Under consideration for publication in Formal Aspects of Computing

Partial Evaluation of String Obfuscations for Java Malware Detection

Aziem Chawdhary, Ranjeet Singh and Andy King
School of Computing, University of Kent, CT2 7NF

Abstract.
The fact that Java is platform independent gives hackers the opportunity to write exploits that can target users on any platform, which has a JVM implementation. Metasploit is a well-known source of Java exploits and to circumvent detection by Anti Virus (AV) software, obfuscation techniques are routinely applied to make an exploit more difficult to recognise. Popular obfuscation techniques for Java include string obfuscation and applying reflection to hide method calls; two techniques that can either be used together or independently. This paper shows how to apply partial evaluation to remove these obfuscations and thereby improve AV matching. The paper presents a partial evaluator for Jimple, which is an intermediate language for JVM bytecode designed for optimisation and program analysis, and demonstrates how partially evaluated Jimple code, when transformed back into Java, improves the detection rates of a number of commercial AV products.

1. Introduction

Java is both portable and architecture-neutral. It is portable because Java code is compiled to JVM byte code for which interpreters exist, not only for the popular desktop operating systems, but for phones and tablets, and as browser plug-ins. It is architecture-neutral because the JVM code runs the same regardless of environment. This presents a huge advantage over native code languages, such as C/C++, but also poses a major security threat. If an exploit levers a vulnerability in a JVM implementation, it will affect all versions of a JVM that have not closed off the vulnerability, and affect those users who have not updated their JVM.

JVM vulnerabilities have been used increasingly by criminals in so-called client side attacks, often in conjunction with social engineering tactics. For example, a client-side attack might involve sending a pdf document [Nat13] that is designed to trigger a vulnerability when it is opened by the user in a pdf reader. Alternatively a user might be sent a link to a website which contains a Java applet which exploits a JVM vulnerability [Rap13] to access the user's machine. Client-side attacks provide a way of bypassing a firewall that blocks ports to users' machines and, are proving to be increasingly popular: last year many JVM exploits were added to the Metasploit [Rap14] package, which is a well-known and widely-used penetration testing platform. This, itself, exacerbates the problem. As well as serving penetration testers and security

Correspondence and offprint requests to: Dr. A. Chawdhary, School of Computing, University of Kent, CT2 7NF, UK.
email: a.a.chawdhary@kent.ac.uk

engineers, a script kiddie or a skilled blackhat can reuse a JVM vulnerability reported in Metasploit, applying obfuscation to avoid detection by up-to-date AV detection software.

Anecdotal experimental evidence suggests that commercial AV vendors use Metasploit as a source of popular attack vectors, since exploits from Metasploit are typically detected if they come in an unadulterated form. One can only speculate what techniques an AV vendor actually uses, but detection methods range from entirely static techniques, such as signature matching, to entirely dynamic techniques in which the execution of the program or script is monitored for suspicious activity. In a signature matching, a signature (a hash) is derived, often by decompiling a sample, which is compared against a database of signatures constructed from known malware. Signatures are manually designed to not trigger a false positive which would otherwise quarantine an innocuous file. Dynamic techniques might analyse for common viral activities such as file overwrites and attempts to hide the existence of suspicious files though, it must be said, there are very few academic works that address the security classification of Java applets [SKV12].

To address the weaknesses of AV detection techniques, we propose to use partial evaluation to remove string obfuscations and reflective method calls and thus improve the detection of malicious Java applets. Partial evaluation can be considered to be a hybrid approach [Hat98, JGS93], in which parts of the program are selectively executed. The fundamental idea is to partition variables into one of two types: static and dynamic. The static variables are those which are bound to known values at compile-time such as string and integer constants; all other variables are considered to be dynamic since their bindings cannot be determined until the program executes. This partitioning of variables can either be inferred prior to partial evaluation by applying binding-time analysis [JGS93, Section 4.4.6] in the so-called offline approach [JGS93, Section 7.2.2], or in tandem with partial evaluation in the so-called online approach [JGS93, Section 7.2.1]. A partial evaluator specialises the program using the values of the static variables to simplify statements and expressions to derive a new program known as the residual. Crucially the original program and the residual have the same observable semantics.

Partial evaluation warrants consideration in malware detection because it is more powerful than merely tracing a sequence of function calls; it is geared towards working over incomplete data which inevitably arises when a program is evaluated without complete knowledge of its environment. Partial evaluation is appropriate for improving anti-virus (AV) detection because Java exploits are often obfuscated by string obfuscation and reflection. Although reflection is designed for applications such as development environments, debuggers and test harnesses, in the context of malware, it can be applied to hide a method call that is characteristic of the exploit. This paper shows how partial evaluation can be used to deobfuscate malicious Java software and shows that AV scanning can be improved by matching on the partially evaluated JVM bytecode, rather than original JVM bytecode itself.

1.1. Contributions

This paper describes how partial evaluation can deobfuscate malicious Java exploits; it revisits partial evaluation from the perspective of malware detection which, to our knowledge, is novel. The main contributions are as follows:

- The paper describes the semantics of a partial evaluator for the Jimple [VH98] language. Jimple is a typed three-address intermediate representation for Java bytecode that is designed to support program analysis and, conveniently, can be decompiled back into Java for AV matching.
- In the spirit of revising the conference paper for submission to this journal, the partial evaluator and its semantics has been completely rewritten and formalised, taking inspiration from partial evaluators for logic programs [LS91]. Unusually for works on partial evaluation of Java-like languages, we have formalised our partial evaluator using a small-step operational semantics. This shows a clear link between the operational semantics of Java and its partial evaluation semantics. Our formalisation presented in this paper leads to short correctness proofs, and is itself a useful advance independent of our use case. The partial evaluator defined is tuned for malware detection, but is general enough to be applied to other problem domains.
- The paper shows how partial evaluation can be used to remove reflection from Jimple code, as well as superfluous string operations, that can be used to obfuscate malicious Java code.
- Experimental evidence shows that malicious Java applet detection rates can be improved using partial evaluation.

```
java.security.Permissions o = new java.security.Permissions();
o.add(new AllPermission());

Class<?> c = Class.forName("java.security.Permissions");
Object o = c.newInstance();
Method m = c.getMethod("add", Permission.class);
m.invoke(o, new AllPermission());
```

Listing 1. Method call and its obfuscation taken from CVE-2012-4681

2. Primer on Java Obfuscation

This section will describe techniques that are commonly used to obfuscate Java code to avoid AV detection. The obfuscations detailed below are typically used in combination; it is not as if one obfuscation is more important than another.

2.1. Reflection Obfuscation

An AV filter might check for the invocation of a known vulnerable library function, and to thwart this a malicious applet frequently uses reflection to invoke vulnerable methods. This is illustrated by the code in listing 1 which uses the `Class.forName` static method to generate an object `c` of type `Class`. The `c` object allows the programmer to access information pertaining to the Java class `java.security.Permissions`, and in particular create an object `o` of type `java.security.Permissions`. Furthermore, `c` can be used to create an object `m` that encapsulates the details of a method call on object `o`. The invocation is finally realised by applying the `invoke` on `m` using `o` as a parameter. This sequence of reflective operations serves to disguise what would otherwise be a direct call to the method `add` on an object of type `Permissions`.

2.2. String Obfuscation

Malicious applets will often assemble a string at run-time from a series of component strings. Alternatively, a string can be encoded and then decoded at run-time. Either tactic will conceal a string, making it more difficult to recognise class and method names, thereby improving the chances of outwitting a signature-based AV system. Listing 2 gives an example of a string reconstruction method that we found in the wild, in which the string `java.lang.SecurityManager` is packed with numeric characters which are subsequently removed at runtime. Listing 4 illustrates an encoder which replaces a letter with the letter 13 letters after it in the alphabet. The encoded strings are then decoded at run-time before they are used to create a handle of type `Class` that can, in turn, be used to instantiate `java.lang.SecurityManager` objects.

2.3. Other Obfuscations

There is no reason why other obfuscations [CN09] cannot be used in combination with reflection and string obfuscation. Of these, one of the most prevalent is name obfuscation in which the names of the user-defined class and method names are substituted with fresh names. For example, the name `getStr` in Listing 2 might be replaced with a fresh identifier, so as to mask the invocation of a known decipher method.

3. Partial Evaluation of Jimple

Jimple is a widely used intermediate language for Java program analysis. Jimple [VH98] sits in between JVM bytecode [LYBB13] and higher-level JVM languages such as Java. Jimple uses a stack-less representation, in contrast to JVM bytecode, which greatly simplifies program analysis for Java programs. Rather defining the partial evaluator for a high-level Java-like language, we use a simplified form of the Jimple intermediate language. The implementation of the partial evaluator described in this paper is implemented using Jimple

```
public static String getStr(String input) {
  StringBuilder sb = new StringBuilder();

  for(int i = 0; i < input.length(); i++) {
    if(!(input.charAt(i) >= '0' && input.charAt(i) <= '9')) {
      sb.append(input.charAt(i));
    }
  }
  return sb.toString();
}

String str = "1j2a34v234a.324l324an324g23.4S234e3c24u324r3i4t324y23M4a23n4ag234er";

Class<?> c = Class.forName(getStr(str));
```

Listing 2. String Obfuscation with numeric characters

```
 public static java.lang.String getStr(java.lang.String)
    {
        java.lang.String r0, $r4;
        java.lang.StringBuilder $r1, r2;
        int i0, $i1;
        char $c2, $c3, $c4;

        r0 := @parameter0;
        $r1 = new java.lang.StringBuilder;
        specialinvoke $r1.<init>();
        r2 = $r1;
        i0 = 0;
     label0:
        $i1 = r0.length();
        if i0 >= $i1 goto label3;
        $c2 = r0.charAt(i0);
        if $c2 < 48 goto label1;
        $c3 = r0.charAt(i0);
        if $c3 <= 57 goto label2;
     label1:
        $c4 = r0.charAt(i0);
        r2.append($c4);
     label2:
        i0 = i0 + 1;
        goto label0;
     label3:
        $r4 = r2.toString();
        return $r4;
    }
```

Listing 3. Jimple Representation of `getStr` method from Listing 2

and the Soot program analysis framework, thus we are using the same foundation for our formal semantics. In this section we define a partial evaluator for Jimple using a small-step style operational semantics. We define an online partial evaluator to remove string obfuscation and reflection from Java bytecode. The online approach partially evaluates statements on-the-fly, in contrast to offline partial evaluation, which performs a binding time analysis prior to specialisation.

3.1. Jimple

To define our formal semantics we split the Jimple instructions into one of three categories: local declarations, simple statements and control instructions. Each Java method is built of basic blocks containing these instructions. We make a number of assumptions:

- Each basic block is labelled.

```
public static String rot13(String s) {
    StringBuffer sb = new StringBuffer();
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if       (c >= 'a' && c <= 'm') c += 13;
            else if  (c >= 'A' && c <= 'M') c += 13;
            else if  (c >= 'n' && c <= 'z') c -= 13;
            else if  (c >= 'N' && c <= 'Z') c -= 13;
            sb.append(c);
        }
        return sb.toString();
    }

String str = "wnin.ynat.FrphevglZnantre";

Class<?> c = Class.forName(rot13(str));
```

Listing 4. String obfuscation using the rot13 substitution cipher

$$
\begin{array}{lll}
d ::= \epsilon & s ::= \epsilon & c ::= \texttt{goto}\ \ell \\
\quad \mid \texttt{var}\ t\ x & \quad \mid x = e & \quad \mid \texttt{if}\ e\ \texttt{goto}\ \ell_1;\ \texttt{goto}\ \ell_2 \\
\quad \mid d_1; d_2 & \quad \mid x = y.f & \quad \mid \texttt{return}\ x \\
 & \quad \mid y.f = x & \\
e ::= c & \quad \mid x = \texttt{@this} & b ::= s; c \\
\quad \mid x & \quad \mid x = \texttt{@param}i & \\
\quad \mid e_1 \oplus e_2 & \quad \mid x\ \texttt{= new}\ C & \\
 & \quad \mid x\ \texttt{= vinvoke}(y, m(\vec{t}), \vec{z}, t') & \\
 & \quad \mid x\ \texttt{= sinvoke}(C.m(\vec{t}), \vec{z}, t') & \\
 & \quad \mid s_1; s_2 &
\end{array}
$$

Fig. 1. Syntax of Jimple

- The first block in any Java method contains all the local variable declarations needed for the method.
- Each block contains a (possibly empty) sequence of simple instructions followed by a single control instruction which ends the block.
- To simplify our presentation, we assume Jimple conditional jumps have two branches which does not compromise generality since this can be achieved by a simple program transformation.

The Jimple language syntax is summarised in Fig 1. In the sequel we provide a commentary on how these syntactic objects are handled in the partial evaluator. It should be noted that this Jimple syntax differs from the dialect of Jimple produced by the Soot analysis framework [EN08]. An example of Jimple code produced by Soot is presented in Listing 3. Our differences in syntax simplify the presentation of the formal semantics. Specifically, we require all variable declarations to be in the first basic block of a method. We change the syntax of if statements to always include an else branch rather than a default fall through to the next statement. One can translate between this dialect and Soot Jimple via straightforward program transformations.

Variable declarations are of the form `var` t x, for some type t and some variable x. The effect of this is to allocate storage for the variable and set a default value. To simplify our semantics we assume that all variable declarations occur at the beginning of a method.

Expressions consist of integers, and we model Boolean values with 0 and 1 for the sake of simplicity. An expression is either a constant c, a local variable x or some arithmetic operator on expressions. We assume that the Java bytecode is type correct, thus not allowing a variable of type `String` to be added to an integer for example. This again simplifies the partial evaluator.

