless has significant differences to Java bytecode. We explain how we model CIL's distinct features in Jimple without extending the Jimple language itself. We chose to avoid extensions to Jimple in order to allow researchers to reuse existing analyses based on Jimple without requiring changes or extensions. We hope that these ideas help further broadening the applicability of existing (also other) static-analysis frameworks to other languages and platforms.

The remainder of the paper is structured as follows. In Section 2, we discuss how code is organized in CIL as opposed to Java. In Section 3, CIL's type system and method calling conventions are discussed, before going into the details of selected CIL-specific language constructs in Section 4. Afterwards, we present how the

CIL frontend is architected in Section 5, and present the limitations of this work in Section 6. We evaluate our work in Section 7 before we present related work in Section 8 and conclude in Section 9.

2. Code Organization in CIL

In Java, every class gets compiled into its own class file. Multiple class files can be packaged together into one JAR archive. In CLI, on the other hand, a compiled class itself has no representation in the file system. Instead, the CLI runtime operates with a collection of classes stored in an *assembly* file. While at a first glance assemblies seem conceptually similar to JAR files, they are artifacts in their own right, resembling a logical package on the language level. For instance, visibility levels include an option for assembly-wide visibility for classes, methods, and fields. Code can use reflection to access an assembly and, e.g., list all classes inside it. Assemblies can be signed and used for code security, i.e., by defining that certain code may only be called from code within assembly that was signed with a specific key.

As Java has no notion of assembly visibility, nor has Jimple, our generation of Jimple code from CIL widens assembly-wide visibility (`internal` in C#) to public visibility. While this may impact some specific analyses, it retains compatibility in general. CIL bytecode represents security restrictions such as signatures and permissions as API calls or attributes in the original code. CLI attributes are similar to Java attributes and are translated into Jimple as Tags attached to the appropriate Jimple artifacts. An analysis usually does not need to cover the semantics of such attributes unless it directly targets security problems. In thus case, it would have to precisely model the correct target platform in either case.

Inside an assembly, classes are structured in namespaces, similar to Java's packages. In Jimple, we therefore represent namespaces as packages. C# also allows one to define aliases for namespaces, but these aliases are resolved to their original names by the compiler and are thus not a challenge for the work presented here. Note that CIL bytecode can reference the same class in the same namespace from two different assemblies. This is not possible in Jimple, unless we treat assembly names as namespace prefixes.

Furthermore, .net languages such as C# support *partial classes*. In a partial class, methods and fields can be scattered among multiple source files which get merged into one complete class at compile time. Conflicting definitions lead to a compiler error. It is also possible to only write the signature of a method in one source file and the implementation in another one. Due to the compile-time merging, the front-end presented in this work can, however, be oblivious to this language feature.

3. The CIL Type System

Java distinguishes between Objects and numeric/Boolean values, the former of which always use a pass-by-reference semantics, while the latter use pass-by-value. CIL offers a richer model: on top of regular objects it supports also *value types* called *structs* that are passed by value. All structs have an implicit empty constructor and therefore do not need to be initialized explicitly. The primitive data types such as `int` and `float` are system-defined structs which are declared in the system assembly `mscorlib`. Enums are structs as well, merely defining fixed values that can be type-checked by the compiler.

```
 1 void test() {
 2   int a = 5;
 3   calcRef(ref a);
 4   Console.WriteLine(a);
 5
 6   int[] b;
 7   calcOut(b);
 8   Console.WriteLine(b[1]);
 9 }
10 int calcRef(ref int a) {
11   a = a + 3;
12 }
13 int calcOut(out int[] a) {
14   a = new A[3] { 1, 2, 3 };
15 }
```

Listing 1. Calling Conventions in C#

Since Jimple is based on Java, it cannot directly represent this distinction. We represent CIL classes and structs as normal classes in Jimple. Whenever such Jimple classes are used in the code, we, however, need to correctly emulate the semantics of the calling convention. We achieve this by cloning the objects that correspond to structs before passing them as method parameters (or base objects for virtual calls). The clone is a member-wise deep-clone on structs only. This means that if a struct contains another struct (recall that primitive types are structs as well in CIL), its value is copied recursively. If a struct contains a reference to a class, this reference is set to null in the clone.

