

Self-Protecting Mobile Agents Obfuscation Techniques Evaluation Report

Lee Badger
Larry D'Anna
Doug Kilpatrick
Brian Matt
Andrew Reisse
Tom Van Vleck

NAI Labs

Report #01-036

November 30, 2001¹

Abstract

This document presents an analysis of various program obfuscation techniques performed as part of the Self-Protecting Mobile Agents (SPMA) project. We have developed the Java Binary Enhancement Tool (JBET) to explore the real-world tradeoffs in translating and obfuscating Java bytecodes. We consider methods for obfuscating long-term data values, temporary data values, control flow, memory management and type representation. We present techniques that have considerably more power than other available obfuscators. No tools or methods known to the authors can quickly de-obfuscate programs that have been transformed using the methods presented here; we believe that programs transformed using these methods can be used in hostile environments for a limited time without revealing their algorithms and contents. We consider obfuscation methods for Java programs that may impose a cost in execution time and program size.²

1 Introduction

Obfuscation transforms a program into another program that has equivalent behavior but which is harder to understand. This report describes automatable techniques for obfuscating computer programs; that is, techniques that can be applied programmatically to software without requiring human assistance. Successful program obfuscation would confer a number of benefits, including:

1. protection of transient secrets in programs,

¹ Updated March 22, 2002.

² This research was funded under DARPA contract N66001-00-C-8602.

2. license management for networked (e.g., client/server) software,
3. temporary protection of digital watermarks in programs,
4. software-based tamper resistance, and
5. protection of mobile agents.

Obfuscation is substantially weaker than cryptography – a sufficiently determined attempt to de-obfuscate a program and recover its secrets will always succeed. Such an attack on the obfuscation, however, is likely to require substantial resources (i.e., personnel) as well as time. The goal of obfuscation is to raise dramatically the cost of reverse engineering in order to deter attacks on low-to-moderate value programs, and to delay attacks on high-value programs. For some applications (such as mobile agents), even a modest delay can be instrumental to a system’s survival.

We have extended the research of Collberg et al [CTL97a], [CTL98a], [CTL98b], [CT00] and Wang et al [W00], [WHK+00], [WDH+01] to obfuscation of Java programs, and built an extensible tool that supports the obfuscation techniques they describe and additional obfuscation techniques proposed by us. Because of the many restrictions of the Java runtime system, we believe that Java is a particularly difficult environment for obfuscating transformations, and that the techniques we identify for Java will likely work even better in less restrictive environments, such as natively compiled languages.

We have built a tool for analyzing and transforming Java bytecodes: JBET (Java Binary Enhancement Tool). We have used our tool to explore the space of possible obfuscation techniques. About half of the obfuscation techniques discussed in this report have not been implemented yet. We have, however, formulated our techniques based on experience with JBET; this experience has been critical in discriminating between feasible and unfeasible techniques, and in understanding tradeoffs between obfuscation, code/data size, and performance.

A considerable amount of confusion has surrounded the term obfuscation; to avoid confusion, we begin with normal dictionary definitions:

obfuscate – to make obscure

obscure -- shrouded in or hidden by darkness; not clearly seen or easily distinguished

program -- a sequence of coded instructions that can be inserted into a mechanism (as a computer)³

Our goal here is to describe automatable techniques that read in a program P , and an obfuscation policy p , and then generate a new, *obfuscated program* OP .

³ Definitions from Merriam-Webster’s Dictionary.

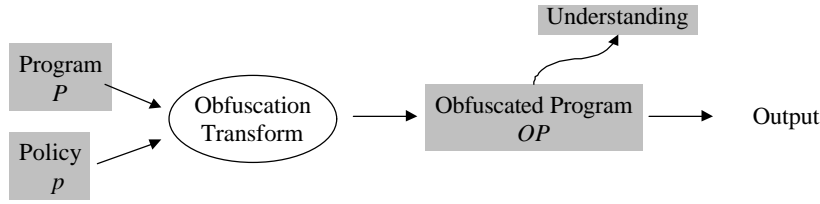


Figure 1

Figure 1 displays the concept: both the original program and the policy feed into a transformation procedure that generates the obfuscated program. After some period of time and expended effort, an attacker can gain understanding of OP . It has been postulated that the program can run safely for a limited time.[Hohl98] As noted above, obfuscation provides weaker protection than cryptography, and therefore a judgment must be made about how long the obfuscation can be trusted to protect the program.

For our objectives, it is important that OP have two key properties:

1. given identical input data, the behaviors of P and OP are semantically identical at a specified interface, and
2. the relationship between OP 's state and OP 's behavior is obscure to any observer who has not seen p .

Property 1 merely asserts that, at some interface of interest (e.g., library APIs, system calls), OP either behaves exactly like P , or OP 's behavior has exactly the same effects as P 's behavior.⁴ If the interface is chosen reasonably, OP can be used wherever P can be used.

Property 2 asserts that knowledge of p is necessary for an observer to understand, without de-obfuscating OP , how OP 's behavior is driven by OP 's state. In this context, we use the term “state” to designate both data held in variables, and data held in instructions. Under the assumption that an observer cannot de-obfuscate the program, property 2 implies two important limitations on attackers:

1. an attacker cannot observe sensitive information carried in the program, and
2. an attacker cannot modify selected parts of the program to change its behavior in a predictable way.

If they can be provided, these properties can be used to provide software-based protection in a variety of contexts, e.g., software-based tamper resistance, watermarking, enforcement of licensing, safe mobile agent systems.⁵

Using property 2, we can characterize the space of possible attacks on an obfuscated program:

⁴ For example, if P issues a write() call to output 100 bytes, OP in some cases could issue two write() calls of 50 bytes each.

⁵ See [CTL97a, Hol98, BGI+01,WHK+00, W00, WDH+01] for alternative definitions of program obfuscation and related terms such as black-box security.

Attacks that expose sensitive information. In this class of attacks, an attacker learns sensitive information either by static analysis of *OP*'s code or state, or by dynamic analysis (i.e., running *OP* with various inputs and then studying *OP*'s behavior, and tracing its state from specific points in its execution). In either case, the attacker must identify a part of *OP*'s code or state, and also identify how to interpret the code or state. For example, if the sensitive value is an integer, the attacker must both find the place (or places) where the integer is stored in *OP*, and be able to convert the integer's obscure representation into a standard representation, such as 32 contiguous bits.

Attacks that change behavior. In this class of attacks, an attacker identifies a controlling part of *OP*'s code or state, and modifies that part so that *OP* behaves in a new, but predictable way. For example, a program that compares an input to a stored constant could be modified to compare the input to a different stored constant. As with the first class of attacks, the attacker's objective is to identify and interpret a part of the program's code or state, and the attacker may use either static or dynamic analysis.

The relative ease or difficulty of these classes of attack depends in part on what assumptions we make about the resources available to the attacker. In all cases, we assume that the attacker does not have access to the obfuscation policy p . Other assumptions will affect the work factor for the attacker, for example:

The attacker has complete control of the execution platform. We assume that the attacker has complete control over the execution platform (e.g., the Java Virtual Machine, system calls). This implies that the attacker can trace and profile the execution of *OP*, and can run a debugger on *OP*.

The attacker has source code for P . Since P has the same behavior as *OP* at a given interface, knowledge of P could assist the adversary in analysis of *OP*. For example the attacker might compare the basic blocks of P and *OP* to identify which parts of *OP* perform the known functions of P . Furthermore, knowledge of P 's algorithm could allow the attacker to identify parts of *OP* as being related to identifiable functions of P .

The attacker has seen all of the input data to *OP*. Seeing all the input to *OP* could help the attacker understand how *OP* initializes itself. **Note:** if the attacker also has the source code of P , the attacker can predict *OP*'s behavior since the attacker can simulate it using P . We generally assume that the attacker does not have *both* the source code of P and all of the input read by *OP*. In the case of mobile agents, it will usually be the case that a single attacker (at a node) will not have access to all of the input data to *OP*. For stand-alone applications, however, the attacker probably can have all of the input data via tracing the application's system calls.

The attacker has the source code of the obfuscation tool. We assume that the source code of the obfuscation tool is open and well-known. Any obscurity that could be provided by secret source code to the obfuscation tool would be extremely fragile if the tool became widely used.

The attacker is able to conduct dynamic analysis. We generally assume that the attacker can perform repeated tests on *OP* using different input data to analyze *OP*'s behavior. This gives the attacker considerable leverage in discovering sensitive information held in *OP*. For example, if *OP* holds a sensitive constant (e.g., the maximum price a customer is willing to pay), the attacker can run *OP* repeatedly using different inputs and observe the threshold value where *OP* changes its behavior (e.g., by refusing to purchase). The application of obfuscation to mobile agent systems is a special case where dynamic analysis can be prevented if mobile agents only run when in communication with their peers on benign systems.

The attacker has limited time to compromise *OP*. Given enough time, we believe that a determined attacker will always be able to de-obfuscate *OP*.

While attack classes and attacker assumptions can provide some insight about the feasibility of obfuscating transforms, they are too high-level to support conclusions about the relative costs and benefits of obfuscating transformations. As with attacks on existing computer systems, attacks on obfuscation are often based on exploiting fairly low-level details. In conventional attacks, low-level details of system interfaces and implementations are misused to gain unauthorized access. In attacking obfuscation, low-level details of execution formats and instruction sequences are used to gain information that can then be used to identify and interpret program states.

The interplays between attackers and conventional computer defenses are often characterized as “arms races” since systems are imperfect and attackers are continually seeking to discover vulnerabilities that have been overlooked by the defenders. Usually, if the defender is willing to expend more effort, the defenses can be improved. We view the interplay between attackers and defenders with respect to obfuscating transforms in similar terms. In the case of obfuscation, however, the role of automated tools is perhaps even more central. Automatic obfuscation tools can generate data structures that severely stress manual analysis and require many hours (or days) of analysis to unravel. The attackers must therefore rely on automated tools to attempt to analyze obfuscated programs. The arms race is between the writers of defending (obfuscating) tools and the writers of attacking (de-obfuscating) tools.

We have investigated available freeware and commercial Java obfuscators and decompilers. The obfuscation methods described in this report are substantially more complex than those currently available. This comes with a cost: most existing obfuscators' output programs are smaller than their input, and most produce output programs that execute at more or less the same speed as the original. Our techniques attain substantially more obfuscation by abandoning this constraint.

In our prototyping experience, we have encountered several aspects of the Java platform that require special handling when implementing obfuscation techniques. Additionally, we have discovered that the interfaces between an obfuscated program and its enclosing environment raise obfuscation issues that are unique to the boundary. Finally, we have discovered (and this is not surprising) that obfuscating transforms (e.g., that make data access and control flow dispersed or indirect) carry significant space and performance overheads. This corresponds with prior work done in the context of C

[W00,WDH+01,WHK+00]. Thus, the selection of such techniques depends significantly on the frequency of the operations being obscured. Different techniques should be used with lightweight operations than are used with heavyweight operations. To address these issues and identify appropriate tradeoffs, this report is organized as follows:

- Java Platform Effects on Obfuscation
- Interfacing with the Enclosing Environment
- Control Flow Obfuscation
- Heap-allocated Storage and the Type System
- Parameterized Obfuscation of Long Term Variables
- Transient Variable Obfuscation
- Related Work
- Conclusions

2 Java Platform Effects on Obfuscation

The Java virtual machine (JVM) is a restricted execution environment. Many JVM features, such as typed memory management, are useful to application developers but inhibit program obfuscation. Others ensure that Java bytecodes are safe for the host environment, preventing stack overflow and other memory reference attacks, but restrict the control flow of the program. Our choice of the Java platform, for its mobile agent facilities and relatively simple binary editing, means that we must work within the restrictions of the Java virtual machine.

The remainder of this section describes restrictions of the JVM, and features that we are not addressing at this stage because they are infrequently used and, in some cases, difficult to implement.

2.1 Java Virtual Machine Restrictions

The Java virtual machine goes to significant lengths to protect the system from potentially malicious code. All code output by our tool must pass the checks enforced by the JVM. This section now discusses particular restrictions that an obfuscation tool will have to address.

Pointer arithmetic. Java is type safe, enforcing that pointers are only the target of operations that can target pointers. Since integer operations such as add and multiply cannot target pointers, pointer arithmetic is not possible with the Java pointer type. Fortunately, all Java objects have a single base type, `java.lang.Object`.

Type system. Each Java class file contains the type information for a single class. An attacker with the original source can limit the scope of his reverse engineering efforts significantly if he knows which classes he wishes to modify. Further, the explicit use of casts and virtual method calls makes it easy to determine the real

types from any trivially transformed type graph. We therefore wish to obscure the class structure of the obfuscated program.

Restricted control flow. Java class files have a fixed structure for methods. All code for a method is stored together, with a header describing the exception handling behavior for that method. The Java verifier checks each instruction to ensure that no matter what control flow path is followed to get to that instruction, the operand stack is the same size and contains all the same types, and the types of all registers are the same.

In addition, general control flow to other methods' code is restricted, as only method call and return operations (or exceptions) can transfer control to other methods, and only to particular points within that code.

Hardcoded jump targets. No virtual machine opcode provides jump to address behavior; all jump targets are hardcoded into the class files.

Fixed method size. Java methods must be smaller than 65,536 bytes. This size limitation complicates some transformations.

2.2 Deferred Java Features

Some features of the Java language and virtual machine are less commonly used, or should be considered only after the basic features have been covered. Obscuring the use of these features will not be examined in this report.

Floating point arithmetic. Based on our experience with developing software, we expect that most programs will primarily use integers, and that nearly all use of floating point can be easily converted to integers. Therefore, effort spent in transforming and obfuscating floating point arithmetic is of minimal use to this project at this time.

Reflection. Java allows programs to easily examine their own classes and types through the `java.lang.reflect` package. Supporting this functionality so that a program can get pre-obfuscation structure would require that the program have significant information useful to the attacker hidden internally. We do not believe that this feature is commonly used by self-contained software, and see little reason to support it at this time. Given the experience with C, programs that require some form of reflection can usually simply implement the subset of the reflection functionality that they actually require.

Serialization. Similar to reflection in implementation, a completely general implementation of serialization requires full type information to be preserved. Since we would like to remove as much semantic information from the program as possible, serialization would be difficult to implement. Java programs like Applets making heavy use of serialization and exchanging objects with external programs are not good candidates for obfuscation due to the detail of information exchanged. Note that just because we do not intend to provide a general

implementation of serialization does not mean that we will not support limited serialization for statically determined types.

Synchronization interface Use of synchronization primitives is easy for an attacker to detect, particularly since synchronization instructions are used far less frequently than any other instructions. An attacker can match synchronization instructions from the original source code to those encountered during execution of the obfuscated program. We have not explored emulating synchronization, but we believe the solutions include simulating multiple threads within one Java thread and introducing extra synchronization code or dummy threads.

Future versions of Java (1.4) promise to address parts of this issue by making single threaded I/O-intensive programs possible, through wrappers for system calls like `select()` or `poll()`. Initially, we do not expect to protect the synchronization of threaded programs.

3 Interfacing with the Enclosing Environment

Interfacing with the host presents a number of problems for an obfuscated program, especially in Java. All data must be converted to readable formats when passed to the host, and all objects passed to the host (or obtained from the host, in the example of file stream objects), must be of proper Java type. The attacker can see both input and output during tracing. By themselves, these two capabilities give the attacker a great deal of insight into the behavior of the program [FHS+96]. In combination with advanced debugging tools, these capabilities become even more potent.

An attacker, for instance, can trace the execution of the program, checkpointing changed state at every control-flow point. This trace data then allows the attacker to run the program backwards [Balzer69], finding the places where important data was created or decisions made. Any point of interaction with the enclosing environment represents an interesting point to start the analysis, and a known point of equivalence with the original program. By working backward from the point of interaction, the attacker can examine a much smaller amount of code to determine where the decision that led to the interaction was made and thus shorten the amount of time required to tease out detail of interest.

The attacker can also slice the program [Tip95] to discover the sequence of statements that influence an output value. Static slicing by deleting irrelevant statements from a program can be defended against by obfuscation transformations that prevent static recovery of the control flow graph (see Section 4). Dynamic slicing, which traces the execution of a program under specific inputs, can identify the actual control flow taken as well as intermediate values developed during computation, so that, for example, if a credit limit is stored in encrypted form in an agent, but decrypted and used in a comparison during computation, the decrypted form would be visible during a dynamic slicing trace. Systematic dynamic slicing of a program with multiple inputs could enable an analyst to discover object code that is never used, or to associate particular object code with particular input arguments.