Our simple instructions consist of assignments, field lookup and update, assignment of the current receiver object ($x = \texttt{@this}$), assignment of a parameter and new object instantiation. We have two method call instructions, one for dynamic methods and the other for static methods. The x `= vinvoke(`y`,`$m(\vec{t})$`,`$\vec{z}$`,`t'`)` invokes method m, which takes parameters of type $\vec{t}$, on receiver object y, with actual parameters $\vec{z}$. The method has return type t' and the result is assigned to x. The static invoke instruction, x `= sinvoke(`$C.m(\vec{t})$`,`$\vec{z}$`,`t'`)`, calls method m, with parameters of type $\vec{t}$, on the class C, with actual parameters $\vec{z}$. The return type is t' and the result is assigned to x. Lastly we have a sequencing operator. A control instruction is either an unconditional jump, a conditional jump or a return statement. A block is then a (possibly empty) sequence of simple statements followed by a control statement.

3.2. Partial Evaluation Semantics

The partial evaluator is run on each class and method contained in the applet of interest; although the partial evaluator will follow methods calls made by a class, it will discard the residual code and keep the *original* code of the method called. The intention is to only keep residual code of the applet with the aim of exposing any malicious behaviour hidden by string obfuscations and reflective method calls. Note also that our partial evaluator begins evaluating a class with respect to a state where we have no information, i.e. a state which is equivalent to true; contrast this with the more common use of partial evaluation where some inputs are predefined and the program is partially evaluated with respect to these known inputs.

The partial evaluator specialises each basic block in turn as it explores reachable parts of a Jimple program: thus the partial evaluator produces a *residual* piece of code for each block, that is a piece of code which has been specialised with respect to the program state and has the same semantics as the original block. The partial evaluator follows each path through the program until every block has been visited. As there may be multiple ways to reach a block, for example using conditional jumps or loops, the partial evaluator uses the program state to determine if a block should be processed: if the block has been previously visited with a program state that subsumes the current one it is no longer necessary to reexamine this block, reminiscent of similar techniques used in top-down abstract interpretation and interpolant based model-checking. Note that subsumption here means logical implication: a state subsumes another if it logically entails the other.

The partial evaluator aims to remove reflective API calls in Java programs and hence treats methods returning strings differently to other methods: if the partial evaluator can determine the value of the string, the call to the method is removed and replaced with string object instantiation. This behaviour is parametric to the semantics: if a user wants to focus on other properties, involving booleans or integers for example, then the rules can be adapted accordingly. The partial evaluator follows method calls and if a return value can be determined this is propagated back to the callee, otherwise the return value is unknown.

Specialised programs returned by the partial evaluator are polyvariant [Bul84], meaning that each block may have multiple residual blocks since there may be multiple entry states. For example, join blocks after a conditional may have two different entry states: with each residual block we store the corresponding entry state and leave the decision about merging to the user. This degree of polyvariance is controllable in our analysis, using the abstract function, allowing a user to tune the degree of polyvariance.

- The partial evaluation semantics of simple statements and expressions is in Fig. 2 and 3.
- The semantics of method calls is split into two: one general formulation in Fig 5 and a more specific set of rules for the Java reflection API (Fig. 4).
- The partial evaluation algorithm is defined in Fig. 6, which contains rules for control statements and rules for partial evaluation.
- Once the partial evaluation algorithm has terminated, a renaming phase (Fig. 7) renames apart blocks with multiple entries.

The partial evaluator is defined using a small-step operational semantics. A Java method is represented by a map $\Gamma : \mathit{label} \to \mathit{block}$, which maps labels to the associated block of code. A program state is represented by a store $\sigma : \mathit{loc} \to \mathit{value}$ and an environment $\rho : \mathit{var} \to \mathit{loc}$. Semantically all values are allocated to the store for simplicity. Instantiated objects are considered to be values and are written as a pair $C : \rho$; the object $C : \rho$ has an associated variable environment ρ that assigns locations to its fields.

A state is then a pair consisting of a store and an environment. Note however that within the body of a method the environment will mostly stay constant: this follows from the assumption that all variable

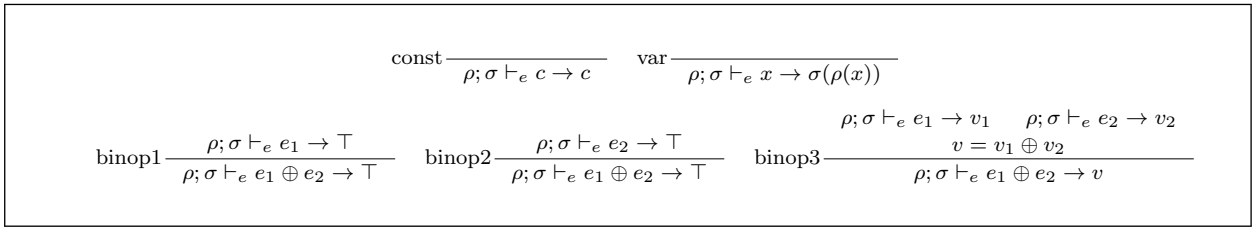


Fig. 2. Operational Semantics for Expressions

declarations are performed at the beginning of a method. This simplifies the formalism and fixes the value of the environment in the semantic rules. The partial evaluator will produce a new program $\Sigma : \mathit{label} \times \mathit{store} \to \mathit{stmt}^*$. Each label is associated with a store which holds before execution of the block associated with the label. The entailment relation is defined on values as $v \models v$, $v \models \top$ and $\top \models \top$. This induces a join defined by $v \sqcup v = v$, $v_1 \sqcup v_2 = \top$ if $v_1 \neq v_2$ and $v \sqcup \top = \top \sqcup v = \top \sqcup \top = \top$. Entailment is lifted pointwise to stores by $\sigma \models \sigma'$ iff $dom(\sigma) = dom(\sigma')$ and $\sigma(l) \models \sigma'(l)$ for all $l \in dom(\sigma)$. Join lifts to stores by $\sigma \sqcup \sigma' = \lambda l.\sigma(l) \sqcup \sigma'(l)$. Finally we have two helper functions: entryLabel takes a method and returns the label of its entry block and entryDecls takes a method and returns a sequence of variable declarations for the method.

3.2.1. Expressions

The semantics for expressions is defined in Fig.2. The const rule is standard and returns a constant value. The var rule looks up the environment and store to get the value of x. The binop operators evaluate a binary operator and either return a value (binop3) or return $\top$ in case one of the operands evaluates to $\top$ (rules binop1 and binop2).

3.2.2. Simple Statements

The semantics for simple statements returns an updated state but also a *residual*, a specialised version of the instruction with respect to the input state σ. The rules for simple statements ($\vdash_s$) take a fixed program Γ, environment ρ and receiver object o, and process a simple statement along with an input state and return a residual and output state. The rule expr-val evaluates an expression and assigns the result to x. Since the value of the expression is known the statement can be specialised to the assignment of a constant value. If the value is not known (rule expr-top) the residual is the original statement.

The rules for field lookup and update work in a similar fashion: if the value of the field is known then the field lookup is replaced with a simple assignment to a value; otherwise the original instruction is retained (rules fld-rd-val and fld-rd-top respectively). Similarly for the field update (fld-wr-val and fld-wr-top).

The rules this and param work as standard and do not rewrite the original instruction. The new rule is more complicated as a new object needs to be instantiated: new entries in the environment are created, with a default value in the store, corresponding to the number of fields in the object. The newly instantiated object is then assigned to x. This new object has an associated environment ρ' mapping the objects fields to the correct location in the store. The rules for sequencing are standard: the first statement is executed followed by the second and then the residuals from each are concatenated (rules seq1 and seq2).

3.2.3. API Calls

Java malware makes heavy use of the Java reflection API, which the partial evaluator aims to specialise away. Fig 4 contains rules specifically for the Java reflection API. These rules are crucial to reasoning about the Java reflection API, heavy use of which can cause anti-virus tools fail when analysing Java code. In order to reason about the Java reflection API, we also need to reason about the Java String class. This is explained further in the next section about method invocation.

The forName rule handles the `forName` method, which given a class name, returns the class descriptor object associated with the given class name. Semantically this is handled by instantiating a Class descriptor object and setting the `class` field to contain the given class name. The newInstance rule handles calls to `newInstance`, which instantiates an object represented by the given class descriptor. The getMethod rule

$$\text{decl}\ \frac{l \notin dom(\sigma) \quad \rho' = \rho[x \mapsto l] \quad \sigma' = \sigma[l \mapsto \mathsf{default}(t)]}{\vdash_d \langle \texttt{var}\ t\ x,\ (\rho,\sigma)\rangle \to (\rho',\sigma')}$$

$$\text{decl1}\ \frac{}{\vdash_d \langle \epsilon,\ (\rho,\sigma)\rangle \to (\rho,\sigma)} \qquad \text{decl2}\ \frac{\vdash_d \langle d_1,\ (\rho,\sigma)\rangle \to (\rho',\sigma') \quad \vdash_d \langle d_2,\ (\rho',\sigma')\rangle \to (\rho'',\sigma'')}{\vdash_d \langle d_1; d_2,\ (\rho,\sigma)\rangle \to (\rho'',\sigma'')}$$

$$\text{expr-val}\ \frac{\rho;\sigma \vdash_e e \to v \quad \sigma' = \sigma[\rho(x) \mapsto v]}{\Gamma; o; \rho \vdash_s \langle x\ \texttt{=}\ e,\ \sigma\rangle \to \langle x\ \texttt{=}\ v,\ \sigma'\rangle} \qquad \text{expr-top}\ \frac{\rho;\sigma \vdash_e e \to \top \quad \sigma' = \sigma[\rho(x) \mapsto \top]}{\Gamma; o; \rho \vdash_s \langle x\ \texttt{=}\ e,\ \sigma\rangle \to \langle x\ \texttt{=}\ e,\ \sigma'\rangle}$$

$$\text{fld-rd-val}\ \frac{\sigma(\rho(y)) = C : \rho_0 \quad \sigma(\rho_0(f)) = v \quad \sigma' = \sigma[\rho(x) \mapsto v]}{\Gamma; o; \rho \vdash_s \langle x\ \texttt{=}\ y.f,\ \sigma\rangle \to \langle x\ \texttt{=}\ v,\ \sigma'\rangle} \qquad \text{fld-rd-top}\ \frac{\sigma(\rho(y)) = C : \rho_0 \quad \sigma(\rho_0(f)) = \top \quad \sigma' = \sigma[\rho(x) \mapsto \top]}{\Gamma; o; \rho \vdash_s \langle x\ \texttt{=}\ y.f,\ \sigma\rangle \to \langle x\ \texttt{=}\ y.f,\ \sigma'\rangle}$$

$$\text{fld-wr-val}\ \frac{\sigma(\rho(y)) = C : \rho_0 \quad \sigma(\rho(x)) = v \quad \sigma' = \sigma[\rho_0(f) \mapsto v]}{\Gamma; o; \rho \vdash_s \langle y.f\ \texttt{=}\ x,\ \sigma\rangle \to \langle y.f\ \texttt{=}\ v,\ \sigma'\rangle} \qquad \text{fld-wr-top}\ \frac{\sigma(\rho(y)) = C : \rho_0 \quad \sigma(\rho(x)) = \top \quad \sigma' = \sigma[\rho_0(f) \mapsto \top]}{\Gamma; o; \rho \vdash_s \langle y.f\ \texttt{=}\ x,\ \sigma\rangle \to \langle y.f\ \texttt{=}\ x,\ \sigma'\rangle}$$

$$\text{this}\ \frac{\sigma' = \sigma[\rho(x) \mapsto o]}{\Gamma; o; \rho \vdash_s \langle x\ \texttt{=}\ \texttt{@this},\ \sigma\rangle \to \langle x\ \texttt{=}\ \texttt{@this},\ \sigma'\rangle}$$

$$\text{param}\ \frac{\sigma' = \sigma[\rho(x) \mapsto \sigma(\rho(param_i))]}{\Gamma; o; \rho \vdash_s \langle x\ \texttt{=}\ \texttt{@param}i,\ \sigma\rangle \to \langle x\ \texttt{=}\ \texttt{@param}i,\ \sigma'\rangle}$$

$$\text{new}\ \frac{\langle \vec{f}, \vec{t}\rangle = \mathsf{fields}(C) \quad vars(\vec{l}) \cap dom(\sigma) = \emptyset \quad v_0 = \mathsf{default}(t_0) \quad \cdots \quad v_{n-1} = \mathsf{default}(t_{n-1}) \quad \rho' = \{f_0 \mapsto l_0, \ldots, f_{n-1} \mapsto l_{n-1}\} \quad \sigma' = \{l_0 \mapsto v_0, \ldots, l_{n-1} \mapsto v_{n-1}, \rho(x) \mapsto C : \rho'\}}{\Gamma; o; \rho \vdash_s \langle x\ \texttt{=}\ \texttt{new}\ C,\ \sigma\rangle \to \langle x\ \texttt{=}\ \texttt{new}\ C,\ \sigma'\rangle}$$

$$\text{seq1}\ \frac{}{\Gamma; o; \rho \vdash_s \langle \epsilon,\ \sigma\rangle \to \langle \epsilon,\ \sigma\rangle} \qquad \text{seq2}\ \frac{\Gamma; o; \rho \vdash_s \langle s,\ \sigma\rangle \to \langle r,\ \sigma'\rangle \quad \Gamma; o; \rho \vdash_s \langle s',\ \sigma'\rangle \to \langle r',\ \sigma''\rangle}{\Gamma; o; \rho \vdash_s \langle s; s',\ \sigma\rangle \to \langle r; r',\ \sigma''\rangle}$$

Fig. 3. Operational Semantics for Declarations and Simple Statements

allows calls to `getMethod`, which given a class name and a vector of class objects representing parameters, constructs a Method object which represents a method of name given string and parameters. The invoke instruction allows the programmer to programmatically invoke a method on an object. The receiver object, y, of the invoke method must be a Method class which represents a Java method, the first argument to the method `invoke` is the receiver object upon which the method is going to be invoked on, and the vector of objects represents the parameters to the method invocation. Semantically we translate this to a standard virtual invoke instruction and then return a residual and updated state.

Although we have not observed this in the wild, there is no reason why reflection cannot be applied to a method that obfuscates a string, such as a decryptor. The partial evaluator can follow reflective method calls and multi-layered forms of obfuscation can be handled.

