Kent Academic Repository

Full text document (pdf)

Citation for published version

Singh, Ranjeet and King, Andy (2015) Partial Evaluation for Java Malware Detection. In: Proietti, Maurizio and Seki, Hirohisa, eds. Twenty fourth International Symposium on Logic-Based Program Synthesis and Transformation. Lecture Notes in Computer Science, 8991 . Springer, pp. 133-147. ISBN 978-3-319-17821-9.

DOI

https://doi.org/10.1007/978-3-319-17822-6_8

Link to record in KAR

https://kar.kent.ac.uk/42104/

Document Version

Pre-print

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version.
Users are advised to check **http://kar.kent.ac.uk** for the status of the paper. **Users should always cite the published version of record.**

Enquiries
For any further enquiries regarding the licence status of this document, please contact:
researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at **http://kar.kent.ac.uk/contact.html**

KAR Kent Academic Repository

University of Kent

Partial Evaluation for Java Malware Detection

Ranjeet Singh and Andy King

School of Computing, University of Kent, UK, 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. 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 a typed three-address code suitable for optimisation and program analysis, and also demonstrates how the residual 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 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 that loophole, and well as 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 [14] 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 [1] to access the user's machine. Client-side attacks provide a way of bypassing a firewall that block ports to users' machines and, are proving to be increasingly popular: last year many JVM exploits were added to the Metasploit 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 engineers, a script kiddie and or a skilled black-hat can reuse a JVM vulnerability reported in Metasploit, applying obfuscation so that it is not recognised by even up-to-date AV detection software.

Experimental evidence suggests that commercial AV software use Metasploit as 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 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 classification of Java applets [17].

The essential difference between running a program in an interpreter and partially evaluating it within a partial evaluator is that the latter operates over a partial store in which not all variables have known values; the store determines which parts of the program are executed and which parts are retained in the so-called residual. Partial evaluation warrants consideration in malware detection because it offers a continuum between the entirely static and the entirely dynamic approaches. In particular, one can selectatively execute parts of the program, namely those parts that mask suspicious activity, and then use the residual in AV scanning. This avoids the overhead of full execution while finessing the problem of incomplete data that arises when a program is evaluated without complete knowledge of its environment. On the theoretical side, partial evaluation provides nomenclature (eg. polyvariance) and techniques (eg. generalisation) for controlling evaluation and specialisation. On the practical side, partial evaluation seems to be partially appropriate for AV matching because Java exploits are often obfuscated by string obfuscation and by using advanced language features such as reflection. Although reflection is designed for such applications as development environments, debuggers and test harnesses, it can also be applied to hide a method call that is characteristic of the exploit. This paper will investigates how partial evaluation can be used to deobfuscate malicious Java software; it argues that AV scanning can be improved by matching on the residual JVM code, rather than original JVM code itself.

1.1 Contributions

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

- The paper describes the semantics of a partial evaluator for Jimple [20]. Jimple is a typed three-address intermediate representation that is designed to support program analysis and, conveniently, can be decompiled back into Java for AV matching.
- 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.

```
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 1.1. Method call and its obfuscation taken from CVE-2012-4681

– The paper describes how partial evaluation can be used in tandem with an abstract domain so as to avoid raising an error when a branch condition cannot be statically resolved [18, section 3.3]. In such a situation, one branch might lead to an environment whose bindings are inconsistent with that generated along another. Rather than abort when inconsistent environments are detected at a point of confluence, we merge the environments into an abstract environment that preserves information from both, so that partial evaluation can continue.

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, malicious applets frequently use reflection to invoke vulnerable methods. This is illustrated by the code in listing 1.1 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 operation 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

```
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.324l324an324g23.4
    S234e3c24u324r3i4t324y23M4a23n4ag234er";

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

Listing 1.2. String Obfuscation with numeric characters

method names, and thereby improving the chances of outwitting a signature-based AV system. Listing 1.2 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 1.3 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 also no reason why other obfuscations [6] 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 1.2 might be replaced with a fresh identifier, so as to mask the invocation of a known decipher method.

3 Partial Evaluation

In this section we outline a partial evaluator for removing string obfuscation and reflection from Jimple code, which is a three address intermediate representation (IR) for the Java programming language and byte code. Jimple is supported by the Soot static analysis framework and, quite apart from its simplicity, Soot provides support for translating between Jimple and Java.