To the greatest degree possible and practical, the obfuscated program should make it difficult to track the interactions with the enclosing environment to the calculations of the

program. There are two separate issues to address. First, we seek to make reverse tracking of control flow difficult. Given an external call that utilizes important information, it should be difficult to determine how the information was obtained internally by the program. Second, we seek to reduce the utility of a simple trace of interactions with the enclosing environment.

3.1.1 Obscuring Program Flow Near the External Interface

Section 5 discusses the issues with the internal use of the Java type system. Separating the internal type system from the external interfaces requires multiple stub functions, so that the code that interfaces with the enclosing environment is isolated from the code that is obscured. Unfortunately, simple software-engineering style “isolation of interface” does not isolate the code to any attacker. Static call graphs are easy to construct, and even if those were hidden, dynamic “stack traces” are equally informative and easy to obtain. Stronger methods of isolation may be required.

One possible solution is to leverage the threading problem. This was suggested in [CTL97a]. By isolating calls to the enclosing environment into their own thread, the trace of execution that lead to the call gets significantly harder to determine. If the transfer of control between threads avoids explicit locking⁶ then tracing output to the relevant code could be made significantly more difficult.

The attacker can monitor a given piece of memory to determine which threads access it. This will allow the attacker to find the communication between the thread that invokes an external operation and the thread that uses the information. However, to find out which piece of memory to monitor, the attacker has to watch all pieces of memory accessed by the thread that interacts with the environment. This will take time, and possibly human intervention.

Additionally, covert channels could be used by the threads to convey small amounts of information. For example, one thread might lock and unlock a particular object not to protect any data, but merely to convey information to a different thread (like the location of data to use for the external call).

3.1.2 Obscuring the Program Interface Trace

From the perspective of the attacker, any use of a system provided library is a point of interaction with the environment. For example, the use of `java.util.Vector`, a common data representation, is a visible interaction with the external environment. In the example below, the use of `System.out.println()` instead of `system.out.write()` gives more information at the interaction point than is necessary.

Take the example of the prototypical “Hello, World!” program:

```
public class Hello {
```

⁶ If thread A sets a specific bit without the use of synchronization, thread B is not guaranteed to see that bit set until thread A invokes a “monitorexit” opcode and thread B invokes a “monitorenter” opcode. As long as only thread A ever writes to that bit and thread B only reads it, the behavior is defined, albeit unbounded in time.

```
    public static void main(Strings[] args) {
        System.out.println("Hello, World!");
    }
}
```

This simple program has four places where it interacts with the enclosing environment: the start of “main”, the marshalling of the “Hello, World!” string into a Java String object, the call to `System.out.println()`, and the return from main.

While it is not possible to eliminate all interactions with the enclosing environment, the Java run time environment encourages programs to use semantically meaningful operations. For example, the Java provided container classes (`Vector`, `Hashtable`) will be used by many programs, giving the attacker useful information about the data of a program. A program might use Java provided stream classes to read in data in sizes meaningful to the program.

To reduce this information flow, the obfuscating tool should reimplement many of the Java-provided convenience classes with obfuscated versions. This could be done on an automated basis by incorporating any pure-Java classes into the obfuscated program before the type and data-flow obfuscations, and on a per-class basis by first reimplementing simple native methods in pure-Java terms. Classes that have both native methods and pure-Java methods could be transformed so that the pure-Java methods are incorporated into the obfuscated program and only the native-method invocations are visible to the enclosing environment.

The problem of reducing the information flow of this nature is similar to the problem of reducing covert channels, and similar issues may arise. A secure FTP gateway will make auxiliary fetch requests to hide the actual data requested, similarly, a transformed program could make additional I/O requests from the environment that it does not actually need.

Techniques used by covert channel analysis may be helpful in finding unnecessary interactions, and transforming groups of interactions into interactions with less useful information. It is unknown at this time if any of those techniques can be used in this context.

3.2 Data Translation

Some interactions with the enclosing environment will take complex objects as arguments. In the above example, `System.out.println()` takes a String object. To properly use these interfaces, the obfuscated code will have to transform objects from a complex internal form described in section 6 to the normal external form. Generating the code to map one form to the other is not expected to be difficult. Hiding the code may be.

Code that marshals data into the export form has high value to the attacker, as it describes the obfuscated form in easy-to-understand terms. The code will be easy to identify, as the construction of new objects has its own opcode,⁷ and so this code will require a high

⁷ The Java opcode `new` takes an index into the constant pool that specifies the class of the object to be created. The constant pool can not be changed at run time, so the type of object being created is obvious to

degree of control-flow and memory obfuscation. It may be possible to use a third, internal format to separate the code that interacts with the environment from the code that makes decisions, but we have not yet investigated this possibility.

Our first priority is to reduce calls to external code, as they provide analysis points. Grouping requests, and including some standard classes into the obfuscation domain may help reduce the interactions. To increase the difficulty of working backwards in the code to find out how a decision was made, interactions with the environment should be moved via queuing to a separate thread. Additional spurious input should be requested from the environment to obscure the connection between input and output.

4 Control Flow Obfuscation

Analysis of the control transition instructions within a program can reveal the locations and interpretations of sensitive state, and of state that could be modified to cause a predictable change in program behavior. This information could be used to invalidate the second obscurity property presented in section 1. To address this vulnerability, the control flow constructs of a program must be reorganized to increase the difficulty of such analysis.

Generally, control flow analysis begins with the identification of basic blocks (i.e., sequences of instructions with no branching) and the instructions that transfer control between basic blocks. This information can be used to construct a Control Flow Graph (CFG). A CFG is a graph of basic blocks where the edges of the CFG represent possible flows of control between basic blocks. In a block-structured program, the CFG can provide a substantial amount of information about data flows within a program. Figure 2 gives simple example of pseudo-code and a corresponding CFG.

static analysis. To remain obfuscated, the actual initialization of the object has to be hidden, and the control flow that results in the creation of the object must be hidden.

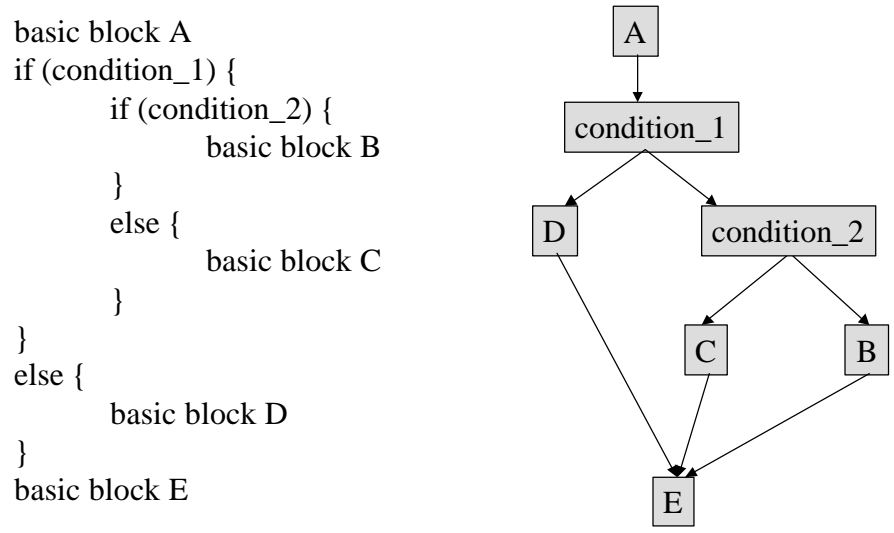


Figure 2

As illustrated by Figure 2, an enumeration of the possible paths through a CFG can be used to determine the relative order of operations, which can be used to derive dataflow information through program slicing [Weiser84] and other dataflow techniques.

For obfuscation purposes, it is important to make the CFG as difficult as possible to obtain. We consider two broad classes of analysis attacks on the control flow of an obfuscated program:

Derivation of edges between basic blocks. An attacker can scan the obfuscated program for the instructions that transfer control between basic blocks. This kind of scanning can typically be performed statically (i.e., without running the program). Additionally, an attacker can profile the execution of the program and observe the orders in which basic blocks are executed. If the attacker is able to use black-box testing (i.e., running the program multiple times on various input values), the attacker can build up a set of edges between the basic blocks.

Pattern matching of basic blocks. In the case where an attacker has access to the original source code of the obfuscated program,⁸ derivation of the CFG may be easier. In particular, the attacker can attempt to match basic blocks from the original program to those of the obfuscated program.

The following sections address these two classes of attacks.

4.1 Obscuring Control Transitions in Java

Unlike the case with some CPU instruction sets, static analysis of JVM instructions is extremely straightforward. A particularly useful aspect of the Java executable format, for

⁸ Note: if the attacker knows the source code, the obfuscation is primarily useful for protecting the current values of state variables.

the attacker, is that instructions are separated from data values (that might otherwise look like instructions), and a static analyzer can therefore always determine what values contribute to the CFG, and what values do not. The absence of pointer arithmetic also simplifies the task of creating a CFG since all the jump targets can be known statically. Finally, semantic information, such as function names and formal parameter lists, are maintained in the Java class file format, which simplifies the analytic task.

Control transition in Java programs is accomplished using any of several mechanisms:

Method invocation and return. Methods are invoked via special JVM instructions (`invokevirtual`, `invokespecial`, `invokestatic`, `invokeinterface`), and returns from methods are accomplished via a family of instructions that each return one of the JVM data types (e.g., `ireturn` returns an `int`, `freturn` returns a `float`). Method invocation instructions refer to a method record held in the constant pool of a class file; this record identifies the method, its class, formal parameters, and special flags (i.e., `static`, `synchronized`). These bytecodes are easily found by scanning a Java class file, and thus method call trees can be easily constructed using static analysis.

Unconditional branching. The JVM includes a `goto` instruction and an instruction for jumping unconditionally to a subroutine (not a method). As with the method instructions, these can be easily found and used to construct a CFG statically.

Conditional branching. The JVM includes a number of instructions that compare the primitive ordinal Java types (`int`, `long`, `float`, `double`) for equality, less-than, etc., and that transfer control to an offset in the instruction array of the current method. These are easily identified for building a CFG.

Block ordering. The ordering of basic blocks within the instruction array represents some of the edges of the CFG (i.e., the edges taken when a conditional branch evaluates to false).

Exception handling. Exceptions can be thrown by the JVM, explicitly by the `athrow` instruction, and implicitly by a number of other JVM instructions when they encounter error conditions. In all cases, these exceptions cause control to transfer to the governing exception handler. Each Java method may define exceptions. Exceptions are represented in the method by a table (interpreted by the JVM) that specifies, for each exception handler, the kind of exception the handler is listening for, a range of instructions in the method's instruction array where the handler is active, and a range of instructions that comprise the exception handler. Exceptions cause control transfers to handler instructions. While the exception structure is somewhat more complex than the previous control transfer mechanisms, it can still be statically analyzed to construct a CFG.

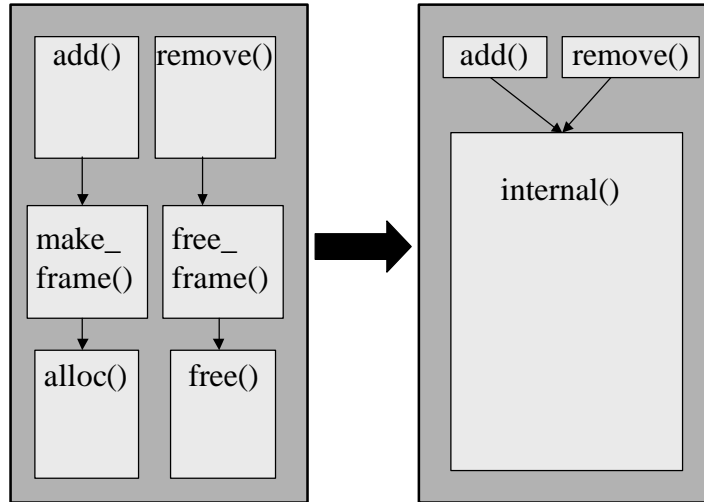


Figure 3

Because Java programs employ highly visible mechanisms, and maintain a considerable amount of symbolic information about them, analysis of control transitions is fairly easy. Our approach to protecting control transition information is to replace visible, high-level mechanisms (such as method calls) with less visible, lower-level mechanisms, and to formulate the lower-level mechanisms for as much indirection and run-time data dependency as is possible without imposing unacceptable performance overheads. The use of indirection and run-time data dependency is not a novel part of this work, and has been investigated in Wang et al. [W00,WDH+01,WHK+00]. Our contribution here is in translating semantically rich and visible JVM structures into low-level structures that are much less revealing, and then applying indirection and run-time data dependency at the lower level. By removing important events that reveal program structure, such as method calls and exceptions, we hope to substantially complicate dynamic analysis. Our approach can be described as five transformations on the program’s original control flow structures:

Merge all internal methods. Our first technique is to merge the internal methods of a class into a single internal method. As discussed in section 3, some methods are externally callable, and they must remain visible to the enclosing environment. Our approach is to translate them into stubs which merely receive the calls and transition to a single internal method. Figure 3 illustrates the effect on the global structure of a Java class. In Figure 3, two methods (`add()` and `remove()`) are externally visible, and the others are internal and are used by `add()` and `remove()`. The transformation replaces `add()` and `remove()` with stub methods which merely pass calls through to an internal method. The other original methods of the class are merged together by our tool to form the method `internal()`.⁹

⁹ The maximum size of a Java method is 64k bytes. Hence, our merge transformation actually generates multiple merged methods as needed. The merged methods, however, do not have a recognizable

Merging methods prevents an attacker from constructing a call graph of the program’s interior. To allow clients of internal to continue to function, we merge the formal parameter lists of all the methods contained in `internal()` and edit the call points so that the extra parameters are provided by callers of `internal()`. Based on the provided parameters, `internal()` switches to the correct code. Most importantly, all the non-stub methods are now encapsulated within `internal()` and calls among them can be represented as local control transfers and without using the visible Java method invocation mechanisms at all. For recursive methods, this implies that the merged method must implement a call stack directly (note: our prototype implements this feature). Stack management code is relatively easy to detect, and therefore obscurity techniques from section 5 are needed to protect the internal stack from reverse engineering. Where methods are not recursive, no stack is required, which results in more obscure control transfers within the unified method.

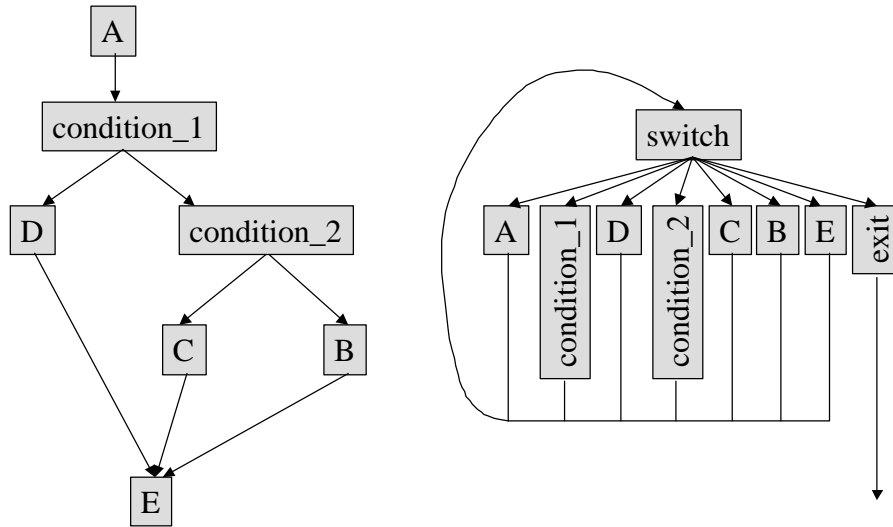


Figure 4

Flatten all branching. The Java branching instructions are highly visible and, if they are preserved, the attacker can use them to statically construct a CFG of the unified method. To avoid this, we use the technique of Wang et al. [W00,WDH+01,WHK+00], which flattens control flow and drives the execution-time basic block ordering using indirectly accessed run-time data. Our approach is slightly different in that we use a single flattened control flow structure for all of a program’s control transitioning (excepting the single `invoke` instruction in each stub method) whereas Wang’s approach creates a separate flattened structure for each function in the program. We have implemented a transform, `switchify`, which performs this conversion on Java bytecodes.

relationship to the original set of methods.

Figure 4 demonstrates the transform, applied to the example from Figure 2. Essentially, the program's control flow is reorganized as a switch statement enclosed in a loop, where the cases of the switch are the basic blocks of the program.