Note that in these rules we have made some simplified models of Java objects: for example a `Class` object in Java does not have a `class` field, but has a `getClass` method. This simplification is purely to aid readability of the rules.

3.2.4. Method Invocation

Fig 5 contains the rules for method calls. The rules distinguish between methods returning a string or non-string values: the reason for this is that deobfuscating string patterns is often crucial to reasoning about reflective method calls in Java.

The rule virtual1 is a fall through case for virtual invoke instructions when other rules are not applicable: in this circumstance nothing is known about the return value and thus x is assigned $\top$ and the residual

$$\text{forName}\ \frac{\begin{array}{c}\sigma(\rho(z)) = \texttt{String} : \rho_0 \qquad \sigma(\rho_0(\texttt{string})) = C \\ \Gamma; o; \rho \vdash_s \langle x\ \texttt{=}\ \texttt{new Class},\ \sigma\rangle \to \langle r,\ \sigma'\rangle \\ \sigma'(\rho(x)) = \texttt{Class} : \rho_1 \qquad \sigma'' = \sigma'[\rho_1(\texttt{class}) \mapsto C]\end{array}}{\Gamma; o; \rho \vdash_s \langle x\ \texttt{=}\ \texttt{sinvoke(Class.forName(String),}\ \langle z\rangle\texttt{,}\ \texttt{Class)},\ \sigma\rangle \to \langle r,\ \sigma''\rangle}$$

$$\text{newInstance}\ \frac{\begin{array}{c}\sigma(\rho(y)) = \texttt{Class} : \rho_0 \qquad \sigma(\rho_0(\texttt{class})) = C \\ \Gamma; o; \rho \vdash_s \langle x\ \texttt{=}\ \texttt{new}\ C,\ \sigma\rangle \to \langle r,\ \sigma'\rangle\end{array}}{\Gamma; o; \rho \vdash_s \langle x\ \texttt{=}\ \texttt{vinvoke(}y\texttt{,}\ \texttt{newInstance(),}\ \langle\rangle\texttt{,}\ C\texttt{)},\ \sigma\rangle \to \langle r,\ \sigma'\rangle}$$

$$\text{getMethod}\ \frac{\begin{array}{c}\sigma(\rho(w)) = \texttt{String} : \rho_0 \qquad \sigma(\rho_0(\texttt{string})) = m \\ \sigma(\rho(y)) = \texttt{Class} : \rho_1 \qquad \sigma(\rho_1(\texttt{class})) = C \qquad \sigma(\rho(\vec{z})) = \vec{t} \\ \Gamma; o; \rho \vdash_s \langle x\ \texttt{=}\ \texttt{new Method},\ \sigma\rangle \to \langle r,\ \sigma'\rangle \\ \sigma'(\rho(x)) = \texttt{Method} : \rho_2 \qquad \sigma'' = \sigma'[\rho_2(\texttt{method}) \mapsto m,\ \rho_2(\texttt{params}) \mapsto \vec{t}]\end{array}}{\Gamma; o; \rho \vdash_s \langle x\ \texttt{=}\ \texttt{vinvoke(}y\texttt{,}\ \texttt{getMethod(String,}\ \vec{\texttt{Class}}\texttt{),}\ \langle w\rangle :: \vec{z}\texttt{,}\ \texttt{Method)},\ \sigma\rangle \to \langle r,\ \sigma''\rangle}$$

$$\text{invoke}\ \frac{\begin{array}{c}\sigma(\rho(y)) = \texttt{Method} : \rho_0 \qquad \sigma(\rho_0(\texttt{method})) = m \qquad \sigma(\rho_0(\texttt{params})) = \vec{t} \\ \Gamma; o; \rho \vdash_s \langle x\ \texttt{=}\ \texttt{vinvoke(}w, m(\vec{t}), \vec{z}, t'\texttt{)},\ \sigma\rangle \to \langle r,\ \sigma'\rangle\end{array}}{\Gamma; o; \rho \vdash_s \langle x\ \texttt{=}\ \texttt{vinvoke(}y\texttt{,}\ \texttt{invoke(Object,}\ \vec{\texttt{Object}}\texttt{),}\ \langle w\rangle :: \vec{z}\texttt{,}\ t'\texttt{)},\ \sigma\rangle \to \langle r,\ \sigma'\rangle}$$

Fig. 4. Operational Semantics for illustrative Java reflection API calls used for obfuscation

$$\text{virtual1}\ \frac{\mathsf{modifies}(m) = \{z_{i_1}, \ldots, z_{i_n}\} \subseteq \mathsf{vars}(\vec{z}) \quad \sigma' = \sigma[\rho(x) \mapsto \top, \rho(z_{i_1}) \mapsto \top, \ldots, \rho(z_{i_n}) \mapsto \top]}{\Gamma; o; \rho \vdash_s \langle \texttt{x = vinvoke(}y\texttt{,}m(\vec{t})\texttt{,}\vec{z}\texttt{,}t'\texttt{),}\ \sigma\rangle \to \langle \texttt{x = vinvoke(}y\texttt{,}m(\vec{t})\texttt{,}\vec{z}\texttt{,}t'\texttt{),}\ \sigma'\rangle}$$

$$\text{virtual2}\ \frac{\begin{array}{c}\{l_0, \ldots, l_{n-1}\} \cap dom(\sigma) = \emptyset \qquad \rho' = \rho[param_0 \mapsto l_0, \ldots, param_{n-1} \mapsto l_{n-1}] \\ \sigma' = \sigma[\vec{l} \mapsto \sigma(\rho(\vec{z}))] \quad o' = \sigma(\rho(y)) = C : \_ \quad \ell = \mathsf{entryLabel}(C.m(\vec{t})) \quad d = \mathsf{entryDecls}(C.m(\vec{t})) \\ \vdash_d \langle d,\ (\rho', \sigma')\rangle \to (\rho'', \sigma'') \qquad \Gamma; o'; \rho'' \vdash_{pe} \{(\ell, \sigma'') \mapsto \bot\} \to \Sigma' \\ v = \mathsf{exitValue}_{\Gamma,\rho''}(\Sigma') = \texttt{String} : \rho_0 \qquad \sigma''' = \mathsf{exitStore}_{\Gamma,\rho''}(\Sigma') \qquad \sigma^{iv} = \sigma'''[\rho(x) \mapsto v]\end{array}}{\Gamma; o; \rho \vdash_s \langle x\ \texttt{=}\ \texttt{vinvoke(}y\texttt{,}\ m(\vec{t})\texttt{,}\ \vec{z}\texttt{,}\ \texttt{String,}\ \sigma\rangle \to \langle x\ \texttt{=}\ \texttt{new String; }x\texttt{.string =}\ v,\ \sigma^{iv}\rangle}$$

$$\text{virtual3}\ \frac{\begin{array}{c}\{l_0, \ldots, l_{n-1}\} \cap dom(\sigma) = \emptyset \qquad \rho' = \rho[param_0 \mapsto l_0, \ldots, param_{n-1} \mapsto l_{n-1}] \\ \sigma' = \sigma[\vec{l} \mapsto \sigma(\rho(\vec{z}))] \quad o' = \sigma(\rho(y)) = C : \_ \quad \ell = \mathsf{entryLabel}(C.m(\vec{t})) \quad d = \mathsf{entryDecls}(C.m(\vec{t})) \\ \vdash_d \langle d,\ (\rho', \sigma')\rangle \to (\rho'', \sigma'') \qquad \Gamma; o'; \rho'' \vdash_{pe} \{(\ell, \sigma'') \mapsto \bot\} \to \Sigma' \\ v = \mathsf{exitValue}_{\Gamma,\rho''}(\Sigma') \qquad \sigma''' = \mathsf{exitStore}_{\Gamma,\rho''}(\Sigma') \qquad \sigma^{iv} = \sigma'''[\rho(x) \mapsto v]\end{array}}{\Gamma; o; \rho \vdash_s \langle x\ \texttt{=}\ \texttt{vinvoke(}y\texttt{,}\ m(\vec{t})\texttt{,}\ \vec{z}\texttt{,}\ t'\texttt{),}\ \sigma\rangle \to \langle x\ \texttt{=}\ \texttt{vinvoke(}y\texttt{,}\ m(\vec{t})\texttt{,}\ \vec{z}\texttt{,}\ t'\texttt{),}\ \sigma^{iv}\rangle}$$

$$\text{static1}\ \frac{\mathsf{modifies}(m) = \{z_{i_1}, \ldots, z_{i_n}\} \subseteq \mathsf{vars}(\vec{z}) \quad \sigma' = \sigma[\rho(x) \mapsto \top, \rho(z_{i_1}) \mapsto \top, \ldots, \rho(z_{i_n}) \mapsto \top]}{\Gamma; o; \rho \vdash_s \langle x\ \texttt{=}\ \texttt{sinvoke(}C.m(\vec{t}), \vec{z}, t'\texttt{),}\ \sigma\rangle \to \langle x\ \texttt{=}\ \texttt{sinvoke(}C.m(\vec{t}), \vec{z}, t'\texttt{),}\ \sigma'\rangle}$$

$$\text{static2}\ \frac{\begin{array}{c}\{l_0, \ldots, l_{n-1}\} \cap dom(\sigma) = \emptyset \qquad \rho' = \rho[param_0 \mapsto l_0, \ldots, param_{n-1} \mapsto l_{n-1}] \\ \sigma' = \sigma[\vec{l} \mapsto \sigma(\rho(\vec{z}))] \qquad \ell = \mathsf{entryLabel}(C.m(\vec{t})) \qquad d = \mathsf{entryDecls}(C.m(\vec{t})) \\ \vdash_d \langle d,\ (\rho', \sigma')\rangle \to (\rho'', \sigma'') \qquad \Gamma; \texttt{null}; \rho'' \vdash_{pe} \{(\ell, \sigma'') \mapsto \bot\} \to \Sigma' \\ v = \mathsf{exitValue}_{\Gamma,\rho''}(\Sigma') = \texttt{String} : \rho_0 \qquad \sigma''' = \mathsf{exitStore}_{\Gamma,\rho''}(\Sigma') \qquad \sigma^{iv} = \sigma''[\rho(x) \mapsto v]\end{array}}{\Gamma; o; \rho \vdash_s \langle \texttt{x = sinvoke(}C.m(\vec{t})\texttt{,}\vec{z}\texttt{,String),}\ \sigma\rangle \to \langle x\ \texttt{=}\ \texttt{new String; }x\texttt{.string =}\ v,\ \sigma^{iv}\rangle}$$

$$\text{static3}\ \frac{\begin{array}{c}\{l_0, \ldots, l_{n-1}\} \cap dom(\sigma) = \emptyset \qquad \rho' = \rho[param_0 \mapsto l_0, \ldots, param_{n-1} \mapsto l_{n-1}] \\ \sigma' = \sigma[\vec{l} \mapsto \sigma(\rho(\vec{z}))] \qquad \ell = \mathsf{entryLabel}(C.m(\vec{t})) \qquad d = \mathsf{entryDecls}(C.m(\vec{t})) \\ \vdash_d \langle d,\ (\rho', \sigma')\rangle \to (\rho'', \sigma'') \qquad \Gamma; \texttt{null}; \rho'' \vdash_{pe} \{(\ell, \sigma'') \mapsto \bot\} \to \Sigma' \\ v = \mathsf{exitValue}_{\Gamma,\rho''}(\Sigma') \qquad \sigma''' = \mathsf{exitStore}_{\Gamma,\rho''}(\Sigma') \qquad \sigma^{iv} = \sigma'''[\rho(x) \mapsto v]\end{array}}{\Gamma; o; \rho \vdash_s \langle x\ \texttt{=}\ \texttt{sinvoke(}C.m(\vec{t}), \vec{z}, t'\texttt{),}\ \sigma\rangle \to \langle x\ \texttt{=}\ \texttt{sinvoke(}C.m(\vec{t}), \vec{z}, t'\texttt{),}\ \sigma^{iv}\rangle}$$

Fig. 5. Operational Semantics for Method Calls

contains the original method invocation instruction. To handle the issue of aliasing across a method call, we also set any parameters which may be modified by the method to $\top$. The $\mathsf{modifies}(m)$ function uses the results of a compositional pointer analysis, for example purity analysis [MRV11], to infer which of the arguments may be modified by the method. For correctness, a simple compositional analysis is sufficient which over-approximates the way a method can mutate its arguments without reference to a specific call state. If a parameter may be modified then its entry in the return state is set to $\top$. The rule virtual2 deals with calls to methods returning a string value: the rule sets up the environment for the method call, runs the local variable declarations and then calls the method by marking the initial block as to be processed with the new environment ρ'' (the first three lines of the rule). The partial evaluator may produce more than one final return value and store. The calls to exitValue and exitStore collapse multiple values into a single value:

$$\mathsf{exitValue}_{\Gamma,\rho}(\Sigma) = \bigsqcup \left\{ \sigma'(\rho(x)) \;\middle|\; \begin{array}{l} (\ell,\sigma) \mapsto (r; \texttt{return } x) \in \Sigma \quad \wedge \\ \Gamma; o; \rho \vdash_s \langle r,\ \sigma\rangle \to \langle r',\ \sigma'\rangle \end{array} \right\}$$

$$\mathsf{exitStore}_{\Gamma,\rho}(\Sigma) = \lambda l. \bigsqcup \left\{ \sigma'(l) \;\middle|\; \begin{array}{l} (\ell,\sigma) \mapsto (r; \texttt{return } x) \in \Sigma \quad \wedge \\ \Gamma; o; \rho \vdash_s \langle r,\ \sigma\rangle \to \langle r',\ \sigma'\rangle \end{array} \right\}$$

The store is then updated with the return value being assigned to x. Since the return value is known the method call can be replaced by a new string object with the returned value (the residual created in the conclusion).

The rule virtual3 deals with methods of non-string type and the method call is not partially evaluated away. However the method call is followed to infer the return state. The reason for not retaining the residual of a method call is to avoid problems with inheritance and method overloading: it is not sound to partially evaluate a method with respect to only one callee; we need to reason about all potential calls in order to soundly specialise a method. One could keep the residual of a method by placing it in a fresh method in a class and changing the callee to call the new method, but this is akin to method inlining. The three static rules are similar to the corresponding virtual rules as described above.