There are two approaches to partial evaluation: online and offline. In the online approach specialisation decisions are made on-the-fly, based on the values of

```
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 1.3. String obfuscation using the rot13 substitution cipher

expressions that can be determined at that point in the specialisation process. In the offline approach, the partial evaluator performs binding time analysis, prior to specialisation, so as to classify expressions as static or dynamic, according to whether their values will be fully determined at specialisation time. This classification is then used to control unfolding, so that the specialisation phase is conceptually simple. The online approach, however, mirrors the structure of the interpreter in the partial evaluator, and hence is easier to present (and ultimately justify by abstract interpretation). We therefore follow the online school.

Figures 1 and 2 present some highlights of the partial evaluator, which specialises sequences of Jimple instructions, that are tagged with labels for conditional jumping. The sequel provides a commentary on some representative instructions. In what follows, l denotes a location in memory, x a variable, and v a value. A value is either a primitive value, such as an integer or a boolean, or an object, or $\top$ which is used to indicate the absence of information. An object is considered to be a class name, C, paired with an environment ρ, together denoted $C : \rho$; C is the name of the class from which the object is instantiated and ρ specifies the memory locations where the fields (internal variables) of the object are stored.

The partial evaluator uses an environment ρ and a store σ to record what is known concerning the values of variables. The environment ρ is a mapping from the set of variables to the memory locations, and the store σ is a mapping from locations to values. The partial evaluator is presented as a function $\mathcal{P}[\![S]\!]\langle\rho, \sigma, o\rangle$, which executes the sequence of instructions S in the context of an environment ρ, store σ and current object o.

$\mathcal{P}[\![var\ t\ x; S]\!]\langle\rho, \sigma, o\rangle =$
$\quad v = \mathsf{default}(t)$
$\quad l = \mathsf{allocate}(v)$
$\quad \rho' = \{x \mapsto l\}$
$\quad \sigma' = \{l \mapsto v\}$
$\quad emit[\![var\ t\ x]\!]$
$\quad \mathcal{P}[\![S]\!]\langle\rho \circ \rho', \sigma \circ \sigma', o\rangle$

$\mathcal{P}[\![x := y \oplus z; S]\!]\langle\rho, \sigma, o\rangle =$
$\quad$ if $\sigma(\rho(y)) = \top \vee \sigma(\rho(z)) = \top$ then
$\quad\quad emit[\![x := y \oplus z]\!]$
$\quad\quad \sigma' = \sigma \circ \{\rho(x) \mapsto \top\}$
$\quad\quad \mathcal{P}[\![S]\!]\langle\rho, \sigma', o\rangle$
$\quad$ else
$\quad\quad v = \sigma(\rho(y)) \oplus \sigma(\rho(z))$
$\quad\quad emit[\![x := v]\!]$
$\quad\quad \sigma' = \sigma \circ \{\rho(x) \mapsto v\}$
$\quad\quad \mathcal{P}[\![S]\!]\langle\rho, \sigma', o\rangle$
$\quad$ endif

$\mathcal{P}[\![x\ :=\ @this; S]\!]\langle\rho, \sigma, o\rangle =$
$\quad emit[\![x\ :=\ @this]\!]$
$\quad \sigma' = \{\rho(x) \mapsto o\}$
$\quad \mathcal{P}[\![S]\!]\langle\rho, \sigma \circ \sigma', o\rangle$

$\mathcal{P}[\![x\ :=\ @parameter_i; S]\!]\langle\rho, \sigma, o\rangle =$
$\quad emit[\![x\ :=\ @parameter_i]\!]$
$\quad \sigma' = \{\rho(x) \mapsto \sigma(\rho(parameter_i))\}$
$\quad \mathcal{P}[\![S]\!]\langle\rho, \sigma \circ \sigma', o\rangle$

$\mathcal{P}[\![x\ :=\ new\ C; S]\!]\langle\rho, \sigma, o\rangle =$
$\quad \langle\bar{t}, \bar{f}\rangle = getFields(C)$
$\quad \bar{l} = \mathsf{allocates}(\bar{f})$
$\quad \bar{v} = \mathsf{defaults}(\bar{t})$
$\quad \rho' = \{f_0 \mapsto l_0, \ldots, f_n \mapsto l_n\}$