The goal of control flow flattening is to provide a framework for control flow indirection. Indirect control flow makes static analysis impossible and dynamic analysis difficult if done well. A short example shows how flattening makes insertion of control flow indirection easy follows. Consider the simple program:

```
public static void main (String [] args)
{
    for (int i = 0; i < 6; i++)
        System.out.println (i);
}
```

This program, which merely counts from 0 to 5 and prints out the values, compiles to the following Java bytecodes:

```
0    iconst_0    # push 0 on the stack
1    istore_1    # store 0 in local variable 1
2    goto 15
5    getstatic  java/lang/System.out : Ljava/io/PrintStream;
        # push the object ref for System.out
        # onto the stack
8    iload_1     # push local variable 1 onto the stack
9    invokevirtual java/io/PrintStream.println : (I)V
        # call println(i)
12   iinc 1 1     # increment local variable 1
15   iload_1     # push local variable 1 onto the stack
16   bipush 6    # push 6 onto the stack
18   if_icmplt 5 # goto 5 if local variable 1 is < 6
21   return
```

Control flow flattening converts this to the following instructions, which drive control through the `tableswitch` instruction:

```
        iconst_1
switch:
        tableswitch 1:block1 2:block2 3:block3 4:block4

default:
        return
block1:
        iconst_0
        istore 6
#        iconst_3
# insert control flow indirection here
# (code that calculates 3)
        goto switch
block2:
        getstatic java/lang/System.out
        iload 6
        invokevirtual java/io/PrintStream.println :(I)V
        iinc 1 6
        iconst_3
```

```

                                goto switch
block3:
                                iload 6
                                bipush 6
                                if_icmplt jump
#                                iconst_3
# insert control flow indirection here
# (calculate 3)
                                goto switch
                                jump:
#                                iconst_2
# insert control flow indirection here
# (calculate 2)
                                goto switch
block4:
#                                iconst_1
# insert control flow indirection here
# (calculate 1)
                                goto switch

```

The switch statement allows any block to be selected with 'goto switch'. Before each jump to the switch instruction, an opaque calculation produces the next block value. `System.out.println` is left as is since it is an external object.

Use indirection to control branching. In the example above, all control flows drive through a single switching structure, but constant values are pushed on the stack and then consumed by the `tableswitch` instruction as it decides which basic block should execute next. Although there is only a single conditional branching instruction in the program, the data fed to it in the form of constants would be easy to analyze. By scanning for constant values pushed on the stack prior to the `goto` instructions that jump to the `tableswitch` instruction, an attacker could create the CFG. To address this vulnerability, we borrow techniques from section 7 to obscure the values that feed to the `tableswitch` instruction. Essentially, the techniques in section 7 encode constant values in a variety of quick-lookup data structures (for example, short traversals through runtime graphs) that execute quickly at runtime but are resistant to static analysis.

Order blocks arbitrarily. Some edges in the CFG are implicitly specified by block ordering. Once the code has been switchified, however, implicit edges are lost and cannot be used by an attacker. Because of this, there is no cost to arbitrarily ordering the basic blocks managed by the `tableswitch` instruction, thus further obfuscating the edges of the CFG. This technique is also suggested in [W00].

Fold exception handling into normal computational structures. Many exceptions are thrown explicitly using the `athrow` instruction. These exceptions can be re-implemented by replacing `athrow` instructions with appropriate `goto` instructions to the `tableswitch` instruction, and implementing the exception-handling code using normal basic blocks encompassed by the flattened control structure. For exceptions thrown as side effects of JVM instructions, or by the JVM itself, it is necessary to encompass the entire program in a single generic exception handler

that catches all such exceptions and then transfers control back to the `tableswitch` instruction.

We have implemented method merging and the basic flat control flow approach, along with support for simple exceptions. Our experience so far is that these techniques can be implemented using generic transforms. We have applied these transforms to a significant part of the JBET tool itself. We have observed, however, a significant performance penalty for these techniques. We have not yet conducted a full suite of performance tests, but our preliminary experience with these techniques indicates that obfuscated programs may execute at less than 1/10'th speed as compared with non-obfuscated programs. Optimizations may be possible, however, along with approaches that tune the obfuscation techniques to minimize their impact on performance critical sections of code.

4.2 Obscuring Basic Blocks in Java

Pattern matching basic blocks or execution counts of basic blocks with the original program could allow an attacker to link obfuscated blocks with blocks from the original program. Similarly, pattern matching of basic block frequencies could allow an attacker to statistically analyze the obfuscated program, thus finding out, for example, where important functions are performed in the program. Additionally, basic blocks that implement support features for obfuscation, such as implementation of function call stacks, are likely to be frequent events during execution and could help the attacker. We address these possibilities using five central strategies:

Split original basic blocks. A basic block can be arbitrarily split into multiple basic blocks that happen to run in sequence. This can complicate pattern matching attacks based on knowledge of the original code.

Change temporary variable processing. A basic block's instructions often embody numerous arbitrary decisions, such as which slots in the local variable table are used for temporary values, the number of temporary variables at use at one time, etc. Some of these decisions can be remade during code obfuscation, thus resulting in basic blocks that perform the same functions, but with different instruction signatures.

Add spurious instructions into basic blocks. Additional instructions that perform no actual purpose can be added, to further obscure individual basic blocks. This is also suggested in [W00].

Duplicate differently specialized basic blocks. A single basic block can be transformed into a number of functionally identical, but different basic blocks by varying temporary variable processing and fluff instructions. Control transfers through the `tableswitch` instruction arbitrarily select among these. We believe that this technique can be used to complicate, or possibly even prevent, statistical analysis of the runtime behavior of an obfuscated program. This is also suggested in [W00].

Interleave basic blocks with obfuscation support functions. An obfuscated program, particularly a mobile agent, may have auxiliary functions to perform. These

functions could be interleaved as their own basic blocks within the program, but by adding them to some of the basic blocks of the original program, we anticipate that we can further obscure the identities of the original basic blocks while accomplishing additional work.

Introduce unnecessary threading. Introducing parallelism into the obfuscated program can increase resistance to static analysis. Independent code regions could be discovered, and unsynchronized threads could run that code. Dependent regions could be split and executed in separate threads also, but synchronization would be necessary [CTL97a].

Use of non-general protocols (for general method calls). Any general method call implementation will have a detectable signature, because of the number of data structures that are involved with method calls in a typed language: virtual tables, register stacks, parameter passing, and control transferred to similar targets (In Java, the operand stack is empty at the start of a method). In addition, the sets of data accessed by different methods (or recursive invocations), will be different as the called method does not access the caller's data. Therefore it is helpful to include spurious method calls and returns in the obfuscated program and to inline and outline methods when possible.

If the call graph of the original program does not have many cycles, different call and return protocols can be used, making pattern matching more difficult. A method can have a non-general call protocol (one that cannot support many method call features, such as distinct stacks or method recursion) if it can never occur twice on the call stack. Many options for non-general call protocols are available; parameters could be accessed directly, the same storage area for temporary locals could be used, a call stack data structure is not needed. Exception handling also must be determined at compile-time for a non-general call protocol. The greatest obscurity advantage comes from using as many different and non-general method call protocols as possible, limiting the usefulness of searching for data access patterns in the program.

A simple example of a non-general method call protocol that would work for any method called from a fixed number of places, and cannot call itself, is simply to allocate the space for the called method's registers in the caller's stack frame, and code the method call and return as ordinary jumps. The jump for return could use a return address stored by the caller.

In summary, control flow obfuscation can be divided into two basic areas: obfuscation of transitions between basic blocks, and transforms that hide the basic blocks themselves. We will implement obfuscation of control transitions by using indirection for all transitions. However, doing this alone is not enough; the connection between basic blocks in the original program and in the obfuscated program must also be obscured, to prevent an attacker from collecting statistics on execution. This combination of techniques should make tracing program execution substantially more difficult.

5 Heap-Allocated Storage and the Type System

In addition to increasing an attacker’s cost for analyzing control flow of an obfuscated program it is also necessary to increase the cost of analyzing the state variables (i.e., allocated ranges of memory) of an obfuscated program. This bears directly on our second obfuscation property presented in section 1:

2. the relationships between OP ’s state and OP ’s behavior is obscure to any observer who has not seen p .

If an attacker (i.e., an observer) can observe and *understand* state variables in a program, the attacker may be able to derive a considerable amount of knowledge about the program’s behavior. Consider, for example, a program that searches for the lowest price for a good or service by comparing the offering price of each vendor encountered against a set of stored prior offering prices from other vendors. The prior offers must be stored in the state variables of the program. If an attacker can determine which memory locations hold the prior offers and how the offers are represented in those memory locations, the attacker can make an offer that is just barely the “winning” offer. Even more seriously, the attacker can modify the prior offers to fool the program into accepting an offer that is too high. More abstractly, if the attacker can understand the state variables of the program, the attacker can gain an advantage in understanding all of the data-driven behavior of the program.

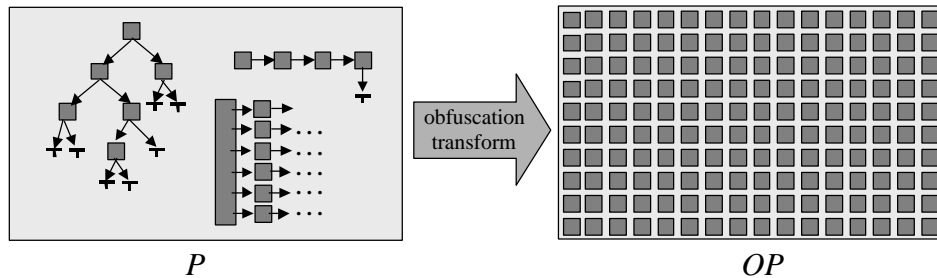


Figure 5

Our goal is to, through an obfuscation transform, impose substantial costs for performing such analysis. Figure 5 illustrates an ideal obfuscation transform. The un-obfuscated program, P , on the left has its state organized using several conventional techniques (trees, lists, tables). Analysis of such structures is fairly easy since an attacker can watch access patterns at runtime and can follow pointer relationships between objects. Our goal is to achieve the state depicted in the figure for OP , in which:

- the state variables appear to be all the same (i.e., they cannot be easily grouped by type or function),
- there is no simple mapping between state variables of P and state variables of OP ,
- although there must be values encoded in the state variables and relationships among them, the value encodings are not recognizable, and the relationships among the variables are similarly not apparent, and

- to an attacker who watches access patterns at runtime and attempts to infer relationships and interpretations of state variable, their access patterns are apparently chaotic and unrelated.

The basic challenge in obfuscating state variables is that usage patterns for state variables are often highly repetitive and obvious, and standard compiler-generated state variable implementation mechanisms produce extremely regular data structures (e.g., stacks, chains). Such regularity gives an attacker easy access points at which to begin analysis. Different runtime environments (e.g., the JVM, libc, Perl) employ different implementation mechanisms and hence have different vulnerabilities to analysis and attack. Our focus is on Java and its vulnerabilities. Unfortunately, because of the many controls that Java imposes on memory access, Java presents a number of challenges for state variable obfuscation:

Memory allocation and garbage collection. In Java, memory is allocated both on the stack and via the heap. For heap-allocated memory, the JVM manages allocated memory, a process known as garbage collection (GC). While this is helpful to programmers, it provides a large amount of information to an attacker. Furthermore, an attacker can monitor, when allocation occurs, which objects reference which other objects, and gain significant information by analyzing the graph of allocated objects at runtime.

Memory type management. In Java, all memory locations are typed, which means that each memory location has a standard interpretation (`boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, or `reference` (to a class, interface, or array)). The JVM instructions indicate the types of memory locations (e.g., `iconst` pushes an `int` constant on the stack, and `lconst` pushes a `long` constant on the stack). A casual examination of the instructions reveals the types of state variables, thus providing valuable information to an attacker.

Pointer restrictions. In Java, programs are written using references, which act like pointers but are restricted. A key aspect of references is that they are strongly typed: a reference indicates the type of object it refers to (or null), and the runtime system prevents the reference from being assigned to any other kind of object. These restrictions prevent a variety of pointer-based obfuscation techniques that are based on construction of invalid pointers or arithmetic manipulation of pointers. References can be trusted by an attacker to reliably indicate the types of objects, and usage patterns of pointers can help an attacker analyze a program.

Class types. Every object used in a Java program has an associated `.class` file, which documents the object's methods and local variables. Thus, information about the data structures used by a Java program are easily available to an attacker, and can be used to analyze a program.

Inheritance and type casting. Java programs frequently cast a reference up or down a class hierarchy. These casts are allowed by the type system. Obfuscating transformations to a Java program that alter supporting data structures must transform casting operations into related casting operations in the new data structures, and must ensure that the new casting operations maintain a program's original behavior.

Virtual method support. Java supports virtual methods, which imply the existence of a per-object virtual method table – this is needed to route functions calls on an object reference, at runtime, to the method that should be used given the object’s actual type. Obfuscating transforms that re-organize data structures, must ensure that calls on a new object have identical behavior to calls on an **original object** (see glossary). This requires the existence of **runtime type information** (see glossary), which must be generated by the obfuscation transform tool.

These Java characteristics present us with two basic approaches for implementing memory obfuscation. The **first approach** is to translate the classes of a program into different, but equivalent, classes that are harder to understand but that operate within the normal Java runtime environment. This approach is constrained by the Java-enforced memory structure limitations described above, and therefore vulnerable to dynamic analysis based on them. The **second approach** is to implement, in the obfuscated program, an unstructured model of memory (e.g., a simple Java array of bytes), and represent Java objects as regions of that space. Within the array, we are free to represent variables as we please, without regard to Java’s memory structuring conventions (i.e., Java permits the elements of Java byte array to be arbitrarily referenced and modified, so objects represented within such an array can have arbitrary structure).

Either approach has advantages and weaknesses, and is non-trivial. Class translation is relatively easier to implement and should have greater performance, but it is impossible, using only class translation, to manipulate pointers or cast regular data to a pointer, and any spurious pointers added can cause memory leaks. Unstructured memory is less vulnerable to some kinds of dynamic analysis since its use of easily-monitored JVM features (i.e., the garbage collector) is more indirect, but the additional code added to implement unstructured memory implicitly “knows” much of the information that we are trying to hide, and could be a single point of obfuscation failure.

Fortunately, no choice is required; both approaches can be used together. Our strategy is to implement a two-phase process where Phase 1 implements class translation, and results from Phase 1 are fed into Phase 2, which implements unstructured memory. By implementing both approaches, we believe we can highly decouple the original memory structures from obfuscated memory structures, resulting in a substantial work factor for attackers.

5.1 Phase 1: Class Translation

In order to prevent the attacker from learning how data is stored, we need to extensively limit his ability to learn about memory, pointers, and types. It is highly beneficial to rearrange the types of the input program. This transformation should be applied before any other obfuscation. In this way, we can force the attacker to try to match three programs instead of two: the original program, the re-arranged program, and the final output.

Our basic strategy for class translation is to represent objects and pointers in as many ways as possible. Use of multiple strategies in one program removes common patterns and makes analysis harder. Ideally, every field access, polymorphic method dispatch, or

type cast should look different from all the others. Implementation of this diversity is nontrivial, because the transformation must also maintain consistency with the original program. For example, if we represent two different classes *A* and *B* differently, then we need to handle the case where a pointer to an *A* is cast to a pointer to a `java.lang.Object`, and then cast back into an *A*. The problem becomes even more complicated if the user performs a type query on a pointer to a `java.lang.Object`.

The Java dynamic memory allocator and type system retain most of the structural and semantic information from the source code. Our goal in obfuscation here is restructure the program so that the semantic information contained by the allocator and type system is meaningless. A number of techniques can be applied at this level:

Rearrange classes. One useful rearrangement is false refactoring: if there are two distinct classes *A* and *B* in the original program, we create a superclass *AB* of both. We can then implement some (arbitrarily chosen) methods of *A* or *B* in the superclass, but using common data storage in the superclass so that use of a method from *A* on an object created as a *B* will corrupt the object. Alternately, the methods in the superclass could be incorrect versions of methods from *A* and *B*.

Another rearrangement is to take some of the fields of *A* (a class from the original program) and put them in a new class of their own, called *B*. Then give *A* a pointer to *B* and give *B* some of the methods from *A*. Other instances of *B* (not associated with any *A*) can be used for spurious data. Many similar rearrangements are possible. If there are *n* fields in the classes of a program, there are 2^n groupings of the fields, even if no duplication is allowed.