3.2.5. Control Flow and Overall PE Algorithm

The overall semantics for the partial evaluator is defined in Fig. 6. Given a program $\Gamma : \mathit{labels} \to \mathit{blocks}$, the partial evaluator will return a $\Sigma : \mathit{labels} \times \mathit{store} \to \mathit{blocks}$. The resultant partially evaluated program is thus a map from labels together with an input state to a block. The high level algorithm uses two rules (labelled $\vdash_{pe}$): the stop rule checks to see if any block has been marked for processing by adding a mapping from the label and input state to $\bot$. If nothing remains to be processed the rule stop terminates partial evaluation and returns the final specialised program. The other case is handled by cont, which takes a label to be processed, and then processes the block (second line in rule cont), resulting in an updated partially evaluated program Σ'. The third line then calls a $\vdash_{pe}$ rule to see if anything else needs to be processed. The abstract function is a parameter to tune the degree of polyvariance in the final partially evaluated program: for example blocks with multiple entries in the partially evaluated program could be merged. Merging can lose some specialisation but will keep the size of the final program manageable. Below is an example instantiation of abstract:

$$\mathsf{abstract}_{mono}(\Sigma) = \begin{cases} \mathsf{abstract}_{mono}(\Sigma'[(\ell, \sigma \sqcup \sigma') \mapsto \bot]) & \text{if } \{(\ell,\sigma),(\ell,\sigma')\} \subseteq \mathit{dom}(\Sigma) \wedge \sigma \neq \sigma' \\ \Sigma & \text{otherwise} \end{cases}$$

$$\text{where } \Sigma' = \{(\ell^*, \sigma^*) \mapsto s^* \in \Sigma \mid \ell^* \neq \ell \vee \sigma^* \not\models \sigma \sqcup \sigma'\}$$

The instance above enforces monovariance: if a label is seen with more than one input state, then the input states are collapsed and this label is marked as to be processed with the merged states. Another instantiation of abstract can be used to reduce the code size of the resultant partially evaluated program by collapsing blocks with the same label and residual into a single block:

$$\mathsf{abstract}_{unique}(\Sigma) = \begin{cases} \mathsf{abstract}_{unique}(\Sigma'[(\ell, \sigma \sqcup \sigma') \mapsto \bot]) & \text{if } \{(\ell,\sigma) \mapsto r, (\ell,\sigma') \mapsto r\} \subseteq \Sigma \wedge \sigma \neq \sigma' \\ \Sigma & \text{otherwise} \end{cases}$$

$$\text{where } \Sigma' = \{(\ell^*, \sigma^*) \mapsto s^* \in \Sigma \mid \ell^* \neq \ell \vee \sigma^* \not\models \sigma \sqcup \sigma'\}$$

Rules to control the exploration of a program are marked with $\vdash_c$, which check to see if the given store

$$\text{covered}\ \frac{\exists(\ell,\sigma') \in dom(\Sigma).\sigma \models \sigma'}{\rho \vdash_c \langle \ell,\ \sigma,\ \Sigma\rangle \to \Sigma} \qquad \text{uncovered}\ \frac{\begin{array}{c}\forall(\ell,\sigma') \in dom(\Sigma).\sigma \not\models \sigma' \\ \Sigma' = \{(\ell^*,\sigma^*) \mapsto s^* \in \Sigma \mid \ell^* \neq \ell \vee \sigma^* \not\models \sigma\} \\ \Sigma'' = \Sigma'[(\ell,\sigma) \mapsto \bot]\end{array}}{\rho \vdash_c \langle \ell,\ \sigma,\ \Sigma\rangle \to \Sigma''}$$

$$\text{return}\ \frac{\Sigma' = \Sigma[(\ell,\sigma) \mapsto r; \texttt{return}\ x]}{\Gamma; o; \rho \vdash_b \langle \ell,\ \sigma,\ r,\ \sigma',\ \texttt{return}\ x,\ \Sigma\rangle \to \Sigma'}$$

$$\text{block}\ \frac{\Gamma; o; \rho \vdash_s \langle s,\ \sigma'\rangle \to \langle r',\ \sigma''\rangle \quad \Gamma; o; \rho \vdash_b \langle \ell,\ \sigma,\ r; r',\ \sigma'',\ c,\ \Sigma\rangle \to \Sigma'}{\Gamma; o; \rho \vdash_b \langle \ell,\ \sigma,\ r,\ \sigma',\ s; c,\ \Sigma\rangle \to \Sigma'}$$

$$\text{if-top}\ \frac{\begin{array}{c}\rho; \sigma' \vdash_e e \to \top \\ \Sigma' = \Sigma[(\ell,\sigma) \mapsto r; \texttt{if}\ e\ \texttt{goto}\ \ell_1;\ \texttt{goto}\ \ell_2] \\ \rho \vdash_c \langle \ell_1,\ \sigma',\ \Sigma'\rangle \to \Sigma'' \qquad \rho \vdash_c \langle \ell_2,\ \sigma',\ \Sigma''\rangle \to \Sigma'''\end{array}}{\Gamma; o; \rho \vdash_b \langle \ell,\ \sigma,\ r,\ \sigma',\ \texttt{if}\ e\ \texttt{goto}\ \ell_1;\ \texttt{goto}\ \ell_2,\ \Sigma\rangle \to \Sigma'''}$$

$$\text{if-true}\ \frac{\rho; \sigma' \vdash_e e \to 1 \quad \Gamma; o; \rho \vdash_b \langle \ell,\ \sigma,\ r,\ \sigma',\ \texttt{goto}\ \ell_1,\ \Sigma\rangle \to \Sigma'}{\Gamma; o; \rho \vdash_b \langle \ell,\ \sigma,\ r,\ \sigma',\ \texttt{if}\ e\ \texttt{goto}\ \ell_1;\ \texttt{goto}\ \ell_2,\ \Sigma\rangle \to \Sigma'}$$

$$\text{if-false}\ \frac{\rho; \sigma' \vdash_e e \to 0 \quad \Gamma; o; \rho \vdash_b \langle \ell,\ \sigma,\ r,\ \sigma',\ \texttt{goto}\ \ell_2,\ \Sigma\rangle \to \Sigma'}{\Gamma; o; \rho \vdash_b \langle \ell,\ \sigma,\ r,\ \sigma',\ \texttt{if}\ e\ \texttt{goto}\ \ell_1;\ \texttt{goto}\ \ell_2,\ \Sigma\rangle \to \Sigma'}$$

$$\text{unfold}\ \frac{\Gamma; o; \rho \vdash_b \langle \ell,\ \sigma,\ r,\ \sigma',\ \Gamma(\ell'),\ \Sigma\rangle \to \Sigma'}{\Gamma; o; \rho \vdash_b \langle \ell,\ \sigma,\ r,\ \sigma',\ \texttt{goto}\ \ell',\ \Sigma\rangle \to \Sigma'}$$

$$\text{specialise}\ \frac{\Sigma' = \Sigma[(\ell,\sigma) \mapsto r; \texttt{goto}\ \ell'] \qquad \rho \vdash_c \langle \ell',\ \sigma',\ \Sigma'\rangle \to \Sigma''}{\Gamma; o; \rho \vdash_b \langle \ell,\ \sigma,\ r,\ \sigma',\ \texttt{goto}\ \ell',\ \Sigma\rangle \to \Sigma''}$$

$$\text{stop}\ \frac{\forall(\ell,\sigma) \in dom(\Sigma).(\ell,\sigma) \mapsto \bot \notin \Sigma}{\Gamma; o; \rho \vdash_{pe} \Sigma \to \Sigma} \qquad \text{cont}\ \frac{\begin{array}{c}(\ell,\sigma) \mapsto \bot \in \Sigma \\ \Gamma; o; \rho \vdash_b \langle \ell,\ \sigma,\ \epsilon,\ \sigma,\ \Gamma(\ell),\ \Sigma\rangle \to \Sigma' \\ \Gamma; o; \rho \vdash_{pe} \mathsf{abstract}(\Sigma') \to \Sigma''\end{array}}{\Gamma; o; \rho \vdash_{pe} \Sigma \to \Sigma''}$$

Fig. 6. Operational Semantics for Control-flow and Blocks

and label is already subsumed by an entry in the program Σ (rule covered): in this case we no longer need to carry on partially evaluating ℓ, since there is already an entry Σ which subsumes the store σ. This allows us to terminate and not explore paths which do not add anything new to the final partially evaluated program. The uncovered rule checks to see if the store is subsumed and if not, marks ℓ and σ to be processed, which will be done by the rule cont later in the derivation.

Control flow instructions ($\vdash_b$) require more contextual information in comparison to simple instructions: this is because a control instruction will insert (or not as is necessary) a new block into the partially evaluated program Σ. A control flow rule takes a six-tuple: the label of the current block, the state at the start of the block, a residual, the current state, a control instruction and a map representing the current partially evaluated program. This returns a new partially evaluated program. The return rule is straightforward: since we are the end of a block we can insert a new block into the partially evaluated program Σ, where the block is labelled ℓ and has instructions from the residual computed so far for the block r together with the return instruction.

A block rule processes a block: first the simple statements are evaluated to return a residual and state and then a control instruction is evaluated. This control instruction will return an updated partially evaluated program.

$$\text{rename}\frac{\exists\{(\ell^\dagger,\sigma^\dagger),(\ell^\dagger,\sigma)\}\subseteq dom(\Sigma).\sigma^\dagger\neq\sigma \quad \ell^\ddagger=\mathsf{freshlabel}(\Sigma) \quad \Sigma'=\mathsf{rename}_{(\ell^\dagger,\sigma^\dagger,\ell^\ddagger)}(\Sigma)}{\vdash_r \Sigma\to\Sigma'}$$

$$\text{rename1}\frac{}{\vdash_r \Sigma\to\Sigma} \qquad \text{rename2}\frac{\vdash_r \Sigma\to\Sigma' \quad \vdash_r \Sigma'\to\Sigma''}{\vdash_r \Sigma\to\Sigma''}$$

Fig. 7. Rules for Repeated Renaming

The rules for conditional jumps are split into three cases: the first case is where the condition cannot be evaluated (rule if-top), in which case both branches need to be partially evaluated. Since the condition cannot be evaluated the original instruction needs to be kept, both branches are marked to signal that they must be processed by using one of the $\vdash_c$ rules. If the condition can be evaluated to either true or false (if-true, if-false respectively), then the conditional jump can be translated to an unconditional jump and is then processed by either the unfold or specialise rule for unconditional jumps.

Unconditional jumps are handled by either the unfold or specialise rule: this is left as a choice to the user. The difference between the two is that unfold will fuse two blocks together, whereas specialise will keep the structure of the blocks. The unfold rule executes the block pointed to by the goto instruction. The specialise rule adds the current residual (and current goto) to the specialised program, and then calls the covered rule to see if the target block should be processed. In the implementation, goto rules use the unfold rule unless the jump is to a loop head.

3.2.6. Renaming

Once the partial evaluator has finished the final program Σ many contain multiple entries for a label ℓ, however each block must have a unique label. Fig 7 defines a set of renaming rules to remove this ambiguity. The rules rename1 and rename2 take a partially evaluated program and keep applying the rename rule until no more changes occur. The key rule is the rename rule which checks if a label $\ell^\dagger$ has multiple blocks, generates a fresh label and then calls the rename function.

Definition 3.1.

$$\mathsf{rename}_{(\ell^\dagger,\sigma^\dagger,\ell^\ddagger)}(\Sigma)=\{\mathsf{renamehead}_{(\ell^\dagger,\sigma^\dagger,\ell^\ddagger)}(\ell,\sigma)\mapsto\mathsf{renametail}_{(\ell^\dagger,\sigma^\dagger,\ell^\ddagger)}(r,\sigma)\mid(\ell,\sigma)\mapsto r\in\Sigma\}$$

$$\mathsf{renamehead}_{(\ell^\dagger,\sigma^\dagger,\ell^\ddagger)}(\ell,\sigma)=\begin{cases}(\ell^\ddagger,\sigma^\dagger) & \text{if } (\ell,\sigma)=(\ell^\dagger,\sigma^\dagger)\\ (\ell,\sigma) & \text{otherwise}\end{cases}$$

$$\mathsf{renametail}_{(\ell^\dagger,\sigma^\dagger,\ell^\ddagger)}(r,\sigma)=\begin{cases}r';\texttt{goto } \ell^\ddagger & \text{if } r=(r';\texttt{goto } \ell^\dagger)\wedge\Gamma;o;\rho\vdash_s\langle r',\ \sigma\rangle\to\langle r'',\ \sigma'\rangle\wedge\sigma'\models\sigma^\dagger\\ r & \text{otherwise}\end{cases}$$

The instantiated rename takes a label $\ell^\dagger$, a state $\sigma^\dagger$ and a new label $ell^\ddagger$, and renames the entry $(\ell^\dagger,\sigma^\dagger)$ to the new label $\ell^\ddagger$, and also searches for a blocks which jumps to $\ell^\dagger$ and then changes the jump to $\ell^\ddagger$.

3.3. Formalisation

Lloyd and Shepherdson put the partial evaluation of logic programs onto a firm theoretical foundation by introducing a notion of closure [LS91]. A residual (logic) program is said to be closed with respect to a set of calls if any call from the set to the residual can only give rise another call that also falls within that set. In this section, we formalise and justify our partial evaluator for Jimple using an analogous idea. We show that the act of partially evaluating Jimple also amounts to that of computing a form of closure (subject to parameters that control the degree of polyvariance). We show that the act of generating different versions of a block, each specialised against a different σ, can be realised as a simple post-processing step, that preserves closure.

We start the justification of our partial evaluator, by showing the residual generated from a straight-line

sequence of simple assignments can faithfully simulate the unspecialised code if invoked with a store ψ that contains more information that the store σ that it was specialised against.