Note that programmers can also overwrite the default calling convention as shown in Listing 1. In line 3 an `int` value (which is a struct and thus normally passed by value) is explicitly passed by reference. In the `test()` method, the new value will thus be available which is why line 4 will output `8`. We model this behavior by not cloning structs when they are explicitly passed by reference. System-defined primitive structs such as `int` are modeled through classes just like any other struct.

CIL further supports the `out` keyword. This can be used in the same place as the `ref` keyword and makes a parameter behave similar to a return value as shown in line 7 of the example. The output line 8 is `1` from the array constructed in the callee `calcOut`. This is different to `ref` where a reference is passed in and members of the referenced object can be changed by the callee, but the callee cannot pass back a totally different object. In the case of `out`, nothing goes in and the callee defines the value to be returned. It is important to correctly model this behavior. If we assume a pure by-reference semantics as in Java, false positives can occur in analyses such as taint tracking. Furthermore, not handling the `out` keyword can lead to uninitialized variables as shown in the example. However, the `out` keyword cannot directly be represented in Jimple. Just like Java, Jimple only allows a single return type for a method and assumes all parameters to be input data. We solve this challenge by automatically boxing the respective parameters. For each type that is passed as an `out` parameter, we automatically generate a boxing class that stores the actual data in a field. When the call returns, the data is read from the field.

Structs differ from objects also in the way they are allocated. Before calling a method on an object or accessing one of its fields, the object's constructor must be called. For a struct, this is not necessary. When a struct is declared, the runtime automatically allocates the required memory and fills it with zeroes. Semantically, this initializes all fields to the default values of their respective types, i.e., zero for numeric types, and null for references. In Jimple, just like in Java, such implicit initialization does not exist. Therefore, we create explicit calls to the default constructor for each declared struct before any other method code is created. This simulates the runtime's initialization behavior.

4. Modeling the CIL Language Features

In this section, we describe some of the distinct language features of the CIL language and how we model them in the Jimple IR. Due to space constraints, we limit ourselves to the most important features. Recall that the goal is re-usability of existing analyses which requires us to avoid changed or extensions to the Jimple IR.

4.1 Generics

```
1 void test() {
2   List<A> lst = getList();
3   A a = lst.get(0);
4   bar(a.toString());
5 }
```

Listing 2. Generic Lists in C#

While Java and the .net languages such as C# all support generics, their handling at compile time is fundamentally different. The example in Listing 2 works for both Java and C#. The Java compiler erases generics by reducing them to the closest common supertype. Therefore, the return type `List<Object>` of method `getList()` will be reduced to `List`. The types of local variables are erased, because they are not needed in the bytecode. A static analysis tool such as Soot must reconstruct these local types from the interface types (parameter types, method return types) and the operations performed on the local variables. In the example, it can only infer `java.lang.Object` as the type for variable `a`. Recall that no generic-type information is available for the list, and thus the return type of the `get()` method is `java.lang.Object` as well. Therefore, the call to `toString()` in line 4 can lead to the `toString()` implementation of `java.lang.Object` or any subtype, making the callgraph greatly imprecise.

In CIL, type information is preserved on local variables as well as on generics. From the CIL bytecode, it is immediately apparent that variable `a` is of type `A` and not of type `System.Object` (which is CIL's equivalent to `java.lang.Object`). Therefore, the CIL frontend can simply inject a typecast. This has several advantages. Firstly, the time-consuming process of type inference is avoided. Secondly, the use of generic types, most commonly collections, does not reduce the callgraph precision in comparison to explicitly-typed specialized classes. In the example, the set of possible callees for the call in line 4 is directly limited to `A.toString()` or overrides in subtypes.