Disrupt one-to-one mappings. In order to avoid revealing the extent of **user objects** (see glossary), we must make sure that a single user object does not correspond to a single **synthetic object** (see glossary), in other words, all user types must be refactored. User objects of different types should be represented using the same synthetic type, and objects of the same type should be split into several different objects that are linked together. One can take this to the extreme and represent all objects using a single generic **chunk** (see glossary) type. Clearly using one chunk type preserves less information than using several.

Parameterize field accesses. A primary weakness of class translation is that we cannot manipulate pointers in arbitrary ways as can be done in other environments, such as C. In this approach, we essentially keep several different representations of pointers to the same user type, where the different representations of pointers point to different chunks within the representation of the user object. The output code can then switch the representations of its pointers at will in order to vary and simplify field access code. It should be noted that the representation for a **user pointer** (see glossary) (as well as the representation for any transient variable) needs to be statically determined. Thus we are not quite free to change representations anywhere we want, but must do some analysis to determine where it is safe.

Figure 6 illustrates both parameterized field access and how one-to-one mappings can be disrupted. In the figure, an original class, `user` is represented as several

instances of a synthetic class `syn`. R_1 , R_2 , and R_3 are representations of a pointer to a `user`. Any of these representations can access any field of `user`, but they each have a different procedure for doing so. The obfuscator would statically determine which representation to use for any particular variable that is a pointer to a `user`. Furthermore, the obfuscator can easily switch a pointer variable from one representation to another. This operation looks like following a next pointer in a linked list, which should be common enough to be quite obscure.

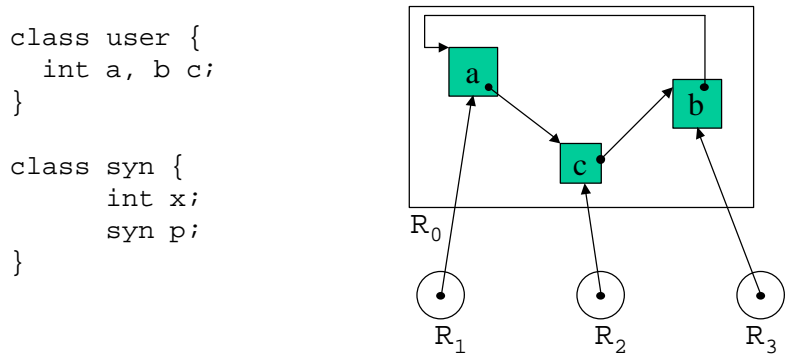


Figure 6

Unfortunately, each time we add a new representation for a particular `user` type, the code to access the fields of that type becomes more complicated (it must first determine which representation is being used and then take a different branch for each representation). This code must be repeated in the program, so the attacker can search for it. This problem can, however, be alleviated by identifying blocks that use particular `user` fields and keeping a pointer to the specific chunk that contains that field for the entire block.

Apply control flow techniques for field access. Another technique to hide field access code is to apply basic block and control flow obfuscation to it. In addition, the code for a particular field access can often be inserted significantly before it actually occurs in the original program (or after, in the case of a write). Mixing field access code with the program's original control flow should go a long way for making it less obvious.

Insert spurious references. The attacker's problem can be made substantially harder by the insertion of new references that add additional linkages among the objects of a program. These spurious references can dramatically increase the apparent complexity of the program's data structures, but they must be managed, and they must also be traversed appropriately to prevent statistical techniques from flagging them as spurious. Management of spurious references can be very difficult, especially in the presence of garbage collection, in which it is critical to eventually remove spurious references to allow legitimate objects to be garbage collected. The following techniques can be used to ensure that legitimate objects are eventually collected. One is to copy over spurious pointers periodically so that, eventually, all the spurious pointers to chunks without real pointers to them will disappear, and thus get garbage collected. Unfortunately this has an obvious

statistical signature. Another approach is to switch the roles of fields from spurious to real every so often (toggling a flag so other code knows about it) to confuse analysis. Levels could be assigned to spurious pointers and spurious pointers could be copied only if the target pointers have lower levels. Pre-allocation of spurious data, serving to anchor spurious pointers, could further complicate analysis. The problem, of course, is in keeping tabs on how much such data has been allocated so that it remains bounded. Additional approaches can further complicate analysis for the attacker.

It should be noted that these techniques for adding spurious pointers are quite tricky to implement correctly and obscurely. Spurious pointers are necessary, however, because without them the attacker will know exactly where the user pointers are. Spurious references are somewhat vulnerable to statistical attack, however, when used on conjunction with other techniques presented here, we believe they will substantially add to an attacker's work factor.

5.2 Phase 2: Unstructured Memory

The second phase of our memory obfuscation is to implement a simple array of bytes, and to generate JVM instructions that access the array. Since the elements of an array in Java can be arbitrarily read and written, and since one entry can serve as an index to another, an array is a suitable abstraction of physical memory. In Phase 2, we implement state variables as regions within an array of bytes, and perform all management of the storage ourselves without using the semantically revealing JVM allocation and garbage collection features.

By directly implementing memory management features, we are able to do much more with pointers than we would be able to otherwise. In such a scheme, pointers are indices, and elements of the array can be either pointers or data, and may change roles as the program runs. We also have more freedom to work with spurious pointers. Although we cannot obscure the fact that a dereference of an array element has occurred, we are free to do pointer arithmetic, and generally manipulate pointers to maximize obscurity. In Phase 2, we focus on obscurely supporting four key Java features using our implementation of unstructured memory:

Allocation Hiding. It is important that the attacker can determine as little as possible about allocation, especially what is being allocated. Our use of a pre-allocated array protects us from analysis based on watching the JVM's memory allocation mechanism, but we must still manage allocation of elements within our array. Several techniques will help to obscure array allocation operations. First, we will randomly modify (within a bound) the amounts of array memory allocated, so that exact allocation sizes cannot be used to understand the objects being created. Secondly, we will allocate memory in equal sized chunks and build up user objects from them. Thirdly, we will pre-allocate numbers of objects and keep them in an allocated state so that array allocation operations do not correlate in any simple way with the current state of the program's algorithm.

Smart Garbage Collection. Unlike the case in Phase 1, the garbage collection code that we generate for Phase 2 can know which pointers are real and which are spurious, and perform proper sweeping based only on the real ones.

Because allocation of a single chunk is so simple and frequently used, and because chunks do not correspond to any particular element of the original program, we will not attempt to make the de-allocation of individual chunks obscure. Instead we will concentrate on obscuring the management of user objects. We will associate with each chunk an in-use flag and periodically de-allocate all chunks that are not in use (this is the sweep phase of a standard mark and sweep garbage collector). We will rely on the obscurity of the code that sets these flags to keep de-allocation of user objects opaque (this is the “mark” phase of a garbage collector).

Marking can be implemented as a series of virtual methods that hit the in-use flag and recurse. We can make the marking covert by frequently referencing the in-use flag when a chunk is accessed, then the actual marking procedure will look much more like normal code. Furthermore, marking need not occur all at once. It can be spread out among other execution. Mixing the marking pass with normal execution will cause memory leaks, but they can be made non-deterministic, and therefore a chunk that was missed in one GC pass will likely be collected in the next pass.

Complex Pointers and Type Queries. A general problem that we encounter in obfuscation is to maintain consistency while at the same time producing the most varied and diverse output that is possible. For instance, we would like every function call in the output program to look different, but if two pieces of code call the same function with a different call protocol then clearly the program will not be correct. This problem is especially pernicious concerning our representations of user objects. Clearly (except for some special cases) the entire program must agree on a representation for each type, because otherwise some code would be accessing the fields of objects in a way that is not compatible with other code in the program. Furthermore, we must somehow deal with the representations of polymorphic base classes. For instance, if A and B are children of C, and we have a pointer to a C, then we must be able to access C’s fields in a consistent manner, even if our pointer really points to an A. Moreover, we need to be able to support type queries on a pointer to C (`instanceof` and `java.lang.Object.getClass`). The way normal programming languages deal with this problem is to prefix an instance of a subclass object with an instance of its base class contiguously in memory. Additionally, conventional languages implement a link from objects to their class structures so that type queries can be easily answered by traversing the link. From an obfuscation point of view, this traversal and layout is very undesirable since it reveals object structures and also global runtime type information to even casual inspection.

To address these issues, we use **complex pointers** (see glossary); instead of simply storing a **simple pointer** (see glossary) to an object, we store a simple pointer along with some information about the object. This information could include the type of the object, or the address of a virtual method table, or even the

address of a particular virtual method. This allows us to perform type queries using only local data, or by calling a method.

Complex pointers give us considerably more flexibility than we could otherwise hope for. If we only use a simple pointer to represent a pointer to a `java.lang.Object`, then we are forced to give every object memory representation in the program a common prefix, which is quite undesirable. With complex pointers we are able to represent different class hierarchies in totally different ways.

Complex pointers can also allow us to vary field access and type query code even more, by storing hints about an object in the pointer. If the type of an object is stored in the pointers to it, then no dereference is needed to support an `instanceof` call. We can also put the **representation parameters** (see glossary) in the pointers, so the dereference occurs after the if-then that decides which protocol to use to access a field. If we store the type information for objects of type `C` in the pointers as well as the objects, then we have two different ways of type-querying a `C`-pointer.

For example, let `C` be a class as above and let `p` be a pointer to `C`. Let `p.foo` denote a value that is stored locally as part of the representation of `p`, and let `p->foo` as a value that is stored in what `p` points to. Let `x` be a local variable and let `y` be a field of `C`. Here are a few different representations for `x = p->y`:

- `x = p->y` in this case A, B and C all store `y` in the same place
- `if (p->(I am an A))`
 `x = p->y` where `p` is interpreted as a pointer to an A
 `else`
 `x = p->y` where `p` is interpreted as a pointer to an B
- `x = (p.m)()` `m` is a locally-held pointer to a function that retrieves `y`
- `if (p.(I am an A))`
 `x = p->y` where `p` is interpreted as a pointer to an A
 `else`
 `x = p->x` where `p` is interpreted as a pointer to an B

The number of permutations along these lines is quite extensive. We believe that these techniques, combined with basic-block and control flow obfuscation are able to substantially disguise type-queries and field access.

Virtual method calling. Virtual methods present a challenge for obfuscation because they must reference runtime type information. Typically, such information is represented globally within a program, and references to it both reveal the specific types of objects and global (to the program) type information. Our basic approach

to this issue is to localize the type information within a program so that decisions about which virtual methods to call can be made locally, without revealing global structures. Additionally, we seek to make the local determinations themselves as obscure as possible. In some cases we will be able to statically determine the type of an object at obfuscation time. If this is true then no runtime dispatch is needed, we can simply jump to the proper method, or perform computations that look like a dispatch but are not. Clearly this is the ideal situation, from an obfuscation point of view.

Unfortunately, it is not always possible to statically determine the type of an object. Fortunately, there are usually only a couple of possibilities for the type of an object, and hence for the identity of the method to call. When the method is dynamically determined, we can examine the type of the object and use normal branching to select the method, and the normal branching can employ the control flow obfuscation techniques from section 4. The dispatch code can either be placed at the point where the virtual method is called, or it can replace the base class method itself, which can then be called statically (this elaborates on techniques in section 4).

In the presence of numerous derived classes, this technique may generate excessive dispatching code, in which case a reversion to normal code pointers may be the best tradeoff between security and space overheads. These code pointers could be stored individually in each object (as if they were a normal field) or in the complex pointers to those objects, or they could be stored somewhere else entirely, so long as there is some way of locating them from either the complex pointers or the instances of our base class in the heap. Even so, any use of a code-pointer is immediately detectable as such, so their use should be avoided if at all possible. In addition we should make use of code-pointers for other wholly unrelated things such as control flow obfuscation in order to help mask their use in method calls.

If the user derives a subclass from a base class that is outside the realm of our obfuscation then we will need to apply special handling. Instead of keeping pointers to the outside base class, we will keep a pointer to an internal, obfuscated stub that has the same methods as the external base class. When we call a method on the stub it will then dispatch the call to either an external method or an obfuscated method. In this way we are able to generate an **exit point** (see glossary) only if it is really needed. We do not need an exit point in order for obfuscated code to call other obfuscated code, even if that call is through an external interface. `java.io.PrintStream` is one example of a class that is likely to be used in this manner: it is a system class, but the user may create derived classes and use those.

Memory obfuscation is generally a hard problem. We have presented a number of techniques, however, that can very significantly rearrange a program's original data structures while preserving their behavior (except for increases in execution time and memory usage). Particularly in combination, these techniques are promising. A particular strength of many of these techniques is that they admit of parameterization, or

coin tossing, which can be used to automatically generate substantial amounts of binary diversity.

6 Parameterized Obfuscation of Long Term Variables

Many programs of interest will have some bits of data that they wish to protect from the attacker. A mobile commerce agent, for example, might have a minimum price. Virus-detection software may wish to protect the virus-fingerprints. Protected programs will generally have important bits of data that they use in their regular execution that they need to protect.

This section addresses the design and effectiveness of obfuscation transforms for primitive variables, especially integers. For reasons expressed below, we prefer to use obfuscation schemes where randomly chosen parameters influence how the variable is obfuscated. These obfuscation techniques will be evaluated against an adversary that uses both static and dynamic analysis techniques. In particular we wish to determine if any obfuscation techniques provide distinctive de-obfuscation resistance against dynamic analysis. Ideally, these techniques should also be “adjustable” so that the obfuscation policy may be used to increase or decrease the level of obfuscation.

6.1 The Problem

The general goals of parameterized obfuscation transforms are:

- 1) Produce obfuscation techniques that are hard for an adversary to reverse engineer quickly, even with the help of both static and dynamic analysis.
- 2) Use parameterization so that an adversary who defeats a given scheme has as little advantage as possible in defeating the same scheme on a different variable. To a large extent, the details of an obfuscation should be dependant on some random data which needs to be retrieved to undo the obfuscation.
- 3) Support arithmetic operations (such as addition, multiplication, logical operations, or bit-wise operations) directly on the obfuscated version of the variable. We prefer to perform as many operations as practical in obfuscated space. Some schemes will allow operations on the obfuscated variable that are comparable to operations on the unobfuscated variable, at least for important ranges. These operations are may only be correct over sets of values that are important to the program and may not be correct when applies to the entire range of values available.
- 4) Hide the required parameter p .¹⁰ The adversary is assumed to know the transforms used by the tool, so the process of determining which transform is in use, and the parameter it uses, need to be difficult.

¹⁰ In this section we will use the following notation. The process of obfuscating a parameterized variable can be represented as three tuple (A, p, e) where e is an un-obfuscated value/variable, A is the obfuscation transform algorithm type and p is a runtime parameter that modifies the behavior the transform in some way. A may also be a parameter since how a variable is (de)obfuscated may be determined at runtime

- 5) **Obscure relationship between variables.** The attacker should not be able to assume that blocks with similar obfuscation schemes refer to the same, or related variables.

6.1.1 Variable Use

How a variable of particular type is used is a significant factor in the suitability of an obfuscation transform. A variable that is only used as a reference value needs to support a smaller number of operations than a variable that is used as an accumulator. A variable that is not used at all, and only transmitted to another system, requires support for even fewer operations.

If a program does not use a variable except to transmit the obfuscated variable to another system, then the obfuscation transform has essentially the same requirements as an encryption operation such as AES or RSA. In such a rare situation, the most difficult problems may be protecting the key, and preventing dynamic analysis from seeing sensitive data in the clear. The types of program secrets that we focus on in this section are contained in variables that are actually used by a program, e.g., operands in arithmetic expressions.

We now present a number of examples of how obfuscated variables might be used. For the purposes of evaluating candidate obfuscation techniques, we will consider the use of parameterized obfuscated variables (e.g., `secret_int`, `secret_total`) in the following code fragments. We assumed that the adversary has identified a code fragment as interesting and is capable of performing some dynamic analysis on at least the fragment during the period of time the obfuscated program is under attack and still have some time to make use for the result of the analysis.

For integer variables:

1. **Simple counter bound** (with optional use of the counter within the loop).

```
for (int i = 0; i < secret_int; ++i) {  
    bar(foo(a),b);  
}
```

If `secret_int` is non-negative then the loop test can be replaced by `i != secret_int` which typically allows for easier obfuscated evaluation. Often the value of the loop counter is used within the loop, i.e., replace `bar(foo(a),b)` with `bar(foo(i),b)`.

If the counter is obfuscated so that it can be compared with `secret_int`, an adversary who has access¹¹ to an obfuscated version of this code fragment *may* see the following:

- The obfuscation of zero in a manner compatible with the obfuscation of `secret_int`.
- The obfuscated `++i` operation in a manner compatible with the obfuscation of `secret_int`.