Lemma 3.1 (intraprocedural residual safety). Let $\psi \models \sigma$ and $\Gamma; o; \rho \vdash_s \langle s,\ \sigma\rangle \to \langle s',\ \sigma'\rangle$. Then

- $\Gamma; o; \rho \vdash_s \langle s,\ \psi\rangle \to \langle r,\ \psi'\rangle$
- $\Gamma; o; \rho \vdash_s \langle s',\ \psi\rangle \to \langle r',\ \psi'\rangle$

Note that the lemma is tagged as intraprocedural because the proof (which is relegated to the appendix) does not argue correctness for calls to `vinvoke` or `sinvoke`. Thus the justification is for intraprocedural specialisation, in which the blocks within a method are specialised, but blocks across method boundaries are not combined. (Formal justification for interprocedural specialisation would appear to be a major study within itself, and therefore we leave this for future work.) The above lemma is used to establish the following result which asserts that the partial evaluation rules presented in Fig. 6 result in a Σ' which is a specialisation of Γ. The intuition is that if $(\ell, \sigma) \mapsto r; c \in \Sigma'$ and r is invoked with a store ψ that contains more information than σ then the residue r behaves as the original block $\Gamma(\ell)$.

Lemma 3.2 (intraprocedural residual safety without unfold). Let $\Sigma = \{(\ell^*, \sigma^*) \mapsto \bot\}$ and $\Gamma; o; \rho \vdash_{pe} \Sigma \to \Sigma'$. If $(\ell, \sigma) \mapsto r; c \in \Sigma'$ then $\Gamma(\ell) = s; c$ and for all $\psi \models \sigma$ it follows

- $\Gamma; o; \rho \vdash_s \langle r,\ \psi\rangle \to \langle r',\ \psi'\rangle$
- $\Gamma; o; \rho \vdash_s \langle s,\ \psi\rangle \to \langle r'',\ \psi'\rangle$

The above result follows only when the unfold rule is not applied; it is scaffolding given to lead the reader into the following result, with drops this restriction. In what follows, if $(\ell, \sigma) \mapsto r; c \in \Sigma'$, then r is formed for a sequence of residuals r_i each of which is in one-to-one correspondence with a block s_i labelled with ℓ_i. Note too that properties 1b, 1c and 1d state that the control instruction c_i ensures that s_i is an immediate predecessor of s_{i+1} in execution order. Furthermore, if the cumulative residual r is invoked with a store ψ with more information than the store σ_0 then r yields the same store as that obtained by executing the original blocks s_i in sequence.

Proposition 3.1 (intraprocedural residual safety with fold). Let $\Sigma = \{(\ell^*, \sigma^*) \mapsto \bot\}$ and $\Gamma; o; \rho \vdash_{pe} \Sigma \to \Sigma'$. If $(\ell, \sigma_0) \mapsto r; c \in \Sigma'$ then

1. there exists $\{\ell_0 \mapsto s_0; c_0, \ldots, \ell_{k-1} \mapsto s_{k-1}; c_{k-1}\} \subseteq \Gamma$ such that $r = r_0; \ldots; r_{k-1}$ where
   (a) $\forall i \in [0, k-1].\Gamma; o; \rho \vdash_s \langle s_i,\ \sigma_i\rangle \to \langle r_i,\ \sigma_{i+1}\rangle$.
   (b) $\forall i \in [0, k-2].(c_i = \texttt{if}\ e_i\ \texttt{goto}\ \ell_{i+1};\ \texttt{goto}\ \ell') \Rightarrow (\ell' = \ell_{i+1}) \vee \rho; \sigma_{i+1} \vdash_e e_i \to 1$.
   (c) $\forall i \in [0, k-2].(c_i = \texttt{if}\ e_i\ \texttt{goto}\ \ell';\ \texttt{goto}\ \ell_{i+1}) \Rightarrow (\ell' = \ell_{i+1}) \vee \rho; \sigma_{i+1} \vdash_e e_i \to 0$.
   (d) $\forall i \in [0, k-2].(c_i = \texttt{goto}\ \ell') \Rightarrow (\ell' = \ell_{i+1})$.
2. moreover if $s = s_0; \ldots; s_{k-1}$ then for all $\psi \models \sigma_0$ it follows
   (a) $\Gamma; o; \rho \vdash_s \langle r,\ \psi\rangle \to \langle r',\ \psi'\rangle$
   (b) $\Gamma; o; \rho \vdash_s \langle s,\ \psi\rangle \to \langle r'',\ \psi'\rangle$

The above proposition relates a single (cumulative) residual r to blocks in the original program Γ but it does not guarantee that Σ' contains a block that is specialised for every store σ that can possibly reach it. This notion of transitive reachability is formalised as closure below:

Definition 3.2. Given a fixed o and ρ, Σ is closed iff for all $(\ell, \sigma) \mapsto s; \texttt{goto}\ \ell' \in \Sigma$ there exists $(\ell', \sigma^*) \in \Sigma$ such that $\sigma' \models \sigma^*$ where $\Gamma; o; \rho \vdash_s \langle s,\ \sigma\rangle \to \langle r,\ \sigma'\rangle$.

The main result can now be stated: when appropriately invoked, the partial evaluation rules presented in Fig. 6 yield a Σ' that is closed. The lemma that follows formalises post-processing. It asserts that renaming labels, in the way prescribed in Fig. 7, does not compromise closure.

Proposition 3.2 (partial evaluation ensures closure). $\Sigma : \{(\ell_0, \sigma_0) \mapsto \bot, \ldots, (\ell_{k-1}, \sigma_{k-1}) \mapsto \bot\}$ and $\Gamma; o; \rho \vdash_{pe} \Sigma \to \Sigma'$. Then Σ' is closed.

Proposition 3.3 (renaming preserves closure). Given a fixed o and ρ, if Σ is closed and $\vdash_r \Sigma \to \Sigma'$ then Σ' is closed.

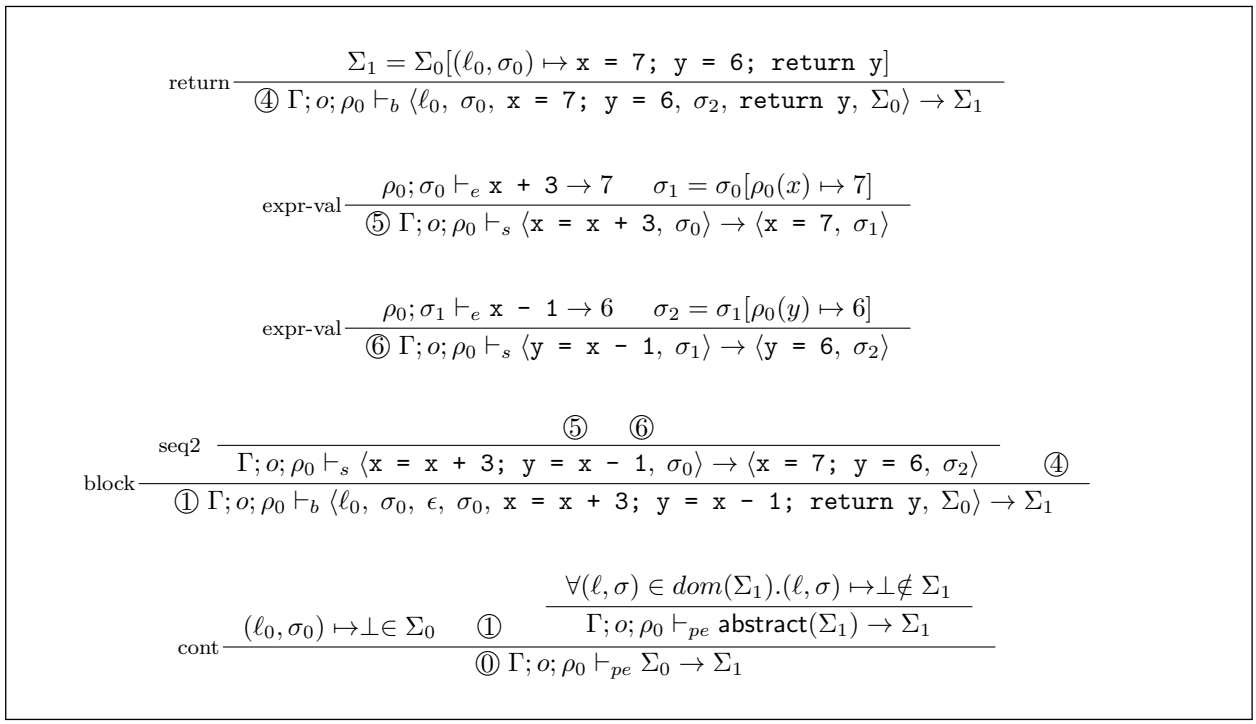


Fig. 8. Derivation Tree for Example 3.1

By repeated renaming Σ' can be reduced to a form Σ'' for which if $\{(\ell, \sigma), (\ell', \sigma')\} \subseteq dom(\Sigma'')$ it follows that $\ell \neq \ell'$, that is, labels are unique. Then the store σ elements of each $(\ell, \sigma) \in dom(\Sigma'')$ are inconsequential and Σ'' can be read-off as a Jimple program.

3.4. Examples

Example 3.1 (Sequence of assignments). This example illustrates partial evaluation on the following program for some object o:

$\Gamma(\ell_0) =$ `x = x + 3; y = x - 1; return y`

Suppose $\rho_0 = \{x \mapsto l_0, y \mapsto l_1\}$, $\sigma_0 = \{l_0 \mapsto 4, l_1 \mapsto 0\}$ and abstract$(\Sigma) = \Sigma$. Partial evaluation begins with $\Sigma_0 = \{(\ell_0, \sigma_0) \mapsto \perp\}$. The partial evaluation tree is presented in Fig. 8. The root of the derivation tree (sub-tree ⓪) starts partial evaluation with Σ_0. The block of code is partially evaluated in sub-tree ①, which evaluates the simple assignments in ⑤ and ⑥. The return instruction is then evaluated in sub-tree ④. Once the block has been evaluated, the derivation then checks to see if there are any more blocks to partially evaluate, of which there are none. The derived partially evaluated program is:

$\Sigma_1(\ell_0, \sigma_0) =$ `x = 7; y = 6; return y`

Note the the assignment to `x` is redundant, but this can be optimised away by a standard compiler analysis; the partial evaluator inspects each instruction without access to contextual information about future uses of the result of the current instruction and thus cannot infer if an assignment is redundant. The partial evaluator could use information from a liveness analysis but we omit this for clarity.

Example 3.2 (Loop unrolling). This example will illustrate how loops can be completely unrolled if we know the valuation of the loop condition. Suppose that $\rho_0 = \{x \mapsto l_0\}$ and $\sigma_0 = \{l_0 \mapsto 3\}$. Suppose that the

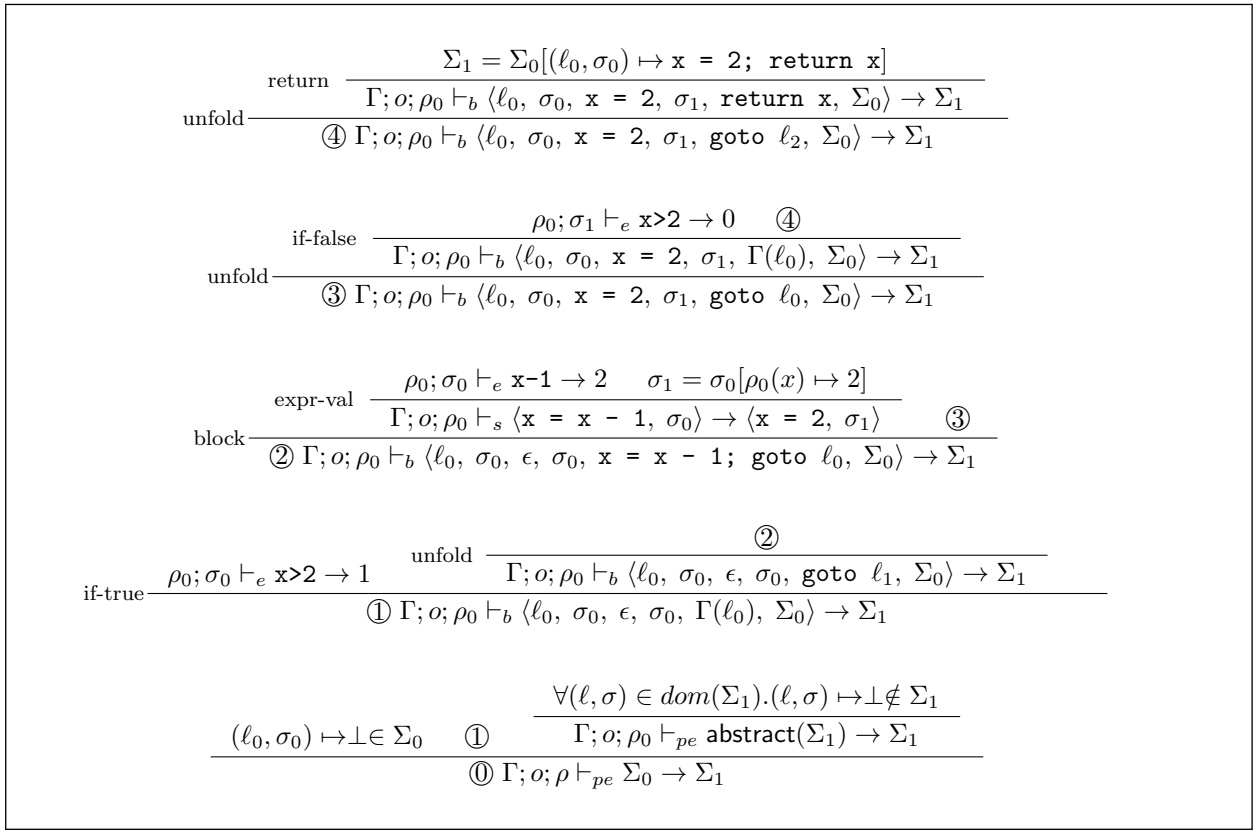


Fig. 9. Derivation for Example 3.2

program is the following:

$$\Gamma = \begin{cases} \ell_0 & \mapsto & \texttt{if x>2 goto}\ \ell_1\texttt{; goto}\ \ell_2 \\ \ell_1 & \mapsto & \texttt{x = x - 1; goto}\ \ell_0 \\ \ell_2 & \mapsto & \texttt{return x} \end{cases}$$

The partial evaluation begins with $\Sigma_0 = \{(\ell_0, \sigma_0) \mapsto \bot\}$ and the derivation tree is shown in Fig 9. Tree ⓪ starts partial evaluation by entering the loop in sub-tree ①, since $\sigma_0(\rho_0(x)) = 3$. The loop body is executed in tree ② and then the loop exists in tree ③, with the exit block being evaluated in ④. The resultant partially evaluated program is:

$\Sigma_1(\ell_0, \sigma_0) = \texttt{x = 2; return x}$

Notice that the partially evaluation has collapsed the blocks down into one: this was due to the application of the unfold rule. Alternatively, the semantics could have chosen to keep the blocks separate using the specialise rule and this is left as a choice in the implementation.