Inside the generic class (the `List` class in the example), the generic types are reduced to base types, similar to the type reduction performed in the Java compiler. If the generic type is a class type, it is reduced to `System.Object`, unless it is explicitly declared to be the subclass of some other, more concrete type. In the latter case, the generic type is reduced to that given supertype. Note that the frontend needs to apply name mangling for generic classes. As generics are explicit in CIL, it is legal to have two different classes with the same name that only differ in the number of type variables. It is, however, not legal to have the same class name and same number of variables multiple times even if the generic types are limited to subclasses of distinct superclasses. This means that only the number of generics is relevant, not any further information about them. The frontend uses this restriction to create unique names for generic classes. In the example, the `List` class with one type variable becomes `List__1`.

```
 1 interface IFace<out T> {
 2   T get();
 3 }
 4 interface IFace2<in T> {
 5   void add(T data);
 6 }
 7 void test() {
 8   IFace<String> if_string = factory();
 9   IFace<Object> if_object = if_string;
10   Object obj = if_object.get();
11
12   IFace<Object> if2_object = factory2();
13   IFace2<String> if2_string = if_object;
14   if2_string.add(new Object());
15 }
```

Listing 3. Co- and Contravariance in C#

CIL also allows covariance and contravariance on generic classes. In the example in Listing 3, the interface `IFace` declares an `out` type variable. The `out` keyword indicates that this variable may only be used in place of return types or as `out`-parameter types of methods. This restriction makes it safe to broaden the type through covariance as shown in line 9. Attempting to use the generic parameter `T` as in `in` parameter in the interface leads to a compiler-time error complaining about unsafe covariance. Our frontend assumes that all bytecode to be processed passes type checking. Recall that we generate typecasts to map generic types to actual ones. The type of the generic interface is, just as in Java, independent of the concrete instances of any type variables. Handling covariance is therefore trivial. The frontend only needs to downcast the return type. Similarly to covariance, CIL also allows contravariance on assigments as shown in line 13 in Listing 3. The interface `IFace2` uses an `in`-type variable `T`, which is restricted to incoming parameters of method calls. Trying to use it for `out` parameters or return values of methods will cause type checking to fail. Under the assumption that all code processed by the frontend type checks, this is also handled through an implicit downcast similar to the case of covariance.

Similar to generic classes, CIL also supports generic methods. As with classes, the generic-type information is persisted in the bytecode. The frontend uses overloading to implement the same technique as with classes. The original generic method is reduced to concrete base types to which concrete overloaded methods provide type-safe access. The overloaded methods implement all necessary typecasts before and after calling the original method.

4.2 Operator Overloading

The .net languages such as C# support operator overloading. In the CIL code, custom operator implementations are treated as normal function calls which makes them easy for Soot to handle. Neither the client analysis nor the CIL front-end need to provide special treatment for this language construct.

4.3 Properties

```
1 class TestClass() {
2   private int m_id;
3   public int id {
4     get { return m_id };
5     set { m_id = value; }
6   }
7   public String data { get; set; }
8 }
```

Listing 4. Properties in C#

The .net languages support properties as shown in Listing 4. Properties are used with the same syntax as fields when storing or retrieving values, but are conceptually similar to getter and setter methods. In fact, the compiler automatically converts the property code into methods. The property in line 3 in Listing 4 is an explicit property definition, similar to writing getter and setter methods in Java. Due to this automatic conversion, the frontend does not need any special handling for properties and can simply keep the method invocations generated by the compiler.

C# also supports a simplified syntax for properties that do not need any additional code, and only store a value. Line 7 declares such a property. The compiler automatically generates a private field for it, as well as getter and setter methods that store and retrieve the value for this field. The implications for the frontend are the same as for explicit properties. The handling of *indexers* is similar to properties. In C#, an indexer is a property that can be accessed with an index, similar to an array. In the CIL bytecode, this is translated into getter and setter methods that take the index as a parameter.