¹¹ The use of the term *access* in this section means that the adversary has identified the fragments as potentially interesting and can perform both static and dynamic analysis on the fragment.

- The test for equality. In many schemes, this may be simply a bit equality test.
- If the body of the loop contains `bar(foo(i),b)`, then the attacker may see de-obfuscation compatible with the obfuscation of `secret_int`.
- If the body of the loop contains `bar(foo(i),b)` and the de-obfuscation occurs in `foo()`, then the attacker may see obfuscation technique support for all of the operations needed by `foo()`.

2. Secret minimum integer.

```
if ((foo = bar()) < secret_int ) {
    secret_int = foo;
}
```

Unlike example 1, here we cannot replace the `<` operator with equal or not equal operators.

If the counter is obfuscated so that it can be compared with `secret_int`, an adversary who has access to an obfuscated version of this code fragment *may* see the following:

- The obfuscation of `foo` a manner compatible with `secret_int` or the some process inside `bar()` that results in a compatibly obfuscated value. Care needs to be taken, as in example 1, that the obfuscation techniques in `bar()` and in this code do not weaken the control flow obfuscation.
- The obfuscated relational test `<` of `foo` and `secret_int` or the de-obfuscation of both and the use of the normal relational operations

3. Bounded accumulator (where the values of `foo()` may be quite large).

```
secret_total = 0; /* run time or compile time */
while (secret_total < secret_int) {
    secret_total = secret_total + foo();
}
```

Similarly to example 2, if `secret_total` is obfuscated so that it can be compared with `secret_int` and added to `foo()`, an adversary who has access to an obfuscated version of this code fragment *may* see the following:

- The obfuscated relational test `<` of `secret_total` and `secret_int`.
- The obfuscated arithmetic `+` of `foo()` and `secret_total` or the de-obfuscation of both and the use of the normal addition followed by re-obfuscation.
- Either the obfuscation of the return value of `foo()` in a manner compatible with `secret_total`, either before return or after. Care should be taken that, as in example 1, the use of obfuscation techniques do not weaken the control flow obfuscation.

4. Simple arithmetic using secret integers.

```
secret_int_A = (secret_int_B + secret_int_C)* secret_int_A;
```

If the variables are obfuscated so that both the + and * operations have equivalents in obfuscated form, then an adversary who has access to this code will see the following:

- The obfuscated arithmetic + and *.

The use of compatible obfuscations results in transitive relationships (except where broken by de-obfuscation or obfuscation conversion) between the obfuscated variables. We refer to these transitive relationships between variables as “operational transience.” These relationships may severely limit the obfuscation diversity of the program.

5. **Complex calculations using secret integers** (usually this is done using multi-precision integers mod n).

In this example both the exponent and the message need protection.

```
secret_total = 1;
secret_partial = encrypted_message;
while (secret_exponent_temp != zero ) {
    if (lsb(secret_exponent_temp) == 1 )
        secret_total = secret_total * secret_partial;
    secret_exponent_temp == secret_exponent_temp >> 1;
    secret_partial = secret_partial * secret_partial;
}
## secret_total holds the decrypted (but obfuscated) message
```

An adversary who has access to this code will see the de-obfuscation of these values unless the obfuscation techniques support the equivalent of * on secret_total and secret_partial. The obfuscation of secret_exponent_temp can use a different obfuscation:

- The obfuscation of one (or at least the identification of an obfuscation of one) in a manner compatible with the obfuscation of secret_int.
- The obfuscated arithmetic operation * on secret_total and secret_partial.
- The obfuscated bit shift operation * (or obfuscated division by 2 / multiplication by the multiplicative inverse of 2) on secret_exponent_temp and the obfuscated equivalent of testing the least significant bit of the un-obfuscated value of secret_exponent_temp.

6. **Boolean variables and operations**

```
if (foo AND bar ) {
    total = total + 1;
}
```

An adversary who has access to this code will either see the de-obfuscation of these values or the use of obfuscated techniques equivalent to logical and.

For boolean variables it is particularly important that multiple obscured equivalent values for each state exist. Limiting the obfuscated version of true or false to one of two values gives little obfuscated strength.

7. For variables of any type

- Either the obfuscated variable is obscured during the compile-type obfuscation process (which is unobservable to the adversary) or it happens at runtime. In the following simple example, the un-obfuscated `sensitive_info` has just been obtained by the program from its environment. `sensitive_info` is then obfuscated in such a manner that will enable the result, `secret_integer`, to be used as needed elsewhere in the program.

```
# obfus_tran_A() is implemented by inline bytecodes
secret_integer = obfus_tran_A(param_foo, sensitive_info);
# overwrite
sensitive_info = <unused data>;
```

An adversary who has access to this code may be able to examine `obfus_tran_a(p, arg)` and be able to determine p , therefore being able to de-obfuscate any variable obfuscated in a compatible manner. The adversary may be able to perform correct arithmetic, comparison, etc. operations on the obfuscated variable or related variables guided by knowledge of the algorithms used by the obfuscating tool and/or the application.

- The obfuscated program may choose to de-obfuscate a value and even disclose that value to the external environment, because certain special conditions have been met.

```
# deobfus_tran_A() is implemented by inline bytecodes
sensitive_info = deobfus_tran_A(param_foo, secret_integer);
```

An adversary who has access to this code may be able to deobfuscate this and other values of variables obfuscated in a compatible manner. The adversary may be able to perform correct arithmetic, comparison, etc. operations on the obfuscated variable or related variables guided by knowledge of the algorithms used by the obfuscating tool and/or the application.

Parameterized variables should be resistant to both static and dynamic analysis. We have presented several examples of the observation that operations on related integers are often clustered in the execution sequence of the program. If dynamic analysis can be performed on such a cluster in a sufficiently timely fashion, then the adversary can employ dynamic analysis attacks on either the weakest operation or on the combination of operations by exploiting subtle interrelationships between the available operations.

6.2 Success Criteria

Generally the un-obfuscated state of an obfuscated variable should be the same as the equivalent variable at the equivalent point in the execution of the un-obfuscated

program.¹² That is, obfuscation of variables is not allowed to break the input program. However, this is only relevant to legal input. In the case of “unreasonable” or illegal input to the program, the behavior may diverge in subtle or not so subtle ways from the un-obfuscated version.

Obfuscation solutions have fairly liberal space and CPU performance constraints, since the level of obfuscation for a variable may be controlled by the obfuscation policy, and since some of the variables may be accessed relatively infrequently. Applying more expensive transforms should provide better levels of obfuscation.

We now discuss techniques to generate obfuscation transformation parameters, and then describe the obfuscation techniques themselves.

6.2.1 Obfuscation Parameter Generation Techniques

The parameter related obfuscation evaluation criteria are the following:

- The adversary should have to understand as much about the operation of the obfuscated program as a whole as possible order to determine a variable’s obfuscation parameters.
- The assignment of obfuscation parameters to variables should not assist the adversary in understanding the obfuscated program. For example, access to the same obfuscation parameter by the two different fragments of the obfuscated code should not enable the adversary to immediately conclude that the fragments are related. Using the same parameter for two obfuscated variables could confuse the adversary into thinking that two unrelated variable are related, but if the adversary breaks one variable then he will know more about the other.
- The set of possible parameters for a variable should be large enough and be uniformly randomly distributed for that exhaustive search for the correct parameter is sufficiently hard.
- The obfuscation parameters should exposed as infrequently as possible and as for as short as time period as possible. The variable’s obfuscation transform may facilitate minimizing obfuscation parameter exposure.

6.2.2 Data Representation/Protection Techniques

There should be a significant number of parameterized obfuscation transforms for each primitive data type included in the obfuscation library. The obfuscation transform evaluation criteria are the following:

- If possible it should be obscure which type of obfuscation transform is being used to protect a particular variable. The adversary is assumed to have a copy of the obfuscation system and therefore know the available set of transforms but ideally will need to expend some effort to determine which transform is being used to protect a

¹² Here we ignore the idea of improperly or inconsistently obfuscated variables used in decoy code in the obfuscated program.

particular variable. Preferably some degree of *simple inspection* should not be sufficient to determine which obfuscation transform algorithm type is being used to protect a variable. The goal is to force the adversary to expend significant resources just determining which obfuscation technique is being used on a particular variable.

- The adversary that does not know a variable's obfuscation parameter should not be able to de-obfuscate a variable except by exhaustive search. Small changes in the parameter should result in significant changes in the representation or else much larger parameters are needed
- Ideally the obfuscation transformation should allow for operations analogous to basic operations defined for the data type to be performed directly on the obfuscated version of the variable without exposing how obfuscation or de-obfuscation are performed.
 - For integer variables the basic operations are

Add (and increment), additive-inverse/subtraction, multiply, multiplicative inverse, remainder, and bit manipulation operations such as shifts and bit-wise AND, OR, NOT and XOR.

As we saw in section 6.1.1 there are some (limited) situations where no operation need be supported on a variable (simple decoding). More frequently, only a few operations (e.g., only increment by one) need to be supported over (perhaps) a small subset of the Java integers.¹³

However, effectively obfuscating Java integers, in the general case, is difficult. Large ranges of values need to be supported over several operations. A significant difficulty, as noted in section 6.1.1, is created by operational transience; a given variable may have to support operations that it does not directly use. For example, let A, B, C, D, and T be obfuscated integers and the statements

```
T = A * B;
```

and later (without changing T)

```
D = C + T;
```

appear in the program. Then all four variables may need to use the same obfuscation technique, one that supports both + and *. As a practical matter all four variables will use equivalent obfuscation parameters.

The Java virtual machine does not indicate arithmetic errors except for divide by zero during integer divide and remainder operations. This error is handled by throwing an `ArithmeticException`. Obfuscated integers do not support divide by zero error handling when the divisor is obfuscated, since doing so exposes information about the divisor. There is a risk that we may therefore break conforming programs by failing to properly handle divide-by-zero.

¹³ Small loops, in particular, have these characteristics.

- For boolean variables the basic operations are
Test, AND, OR, NOT.

6.3 Obfuscation Techniques

First we describe two potential solutions for obfuscating parameters used to control the obfuscation of basic variable types. Then we describe obfuscation transforms that employ parameters to obfuscate basic variables, especially integers.

6.3.1 Obfuscation Parameter Generation Techniques

- **Utilizing obfuscated control flow** – (Block Keys).

In this approach parameters used for (de)obfuscating variables are constructed as part of the series of executions of blocks of code. In order for the adversary to be able to determine an obfuscation parameter at a point in the execution of the program, the adversary must know the correct execution sub-sequence for the program.

For example, a obfuscated program consists of hundreds of blocks of code, including block A which is followed by block B which is followed by block C where the obfuscated variable v is operated on by `Func()` using parameter variable pv_1 . Block C is followed by block D after `Func()` is used. Labels E, F, etc. refer to other blocks of code in the obfuscated program which are not part of the execution sub-sequence A, ..., B, ..., C, ..., D.

In this example the variables i , j , k , and l are *not* equal when they are used.

Block A contains:

```
## Where T(i) refers to the parameter variable pv1
## Where pv1 is know to be equal to w prior
## to executing the following statement
T(i) = T(i) + x
```

Block B contains:

```
## Where S(j) refers to the parameter variable pv1
S(j) = S(j) * y ## alternatively S(j) = S(j) OR y
```

Block C contains:

```
## Where P(m) refers to the parameter variable pv1
## Func() is an unknown binary operation that produces
## a result of the same form as its operands
W(k) = W(k) + a
## P(m) is equal to ((w+x)*y)+a
Z = Func(P, m, V, X)
```

Block D contains:

```
## Where R(k) refers to the parameter variable pv1
```

```
R(k) = R(k) + z
```

Block E contains:

```
## Where Q(1) refers to the parameter variable pv1  
L = Q(1) + zz
```

If the blocks are executed in a different order then pv_1 will be incorrect when `Funct()` is used. The use of array elements to refer to pv_1 is to make static analysis more difficult by requiring the adversary to perform alias analysis. The use of array elements and performing misleading assignments to obfuscation parameters should make associating blocks of code with one another more difficult and force the adversary to perform extensive control flow analysis in combination with alias analysis in order to determine the values of obfuscation parameters when they are actually used to control obfuscation transforms. The memory location of variable pv_1 can be accessed unnecessarily as in blocks D and E in the example, or reused for unrelated purposes to inhibit dynamic analysis.

- **Environment testing.**

In this approach parameters are derived from hopefully obscure information that is obtained by the obfuscated program during execution, typically this information is provided by the system running the obfuscated program, including the state of the obfuscated program. The obfuscated program only operates correctly on systems providing it the correct information, and the obscurity is obtained in part by the degree of difficult the adversary may have in determining what information provided by the system, to the obfuscated program, is relevant. This approach is loosely based on the “Clueless Agents” work of Riordan and Schneier [RS98].

A simple example is the following. The value X is hidden by creating $\text{Hash}(X)$. Only $\text{Hash}(X)$, not X , is stored in the obfuscated program. The program runs and queries the operating system using some interface, or examine the state of the program, perhaps several times. The results Y_1, Y_2, \dots are hashed creating $\text{Hash}(Y_1), \text{Hash}(Y_2), \dots$ and compared with $\text{Hash}(X)$. If $\text{Hash}(Y_i)$ exactly equals $\text{Hash}(X)$ then Y_i , or some other function of Y_i , is the de-obfuscation of $\text{Hash}(X)$. Unless the adversary’s environment provides the proper input(s) to the hash function the adversary will not be able to recover X . This methods and similar techniques can be found in [RS98].

For example Block A contains:

```
## DeObfs is a generic parameterized de-obfuscation function  
## Test() extract some potentially obscure information from  
## the state of the program  
x = Test()  
if (y == Hash(x) ) then  
    z = DeObfs(x, v)
```

This technique depends on the individual configuring the obfuscation policy knowing, either something about the environment on the machines that the obfuscated

program should run on that the adversary does not know or would find hard to duplicate, or knows some aspect of the obfuscated program's state that the adversary is likely to change during some types of attacks. If the adversary can attack the obfuscated program on one of the hosts for which the obfuscated program was configured to run (or knows the host very well), made no discernible changes to the obfuscated program, or how the program executes, then this technique would fail. However this technique can significantly increase the work factor of the adversary.

6.3.2 Data Representation/Protection Techniques

1. **Symmetric key block and stream ciphers** (DES, AES, Pohlig-Hellman, SEAL etc.).

This approach replaces integers with the encryption of the integer (and perhaps other information) using some symmetric key block cipher such as DES [FIPS 46-3] or AES [AES] or a stream cipher such as SEAL [RC94]. Multiple integers can be encrypted at the same time using stream cipher, or a large block cipher such as AES in 256 bit mode, or by using various chaining techniques such as Cipher Block Chaining [FIPS 46-3] with a smaller block cipher.

The obfuscation parameters, the encryption key and perhaps initialization vector, are the same for both obfuscation (encryption) and recovery (decryption). Most such ciphers do not support any operations on obfuscated data except, usually, equality testing and obfuscated incrementing (by performing repeated encryptions¹⁴) which together are enough to create a Simple Counter Bound.

An exception to the limitations of most block and stream ciphers is the Pohlig-Hellman [PH] symmetric key cryptosystem, which also supports obfuscated multiplication.

$$C = M^e \text{ mod } p \quad (p \text{ is a prime})$$

$$M = C^d \text{ mod } p \quad (e * d = 1 \text{ mod } (p-1))$$

The Pohlig-Hellman Cryptosystem

In addition to performing obfuscated multiplications, the Simple Counter Bound pattern from section 6.1.1 can be implemented by either converting obfuscated increment into obfuscated multiplication by the appropriate mapping of upper and lower bounds or by repeated encryptions. Note, however, as in the case of using repeated encryptions, de-obfuscation will likely be needed if the loop counter is used for other purposes.

The obfuscation parameters for Pohlig-Hellman based obfuscated multiplication are *not* the same as the obfuscation and de-obfuscation parameters. Obfuscated multiplication only uses (and exposes) the modulus p , which alone does not allow the

¹⁴ The obfuscator repeatedly applies a block cipher n times (the loop bound) to a given input to create a value, v , which serves as the loop bound. The obfuscated program reconstructs the input and repeatedly encrypts it until the same value v is obtained. Other cryptographic techniques such as Message authentication codes or cryptographic hash functions (with obfuscation parameters used as part of the input) can also be used to achieve the Simple Counter Bound.

adversary to either obfuscate or de-obfuscate. However, using obfuscation potentially exposes the de-obfuscation parameters and vice versa.

The operations $<$ and $>$ can be implemented in the symmetric key cryptosystem framework without de-obfuscation for reasonable numbers of values by using tables. The table indexing can be achieved by hashing the concatenation of the encrypted values that are used as the operands.