Example 3.3 (Loop preservation). This example shows how loops are handled in the case that the loop condition cannot be evaluated. Assume the program and ρ_0 are as above, but $\sigma_0 = \{l_0 \mapsto \top\}$. Partial evaluation begins with $\Sigma_0 = \{(\ell_0, \sigma_0) \mapsto \bot\}$. The tree (①) signals that the blocks ℓ_1 and ℓ_2 need to be processed (using the uncovered rule). These blocks are then processed by tree ③ and ④ respectively. Tree ③ processes the body of ℓ_1; ⑤ processes the goto at the end of block ℓ_1. In this case the goto is jumping to a location and state that has already been seen hence the covered rule applied, curtailing this branch of the

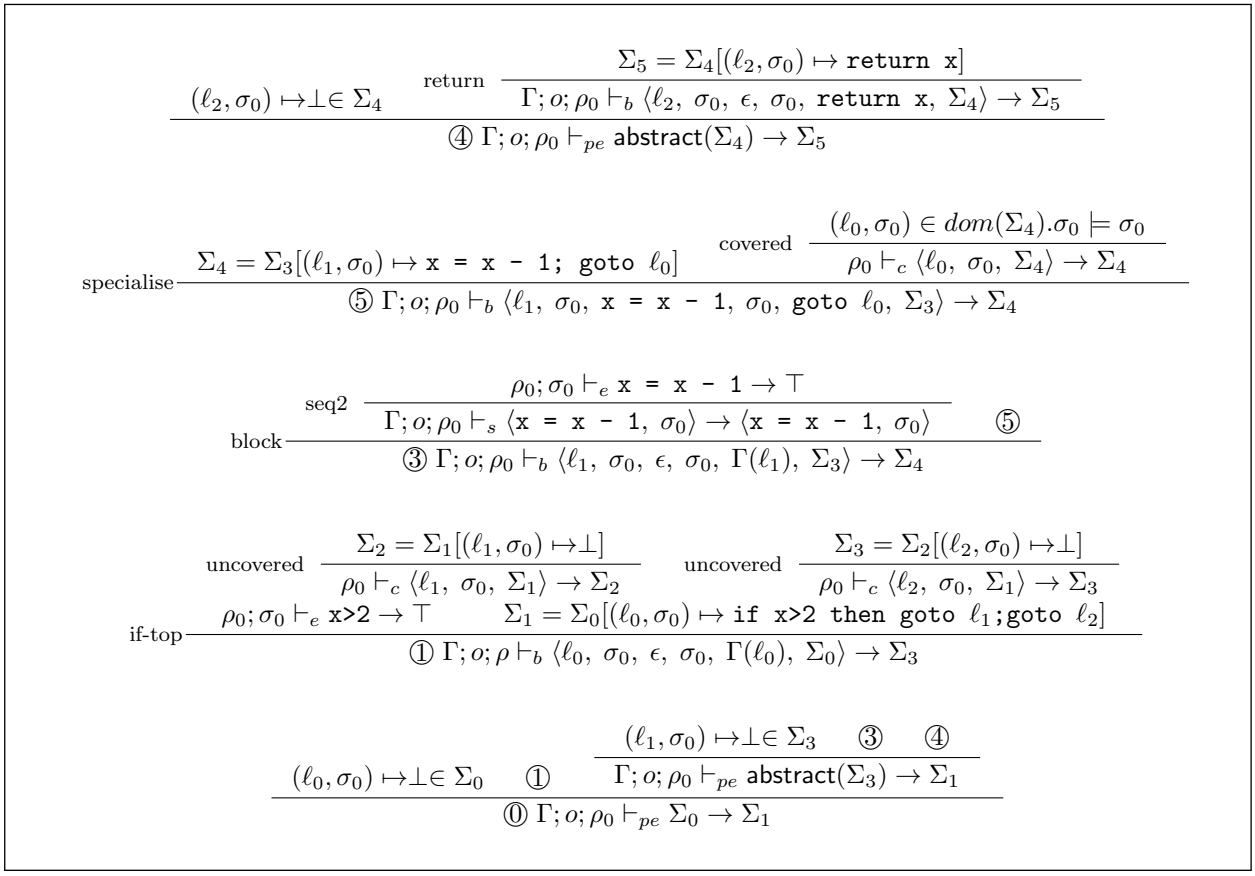


Fig. 10. Derivation for Example 3.3

tree. Tree ④ now processes the other block ℓ_2. The final program is Σ_4 which is

$$\Sigma_4 = \left\{ \begin{array}{lcl} (\ell_1, \sigma_0) & \mapsto & \texttt{x = x - 1; goto } \ell_0 \\ (\ell_0, \sigma_0) & \mapsto & \texttt{if x>2 goto } \ell_1\texttt{; goto } \ell_2 \\ (\ell_2, \sigma_0) & \mapsto & \texttt{return x} \end{array} \right.$$

Notice that the partially evaluated program retains the same structure as the original program, as desired.

Example 3.4 (Specialise vs unfold). The following example illustrates the difference between the unfold and specialise rules for `goto` instructions: in essence unfold removes a `goto` instruction and merges the partially evaluated instructions from the subsequent block into the current block. Specialise retains the structure of the blocks, which is necessary for evaluating loops where the value of the loop condition is unknown and thus the loop structure must be kept. Unfold can be applied when the loop condition can be evaluated and thus the loop unrolled. To highlight these differences, consider the program:

$$\Gamma = \left\{ \begin{array}{lcl} \ell_0 & \mapsto & \texttt{x = x - 1; goto } \ell_1 \\ \ell_1 & \mapsto & \texttt{y = x - 1; return y} \end{array} \right.$$

Let $\sigma_0 = \{l_0 \mapsto 4, l_1 \mapsto 5\}$, $\rho_0 = \{x \mapsto l_0, y \mapsto l_1\}$ and suppose $\Sigma_0 = \{(\ell_0, \sigma_0) \mapsto \bot\}$. The derivation trees are shown in Fig. 11, where the bottom derivation demonstrates the use of unfold and the top derivation illustrates specialise. In the bottom derivation, tree ① executes the body of block ℓ_1: tree ③ executes the `goto` using unfold. In this case, no pending blocks remain to be explored, hence tree ② returns the same Σ_1 and the final program is then

$$\Sigma_1 = \ (\ell_0, \sigma_0) \mapsto \texttt{x = 3; y = 2; return y}$$

$$\text{return}\frac{\Sigma_3 = \Sigma_2[(\ell_1, \sigma_1) \mapsto \texttt{y = 2; return y}]}{④\ \Gamma; o; \rho_0 \vdash_b \langle \ell_1,\ \sigma_1,\ \texttt{y = 2},\ \sigma_2,\ \texttt{return y},\ \Sigma_2\rangle \to \Sigma_3}$$

$$\frac{(\ell_1, \sigma_1) \mapsto \bot \in \Sigma_2 \qquad \text{block}\ \dfrac{\text{expr-val}\ \dfrac{\rho_0; \sigma_1 \vdash_e \texttt{x - 1} \to 2 \qquad \sigma_2 = \sigma_1[\rho_0(y) \mapsto 2]}{\Gamma; o; \rho_0 \vdash_s \langle \texttt{y = x - 1},\ \sigma_1\rangle \to \langle \texttt{y = 2},\ \sigma_2\rangle}}{\Gamma; o; \rho_0 \vdash_b \langle \ell_1,\ \sigma_1,\ \epsilon,\ \sigma_1,\ \Gamma(\ell_1),\ \Sigma_2\rangle \to \Sigma_3} \quad ④}{②\ \Gamma; o; \rho_0 \vdash_{pe} \mathsf{abstract}(\Sigma_2) \to \Sigma_3}$$

$$\text{specialise}\frac{\Sigma_1 = \Sigma_0[(\ell_0, \sigma_0) \mapsto \texttt{x = 3; goto } \ell_1] \qquad \text{uncover}\ \dfrac{\Sigma_2 = \Sigma_1[(\ell_1, \sigma_1) \mapsto \bot]}{\rho_0 \vdash_c \langle \ell_1,\ \sigma_1,\ \Sigma_1\rangle \to \Sigma_2}}{③\ \Gamma; o; \rho_0 \vdash_b \langle \ell_0,\ \sigma_0,\ \texttt{x = 3},\ \sigma_1,\ \texttt{goto } \ell_1,\ \Sigma_0\rangle \to \Sigma_2}$$

$$\frac{(\ell_0, \sigma_0) \mapsto \bot \in \Sigma_0 \qquad \text{block}\ \dfrac{\text{expr-val}\ \dfrac{\rho_0; \sigma_0 \vdash_e \texttt{x - 1} \to 3 \qquad \sigma_1 = \sigma_0[\rho_0(x) \mapsto 3]}{\Gamma; o; \rho_0 \vdash_s \langle \texttt{x = x - 1},\ \sigma_0\rangle \to \langle \texttt{x = 3},\ \sigma_1\rangle} \quad ③}{\Gamma; o; \rho_0 \vdash_b \langle \ell_0,\ \sigma_0,\ \epsilon,\ \sigma_0,\ \Gamma(\ell_0),\ \Sigma_0\rangle \to \Sigma_1} \quad ②}{\Gamma; o; \rho_0 \vdash_{pe} \Sigma_0 \mapsto \Sigma_3}$$

Using Specialise Rule

$$\frac{\forall (\ell, \sigma) \in dom(\Sigma_1).(\ell, \sigma) \mapsto \bot \notin \Sigma_1}{②\ \Gamma; o; \rho_0 \vdash_{pe} \mathsf{abstract}(\Sigma_1) \to \Sigma_1}$$

$$\text{return}\frac{\Sigma_1 = \Sigma_0[(\ell_0, \sigma_0) \mapsto \texttt{x = 3; y = 2; return y}]}{④\ \Gamma; o; \rho_0 \vdash_b \langle \ell_0,\ \sigma_0,\ \texttt{x = 3; y = 2},\ \sigma_2,\ \texttt{return y},\ \Sigma_0\rangle \to \Sigma_1}$$

$$\text{unfold}\frac{\text{block}\ \dfrac{\text{expr-val}\ \dfrac{\rho_0; \sigma_0 \vdash_e \texttt{x - 1} \to 2 \qquad \sigma_2 = \sigma_1[\rho_0(y) \mapsto 2]}{\Gamma; o; \rho_0 \vdash_s \langle \texttt{y = x - 1},\ \sigma_1\rangle \to \langle \texttt{y = 2},\ \sigma_2\rangle} \quad ④}{\Gamma; o; \rho_0 \vdash_b \langle \ell_0,\ \sigma_0,\ \texttt{x = 3},\ \sigma_1,\ \texttt{y = x - 1; return y},\ \Sigma_0\rangle \to \Sigma_1}}{③\ \Gamma; o; \rho_0 \vdash_b \langle \ell_0,\ \sigma_0,\ \texttt{x = 3},\ \sigma_1,\ \texttt{goto } \ell_1,\ \Sigma_0\rangle \to \Sigma_1}$$

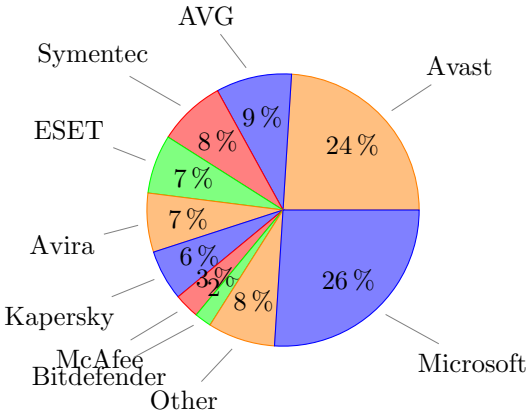
$$\frac{(\ell_0, \sigma_0) \mapsto \bot \in \Sigma_0 \qquad \text{block}\ \dfrac{\text{expr-val}\ \dfrac{\rho_0; \sigma_0 \vdash_e \texttt{x - 1} \to 3 \qquad \sigma_1 = \sigma_0[\rho_0(x) \mapsto 3]}{\Gamma; o; \rho_0 \vdash_s \langle \texttt{x = x - 1},\ \sigma_0\rangle \to \langle \texttt{x = 3},\ \sigma_1\rangle} \quad ③}{\Gamma; o; \rho_0 \vdash_b \langle \ell_0,\ \sigma_0,\ \epsilon,\ \sigma_0,\ \Gamma(\ell_0),\ \Sigma_0\rangle \to \Sigma_1} \quad ②}{\Gamma; o; \rho_0 \vdash_{pe} \Sigma_0 \mapsto \Sigma_1}$$

Using Unfold Rule

Fig. 11. Example contrasting specialise vs unfold rules

CVE	Java Applet Exploit
2012-4681	Remote Code Execution
2012-5076	JAX WS Remote Code Execution
2013-0422	JMX Remote Code Execution
2012-5088	Method Handle Remote Code Execution
2013-2460	Provider Skeleton Insecure Invoke Method

(a) CVEs



(b) AVs and Market Share

Fig. 12. Summary of Common Vulnerabilities and Exposures (CVEs) and Anti-Virus Tools Used

which has amalgamated the two blocks in Γ to a single block with the same semantics.