$\quad \sigma' = \{l_0 \mapsto v_0, \ldots, l_n \mapsto v_n, \rho(x) \mapsto C : \rho'\}$
$\quad emit[\![x\ :=\ new\ C]\!]$
$\quad \mathcal{P}[\![S]\!]\langle\rho, \sigma \circ \sigma', o\rangle$

Fig. 1. Outline of partial evaluator: declarations and assignments

$\mathcal{P}[\![return\ x; \_]\!]\langle\rho, \sigma, o\rangle =$
  $\sigma' = \{\rho(return) \mapsto \sigma(\rho(x))\}$
  $emit[\![return\ x]\!]$
  $\langle\rho, \sigma \circ \sigma', o\rangle$

$\mathcal{P}[\![x\ :=\ \mathsf{virtualinvoke}(obj, m(\bar{t}), \bar{y}); S]\!]\langle\rho, \sigma, o\rangle =$
  $if\ \sigma(\rho(obj)) = \top\ then$
    $emit[\![x := \mathsf{virtualinvoke}(obj, m(\bar{t}), \bar{y})]\!]$
    $\sigma' = \{\rho(x) \mapsto \top\}$
    $\mathcal{P}[\![S]\!]\langle\rho, \sigma \circ \sigma', o\rangle$
  $else\ if\ \sigma(\rho(obj)) = C : \rho'$
    if $C = Method \wedge m = invoke$
      if $\sigma(\rho'(method)) = null$ then error
      $\bar{y}' = \langle y_1, \ldots, y_n\rangle$
      $emit[\![x := \mathsf{virtualinvoke}(y_0, \sigma(\rho'(method)), \bar{y}')]\!]$
      $\sigma' = \{\rho(x) \mapsto \top\}$
      $\mathcal{P}[\![S]\!]\langle\rho, \sigma \circ \sigma', o\rangle$
    $else$
      $\bar{v} = \sigma(\rho(\bar{y}))$
      $B = findMethod(C.m(\bar{t}))$
      $\bar{l} = \mathsf{allocates}(\bar{v})$
      $k = \mathsf{allocate}(result)$
      $\rho'' = \{parameter_0 \mapsto l_0, \ldots, parameter_n \mapsto l_n, result \mapsto k\}$
      $\sigma' = \sigma \circ \{l_0 \mapsto v_0, \ldots, l_n \mapsto v_n\}$
      $\langle \_, \sigma'', \_\rangle = \mathcal{P}[\![B]\!]\langle\rho'' \circ \rho', \sigma', C : \rho'\rangle$
      $\sigma''' = \{\rho(x) \mapsto \sigma''(\rho''(result))\}$
      $\mathcal{P}[\![S]\!]\langle\rho, \sigma \circ \sigma''', o\rangle$

$\mathcal{P}[\![if\ x\ goto\ l; S]\!]\langle\rho, \sigma, o\rangle =$
  if $\sigma(\rho(x)) = 0$ then
    $\mathcal{P}[\![S]\!]\langle\rho, \sigma, o\rangle$
  else if $\sigma(\rho(y)) = 1$ then
    $S = lookup(l)$
    $\mathcal{P}[\![S]\!]\langle\rho, \sigma, o\rangle$
  else if $\sigma(\rho(x)) = \top$ then
    $n\ =\ lookup(S, P)$
    $c\ =\ confluence(l, n)$
    $emit[\![if\ n\ goto\ l]\!]$
    $T_t\ =\ branch(l, c)$
    $T_f\ =\ branch(n, c)$
    $\langle\rho, \sigma_t, o\rangle = \mathcal{P}[\![T_t]\!]\langle\rho, \sigma, o\rangle$
    $\langle\rho, \sigma_f, o\rangle = \mathcal{P}[\![T_f]\!]\langle\rho, \sigma, o\rangle$
    $\sigma' = \{l \mapsto v \mid l \in codomain(\rho) \wedge v = \text{if } \sigma_t(l) = \sigma_f(l) \text{ then } \sigma_t(l) \text{ else } \top\}$
    $\mathcal{P}[\![P\ drop\ c]\!]\langle\rho, \sigma', o\rangle$
  endif

Fig. 2. Outline of partial evaluator: control-flow

3.1 type declarations

A statement *var t x* declares that x is a of type t, where t is either primitive or a user-defined class. Such a declaration is handled by allocating a memory location l using the auxiliary allocate and then updating the environment ρ' to reflect this change. The store is also mutated to map location l to the default value for the type t, which is given by the auxiliary function default. The default values for the primitives types are 0 for *int* and 0 for *boolean*. The default types for object types is *null*.

3.2 new

A statement $x = new\ C$ instantiates the class C to create an object that is represented by a pair $C : \rho$ where the environment ρ maps the fields of C to memory locations that store their values. Different objects $C : \rho_1$ and $C : \rho_2$ from the same class C map the same field variables to different locations. The auxiliary method getFields retrieves the types and the names of the fields of the class C. The function defaults takes a vector of types $\boldsymbol{t}$ and returns a vector of default values that is used to populate the fields, following the conventions of default.