Symmetric key systems can be used to obfuscate an entire parse tree or portions of a tree.

2. Traditional asymmetric key (public key) encryption techniques.

Using asymmetric key encryption systems such as RSA [RSA78], El-Gamal [ElGamal85], or XTR [LV00] for obfuscation has some of the same limitations as the symmetric ciphers; however, obfuscated incrementing and frequently obfuscated multiplication are supported. The most noteworthy differences between the symmetric key and asymmetric key block cipher approaches are:

- Knowledge of the obfuscation/encryption key does not allow an adversary to know the de-obfuscation/decryption key. This is true if the public key of public key cryptosystems is used for obfuscation and the private key is used for de-obfuscation. Some public key cryptosystem, such as RSA allow for the roles of obfuscation key and de-obfuscation key to be reversed.
- Some public key cryptography based systems have the property that knowledge of either the obfuscation/encryption key or de-obfuscation/decryption key does not allow an adversary to recover the other key, *at the same time*. For example with RSA cryptosystem if both the private and public exponents are large hard to guess values, then both key are protected against the disclosure of the other key. We will call this RSA variant, bi-directional RSA.

Though not normally used in cryptography, bi-directional RSA is quite straightforward.

$$C = M^e \bmod n \quad (n = pq, \text{ both } p \text{ and } q \text{ are primes})$$

$$M = C^d \bmod p \quad (e * d = 1 \bmod (p-1)(q-1))$$

The RSA Cryptosystem

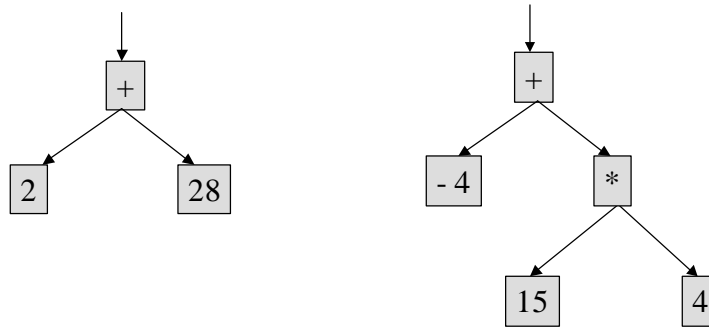
The algorithm supports obfuscated multiplication and the Simple Counter Bound example from section 6.1.1 can be supported by the appropriate mapping of upper and lower bounds and converting increment into multiplication Note, however, that de-obfuscation may be needed if the loop counter is used for other purposes.

The operations $<$ and $>$ can be implemented in the asymmetric key cryptosystem framework without de-obfuscation for reasonable numbers of values by using tables. The table indexing can be achieved by hashing the concatenation of the encrypted values that are used as the operands.

Asymmetric key systems can be used to obfuscate an entire parse tree or portions of a tree.

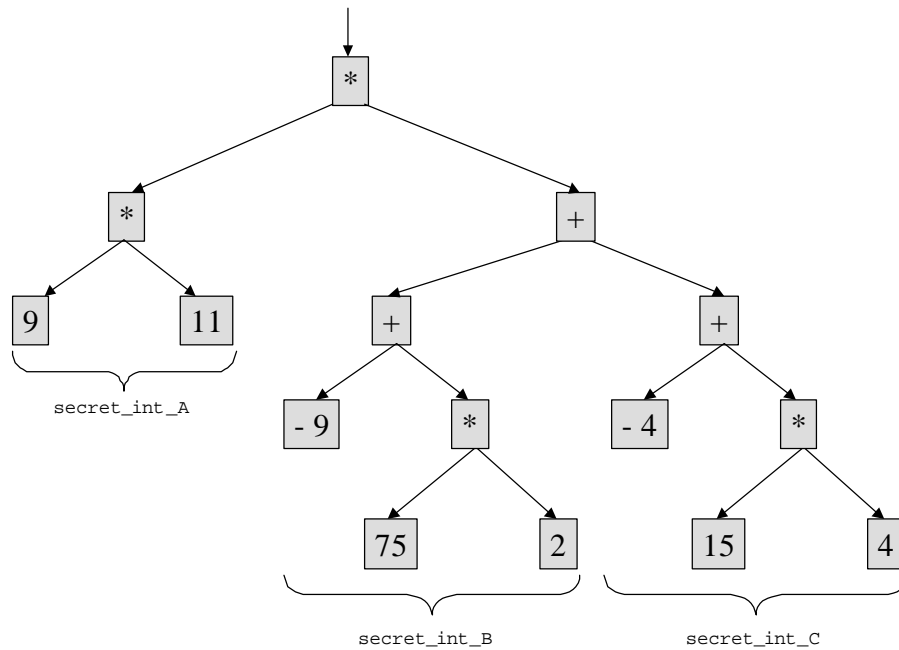
3. Integers stored as parse trees.

This approach replaces integers with a parse tree like structure for an expression that evaluates to the integer being obfuscated. Operations on obfuscated variables, such as addition create new parse trees that combine the parse trees that represent their operands. For example the value 56 could be represented by either of these two trees



Two Parse Trees for the value 56.

depending on the value of the obfuscation parameter. The Simple Arithmetic example from section 6.1.1, $secret_int_D = (secret_int_B + secret_int_C) * secret_int_A$; might result in the following parse tree being created for $secret_int_D$



Parse Tree for $secret_int_D = (secret_int_B + secret_int_C) * secret_int_A$

Obfuscating integers by only using parse-trees is potentially vulnerable to analysis tools. Memory scanning tools could identify trees by their structure. Analysis could be defeated by obfuscating the operands and operators by using (combinations of) other obfuscation techniques such as: the encryption techniques, tables, or re-mapping of integers through simple permutations.

Another approach to obfuscating the parse trees is to use an obfuscation parameter to control the application of reversible changes to the tree such as swapping nodes or adding new sub-trees to the tree.

The main problem with this technique is that it only queues operations for later execution rather than actually executing operations. If the parse tree techniques is combined with other obfuscation techniques that need to use de-obfuscation to enable arithmetic, bit-wise, or other operations, then the de-obfuscations may be batched together and happen less frequently.

Using parse trees may work against some of the code obfuscation techniques described in section 4. If the adversary discovers a parse tree or a location where parse tree evaluations occur then he potentially can discover correct sequences of operations/ blocks of code, that the code obfuscation may otherwise obscure.

Branches based on conditions such as the relative values of two variables cannot in general be taken without de-obfuscation or unacceptable complexity,¹⁵ so the obfuscation tree may need to be de-obfuscated frequently significantly reducing the technique's value.

The properties of the tree (i.e., size, depth) may encode some information but use of this mechanism will be hard to hide and has low bandwidth.

4. Obfuscate integers by mapping integers into (well-known or obscure) mathematical structures that are not based on cryptosystems.

The minimum requirement for this type of technique is that some subset of the integers that is of interest (such as all integers from zero to one hundred) is supported.

Whether the integers beyond the range of interest are supported correctly or incorrectly or not defined is not very important. Since these techniques are not based on strong cryptography, certain desirable properties, such the de-obfuscation transform's resistance to attacks based on known obfuscated, de-obfuscated pairs, may not be present.

- Chinese Remainder Theorem mapping

There is a well known result from number theory that, given any integers $a_1, a_2, a_3, \dots, a_j$ and a set of relatively prime positive integers $m_1, m_2, m_3, \dots, m_j$ there

¹⁵ The parse tree techniques could be extended with node types that support some conditional branches, embedding the branch condition and links to the necessary parse trees to perform either branch. However using such techniques are very complex with poor performance and cannot handle operation that cannot be expressed as a tree, such as I/O.

exists a unique $0 \leq X < (N = \prod m_j)$ such that $X = a_j \pmod{m_j}$. We can obfuscate an integer X by encoding it in its Chinese Remainder Theorem (CRT) representation. The obfuscation and de-obfuscation parameters are the value of N and its partial factorization, $m_1, m_2, m_3, \dots, m_j$.

Normal arithmetic operations such as addition and multiplication can be performed on the CRT representation. However, performing these operations risks exposing the $m_1, m_2, m_3, \dots, m_j$ values, and consequently allowing the adversary to perform obfuscation operations on variables obfuscated using the same parameters. Certain values such as 0, 1, 2, ... do not obfuscate well.

While not directly supported by the CRT, $<$ and $>$ operations can be implemented in the CRT framework without de-obfuscation for reasonable size ranges of values by using tables. The table indexing can be achieved by hashing the concatenation of the CRT representations of the operands.

- Permutations and other mappings of ordered lists onto themselves

We can obfuscate integers into permutations. The obfuscation parameters can be used to control the size of the set of elements, a permutation/mapping that corresponds to incrementing. Let zero be defined as the increment operator, one defined as applying the increment operator twice and so on.

For example¹⁶ let the permutation (1 2) (7 6 5 4 3 8) be the increment operator, and zero be obfuscated as (1 2) (7 6 5 4 3 8), therefore: one is obfuscated as (1) (2) (7 5 3) (8 6 4), and two is obfuscated as (1 2) (7 4) (8 5) (6 3), etc.

This technique is sufficient for implementing addition (by composition of permutations) and therefore incrementing. The $<$ and $>$ operations can be implemented in the permutation framework without de-obfuscation for reasonable size ranges of values by using tables. The table indexing can be achieved by hashing the representations of the permutations used as operands. The obfuscation operation is equivalent to the de-obfuscation operation.

- Other number theory tricks

Another obfuscation technique maps the integer variable y into the following value:

$$(Y = y * p)$$

where p is a prime integer larger than the absolute value of all un-obfuscated integers that need to be represented using p .

De-obfuscation is performed by removing the factor p from an obfuscated value X . Take the integer part I of the logarithm base p of X . Divide X by p^I to obtain the de-obfuscated value.

Adding and multiplying two obfuscated integers W and Y becomes:

¹⁶ The notation (1 3 2) means the symbol in the first position is mapped to the third position, the symbol in the third position is mapped to the second position, and the symbol in the second position is mapped to the first position.

$$W + Y = (p * (w + y)) \text{ and } W * Y = (p^2 * w * y)$$

In either case we don't have to expose the obfuscation parameter p .

Two obfuscated values will have p as part of their greatest common divisor, and the attempted factorization of a single obfuscated value will likely expose p . Therefore this approach is rather weak.

Still another obfuscation technique, maps the integer variable y into the following value (n is a composite of two nearly equal primes controlled by the obfuscation parameter a is chosen at random):

$$(Y = a * n + y), \quad (W = b * n + w),$$

and the arithmetic operations are:

$$W + Y = ((a+b) * n + (w + y)) \text{ and}$$

$$W * Y = (n(n + b*w + a*y + b*a) + w * y)$$

The value n is larger than the absolute value of all un-obfuscated integers that need to be represented using n . The operation $u = U \text{ mod } n$ de-obfuscates U . Neither addition nor multiplication exposes the parameter.

The adversary may still be able to find n without witnessing an obfuscation or de-obfuscation operation if the entropy of the un-obfuscated values is low.

If a and b are the same then the normal integer operators $<$ and $>$ work on the obfuscated values. However the difference between two such obscured variables is no longer obfuscated. The compromise of a single obfuscated value breaks the obfuscation of all values using the same obfuscation parameter.

Both of these obfuscation techniques can produce very large obfuscated values if p or n is large and expressions using many obfuscated multiplications are evaluated. We could either use the Java big integer package to implement the technique or perform mod operations on intermediate values of these expressions.

5. Packing words (integers and/or booleans).

This approach attempts to protect variables from dynamic analysis of memory references. Instead of separate words being used to store different variables, multiple variables are packed into the same word so that the adversary is presented with a storm of events if he sets a break point on all references to a word during dynamic analysis.

- bit packing

Each small set of bits is used store a separate variable, such as a boolean or small integer. The parameters for the variables control which bits of the word represent the variable. Addition, multiplication, etc. are defined for the appropriate sub range of the integers. How bit fields are that are not part of a variable are handled in the integers has not been defined but treating these fields as small integers may be the best approach. Performing operations in the fields will expose the positions of the fields

- prime number based packing

The value of a word (treated as an unsigned integer) mod a prime number or composite number is the value of a separate variable. For example if a word that contains 25 can be used to represent two variables, one variable containing 7 (mod 9) and the other variable containing 12 (mod 13). In this example 9 and 13 are the parameters.

This technique is CRT in small fields so addition, multiplication etc. are defined but the parameters are vulnerable.

6. Other obfuscations of booleans.

An approach to boolean variable obfuscation is to use integer variables to represent boolean variables and assign many possible integer values to `true` and similarly for `false`, for example whether the integer variable is divisible by some value can be used to determine the corresponding boolean value. Example 6 from section 6.1.1 can be replaced by

```
if ( ( gcd(foo * bar ) % obfuscation_param1 ) == 0 ) {
    total = total + 1;
}
```

or if logical OR is required then we can use

```
if ( ( (foo * bar ) % obfuscation_param1) == 0) {
    total = total + 1;
}
```

where `rem` is the remainder operation.

This approach does not obscure that a AND or a OR (or their respective complements) is being executed. The following example obscures the Boolean operation at the cost of greater complexity, i.e., more obfuscation parameters are needed.

```
if ( (( foo + obfuscation_param1) *
      ( bar + obfuscation_param2)) % obfuscation_param3)) == 0) {
    total = total + 1;
}
```

In this approach the variable `foo` can be set to `true` by executing

```
foo = X;          /* where X == 15 */
```

followed by

```
foo = foo + Y[I]; /* where Y[I] == 13 */
```

and then use 14 for value of `obfuscation_param3`. If the use of the remainder operation is too strong an indication that a boolean operation is being performed then other operators such as less than can be used.

6.4 Analysis

6.4.1 Obfuscation Parameter Generation Techniques

The flow control-based obfuscation parameter generation approach appears to be the most viable single approach if care must be taken that the use of the technique does not reduce the effectiveness of flow control obfuscation.

The clueless agent approach in our situation presumes that the program's obfuscator has information about the proper execution environment or proper execution of the obfuscated program that an adversary cannot readily obtain and that varies sufficiently with different obfuscations so that this technique is re-usable against the same adversary with the same application. This may be assuming too much.

Both techniques can be used to detect program tampering and a combination of the two techniques is more effective than either approach and therefore the combination of the techniques is the best choice.

In general, we expect that restrictions on the operations required by a variable will drive the selection of obfuscation techniques. Our ability to properly select a variety of useful techniques is dependent on the ability of our tool to do useful control-flow analysis.

6.4.2 Data Representation/Protection Techniques

The merits of the obfuscation data transforms are less clear than the merits of the obfuscation parameter generation techniques.

Incrementing and equality can be achieved by many different techniques, including techniques such as repeated multiplications of a starting matrix S , by another matrix M , resulting in the sequence of matrices, $S*M$, $S*M*M$, $S*M*M*M$, etc. The loop bound is derived from the determinate of the proper matrix in the sequence.

The use of (a)symmetric cryptographic algorithms to protect parse trees offer a good solution in situations where parse trees make sense. Care must be taken that the cryptographic algorithm is not so distinctive that it quickly becomes a target for analysis. An adversary can observe the expansion of a parse tree without being able to even de-obfuscate the older parts of the tree. However we believe that parse trees typically only reduce the exposure of the de-obfuscation transform or un-obfuscated variables by a modest amount because in most situations the parse tree will need to be evaluated after few operations have been performed on the tree.

This transform ($Y = a * n + y$), is attractive for general use since both obfuscated addition, and obfuscated multiplication, are supported and an adversary cannot readily de-obfuscate such variables even when able to perform dynamic analysis of expression using both obfuscated addition and multiplication. Unfortunately comparison tests are not supported.

If the number of integers that must be supported is not too large, and performance goals are modest, then bi-directional RSA is a good choice. Obfuscated multiplication done directly, while obfuscated addition and obfuscated comparison can be performed by table lookup. Also the disclosure of either the obfuscation or the de-obfuscation parameter

does not compromise the other. Dynamic analysis of multiplication does disclose either the obfuscation parameter or the de-obfuscation parameter.

There are a number of issues with using tables for addition. The table should not support 0 and or 1. Both of these values need to be handled (if necessary) by combination of expression rewriting (including testing for special values). For example in order to implement the Simple Counter Bound example in Section 6.1.1 with this technique the code fragment could be rewritten as.

```
for (int i = obfus_param;
     i != secret_int*onfus_param - obfus_param;
     i=i + obfus_param) {
    bar(foo(a),b);
}
```

Where `obfus_param` is an obfuscated constant that is determined at program obfuscation time.