In contrast, the top derivation uses specialise in the tree ③. This marks block ℓ_2 to be processed which is processed by tree ②. With the specialise rule the final program is

$$\Sigma_3 = \begin{cases} (\ell_0, \sigma_0) & \mapsto & \texttt{x = 3; goto } \ell_1 \\ (\ell_1, \sigma_1) & \mapsto & \texttt{y = 2; return y} \end{cases}$$

keeping the original block structure of Γ.

4. Experiments

We developed a prototype partial evaluator for Jimple focused on the Java string and reflection API. This reduced the engineering effort required: we implemented a partial evaluator in Scala, following the description in section 3, only providing functionality for String, StringBuffer and StringBuilder classes. This was achievable since Java String objects are accessible to Scala. Scala's parser combinator library also makes it straightforward to engineer a parser for Jimple.

To assess how partial evaluation can aid in AV matching, a number of applet malware samples from the Metasploit exploit package [Rap14] were obfuscated using the techniques outlined in section 2. Details of the samples are given in Fig. 12a; the samples were chosen entirely at random. So as to assess the effect on partial evaluation against a representative AV tool (rather than a strawman), we compared the detection rates, with and without partial evaluation, on eight commercial AV products. Together these products cover the majority of the global market, as reported in 2013 [OWA13] and is illustrated in the chart given in table 12b. Conveniently, VirusTotal [Sis14] provides a prepackaged interface for submitting malware samples to all of these products, with the exception of Avira, which is why this tool does not appear in our experiments.

In our experiments we added obfuscation to our samples and then checked if the partial evaluator could remove the obfuscation and thereby restore the AV detection rates. Not all of the randomly selected samples used string or reflection obfuscation, hence these specific obfuscations were added manually following guidelines [Sec12] that have been proposed for evading AV detection by security researchers. String obfuscation was repeatedly applied to the applets until the detection rates reported by VirusTotal changed. Starting from an unobfuscated applet, reflection was also added, again until some AV products could be successfully bypassed. Then the string obfuscations were applied in tandem with the reflection obfuscations. The results of these obfuscation experiments are shown in Fig. 13. It is important to appreciate that the obfuscations used in the fourth experiment include all those obfuscations introduced in the second and third experiments and no more. Finally, the partial evaluator was applied to the applet code employing both forms of obfuscation, and then the partially evaluated applets passed back to VirusTotal for reassessment.

The results show that in most cases the AVs detect most of the exploits in their unadulterated, original form. Exploits CVE-2012-5088 and CVE-2013-2460 go the most undetected, which is possibly because

Exploit Name	Microsoft	Avast	AVG	Symantec	ESET	Kaspersky	McAfee	Bitdefender
CVE	No Obfuscation							
2012-4681	✓	✓	✓	✗	✓	✓	✓	✓
2012-5076	✓	✗	✓	✓	✓	✓	✓	✗
2013-0422	✓	✓	✗	✓	✓	✓	✓	✗
2012-5088	✗	✗	✗	✗	✓	✗	✓	✗
2013-2460	✗	✓	✓	✗	✓	✗	✓	✗
CVE	String Obfuscation							
2012-4681	✗	✓	✓	✗	✓	✓	✓	✓
2012-5076	✗	✗	✓	✓	✗	✗	✓	✗
2013-0422	✗	✗	✗	✓	✗	✗	✓	✗
2012-5088	✗	✗	✗	✗	✓	✗	✓	✗
2013-2460	✗	✗	✓	✗	✗	✗	✓	✗
CVE	Reflection Obfuscation							
2012-4681	✓	✓	✓	✗	✓	✗	✗	✗
2012-5076	✗	✗	✗	✓	✗	✓	✓	✗
2013-0422	✓	✓	✓	✓	✓	✓	✓	✗
2012-5088	✗	✗	✗	✗	✓	✗	✓	✗
2013-2460	✗	✗	✓	✗	✓	✗	✓	✗
CVE	String and Reflection Obfuscation							
2012-4681	✗	✓	✓	✗	✗	✗	✓	✗
2012-5076	✗	✗	✗	✓	✗	✗	✓	✗
2013-0422	✗	✗	✗	✓	✗	✗	✓	✗
2012-5088	✗	✗	✗	✗	✓	✗	✓	✗
2013-2460	✗	✗	✓	✗	✗	✗	✓	✗

Fig. 13. Impact of Obfuscations on Vulnerability Detection

both exploits make extensive use of reflection. It is interesting to see that the product with the highest market share (Microsoft) was unable to detect any of the exploits after string obfuscation, which suggests that removing this obfuscation alone is truly worthwhile. Moreover, after introducing reflection obfuscation the AV detection count for each exploit drops significantly. Furthermore, applying reflection with string obfuscation is strictly stronger than applying string obfuscation and reflection alone. CVE-2012-4681 represents an anomaly under McAfee since reflection obfuscation impedes detection whereas, bizarrely, using reflection with string obfuscation does not. Interestingly, McAfee classifies this CVE with the message `Heuristic.BehavesLike.Java.Suspicious-Dldr.C` which suggests that it is using a heuristic behavioural approach which might explain its unpredictability.

The headline result is not reported in the table: it is that the partially evaluated code (the residual) had the same detection rates to those of the original applets (namely those versions taken directly from Metasploit). This suggests that AV matching can be improved by partial evaluation, without having to redistribute the signature database. The caveat is although the samples were randomly selected, and although the obfuscations were added following guidelines outlined by a third-party [Sec12], the partial evaluator was not applied to samples that have actually appeared in the wild.

5. Related Work

Although there has been much work in Java security, partial evaluation and reflection, there are few works that concern all three topics. This section provides pointers to the reader for the main works in each of these three separate areas.

One of the very few techniques that has addressed the problem of detecting malicious Java Applets is Jarhead [SKV12]. This recent work uses machine learning to detect malicious Applets based on 42 features which include such things as the number of instructions, the number of functions per class and cyclomatic complexity [McC76]. Jarhead also uses special features that relate to string obfuscation, such as the number and average length of the strings, and the fraction of strings that contain non-ASCII printable characters. Other features that it applies determine the degree of active code obfuscation, such as the number of times that reflection is used within the code to instantiate objects and invoke methods. Out of a range of classifiers, decision trees are shown to be the most reliable. Our work likewise aspires to be static, though partial evaluation takes this notion to the limit, so as to improve detection rates. Moreover, machine learning introduces the possibility of false negatives and, possibly worse, false positives. Our approach is to scaffold off existing AV products that have been carefully crafted to not trigger false positives, and improve their matching rates by applying program specialisation as a preprocessing step.

One of the many objectives of partial evaluation is to remove interpretive overheads from programs. Reflection can be considered to be one such overhead and therefore it is perhaps not surprising that it has attracted interest in the static analysis community; indeed the performance benefit from removing reflection can be significant [PL01]. Civet [SC11] represents state-of-the-art in removing Java reflection; it does not apply binding-time analysis (BTA) [And93] but relies on programmer intervention, using annotation to delineate static from dynamic data, the correctness of which is checked at specialisation time. Advanced BTAs have been defined for specialising Java reflection [BN00], though to our knowledge, none have been implemented. We know of no partial evaluator for Jimple, though Soot represents the ideal environment for developing one [EN08]. It is interested to note that our formalisation of our online partial evaluator was influenced by a notion of closure [LS91] that has been used to formalise partial evaluators for logic programs; this suggests that the idea is not limited to one particular programming paradigm. Quite apart from its role in deobfuscation, partial evaluation can also be applied in obfuscation [GJM12]: a modified interpreter, that encapsulates an obfuscation technique, is partially evaluated with respect to the source program to automatically obfuscate the source.

Reflection presents a challenge for program analysis: quite apart from writes to object fields, reflection can hide calls, and hence mask parts of the call-graph so that an analysis is unsound. Points-to analysis has been suggested [LWL05] as a way of determining the targets of reflective calls which, in effect, traces the flow of strings through the program. This is sufficient for resolving many, but not all calls, hence points-to information is augmented with user-specified annotation so as to statically determine the complete call graph. The use of points-to information represents an advance over using dynamic instrumentation to harvest reflective calls [HDH04] since instrumentation cannot guarantee complete coverage. Partial evaluation likewise traces the flow of strings through the program, though without refining points-to analysis, it is not clear that it has the precision to recover the targets of reflective calls that have been willfully obfuscated with such techniques as a substitution cipher (rot13).

6. Discussion and Future Work

There are many choices to make when designing a partial evaluator: since this work deals with Java malware making use of the Java reflection API, we have focused on string methods. Our partial evaluator attempts to specialise all methods of return type string. To aid scalability the partial evaluator should only specialise methods whose return value has some direct influence on the calls to the Java reflective API. There are a number of techniques we could use to address this: at one end of the spectrum we could use a simple live-variable analysis to see if the return value is used subsequently; at the other end where we could apply dependence techniques based on abstract non-interference [GM04]. We leave this as future work to investigate.

The connection with circular proofs is intriguing: one way of terminating the partial evaluation is to stop when we revisit a label whose associate program state is subsumed by the previous visit. This is reminiscent of circular proofs [BGP12] on well-founded inductive predicates where an infinite derivation can be cut by virtue of well-foundedness. We again leave this as future work.

Termination analysis is a subfield of static analysis within itself and thus far we have not explored how termination can improve unfolding. We simply unfold loops where the loop bound is known at specialisation time. Future work will apply homeomorphic embedding to supervise unfolding, making use of type information [AGGZP09] available in Jimple. A related issue for future work is ensure that the residual is well-typed. In addition we would like to have a more general method to deal with method overloading and inheritance.

We will also examine how partial evaluation can remove less common forms of Java obfuscation such as control flow obfuscation and serialisation and deserialisation obfuscation, the latter appearing to be as amenable to partial evaluation as string obfuscation. In the long term we will combine partial evaluation with code similarity matching, drawing on techniques from information retrieval. Due to the declining use of Java applets and increasing use of Android, we will be applying the partial evaluation framework to improve the detection of Android malware.

7. Conclusion

We have presented a partial evaluator for removing string and reflection obfuscation from Java programs, with the aim of improving the detection of malicious Java code. Our work puts partial evaluation in a new light: previous studies have majored on optimisation whereas we argue that partial evaluation has a role in anti-virus matching. The partial evaluator defined has uses beyond anti-virus matching, and is general enough to be applied to other partial evaluation scenarios. To this end, a partial evaluator has been designed for Jimple, which was strength tested on five malware samples from the Metasploit exploit framework, obfuscated using string and reflection obfuscation, both separately and in combination. The framework defined in this paper is parametric in a number of options, allowing a user to control the degree of accuracy and computational cost of partial evaluation of Java programs.

Acknowledgements

This work was supported by grant EP/K032585/1 funded by GCHQ in association with EPSRC.

References

[AGGZP09] E. Albert, J. Gallagher, M. Gómez-Zamalloa, and G. Puebla. Type-based Homeomorphic Embedding for Online Termination. *Information Processing Letters*, 109:879–886, July 2009.

[And93] L. Andersen. Binding-Time Analysis and the Taming of C Pointers. In *PEPM*, pages 47–58. ACM, 1993.

[BGP12] J. Brotherston, N. Gorogiannis, and R. L. Petersen. A Generic Cyclic Theorem Prover. In *APLAS*, volume 7705 of *LNCS*, pages 350–367. Springer, 2012.

[BN00] M. Braux and J. Noyé. Towards Partially Evaluating Reflection in Java. In *PEPM*, pages 2–11. ACM, 2000.

[Bul84] M.A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21(5):473–484, 1984.

[CN09] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, 2009.

[EN08] A. Einarsson and J. D. Nielsen. A Survivor's Guide to Java Program Analysis with Soot. Technical report, 2008.

[GJM12] R. Giacobazzi, N. D. Jones, and I. Mastroeni. Obfuscation by Partial Evaluation of Distorted Interpreters. In *PEPM*, pages 63–72. ACM, 2012.

[GM04] R. Giacobazzi and I. Mastroeni. Abstract Non-interference: Parameterizing Non-interference by Abstract Interpretation. In *Principles of Programming Languages*, pages 186–197. ACM, 2004.

[Hat98] J. Hatcliff. An Introduction to Online and Offline Partial Evaluation Using a Simple Flowchart Language. `epository.readscheme.org/ftp/papers/pe98-school/hatcliff-DIKU-PE-summerschool.pdf`, 1998. DIKU Partial Evaluation Summer School.

[HDH04] M. Hirzel, A. Diwan, and M. Hind. Pointer Analysis in the Presence of Dynamic Class Loading. In *ECOOP*, volume 3086 of *Lecture Notes in Computer Science*, pages 96–122. Springer, 2004.

[JGS93] N. D Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[LS91] J. W. Lloyd and J. C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.

[LWL05] V. B. Livshits, J. Whaley, and M. S. Lam. Reflection Analysis for Java. In *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 139–160. Springer, 2005.

[LYBB13] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.

[McC76] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4), 1976.

[MRV11] R. Madhavan, G. Ramalingam, and K. Vaswani. Purity Analysis: An Abstract Interpretation Formulation. In *SAS*, volume 6887 of *LNCS*, pages 7–24. Springer-Verlag, 2011.

[Nat13] National Institute of Standards and Technology. Vulnerability Summary for CVE-2013-3346, 2013.

[OWA13] OWASP. Metasploit Java Exploit Code Obfuscation and Antivirus Bypass/Evasion (CVE-2012-4681), 2013.

[PL01] J. G. Park and A. H. Lee. Removing Reflection from Java Programs Using Partial Evaluation. In *Reflection*, volume 2192 of *Lecture Notes in Computer Science*, pages 274–275. Springer, 2001.

[Rap13] Rapid 7. Java Applet JMX Remote Code Execution, 2013.

[Rap14] Rapid 7. Metasploit, 2014.