4.4 Delegates

```
 1 class TestClass {
 2   delegate void MyDelegate(String inStr);
 3
 4   void test() {
 5     MyDelegate md = delegate (String
           inStr) {
 6       Console.WriteLine(inStr);
 7     };
 8     md += delegate (String in) {
 9       Console.WriteLine("Hello: " + inStr);
10     };
11     md("My String");
12   }
13 }
```

Listing 5. Delegates in C#

The .net framework provides a built-in concept called *delegates* for handling callbacks. The delegate definition in line 2 is conceptually similar to a Java interface containing only a single method. Line 5 creates an instance of the delegate through an anonymous method implementation. At compile time, the C# compiler creates a new class for each delegate. This class is derived from `System.Delegate`. It declares a constructor and an `Invoke()` method that matches the signature of the declared delegate (`void Invoke(String inStr)` in the example). The constructor takes a reference to the enclosing class instance and the pointer to the method to be called when the delegate is invoked (an int). Since there is one class per delegate and not per implementation, this indirection is required. Note that CIL is able to deal with pointers which is used here for internally managing the delegates.

In the bytecode, invoking a delegate is then represented by simply creating an instance of the delegate class and then calling the `Invoke()` method as shown in Listing 6. At compile time, the anonymous inner method is converted into a normal private method with compiler-generated name. In the example, this is `<test>b__0`. Therefore, no special support for anonymous inner methods is required in the frontend. The opcode `ldftn` is responsible for loading the function pointer of this compiler-generated method onto the stack before invoking the constructor of the delegate class. Note that that the actual bytecode is slightly more complex as the instance of the delegate object is not created anew every time, but cached in a compiler-generated field of the `TestClass` class.

```
  ldnull
  ldftn void
      TestClass::'<test>b__0'(string)
  newobj instance void
      TestClass/MyDelegate::.ctor(object,
      native int)
  ldstr "My String"
  callvirt instance void
      TestClass/MyDelegate::Invoke(string)
```

Listing 6. Simplified Bytecode for Listing 5

The generated delegate class is special, though. Delegates are a concept that is native to the CIL and the .net framework. Therefore, the generated class does not contain any real implementation for the `Invoke()` method or the constructor. Instead, the method is declared as *runtime managed*, a special flag in the method's metadata. This flag instructs the runtime environment to not actually call the empty method when it is invoked. Instead, it jumps to the function pointer that was passed in via the constructor and that is now stored in a field of the delegate class. The process of finding the correct method and invoking it is performed inside the CIL runtime, invisible from the program's code.

To allow existing callgraph algorithms to create sound (and ideally precise) callgraphs, our frontend must emulate this behavior in Jimple code. As function pointers (`ldftn` opcode) do not exist in Jimple, it instead creates an artificial *dispatch* class per function pointer. The `ldftn` opcode is then interpreted as creating an instance of the respective dispatch class and pushing it onto the stack. If the target function is an instance function, the `ldftn` opcode also contains a reference to a target object. This target object is stored in a field of the dispatch class. All these artificial dispatch classes implement a common interface `_cil_delegate` that defines an `Invoke()` method. Since the dispatch class is specific to one single function pointer, it can divert a call to its generic `Invoke()` method to the original method that was referenced in the `ldftn` instruction. This allows the dispatch class to complete cover the semantics of the original function pointer. The `System.Delegate` class must then (instead of the `native int` function pointer) store a reference to a `cil_delegate` object, i.e., the common interface of all dispatch classes. When the `Invoke()` method of `System.Delegate` is called, it can just call `Invoke` on its artificial dispatch class which in turn calls the target method. Note that the concept of dispatch classes allows the frontend to uniformly handle function pointers in a similar fashion as other load instructions. This works even if the function pointer is not directly used afterwards, but remains on the stack for a while.

Delegates can also be used in asynchronous callbacks. In this case, the caller wants to invoke a delegate and continue with its own execution before the delegate has finished its work. Therefore, generated delegate classes provide two additional methods: `BeginInvoke` and `EndInvoke`. Instead of calling the synchronous `Invoke()` method, client code can also call `BeginInvoke()`. This method takes as an optional parameter a second callback that gets invoked upon completion. Afterwards, the client code can obtain the result of the computation through a call to `EndInvoke()`. In the CIL bytecode, these two additional methods become part of the generated delegate class and thus are easily modeled in Jimple.