Obfuscating Boolean variables by converting them into obfuscated integers is a good approach to protecting Booleans. The primary questions that require further investigation is whether or not the representations of the Booleans as integers can be made hard to distinguish from generic obscured integers.

7 Transient Variable Obfuscation

To an attacker, the obfuscated program should appear to be a large set of interwoven basic blocks that provides little useful information. We assume that the attacker has access to the original code of the transformed program. When compiled, that code will contain many standard patterns of code that could be matched against code in the obfuscated program. If attackers can match up code in the original to code in the transformed program, they gain significant advantage in decomposing the transformed program so that they can tamper meaningfully. Our goal is to inhibit such matching and other approaches to analyzing the program, in part, through obfuscating transient variables.

The basic block manipulations discussed in section 4.2 rearrange the blocks of code, and disguise loops. However, unique constants or unique local variable manipulations may still give the attacker valuable insight into mapping the obfuscated code to the original code, and methods for finding relationships between blocks of code if the attacker does not have access to the original code. The complex data representations described in section 6 only affect long-term data, as the performance costs for short-term transient variables of these representations are unacceptable. This section addresses cheap transforms for short-term transient variables that should provide useful, short-term variable relationship obfuscation and some degree of data hiding.

For the purposes of this section, we will look at the following code fragment:

```
for (int i = 0; i < 5; ++i) {
    bar(foo(i),i);
}
```

javac generates the following bytecode¹⁷ for that loop

```
        iconst_0          # push constant 0
        istore_1         # pop into var 1 ("i")
        goto loop        #

start:   iload_1          # push "i"
        invokestatic foo # call "foo",
        # leaves retval on stack
        iload_1          # push "i"
        invokestatic bar # call "bar", no retval
        iinc 1 1         # i += 1
loop:    iload_1          # push "i"
        iconst_5         # push 5
        if_icmplt start  # cmp, jlt start
```

As is apparent, the `iinc 1 1` to `if_icmplt` target sequence makes it easy to identify the loop, and to identify the size of the loop. If the attacker can use that information to determine the relevant loop in the original source, he can use this information to start attacking the control flow obfuscation. We wish to obscure the comparisons and increment instructions to prevent the trivial identification of code blocks based on the local variable manipulations.

This section focuses purely on techniques to obfuscate integers. Through various transforms, all data used in a program can be represented as integers,¹⁸ so techniques that obscure integer values have wide applicability.

Two basic approaches to obscuring local variables are to change the meaning of the bits, or to change the location of the bits. The obvious example of the first is represented by the “XOR with constant” scheme, in which the variable is stored in “encrypted” form. An obvious example of the second is the “array chase” scheme, where the actual variable is a cell in an array that is only referenced indirectly. These two approaches can be applied independently, theoretically increasing the available obscurity.

Many techniques here make assumptions about the usage patterns of the local variables. For instance, the “XOR with constant” scheme is very effective if the variable is only set (via constant or results from a method) and tested against for equality. In particular, integer variables are commonly used for operations that require a range significantly more limited than the range of generic integers. Some obscuring methods work particularly well in this context.

¹⁷ The Java bytecode used in this section is intended to be illustrative; they are not in the input format of a specific assembler.

¹⁸ Pointers can be stored in a large array, with the integer referring to the index of the array. Smaller integers can be represented as larger ones, 64 bit integers can be represented as multiple 32 bit integers. While in theory floating point numbers can also be represented as multiple integers, we expect that few programs that intend to run in hostile environments will use floating point, so we have not addressed the issue in this report.

Transient variable obfuscation must have no effect on program correctness. The solutions should not impose unreasonable costs in terms of either performance or space. The transforms discussed in this section will be applied virtually everywhere in the input program, so performance is much more critical than in the case of longer-term variables. Any variables of extra importance, such that the extra costs in time and space are justified, should be considered long term data. Protecting long-term data is discussed in section 6.

The solution should have some strength. In the case of bit-changing transforms, the transforms should make it hard to determine the “key” by static analysis. In the case of bit-hiding transforms, the transform should make it difficult to find the relevant bits by static analysis. In the case of more mathematical transforms, the parameters of the transform should not be clearly visible in the code. These criteria suggest that the solution must be able to operate significantly in transformed space. To the greatest extent possible, the solutions should be parameterized, as defined in section 6. The attacker should have to search for a particular key even if he knows the transform being used.

7.1 Evaluation

We looked at number of different schemes for obscuring local variables: two different flavors of XOR, transforming small integer ranges into MSB, splitting the register along arbitrary lines, rotating the register, shifting the range, array chasing, and table lookup. We also looked at using some of the schemes for long-term storage. All the examined schemes have significant undesirable properties, and we continue to look for preferable schemes.

7.1.1 XOR with Constant

The first scheme is to XOR the local variable with a known constant. This meets the performance goals, as the transform is small and quick. However, it fails the obfuscation goals if the transform is limited to a single block, and is fairly obvious in the absence of that limitation since simple arithmetic and comparisons are not possible.

For example, consider the following bytecode:

```
                ldc 12345678
                istore_1
                goto loop
start:          iload_1
                ldc 12345678
                ixor
                invokestatic foo
                iload_1
                ldc 12345678
                ixor
                invokestatic bar
                iload_1
                ldc 12345678
                ixor
                iconst_1
                iadd
```

```

        ldc 12345678
        ixor
        istore_1
loop:    iload_1
        ldc 12345678
        ixor
        iconst_5
        if_icmplt start

```

Notice that the `ldc 12345678; ixor` pairing constantly shows up during the loop. This is because the simple operations in the loop (`++i`, passing as an argument to a function, and `i < 5`) have to be performed on the "cleartext" value. If the other basic blocks have the same transform applied, the middle of the loop simplifies; however the need to undo the transform for add and compare remains. Additionally, the "key" constant shows up in the code constantly, preventing the secret from being very secret.

This transform is primarily effective when only set and compare operations are required. In this limited case, any mapping of values to a different random set of values would also work and be equally difficult to reverse. The "XOR with constant" scheme has the advantage of allowing rare deobfuscation of the original value, which the "arbitrarily permute values" scheme does not allow.

7.1.2 XOR with Variable

If we know a variable is constant for the length of the loop (i.e., not modified during the loop), we can use it as the constant in the scheme described above.

Unfortunately, this solves little. The attacker can still easily analyze the code. While changing unmodified code dynamically is now harder, writing a tool to undo the obfuscation is still easy, and such a tool could be run on incoming programs in realtime. Additionally, this scheme makes it harder to use the same obfuscation through basic blocks, as the probability that the variable will remain unchanged diminishes. This scheme is useful in the same scenarios as "XOR with constant" and with the same constraints.

7.1.3 Most Significant Bit

For integers with small ranges (like in the example above) the integer can be transformed into a bit-vector, with the most significant bit set representing the value. The advantage of this approach is that many common operations can be performed while the value is in transformed form. For example

```

        iconst_1
        istore_1
        goto loop
start:  iload_1
        invokestatic foo
        iload_1
        invokestatic bar
        iload_1
        iconst_1
        ishl
        istore_1

```

```
loop:      iload_1
          bipush 31
          if_icmplt start
```

This example assumes that the `foo` and `bar` methods have been modified to expect the arguments in transformed form, but even without that assumption the byte code is reasonable short and compact. The integers that define the loop are obscured and the loop will no longer match the obvious pattern of a loop.

However, an attacker familiar with our toolkit will not be fooled. This approach does not lend itself to keys, and can be easily undone statically. Furthermore, the "untransform" sequence is expensive, as either a loop or a \log_2 operation is required. If the transform is not available globally, the performance impact might be unreasonable.

This scheme is most useful when only `set`, `compare`, and `addition/subtraction-by-constant` operations are used. Rare `deobfuscation` operations are also permitted, but the `deobfuscation` operations tend to give the scheme away. In absence of parameterization, this scheme is of limited utility.

7.1.4 Using Offsets

Integers can be represented as a normal integer with a constant offset. For example, the integer 5 might be represented as the integer 11 and an offset of 6. The offset constitutes the key, giving the obfuscation program a significant key space to use.

Performance should not be highly affected. Extra instructions will be required for most arithmetic operations, but they will normally be simple instructions. Similarly, code size will be affected, but the overall affect should be minor.

Many operations will be possible in obfuscated space, however the key will still leak constantly into the generated bytecode. For example, simple addition of two variables will require an adjustment to the result to remove the extra offset. Multiplication of two transformed variables will require an adjustment of the offset times one of the two operands. In either case, an attacker will be able to determine the key easily, through dynamic analysis. Simple loops will not have changed much, and will certainly not have changed forms. The value in obfuscation is minor.

Offsets can be done multiplicatively as well as additively. This makes multiplication simpler, so it is suitable for variables that are frequently multiplied. It should also be noted that oftentimes the compensation for the extra offset or factor does not need to occur immediately after every operation. We can compensate 3 times after 3 operations, instead of compensating once after each one. If the values being computed are not passed to other parts of the program, compensation may not be needed at all. We can simply rearrange our predicates that concern these obfuscated values to have the proper meaning for the uncompensated values.

This scheme easily supports almost any "by-constant" operation and only fails when operated on by variables. De-obfuscation is trivial and not particularly revealing if the attacker cannot easily connect the de-obfuscation to the use.

7.1.5 Operation Counting

Consider the simple obfuscation of the integer variable x into an obfuscated variable, X , containing the following tuple:

$$(a^k * x \bmod p, k)$$

Where k is incremented for each multiply or divide (multiply by the multiplicative inverse) on X until X is re-initialized. By counting multiplies rather than compensating for them each time they occur (see above) we reduce the exposure of a but not p . Multiplying two obfuscated numbers W and Y that have been obfuscated using the same parameter is straightforward.

$$(X, k) * (Y, l) = (X*Y, k+l)$$

Adding two obfuscated integers w and y is more complex. If both w and y have the same value for the k then the value of k for their sum remains the same. However, if the counters are different then an adjustment is necessary which will expose the obfuscation parameter a . Assuming that w has accumulated more a 's than y :

$$(W, k) + (Y, l) = (W + Y*(k-l), k)$$

It is worth noting that under this technique obfuscations of the same value with the same parameter differ by a factor of a^e for some e . If an adversary knows the relationship between the un-obfuscated values, w and y , of any two obfuscated values, W and Y , then he can break the obfuscation of both.

7.1.6 Rotation

Integers could be represented in rotated form. For example, rotating the integer by 16 bits would effectively switch its representation from big endian to little endian.

One advantage of this approach is that most common operations can be performed while the value is transformed. Simple addition with constants merely requires rotating the constant at obfuscation time, addition of two variables in rotated form is practically unchanged (have to handle the carry bit, but that's about it). Multiplication has to be transformed into shift/add sequences, with division handled similarly. Comparisons with constants are also relatively unaffected.

The transform uses a parameter, which represents the degree of rotation. Unfortunately the search space is small.

As with other transforms mentioned above, the total level of obscurity is not as high as is desired. Take the following byte code:

```
        iconst_0
        istore_1
        goto loop
start:  iload_1
        invokestatic foo
        iload_1
        invokestatic bar
        iload_1
        ldc 0x10000
        iadd
```

```

        istore_1
loop:    iload_1
        ldc 0x50000
        if_icmplt start

```

While the changes are minor, so is the obfuscation gained. In particular, the attacker merely wonders why you chose to increment your loop by such a large value unnecessarily. An automated tool might not even notice the transform, and merely return the loop:

```

    for (int i = 0; i < 0x50000 ; i += 0x10000)
        bar(foo(i),i);

```

7.1.7 $X \bullet N \bmod 1$

An interesting transform due to L. Washington [Wash01] is to represent an integer x as:

$$X = a * x \bmod 1$$

where a is some fixed irrational number and X is a real number on the interval $[0, 1)$. The reason a , the obfuscation parameter, is an irrational number is to avoid mapping multiple integers into the same value of X . If a were 13.5 then 12 and 14 would both map into zero. We can avoid this problem in floating point by choosing a value for a with that includes a small fraction, i.e., $a = 13.5 + 1/2^{30}$.

Obfuscated addition is simple and obscure if the same value is chosen for both operands. There are two approaches to obfuscated multiplication, both have limitations. Let $a*x \bmod 1$ and $a*y \bmod 1$, be the operands for an obfuscated multiplication. One approach is to multiply both operands by $a^{-1} \bmod 1$ to obtain x and y , multiply to get $x*y \bmod 1$, then convert to $a*x*y \bmod 1$. The second approach is to multiply $ax \bmod 1$ and $ay \bmod 1$ to get $(a^2)x*y \bmod 1$ and then multiply $(a^2)x*y$ by $a^{-1} \bmod 1$ to obtain $a*x*y \bmod 1$. The later approach has the slight advantage that x and y do not appear “in the clear” while $a*x*y \bmod 1$ is calculated. However, in either approach, the de-obfuscation parameter a is exposed during each multiplication operation.

To prevent exposing the relationship between copies of the same value obfuscated under this technique different values for the obfuscation parameter are needed. However using different values for the obfuscation parameter degrades the obscurity of the obfuscation addition

7.1.8 Register Split

A single integer register could be split among multiple different registers. The question of which bits in the different registers are actually significant is effectively the key, giving this technique a significant number of different instances.

Many operations can be performed in obfuscated space, however not directly. Instead, operations will have to be effectively emulated with careful concern of carrying bits. As a result, performance will suffer and code size may bloat beyond acceptable bounds.

The “key” will be obtainable by examination of the code, as the general case addition or multiplication or comparison code will all have to have hardcoded knowledge of the bit

layout. To the extent that the obfuscation code can do static analysis of the range of data in the local field, the generated code can be incomplete. However, experience with optimizing compilers suggests that this degree of knowledge is less common than is desired.

This transformation can be generalized into any number of bit splitting transforms. There is no reason why the significant bits in different local variables should be contiguous, for example, nor is there any reason why the significant bits should be stored directly in local variable slots except for performance concerns. The general issues identified here relative to the obscurity of the transform apply (to various degrees) to all other forms of this transform.

7.1.9 Array Chasing

Consider the following array:

```
int baz[10] = { 4, 1, 7, 9, 2, 3, 5, 2, 5, 0 };
```

Instead of using a local variable slot for “i”, use `foo[7]`. Instead of using `foo[7]` directly, write the loop as follows:

```
for (baz[baz[4]] = 0; baz[baz[4]] < 5; ++baz[baz[4]])  
    bar(foo(baz[baz[4]]), baz[baz[4]]);
```

In this example, `baz[4]` is 2, and `baz[2]` is our variable. The approach has considerable theoretical strength as it leverages the aliasing problem.¹⁹ Unless you can prove that neither `bar` nor `foo` affect the array `baz`, you can not assume that the repetitious references to `baz[baz[4]]` all refer to the same place. This technique, along with a scheme for dynamically altering the array, is presented in Wang et al. [W00,WDH+01,WHK+00] and is used to obscure control flow and data access.

Unfortunately, in practice, although the adversary may not be able to prove that property in the general case, he knows that our software generates this type of transform and he can watch the array for manipulations. Also, the transform is relatively easy to recognize, as the bytecode sequences to repeatedly dereference the array elements will be readily apparent.

This transform can use parameters, as the values in the array can vary greatly, as can the path. If the starting constant is replaced by a variable (modified in concert with the array) then nearly any set of values can be chosen. The algorithm for generating the array turns out to be fairly straightforward: populate the array with random numbers in the correct range, and then chose N cells from the array as your chain. Putting multiple local variables in the same array is possible, although it does increase the complexity of re-shuffling the array.

This transform is also relatively fast. Although the space required for local variables will grow (by, in this example, over an order-of-magnitude), the time required to undo the

¹⁹ A known NP complete problem [LR91, CTL98a].

transform is minor, consisting of only a few array dereferences, all ordered conveniently for an optimizer.

7.1.10 Table Replacement

Integers with particularly small ranges can be replaced in total by recalculated tables. The register that previously held the integer now only holds an array index. Unary operations are represented by one-dimensional arrays that map the input index into the output index, and binary operations are represented by two-dimensional arrays.