3.3 arithmetical operations

An assignment statement $x := y \oplus z$ can only be statically evaluated if both the variables y and z are bound to known values. In this circumstance the assignment $x := y \oplus z$ is specialised to $x := v$ where v is the value of the expression $y \oplus z$. The store σ' is updated to reflect the new value of x, as the value of x is statically known. Note that the residual includes $x := v$, even though information on x is duplicated in the store σ', so as to permit subsequent statements, with reference x, to be placed in the residual without substituting x with its value v. If there are no statements that reference x then the assignment $x := v$ will be removed by dead variable elimination, which is applied as a post-processing step.

3.4 this and parameters

In Jimple there is a distinguished variable *this* which stores the current object reference which, in the partial evaluator, is modelled with the current object o, that is passed with the environment and the store. An assignment statement $x := @this$ thus merely updates the location $\rho(x)$ with o.

Also in Jimple, a special variable $parameter_i$ is used to store the location of the i^{th} formal argument of a method call, where the first argument has an index of 0. This is modelled more directly in the partial evaluator, so that an assignment statement $x := @parameter_i$ updates the location $\rho(x)$ with the value of this parameter.

3.5 return and virtualinvoke

The statement *return* x writes the value of x to a special variable return, which is subsequently read by virtualinvoke.

The handling of virtualinvoke$(obj, m(\boldsymbol{t}), \boldsymbol{y})$ is worthy of special note, both in the way reflective and non-reflective calls are handled. A reflective call arises when the method m coincides with invoke and the object obj is of type Method. The reflective method call is a proxy for the so-called reflected method call. The reflected method call is not applied to the object obj but an object that is prescribed by the first parameter of $\boldsymbol{y}$. Moreover, the method that is applied to this object is given by obj in a distinguished field that, for our purposes, is called *method*. This field either contains null, indicating that it has been initialised but not been reset, or a string that represents the name of a method that is to be invoked. If the method field is null then an error is issued, otherwise the string stored in the field method is used to generate the residual. Note that the reflected method call is not invoked; merely placed in the residual. Note too that the first argument of virtualinvoke in the residual is y_0 whereas the last is the vector $\boldsymbol{y}'$ which coincides with $\boldsymbol{y}$ with the exception that the first element has been removed. The first argument is the variable name (rather than the object itself) to which the residuated method will ultimately be applied; the third argument is the list of actual arguments that will be passed to the method on its invocation.

In the case of a non-reflected call, the values of the parameters are looked up, and then an auxiliary function findMethod is applied to find a block B which is the entry point into the method of the class C whose signature matches $m(\boldsymbol{t})$. The function allocates is then called to generate fresh memory locations, one for each actual argument $y_0, \ldots, y_n$. The environment is then extended to map the distinguished variables $parameter_0, \ldots, parameter_n$ to fresh memory locations, so as to store the actual arguments. The partial evaluator is then recursively involved on the block B using $C : \rho'$ as the object. The net effect of the method call is to update the store σ' which is used when evaluating the remaining statements S.