A delegate can not only be used to provide a callback to a single method, but also provides multicast support. In the example in Listing 5, a second implementation is added in line 8. When the delegate is invoked, both implementations are called. If a delegate returns a value, the default multicast operation return the value computed by the last invoked implementation. Multicast is, just like unicast, handled by the runtime. The generated delegate class is no different to unicast except for it being derived from `System.MulticastDelegate` instead. To model adding another recipient to a delegate, the compiler first creates a second instance of the delegate class. It then issues a call to the static `Combine()` method the `System.Delegate` API class. This method takes both instances of the delegate class and returns a combined instance. Again, the exact function pointer handling happens inside the CIL runtime and not in user code. In the frontend, we model multicast by chaining dispatch methods. We provide artificial implementations of system methods such as `Delegate.Combine()`. The `Combine()` method, for instance, takes two dispatch classes, and creates a new instance of the first one. This first dispatcher has a reference to the second one which is called in `Invoke()` after the local target (i.e., the first dispatcher's target method) has returned.

Furthermore, note that delegates are objects and can thus freely be passed around in the program. It is legal to create a delegate instance of a private method and then pass this delegate to some other code location from which the target method would not be accessible otherwise. The frontend solves this issue by making all methods public that are referenced through function pointers.

4.5 Reflection

Similar to class constants in Java bytecode, CIL has data structures for reflectively accessing not only classes, but also methods and fields. All of these handles can be loaded using the `ldtoken` opcode. Depending on the argument of the `ldtoken` opcode, an instance of a particular handle class is created. For a class reference, an instance of the `System.Reflection.RuntimeTypeHandle` class is created and put on the stack. Afterwards, one can use the reflection methods in the system class library to perform operations on the handle such as calling a method or accessing a field.

The CIL frontend detects the type of token being loaded. It constructs one data structure per target that is derived from the respective system data structure. If the CIL code creates a reference to class `A`, the frontend generates an artificial class `_cil_typeref_A` that is derived from `System.Reflection.RuntimeTypeHandle`. The class reference is then modeled through an instance of this artificial reference class instead of an instance of the parent system class. In other words, the `ldtoken` opcode is modeled as creating an instance of the artificial handle class. This technique allows the frontend to keep the semantics of the original target without abstracting all references together in a single class. A Jimple class constant would not correctly capture the semantics of class references being structs with methods and fields in CIL. Furthermore, there are no field or method constants in Jimple which would lead to a non-uniform handling of the three token types in CIL.

5. Implementation

An assembly containing CIL code is a Windows DLL or EXE file with a proper PE header. These files contain the CILcode as additional resources. The EXE files compiled with Microsoft's compilers also contain small bootstrappers written in native code. This code is responsible for invoking the CIL runtime on the contained managed CIL code without additional effort from the user. If no runtime is installed, it offers to download an install it. Consequently, an assembly is a binary file with complex data structures. To avoid having to parse these binary data structures, our front-end uses `ildasm`, the IL disassembler tool shipped with the Microsoft .net framework. The ildasm tool first converts the binary assembly file into a textual disassembly which Soot's front-end then parses and converts into Jimple code.

We implemented our own parser for CIL disassembly files. Soot is implemented in Java and there is no parser for CIL written in Java yet. Existing work on decompiling CIL assemblies has been performed in CIL by the use of the platform's reflection and code-model APIs. The commercial product *.net Reflector* by Redgate [1], for instance, is implemented as a .net application for this reason.

6. Limitations

Some of the .net language features cannot easily be modeled in Jimple. The .net framework, for instance, allows for *mixed-mode DLLs*. Such DLLs are .net assemblies containing CIL code as well as native, platform-specific Windows DLL files. Both parts contain user code. Methods implemented in CIL code can call native methods and vice versa. Native code can construct and use classes in CIL, and various techniques exist for marshaling data transferred between native and CIL code. In contrast to Java's JNI, a mixed-mode DLL in .net integrates native and managed code more tightly. CIL, for instance, supports bytecode instructions that call native methods given their offset in the file. Mixed-mode DLLs are usually written in an extended version of C++ that supports additional modifiers. For the developer, the difference between managed and native code is a matter of adding or leaving away these modifiers. As Jimple is based on Java, the frontend would need to model this tight coupling as explicit calls to native methods which is non-trivial. Therefore, we leave modeling mixed-mode DLLs to future work.