For example, in our loop the integer is known to only have values between zero and five. The operations performed are “increment by one” and “compare less-than”. The following “key” might be generated at compile time:

Value	Index
0	3
1	4
2	2
3	0
4	1
5	5

Given those initial values, the “inc” table would look like:

1	5	0	4	2	NA
---	---	---	---	---	----

That is, 0 (index 3) incremented is 1 (index 4). By definition, we do not care what value goes in index 5, as that value is outside the range seen by the program. Similarly, the “compare less-than” table would look like:

0	0	1	1	1	0
1	0	1	1	1	0
0	0	0	1	1	0
0	0	0	0	0	0
0	0	0	1	0	0
1	1	1	1	1	0

And the loop would become something similar to:

```

    iconst_3          # push constant 0 (index 3)
    istore_1         # pop into var 1 ("i")
    goto loop

start:
    iload_1          # push "i"
    invokestatic foo # call "foo", leaves retval on the stack
    iload_1          # push "i"

```

```

        invokestatic bar    # call "bar", no retval

        aload_2            # load a reference to the increment table
        iload_1            # push "i"
        iaload             # array access
        istore_1           # pop "i"

loop:    aload_3            # load a reference to the less-than table
        iload_1            # push "i"
        aaload            # get the ith column of the table
        iconst_5          # push constant 5 (index 5)
        iaload            # get the 5th entry of the column
        ifeq start        # jump to start unless LessThanTable[i,5]

```

Without knowing the values of the array, it is impossible to determine the size of the loop. If the array is initialized dynamically by the program (only the 0 and 5 values need to be constant) then the attacker must allow the program to run sufficiently to initialize the arrays before any analysis of this section.

Unfortunately for this scheme, the initialized arrays are relatively easy to analyze. In this simple example, the “compare less-than” table is obviously a boolean table (values of only zero or one). As one field is all zeros and no field is all ones, it has to be a less-than table. The field of all zeros is the maximum value. Identity properties will make it easier to reverse engineer addition (+0) and multiplication (*1) tables. To obscure these facts, the generated tables will have to be larger than required. Additionally, the undefined squares (where the value is out of the known range of the integer) will have to be maliciously filled with values that lead into the unused section of the table.

This method is equally suitable for nearly any usage pattern. Unlike most methods discussed above, handling operations by constant and by variable are of equal complexity.

On the downside, the bounds of the variable must be known at obfuscation time, and must be relatively small. The transform generates several tables of non-insignificant size, and may bloat the code beyond acceptable limits. The tables generated will fall quickly to static analysis, once the attacker has identified them and has access to their values. The method cannot be used if the values will be exported, as the “key” used to generate the running program does not know the tables. If export is required, and export table will have to be generated which will obviously reveal the key used.

7.2 Analysis

There is no “one true transform” among the possibilities listed above. The transform chosen will have to be based on the usage patterns of the variables. For the example used in this section, the “Table Replacement” transform is probably the strongest, assuming that the `foo` and `bar` methods can be safely used with transformed data. The “array copy” pattern that the example was intended to model can be used with transformed data, as it merely requires that each value be referenced at least once and only once. Unfortunately to use that method for this example, the obfuscator must be able to prove properties for `foo` and `bar` which might not be practical.

8 Related Work

Our approach, along with work of Hohl [Hohl98] and Wang et. al [WHK+00, W00, WDH+01], aims to delay an attacker, where some other researchers seem to be looking to prevent access entirely. Related research is in the categories of electronic commerce and mobile agent protection, computing with encrypted functions, and in practical obfuscation and decompilation.

8.1 Perpetual Obfuscation

Recently Barak et al. [BGI+01] presented a paper on the theoretical limits of obfuscation techniques. This paper demonstrates that there exists functions for which one cannot create an obfuscator that can prevent an adversary from learning *anything* more about a obfuscated function by operating on the obfuscated version of the function than the adversary could learn by running the function in a block box where the adversary can choose the inputs to the function and see the outputs of the function. This result applies to adversaries that have unbounded time for analysis.

8.2 Program Obfuscation Transforms

A number of authors, Collberg et al. [CTL97a, CTL98a, CTL98b, CT00] and Wang et al. have previously investigated program obfuscation techniques. As noted earlier in this report, our work draws upon their results.

Collberg et al. provides a model for program obfuscation. Their results include the definition and construction of opaque predicates, predicates with outcomes that are known at obfuscation time, but difficult for an obfuscator to deduce [CTL98a], and the development of transforms, such as variable splitting, which obfuscate data structures [CTL98b]. They have shown that some of their techniques are based on the intractability of alias analysis, which is related to the effectiveness of static analysis.

Collberg et al. investigated properties of a large number of potential obfuscation techniques, including techniques for obfuscating general program layout, control obfuscation, data obfuscation, “preventive” obfuscation techniques (i.e., techniques to defeat known de-obfuscators), and opaque constructs. The techniques they examined for general program layout include scrambling identifiers, removing comments, and changing formatting. Their control obfuscation techniques include inserting dead code, interleaving methods, and loop fusion. Data obfuscation techniques they studied include splitting variables, refactoring classes, and merging scalar variables. Their “preventive” obfuscation techniques are designed to defeat known de-obfuscators by use of bogus data dependencies, aliasing parameters, etc. The authors’ opaque constructs include opaque predicates and paralleling programs using multi-threading.

Wang et al. [WHK+00, W00, WDH+01] have further developed control flow and data-flow obfuscation techniques. (They have implemented a number of obfuscation techniques, a sophisticated form of branching flattening, see also Section 4.1 and variable aliasing, related to the techniques of Section 7.1.9 of this report, in a source-to-source obfuscator for C language programs.) The authors studied the performance and precision

of the results of running static analysis tools, the IBM NPIC tool [HBC+99], and the Rutgers PAF toolkit [Rutgers], on outputs of their obfuscator. They also provide a worst-case complexity analysis of their flattening and aliasing techniques against static analysis

In addition to the obfuscation techniques that Wang et al. have implemented, the authors discuss a number of other potential obfuscation techniques. These techniques include multi-threading, variable splitting and obfuscating procedure call and function call interfaces.

8.3 Computing with Encrypted Functions

E-commerce and mobile agent security research has sought a method to compute encrypted functions on the agent platform. Several researchers have sought methods to encrypt a function, send it to a remote location, and execute it such that the execution environment could obtain the result but not know the function. This work is still preliminary: only a few kinds of computations can be done in this way [ST98, SYY99, ACC+01]. The techniques are limited to solving small functions, and some techniques only apply to distributed computations. Those techniques that apply to distributed computations techniques typically rely on on-line trusted servers and have very strong anti-collusion requirements.

8.4 Obfuscation and Decompilation Tools

We examined and tested a number of available Java decompilers and obfuscators, both commercial and freeware, and found plenty of room for future improvement. Some decompilers failed on simple programs. The obfuscators would rename methods and classes, remove debugging information, etc., but most left the code essentially unchanged. A few commercial obfuscators employ simple tactics like constant transformation and loop unrolling.

We conducted a survey of available tools. In the table below, “all” means that the decompiler worked on the entire input, “some” means the decompiler produced output that had some Java source code and some virtual machine opcodes that could not be decompiled, “none” means that the output produced did not even contain some virtual machine instructions, and “fail” means the program did not produce output. All surveyed decompilers were available at no cost (some included source code).

Decompiler	Decompiles Simple program	Output works	Decompiles switchify output			
			support code	blocks	stubs	output works
Jad	All	Yes	Some	Some	All	No
Mocha	Some	No	Fail	Fail	None	No
Jasmine	Some	No	Fail	Fail	None	No
Jode	All	Yes	Some	Some	Fail	No

We surveyed some available obfuscation tools. Most were commercial only, and were not tested. The tested obfuscators only performed simple operations, leaving the bytecodes essentially unchanged in almost all cases.

Obfuscator	Distribution	simple renaming	constant folding	Remove debug info	Anything else
Jode	Java source	Yes	Claimed	Yes	No
Condensity	Commercial, Can buy source license.	Yes	No	Yes	Makes methods final if possible
Crema	commercial with trial	Yes	No	Yes	No
WingGuard	commercial with trial	Yes	No	Yes	No

Jode claimed to perform constant folding, but did not do so with the test programs, which were completely determined at compile time.

9 Conclusion

This document presents work in progress. We have presented a number of techniques that we believe will be useful in obfuscating Java programs. We have not implemented all of the techniques presented here, however we have developed a Java bytecode translation tool, JBET, and implemented several of our control flow techniques. More significantly, we have developed a software foundation for implementing all of the techniques described in this report.

Our experience in implementing JBET has given us valuable understanding of the aspects of the obfuscation problem in Java, and of various tradeoffs that occur naturally when obfuscating bytecodes. Different techniques are appropriate depending on the particular Java features that are being obfuscated. We have explored techniques that are appropriate for the major structural groupings of Java programs: long-term data values, temporary data values, control flow, memory management and type representation. Our preliminary analysis indicates that the techniques presented here can offer considerably more power than those available from the other Java obfuscators of which we are aware. Unlike many obfuscators, we are including techniques that impose runtime and space overheads. This imposes costs, but enables the generation of more obscure executable representations.

The major issue at stake in this research is whether tools based on our techniques can produce obfuscated code that is resistant enough to analysis so that de-obfuscation always requires significant manual analysis (or manual guidance of de-obfuscation tools). If our techniques are sufficient to force a manual component to de-obfuscation, we have provided a positive cost/benefit tradeoff between obfuscation and de-obfuscation since the obfuscation techniques we present here can be automatically applied without human

involvement, and are therefore relatively inexpensive. This cost/benefit relationship could be highly useful for protecting code in environments where code could be frequently (re)obfuscated and run for limited periods of time, such as in mobile agent systems, and smart clients of security-aware servers.

We intend to implement all of the techniques described here in future versions of JBET. While we have explored our techniques in the context of Java, we believe they will also be applicable, with some translation, to other execution environments.

Bibliography

- [ACC+01] J. Algesheimer, C. Cachin, J. Camenisch, and G. Karjoth. “Cryptographic security for mobile code”, Proceedings of 2001 IEEE Symposium on Security and Privacy, Oakland, May 2001, pp. 2-11.
- [AES] “Announcing the Advanced Encryption Standard”, Federal Information Processing Standards Publication ZZZ, U.S. Department of Commerce/National Bureau of Standards, National Technical Information Service, Springfield, Virginia, 2001, <http://csrc.nist.gov/publications/drafts/dfips-AES.pdf>
- [Balzer69] R. Balzer, “EXDAMS --- EXTendable Debugging And Monitoring System”, In AFIPS 1969 Spring Joint Computer Conference, Vol. 34, AFIPS Press, May 1969, *Proc. SJCC*, 1969, pp. 567—580.
- [BGI+01] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S.. Vadhan, K. Yang, “On the (Im)possibility of Obfuscating Programs”, Advances in Cryptology, Proceedings of Crypto'2001, Lecture Notes in Computer Science, Vol. 2139, pp. 1-18.
- [CT00] C. Collberg and J. Thomborson, Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection, University of Arizona Technical Report 2000-0, Feb 2000.
- [CTL98a] C. Collberg, J. Thomborson, and D. Low, “Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs”, IEEE International Conference on Computer Languages, May 1998.
- [CTL98b] C. Collberg, J. Thomborson, and D. Low, “Breaking Abstractions and Unstructuring Data Structures ”, ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Jan 1998
- [CTL97a] C. Collberg, J. Thomborson, and D. Low, A Taxonomy of Obfuscating Transformations , University of Auckland Technical Report #170, Jul 1997
- [ElGamal85] T. ElGamal, “A public key cryptosystem and a signature scheme based on discrete logarithms”, Advances in Cryptology—Proceedings of Crypto' 84 Lecture Notes in Computer Science, Vol. 196, 10–18, 1985.
- [FHS+96] S. Forrest, S. A. Hofmeyer, A. Somayaji, and T. A. Longstaff, “A Sense of Self for Unix Processes,” In Proceedings of 1996 IEEE Symposium

on Computer Security and Privacy (1996).

- [FIPS 46-3] “Data encryption standard”, Federal Information Processing Standards Publication 46-3, U.S. Department of Commerce/National Bureau of Standards, National Technical Information Service, Springfield, Virginia, October, 1999
- [HBC+99] M. Hind, M. Burke, P. Carini and J. Choi, “Inter-procedural Pointer Analysis”, *ACM Transactions on Programming Languages and Systems*, Vol21, No. 4, July 1999, pp 848-894.
- [Hohl98] F. Hohl. “Time limited blackbox security: Protecting mobile agents from malicious hosts”, In *Mobile Agents and Security*, 1419 in LNCS. Springer-Verlag, 1998, pp. 92-113.
- [Lai01] H. Lai, A comparative survey of Java obfuscators available on the Internet, 415.780 Project Report, February, 2001.
- [LR91] W. Landi and B. G. Ryder. “Pointer-induced aliasing: A problem classification”, In *Proceedings of the Eighteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 93--103, Orlando, Florida, Jan. 1991.
- [LV00] A. K. Lenstra and E. R. Verheul, “The XTR public key system”, *Advances in Cryptology, Proceedings of Crypto'2000, Lecture Notes in Computer Science*, Vol. 1880, Springer-Verlag, 2000, pp. 1-19.
- [LY99] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification Second Edition*. Addison Wesley, 1999.
- [PH78] S. Pohlig and M. Hellman, “An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance”, *IEEE Transactions on Information Theory*, 24 (1978), pp. 106–110.
- [RC94] P. Rogaway and D. Coppersmith, “A software-optimized encryption algorithm”, R. Anderson, editor, *Fast Software Encryption, Cambridge Security Workshop (LNCS 809)*, Springer-Verlag, 1994, pp. 56–63.
- [RSA78] R. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems”, *Communications of the ACM*, 21 (1978), pp. 120–126.
- [RS98] J. Riordan and B. Schneier, "Environmental Key Generation Towards Clueless Agents", G. Vinga (Ed.), *Mobile Agents and Security*, Springer-Verlag, Lecture Notes in Computer Science No. 1419, 1998, pp. 15-24.
- [Rutgers] The Prolangs Analysis Framework (PAF). Rutgers University.
<http://www.prolangs.rutgers.edu/public>
- [ST98] T. Sander and C. Tschudin, "Protecting mobile agents from malicious hosts", in *Mobile Agents and Security, LNCS 1419*, G. Vigna (Ed.), Springer-Verlag, 1998, pp. 44-60.

- [SYY99] T. Sander, A. Young, and M. Yung, “Non-interactive CryptoComputing for NC¹”, Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS), 1999.
- [Tip95] F. Tip, “A Survey of Program Slicing Techniques”, Journal of Programming Languages, 3(3):121--189, Sept. 1995.
- [Wash01] L. Washington, October 2001, personal communication.
- [Weiser84] M. Weiser, "Program slicing", IEEE Transactions on Software Engineering SE-10(4) pp. 352-357 (July, 1984).
- [W00] C. Wang. A Security Architecture for Survivability Mechanisms. PhD thesis, University of Virginia, School of Engineering and Applied Science, October 2000.
www.cs.virginia.edu/ survive/pub/wangthesis.pdf.
- [WDH+01] C. Wang, J. Davidson, J. Hill, J. Knight Protection of Software-based Survivability Mechanisms International Conference of Dependable Systems and Networks, Goteborg, Sweden (July, 2001)
- [WHK+00] C. Wang, J. Hill, J. Knight, J. Davidson, Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report CS-2000-12, Department of Computer Science, University of Virginia, 2000

Glossary

block: A piece of code with a start point and a finish point. There cannot be any jumps from other code to the middle of a block, or any jumps out of the block except at the end.

chunk: An allocable piece of memory. User objects are built up from chunks. A chunk could be a synthetic object or a range of an array, or something else.

complex pointer: A user pointer that contains more information than just the location of what it points to. For instance, a complex pointer may include the type of the object it points to.

entry point: A method implemented internally that can be called from external code.

external code: Code that has not been obfuscated.

exit point: A place where internal code must call external code, e.g.,
`System.out.println("Hello World")`.

GC: Garbage collector.

internal code: Code that has been obfuscated.

original object: An object from the original program.

representation parameter: A parameter that determines how a variable is stored

runtime type information: This is a blanket term for any functionality that needs to know what the type of an object is at runtime, for instance: virtual method calls

and type casts. Use of the `java/lang/reflect` package is runtime type information, but we will call that **reflection** instead, since the typical Java program uses only casting and `instanceof`.

simple pointer: A variable or a value that refers to a chunk. Depending on how chunks are implemented, an apparent pointer may be a real java pointer or it may be an integer index into an array.

synthetic object: An object in the output program, i.e., something with a `.class` file.

user pointer: The representation of a pointer from the original program in the output program.

user object: The representation of an original object in the output program

value convolution: In order to determine the meaning of data, an attacker may focus his attention on the **values** a program computes, rather than the variables the program stores them in. In this case copying data around in order to obscure what it is, where it came from and where it's going does little good. For this reason it is imperative that the program routinely uses completely unrelated data in its computation, so it is difficult to determine which inputs to a value are really salient. This strategy is called value convolution.