Note that this formulation assumes that method calls are side-effect free. Although this is true for string obfuscation methods that we have seen, for full generality the partial evaluator should be augmented with an alias analysis, in the spirit of side-effect analysis [9], that identifies those variables that are possibly modified by a method call, hence must be reset to $\top$.

3.6 goto

Specialising a conditional jump in Jimple is not conceptually difficult, but is complicated by the way if x goto $l; S$ will drop through to execute the first instruction of the sequence S if the boolean variable x is false. This makes the control-flow more difficult to recover when the value of x is unknown. When x is known to be true, however, the partial evaluator is merely redirected at a block B that is obtained by looking up the sequence whose first instruction is labelled

by l. Conversely, if x is known to be false, then partial evaluation immediately proceeds with S.

In the case when x has an undetermined value, the partial evaluator explores both branches until the point of confluence when both branches merge. Then the partial evaluator continues at the merge point, relaxing the store to σ' so that it is consistent with the stores that are derived on both branches. Note that partial evaluation does not halt if the stores are inconsistent; instead it will unify the two stores by replacing any inconsistent assignment for any location with an assignment to $\top$. Note that it is only necessary to unify those locations that are reachable from the variables that are currently in scope.

To realise this approach, an auxiliary function lookup is used to find the position n of the first statement of S in the list P which constitutes the statements for the currently executing method. This is the position of the first statement immediately after the conditional branch. Then a function confluence examines the control-flow graph of the method so as to locate the confluence point, of the true and false branches, identified by the index c of S. Both branches are evaluated separately with two copies of the environment, until the confluence point where the two environments are merged. Partial evaluation then resumes at the confluence point, which corresponds to the instruction sequence P drop c, namely, execution is continued at the c^{th} instruction of the sequence P.

3.7 Example

Listing 1.4 gives the before and after for a method call that is obfuscated by reflection and string obfuscation, using the ROT13 simple letter substitution cipher given in Listing 1.3. The residual Jimple code is presented as Java for readability. Completely unfolding the `rot13` method call decrypts the string `qbFbzrguvat` as the string `doSomething`. This string statically defines the value of the object `method`, allowing `method.invoke(m, null)` to be specialised to `m.doSomething()`, thereby removing the reflective call. Note that the variables `encrypted` and `method` cannot be removed without dead variable elimination.

4 Experiments

To assess how partial evaluation can aid in AV matching, a number of known applet malware samples from the Metasploit exploit package [2] were obfuscated using the techniques outlined in section 2. Details of the samples are given in Fig. 3; the samples were chosen entirely at random. So as to assess the effect of partial evaluation against a representative AV tool, 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 [15] and is illustrated in the chart given in Figure 3. Conveniently, VirusTotal [19] 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.