[1] http://www.red-gate.com/products/dotnet-development/reflector/

7. Evaluation

In this section, we evaluate the performance of the CIL frontend presented in this paper. We furthermore use the frontend to apply existing analyses to CIL bytecode. We also report on experiments on a recent malware sample for Android that uses CIL code to hide its malicious behavior from state-of-the-art detection tools. It exploits that most of these tools do not support CIL code, although CIL code can be run on Android using the Mono framework.

7.1 Performance

A new frontend to any static analysis framework should be able to efficiently handle even large input files. Note that the implementation of the frontend is not yet fully stable and functional for every corner case, which is why the performance data reported here is preliminary. As we have not yet spent any explicit effort on performance optimization, it can be seen as an upper bound for the computation time.

When parsing a simple "Hello World" program written in Java using the ASM-based Java bytecode frontend, Soot loads 216 system classes this program depends on. When loading the semantically equivalent program written in C# using the CIL frontend, Soot needs to load only 114 classes, due to the different structure of the .net framework's runtime. These classes are contained in the `mscorlib` system assembly whose binary is about 5 megabytes and whose disassembly is about 55 megabytes in size. In total, for Microsoft .net framework version 4.0.30319 x64 `mscorlib` comprises more than 3,200 types, 28,300 methods, and 14,200 fields.

In the case of Java, the Jimple conversion requires about one second. In the case of CIL, the current frontend requires six seconds. This excludes the additional time required by Microsoft's external ILDASM tool to disassemble the bytecode. From our experience, however, this time is negligible. At the moment, the biggest bottleneck appears to be the parsing of the huge input text file. We plan to improve the performance in future work.

7.2 Cross-Platform Cross-Language Malware for Android

Mobile devices are used to process a great amount of sensitive information such as banking or health data. Furthermore, these devices are equipped with a broad variety of sensors such as for location (GPS) or acceleration. They can also impose charges at the cost of the user by sending SMS messages to costly premium-rate telephone numbers. Unsurprisingly, these features have attracted miscreants who develop and provide malicious apps. Due to the great market share of Android (more than 80%), most mobile malware is developed for Android. Regardless of the programming language used to develop an app, it must be compiled to Dalvik bytecode. Dalvik is a register-based bytecode language that is specially optimized for resource-constrained mobile devices.

Since there is no compiler from .net languages such as C# to Dalvik, any Android application that wishes to use .net must have its CIL code intepreted, through a special version of the Mono framework (an open-source implementation of a CIL runtime) compiled for ARM. The Mono execution environment runs side-by-side with Android's normal Dalvik runtime. A recent malware sample (identifiable through its package name `com.tinker.gameone`) uses CIL to hide its malicious code from purely Dalvik-based analyses, as they are commonly used by mobile app stores to block malware from the store. In the `gameone` malware, these analysis only see the Dalvik bytecode of the benign Mono framework, which is why the app made it into several stores. According to *virustotal.com*, even at the time of writing the paper, the malware was only detected by 29 out of 56 popular anti-virus tools. Those tools use signature-based matching, i.e., can only re-identify the malware once its malicious behavior has been manually identified.

The malware protects its assemblies from being disassembled by associating the `SuppressIldasmAttribute` with its assembly. Recall that in CIL assemblies are proper entities and may thus have attributes associated. This particular attribute is checked by `ildasm`. If present, `ildasm` refuses to disassemble the assembly. We countered this protection by removing the attribute before disassembling the file. In the Jimple code, the class `FBAccount.TinkerAccount` contains a method `AccountSend Data()`. This method calls a number of methods with obfuscated names, which are, however, only wrappers around API calls. In the end, the code calls the method `PushToServer()` in the class `AppData.ClientDataManager`. With our frontend, Soot was able to find call sites for methods such as `getResult` in `System.Net. Http.Http ResponseMessage` as they appear in `PushToServer()`. In total, these methods send the user's Facebook credentials over the internet. Although not yet tested, we are confident that existing data-flow analyses or slicing techniques can be applied as well without modifications to the analysis as the data flow is fairly trivial in this malware app. It furthermore allows the human analyst to read convenient Jimple code instead of the stack-based CIL disassembly. Therefore, using our frontend it would have been easy to detect this malware with existing analysis techniques.