[SC11] A. Shali and W. R. Cook. Hybrid Partial Evaluation. In *OOPSLA*, pages 375–390. ACM, 2011.

[Sec12] Security Obscurity Blog. Java Exploit Code Obfuscation and Antivirus Bypass/Evasion (Blog Post). `http://security-obscurity.blogspot.co.uk/2012/11/java-exploit-code-obfuscation-and.html`, 2012.

[Sis14] H. Sistemas. VirusTotal Analyses Suspicious Files and URLs, 2014. `https://www.virustotal.com/`.

[SKV12] J. Schlumberger, C. Kruegel, and G. Vigna. Jarhead: Analysis and Detection of Malicious Java Applets. In *ACSAC*, pages 249–257. ACM, 2012.

[VH98] R. Valleé Rai and L. J. Hendren. Jimple: Simplifying Java Bytecode for Analyses and Transformations. Technical Report TR-1998-4, McGill University, 1998.

A. Proofs

Lemma A.1 (monotonicity of expression evaluation). Suppose $\oplus$ is monotonic. Then if $\sigma \models \sigma'$, $\rho; \sigma \vdash_e e \to v$ and $\rho; \sigma' \vdash_e e \to v'$ it follows $v \models v'$.

Proof of Lemma A.1 Proof by induction.

- Let $e = c$. By rule *const* it follows $\rho; \sigma \vdash_e e \to c$ and $\rho; \sigma' \vdash_e e \to c$.
- Let $e = x$. By rule *var* it follows $\rho; \sigma \vdash_e e \to \sigma(\rho(x))$ and $\rho; \sigma' \vdash_e e \to \sigma'(\rho(x))$ where $\sigma(\rho(x)) \models \sigma'(\rho(x))$.
- Let $e = e_1 \oplus e_2$.
  - Consider rule *binop*1. Since $\rho; \sigma \vdash_e e_1 \to \top$ by induction it follows $\rho; \sigma \vdash_e e_1 \to \top$ hence $\rho; \sigma \vdash_e e \to \top$.
  - Consider rule *binop*2. Let $e = e_1 \oplus e_2$. Since $\rho; \sigma \vdash_e e_2 \to \top$ by induction it follows $\rho; \sigma \vdash_e e_2 \to \top$ hence $\rho; \sigma \vdash_e e \to \top$.
  - Consider rule *binop*3. Let $\rho; \sigma \vdash_e e_1 \to v_1$, $\rho; \sigma \vdash_e e_2 \to v_2$, $\rho; \sigma' \vdash_e e_1 \to v_1'$, $\rho; \sigma' \vdash_e e_2 \to v_2'$. By the inductive hypothesis $v_1 \models v_1'$ and $v_2 \models v_2'$ hence $v_1 \oplus v_2 \models v_1' \oplus v_2'$ by the monotonicity of $\oplus$.

□

Lemma A.2 (monotonicity). Let $\sigma \models \sigma'$ and

- $\Gamma; o; \rho \vdash_s \langle s,\ \sigma\rangle \to \langle r,\ \psi\rangle$
- $\Gamma; o; \rho \vdash_s \langle s,\ \sigma'\rangle \to \langle r',\ \psi'\rangle$

Then $\psi \models \psi'$.

Proof of Lemma A.2 By induction in the style of Lemma A.1. □

Proof of Lemma 3.1 Let $\Gamma; o; \rho \vdash_s \langle s',\ \psi\rangle \to \langle r',\ \psi''\rangle$ so it remains to show $\psi' = \psi''$.

- Consider expr-val i.e. $x = e$.
  - Either $\rho; \sigma \vdash_e e \to v$. Then $s' = (x = v)$ hence $\psi'' = \psi[\rho(x) \mapsto v]$. But $\psi \models \sigma$ thus $\rho; \psi \vdash_e e \to v$ whence $\psi' = \psi[\rho(x) \mapsto v] = \psi''$ as required.
  - Or $\rho; \sigma \vdash_e e \to \top$. Thus $s' = (x = e) = s$ and $\psi' = \psi''$ as required.
- Consider expr-top ie. $\rho; \sigma' \vdash_e e \to \top$. Observe $s' = (x \texttt{ = } e) = s$ thus $\psi' = \psi''$ as required.
- Consider fld-rd-val. Analogous to the expr-val case.
- Consider fld-rd-top. Analogous to the expr-top case.
- Consider fld-wr-val. Analogous to the expr-val case.
- Consider fld-wr-top. Analogous to the expr-top case.
- Consider this. Analogous to the expr-top case.
- Consider param. Analogous to the expr-top case.
- Consider new. Analogous to the expr-top case.
- Consider seq1. Then $s' = \epsilon = s$ whence $\psi' = \psi''$ as required.
- Consider seq2. Then $s = s_1; s_2$. Moreover $\Gamma; o; \rho \vdash_s \langle s_1,\ \sigma\rangle \to \langle s_1',\ \varsigma\rangle$ and $\Gamma; o; \rho \vdash_s \langle s_2,\ \varsigma\rangle \to \langle s_2',\ \sigma'\rangle$ where $s' = s_1'; s_2'$. Let $\Gamma; o; \rho \vdash_s \langle s_1,\ \psi\rangle \to \langle r_1,\ \zeta\rangle$ and $\Gamma; o; \rho \vdash_s \langle s_2,\ \zeta\rangle \to \langle r_2,\ \psi'\rangle$ Since $\psi \models \sigma$ it follows by induction that $\Gamma; o; \rho \vdash_s \langle s_1',\ \psi\rangle \to \langle r_1',\ \zeta\rangle$. By lemma A.2 it follows $\zeta \models \varsigma$ hence by induction $\Gamma; o; \rho \vdash_s \langle s_2',\ \zeta\rangle \to \langle r_2',\ \psi'\rangle$. There $\Gamma; o; \rho \vdash_s \langle s_1'; s_2',\ \psi\rangle \to \langle r_1'; r_2',\ \psi''\rangle$ as required.

□

Proof of Lemma 3.2 By induction. Let $\Sigma_0 = \{(\ell^*, \sigma^*) \mapsto \bot\}$ and $\Gamma; o; \rho \vdash_b \langle \ell,\ \sigma,\ \epsilon,\ \sigma,\ \Gamma(\ell),\ \Sigma_i\rangle \to \Sigma_{i+1}$.

- Observe $(\ell, \sigma) \mapsto r; c \notin \Sigma_0$ hence the result holds vacuously.
- Suppose $(\ell, \sigma) \mapsto \bot \in \Sigma_i$ such that $\Gamma(\ell) = s; c$ and $\Gamma; o; \rho \vdash_s \langle s,\ \sigma\rangle \to \langle r,\ \sigma'\rangle$. Let $\psi \models \sigma$. By lemma 3.1 it follows $\Gamma; o; \rho \vdash_s \langle s,\ \psi\rangle \to \langle r',\ \psi'\rangle$ and $\Gamma; o; \rho \vdash_s \langle r,\ \psi\rangle \to \langle r'',\ \psi'\rangle$ as required.

□

Proof of Lemma 3.1 By double induction. Let $\Sigma_0 = \{(\ell^*, \sigma^*) \mapsto \bot\}$ and $\Gamma; o; \rho \vdash_b \langle \ell,\ \sigma,\ \epsilon,\ \sigma,\ \Gamma(\ell),\ \Sigma_i\rangle \to \Sigma_{i+1}$.

- Suppose $(\ell, \sigma_0) \mapsto r; c \in \Sigma_0$. This contradicts $\Sigma = \{(\ell^*, \sigma^*) \mapsto \bot\}$ hence the result holds vacuously.
- Suppose $(\ell, \sigma_0) \mapsto r; c \in \Sigma_{i+1}$.
  - Suppose unfold is never applied. Then there exists $(\ell, \sigma_0) \mapsto \bot \in \Sigma_i$ such that $\Gamma(\ell) = s_0; c_0$ and $\Gamma; o; \rho \vdash_s \langle s_0,\ \sigma_0\rangle \to \langle r_0,\ \sigma_1\rangle$. Then $r = r_0$ and $c = c_0$. Hence 1a holds. Observe 1b and 1c hold vacuously.
    Let $\psi \models \sigma_0$. By lemma 3.1 it follows $\Gamma; o; \rho \vdash_s \langle s_0,\ \psi\rangle \to \langle r',\ \psi'\rangle$ and $\Gamma; o; \rho \vdash_s \langle r_0,\ \psi\rangle \to \langle r'',\ \psi'\rangle$ as required. Thus 2 holds.
  - Suppose unfold is applied j times. Then $\Gamma; o; \rho \vdash_b \langle \ell,\ \sigma_0,\ r_0; \ldots; r_j,\ \sigma_{j+1},\ \texttt{goto}\ \ell',\ \Sigma_i\rangle \to \Sigma_{i+1}$. Put $\ell_{j+1} = \ell'$ and $\Gamma(\ell_{j+1}) = s_{j+1}; c_{j+1}$. It follows $\Gamma; o; \rho \vdash_b \langle \ell,\ \sigma_0,\ r_0; \ldots; r_j,\ \sigma_{j+1},\ s_{j+1}; c_{j+1},\ \Sigma_i\rangle \to \Sigma_{i+1}$ hence $\Gamma; o; \rho \vdash_s \langle s_{j+1},\ \sigma_{j+1}\rangle \to \langle r_{j+1},\ \sigma_{j+2}\rangle$ and $\Gamma; o; \rho \vdash_b \langle \ell,\ \sigma_0,\ r_0; \ldots; r_j; r_{j+1j},\ \sigma_{j+1},\ c'_{j+1},\ \Sigma_i\rangle \to \Sigma_{i+1}$. Observe that if $c_{j+1} = \texttt{if}\ e_i\ \texttt{goto}\ \ell';\ \texttt{goto}\ \ell''$ and $\ell' \neq \ell''$ then $c'_{j+1} = \texttt{goto}\ \ell'$ if $\rho; \sigma_{i+1} \vdash_e e_i \to 1$ and otherwise $c'_{j+1} = \texttt{goto}\ \ell''$. Moreover if $c_{j+1} = \texttt{goto}\ \ell'$ then $\ell' = \ell_{i+1}$. Thus 1b, 1c and 1d hold as well as 1a.
    Let $\psi \models \sigma_0$. By induction $\Gamma; o; \rho \vdash_s \langle r_0; \ldots; r_j,\ \psi\rangle \to \langle r',\ \psi'\rangle$ and $\Gamma; o; \rho \vdash_s \langle s_0; \ldots; s_j,\ \psi\rangle \to \langle r'',\ \psi'\rangle$. Since $\psi \models \sigma_0$ it follows $\psi' \models \sigma_{j+1}$ hence by lemma 3.1 it follows $\Gamma; o; \rho \vdash_s \langle s_{j+1},\ \psi'\rangle \to \langle r',\ \psi''\rangle$ and $\Gamma; o; \rho \vdash_s \langle r_{j+1},\ \psi'\rangle \to \langle r'',\ \psi''\rangle$ therefore the result follows.

□

Proof for Proposition 3.2 Put $\Sigma_0 = \Sigma$ and let $\Gamma; o; \rho \vdash_b \langle \ell,\ \sigma,\ \epsilon,\ \sigma,\ \Gamma(\ell),\ \Sigma_i\rangle \to \Sigma_{i+1}$ where $(\ell, \sigma) \mapsto \bot \in \Sigma_i$. Then $\Sigma_k = \Sigma'$ for some $k > 0$. Observe that $(\ell, \sigma) \mapsto \bot \notin \Sigma_k$. Now suppose $(\ell, \sigma) \mapsto r; \texttt{goto}\ \ell' \in \Sigma_k$. By rule specialse there exists $0 < j < k$ such that $(\ell, \sigma) \mapsto \bot \in \Sigma_j$, $\rho \vdash_c \langle \ell',\ \sigma',\ \Sigma_j\rangle \to \Sigma_{j+1}$ and $\Gamma; o; \rho \vdash_s \langle r,\ \sigma\rangle \to \langle r',\ \sigma'\rangle$ The result follows by rules covered and uncovered. □

Proof of Proposition 3.3 Proof by induction.

- Consider rule rename. Suppose $\Sigma' = \mathsf{rename}_{(\ell^\dagger, \sigma^\dagger, \ell^\ddagger)}(\Sigma)$. Observe that $\ell^\ddagger$ is fresh, that is, $(\ell^\ddagger, \sigma^*) \mapsto r^* \notin \Sigma$. Let $(\ell', \sigma') \mapsto s \in \Sigma$.
  - Suppose $\ell' = \ell^\dagger$.
    - Consider $(\ell, \sigma) \mapsto r; \texttt{goto}\ \ell' \in \Sigma$ where $\Gamma; o; \rho \vdash_s \langle r,\ \sigma\rangle \to \langle r',\ \sigma'\rangle$ and $\sigma' \models \sigma^\dagger$. Then $\texttt{goto}\ \ell'$ is replaced by $\texttt{goto}\ \ell^\dagger$ but there exists $(\ell^\dagger, \sigma^\dagger) \mapsto s' \in \Sigma'$ so closure is preserved.
    - Consider $(\ell, \sigma) \mapsto r; \texttt{goto}\ \ell' \in \Sigma$ where $\Gamma; o; \rho \vdash_s \langle r,\ \sigma\rangle \to \langle r',\ \sigma'\rangle$ and $\sigma' \not\models \sigma^\dagger$. Then $\texttt{goto}\ \ell'$ is retained but there still exists $(\ell', \sigma'') \mapsto s' \in \Sigma'$ where $\sigma' \models \sigma''$ so again closure is preserved.
  - Suppose $\ell' \neq \ell^\dagger$. If $(\ell, \sigma) \mapsto r; \texttt{goto}\ \ell' \in \Sigma$ then $(\ell, \sigma) \mapsto r; \texttt{goto}\ \ell' \in \Sigma'$ hence closure is preserved.
- Consider rule rename1. Then $\Sigma = \Sigma'$ hence Σ' is closed.
- Consider rule rename2. By induction Σ' is closed hence by induction Σ'' is closed as required.

□