```
//BEFORE
public static void main(String[] args) {
    Main m = new Main();
    String encrypted = "qbFbzrguvat";
    Method method = m.class.getMethod(rot13(encrypted));
    method.invoke(m, null);
}

//AFTER
public static void main(String[] args) {
     Main m = new Main();
     String encrypted = "qbFbzrguvat";
     Method method = m.class.getMethod(rot13(encrypted));
     m.doSomething();
}

//AFTER DEAD VARIABLE ELIMINATION
public static void main(String[] args) {
     Main m = new Main();
     m.doSomething();
}
```

Listing 1.4. Before and after partial evaluation

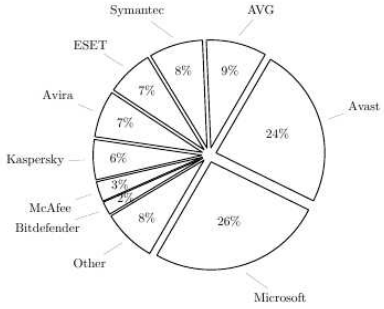
Unfortunately, developing a complete partial evaluator for Jimple is a major undertaking, since it is necessary to support the entire Java API and runtime environment, which itself is huge. To side-step this engineering effort, we implemented a partial evaluator in Scala, following the description in 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 make it is straightforward to engineer a parser for Jimple.) Although other objects could be handled in the same way, we simply took each of these obfuscated CVEs and extracted the Jimple code and methods that manipulated strings. This code was then partially evaluated so as to deobfuscate the string handling. The CVEs were then hand-edited to reflect the residual, and then ran through VirusTotal to check that the effects of obfuscation had been truly annulled. Future implementation work will be to automate the entire process, namely translate the Jimple residual into Java using Soot [8] and then invoke VirusTotal automatically through its public web API.

Table 1 details the detection rates for the AVs given in Figure 3, without obfuscation, with just string obfuscate, with just reflection obfuscation, and with both obfuscations applied. This gives four experiments in all. 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.

The results show that in most cases the AVs detect most of the exploits in their unadulterated form. Exploits CVE-2012-5088 and CVE-2013-2460 go

Fig. 3. CVEs and AVs

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



the most undetected, which is possibly because 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 the removing this obfuscation alone is truly worthwhile. Moreover, after introducing reflection 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.

Most importantly, applying partial evaluation to the CVEs used in the fourth experiment restores the detection rates to those of the first experiment. Thus detection is improved, without having to redistribute the signature database.

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 [17]. 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 [13]. 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

Table 1. Experimental Results

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	✗	✗	✓	✗	✗	✗	✓	✗

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.

The objective 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 [16] . Civet [18] represents state-of-the-art in removing Java reflection; it does not apply binding-time analysis (BTA) [3] 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 [4], 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 [8]. Quite apart from its role in deob-

fuscation, partial evaluation can also be applied in obfuscation [10]: a modified interpreter, that encapsulates an obfuscation technique, is partially evaluated with respect to the source program to automatically obfuscate the source. Program transformation has been proposed for deobfuscating binary programs [5], by unpacking and removing superfluous jumps and junk, again with the aim of improving AV scanning. This suggest that partial evaluation also has a role in binary analysis, where the aim is to make malware detection more semantic [7].

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 [12] 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 [11] 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 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. We also deliberately do not unfold recursive methods, though this is a somewhat draconian limitation. Future work will aim to quantify how termination analysis can be applied in an online setting to improve the quality of malware detection.

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. This would thwart our approach to deobfuscation since the reflected call would be deobfuscated in the residual, but would not actually be evaluated on a given string. Thus we will explore how partial evaluation can be repeatedly applied to handle these multi-layered forms of obfuscation.

We will also examine how partial evaluation can remove less common forms of Java obfuscation such as control flow obfuscation and serialization and deserialization obfusation, the latter appearing to be as amenable to partial evaluation as string obfuscation. In the long term we will combine partial evaluationi with code similarity matching, drawing on techniques from information retrieval.

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. 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.

References

1. Rapid 7. Java Applet JMX Remote Code Execution, 2013.
2. Rapid 7. Metasploit, 2014.
3. L. Andersen. Binding-Time Analysis and the Taming of C Pointers. In *PEPM*, pages 47–58. ACM, 1993.
4. M. Braux and J. Noyé. Towards Partially Evaluating Reflection in Java. In *PEPM*, pages 2–11. ACM, 2000.
5. M. Christodorescu, S. Jha, J. Kinder, S. Katzenbeisser, and H. Veith. Software Transformations to Improve Malware Detection. *Journal of Computer Virology*, 3(4):253–265, 2007.
6. C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, 2009.
7. M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray. A Semantics-based Approach to Malware Detection. *ACM TOPLAS*, 30, 2008.
8. A. Einarsson and J. D. Nielsen. A Survivor's Guide to Java Program Analysis with Soot. Technical report, 2008.
9. A. Flexeder, M. Petter, and H. Seidl. Side-Effect Analysis of Assembly Code. In *SAS*, Lecture Notes in Computer Science, pages 77–94. Springer, 2011.
10. R. Giacobazzi, N. D. Jones, and I. Mastroeni. Obfuscation by Partial Evaluation of Distorted Interpreters. In *PEPM*, pages 63–72. ACM, 2012.
11. 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.
12. 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.
13. T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4), 1976.
14. National Institute of Standards and Technology. Vulnerability Summary for CVE-2013-3346, 2013.
15. OWASP. Metasploit Java Exploit Code Obfuscation and Antivirus Bypass/Evasion (CVE-2012-4681), 2013.
16. 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.
17. J. Schlumberger, C. Kruegel, and G. Vigna. Jarhead: Analysis and Detection of Malicious Java Applets. In *ACSAC*, pages 249–257. ACM, 2012.
18. A. Shali and W. R. Cook. Hybrid Partial Evaluation. In *OOPSLA*, pages 375–390. ACM, 2011.
19. H. Sistemas. VirusTotal Analyses Suspicious Files and URLs, 2014. `https://www.virustotal.com/`.
20. R. Valleé Rai and L. J. Hendren. Jimple: Simplifying Java Bytecode for Analyses and Transformations. Technical Report TR-1998-4, McGill University, 1998.