8. Related Work

The CIL bytecode language is defined as a part of the Common Language Infrastructure (CLI) which is defined in standard ECMA-335 (cli 2012). More precisely, the language parsed by our framework is the textual ILAsm language defined in Part IV of the standard. Others have added documentation on how high-level languages such as C# compile to CIL code (Bock 2008).

Existing work such as the inline reference monitoring for .net programs proposed by Hamlen, Morrisett, and Schneider (Hamlen et al. 2006) is based on Microsoft's ILX SDK. The ILX SDK is capable of reading and writing .net assemblies with the help of OCAML. This toolkit also extends the CIL language with constructs for closures, functions types, thunks, and others (Syme 2001). Microsoft develops the Phoenix Compiler[2] as a research platform for code generation, optimization, and program analysis which can handle native PE binaries as well as CIL code. Phoenix uses a single, strongly-typed intermediate representation. Conceptually, this approach is thus closest to the work presented in this paper. Other purpose-built specialized analysis tools include FxCop (similar to FindBugs (Ayewah et al. 2008) in the Java world) and StyleCop for detecting bad code style. Kieker.NET (Magedanz 2011) is a framework for performing dynamic analysis on .net programs. It re-uses the original Kieker implementation for Java (van Hoorn et al. 2009) by the means of a custom interoperability layer.

[2] `http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx`

9. Conclusions

We have presented a novel frontend for the Soot program analysis framework. The frontend is capable of converting CIL bytecode into Soot's Jimple intermediate representation. We have shown how CIL language constructs can be expressed in Jimple, though Jimple was originally designed for the less expressive Java bytecode language. As future work, we plan to remove the dependency on `ildasm` and implement a conversion directly from the binary CIL datastructures instead of the disassembly text. We are currently working on integrating our frontend into the Soot open-source framework. As the ultimate goal, we hope to apply a mainly unmodified version of the FlowDroid static data flow tracker (Arzt et al. 2014) to .net programs.

References

Common Language Infrastructure (CLI), 2012.

S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Notices*, volume 49, pages 259–269. ACM, 2014.

N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, Sept 2008. ISSN 0740-7459. doi: 10.1109/MS.2008.130.

A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, pages 27–38, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1490-9. doi: 10.1145/2259051.2259056. URL `http://doi.acm.org/10.1145/2259051.2259056`.

J. Bock. *CIL Programming: Under the Hood of. NET*. Apress, 2008.

M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-flow analysis of android applications in droidsafe. In *Proc. of the Network and Distributed System Security Symposium (NDSS). The Internet Society*, 2015.

K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .net. In *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, PLAS '06, pages 7–16, New York, NY, USA, 2006. ACM. ISBN 1-59593-374-3. doi: 10.1145/1134744.1134748. URL `http://doi.acm.org/10.1145/1134744.1134748`.

P. Lam, E. Bodden, O. Lhotak, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Oktober 2011.

F. Magedanz. Dynamic analysis of .net applications for architecture-based model extraction and test generation. Diploma thesis, Kiel University, October 2011. URL `http://oceanrep.geomar.de/15486/`.

D. Syme. Ilx: Extending the .net common {IL} for functional language interoperability. *Electronic Notes in Theoretical Computer Science*, 59 (1):53 – 72, 2001. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/S1571-0661(05)80453-0. BABEL'01, First International Workshop on Multi-Language Infrastructure and Interoperability (Satellite Event of {PLI} 2001).

R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations, 1998.

A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous monitoring of software services: Design and application of the kieker framework. Research report, Kiel University, November 2009. URL `http://oceanrep.geomar.de/14459/`.

I. Watson. Watson libraries for analysis, wala. sourceforge. net/wiki/index. php. *Main Page*.