Self-Protecting Mobile Agents Obfuscation Report

Final report

Larry D'Anna
Brian Matt
Andrew Reisse
Tom Van Vleck
Steve Schwab
Patrick LeBlanc

Network Associates Laboratories

Report #03-015

June 30, 2003

Abstract

This document describes our investigation into software obfuscation for building Self-Protecting Mobile Agents (SPMA).

The original goal of the SPMA project was to develop automated tools to protect mobile agents from attacks by malicious hosts. In development of those tools, we realized obfuscation could not be relied upon to give a reasonable amount of security. Because of this, we redirected the SPMA project to studying obfuscation.

Our conclusions include theoretical results about obfuscation and evidence that supports those results. Our most important conclusion is that there is no general obfuscation problem (i.e. a definition and theory of obfuscation that will always apply). We believe that all automated obfuscation is merely emulation; this will certainly be an area of future research.

We conclude that if software obfuscation is to be useful, it must be employed for a specific purpose (not "obfuscate any program protecting all information"), and use fundamentally new ideas. Future theoretical work on obfuscation will have to define it clearly, and use a restricted set of programs, so that the result of Barak *et al.* [BGI+01] does not apply.

In the course of developing obfuscation tools, we evaluated the properties of programming languages under several obfuscating transforms, concluding that strict type-safe programming languages were the best for obfuscation. In addition, programs specifically designed to be obfuscated will give better results, as the programmers will avoid implementing unobfuscatable constructs.

1 Introduction

Mobile agent technology has the potential to revolutionize network software, but mobile agent technology is fundamentally security-limited. Mobile agents could provide key information capabilities, such as autonomous global searching, information filtering, distributed sensing, price shopping, active networks, micro-transactions, manufacturing,

and large-scale system configuration. As currently conceived, however, many mobile agent solutions cannot be adopted because of security threats: hosts and networks must be protected from malicious agents; agents must be protected from other agents; and agents must be protected from malicious hosts. Of particular concern is that many agent applications require agents to execute on untrusted hosts that have an economic interest in modifying agent behavior or stealing secrets (like credit card numbers) carried by agents. While substantial progress has been made in protecting hosts from agents (e.g., sandboxing [GMP+97], software fault isolation [WLA+93], proof-carrying code [Nec97], operating system access controls), and in protecting agents from other agents (agent separation implemented on hosts, defensive agent KQML interfaces), protection of agents from malicious hosts remains a major problem for agent technology. The key question is: How well can an agent be protected when it is running on a malicious host? If mobile agents cannot be protected, of what value are they?

Computer security studies methods to ensure that a computer system will behave the way the operator wants it to, instead of the way somebody else wants it to. Mobile agent systems (and obfuscation in general) want to achieve the opposite: ensuring a computer system will behave the way the programmer wants it to, instead of the way the user wants it to. In order to derive security results, we make assumptions about the computers, networks, and software involved. Traditionally the most basic of these assumptions is that software of our choice is running on our computer, and that we trust the hardware and all the lower-level layers of software, such as operating systems. Mobile agents cannot trust any of those.

Our research initially aimed to provide a technical basis for building trustworthy agents that perform their missions with confidence even though they sometimes execute on untrusted hosts. Several key requirements must be satisfied for agent systems to realize their potential:

High Mobility. Agents must be free to migrate to, and execute on, a wide variety of hosts that are unknown to the users who launched the agents. Without such mobility, agents will be unable to perform the searching and commercial operations often envisioned for agent technology.

Detached Operation. Agents must operate autonomously, without the need for constant communication with users, and, preferably, without constant communication with trusted infrastructure elements which may or may not exist.

Extended Deployment Periods. Agents must function for extended periods of time, thus allowing users to launch long-term "watcher" agents, that take action only if specified criteria have been met, and other long-term service agents.

Safe Execution. Agents must be free from integrity attacks conducted by malicious hosts or other agents, and must be protected from faulty execution or non-execution by malicious hosts. Agents will also be much more useful if they can carry secrets (such as cryptographic keys or user decision information, such as how much a user would be willing to pay for merchandise).

Realistic Infrastructure Requirements. Agent properties must not rely on unrealistic infrastructure assumptions, such as the assumption that all hosts are trustworthy, that implementation algorithms will remain secret, or that every agent execution environment is implemented by a tamper-resistant hardware peripheral.

These goals cannot be simultaneously met using current technology.

1.1 Technical Approach

We set out to develop tools that translate an individual software agent into a distributed set of tamper-resistant agentlets that is never entirely vulnerable to a single host, and that can detect and recover from compromise of a subset of its elements, providing strong protection by combining three core techniques:

Distributed Agent State. Each agent would be partitioned into a set of communicating programs (agentlets) executing on independent hosts. Critical information would be spread across the agentlets, thus limiting exposure to any proper subset of the hosts.

Obfuscation with Periodic Regeneration. Each agentlet's code and data were to be obfuscated using a variety of techniques (e.g., randomly selected, but equivalent, algorithms and data representations). We initially believed that obfuscation could delay, but not prevent, subversion of agents via reverse engineering. Consequently, we planned to have agentlets periodically expire and be replaced by differently obfuscated versions so that a successful attack on an agentlet was impossible before it expired. Regeneration was going to use information from multiple agentlets (hosts), and hence not be vulnerable to reverse engineering by any single host.

Monitoring and Recovery. Agentlets would be made self-monitoring and able to monitor other agentlets. Using challenge/response techniques, agentlets could automatically exclude compromised agentlets, report the identities of tampering nodes, and replace lost agentlets.

1.1.1 Original Tasks

Our initial plan was to carry out this research in three tasks:

Task 1: Develop source-code translation tools to convert an individual software agent into a set of replicated communicating **agentlets** that collectively manage their navigation to avoid dependence on some hosts that may be colluding, as specified in an agent security specification describing the agentlet protection policies.

Task 2: Develop powerful object-code obfuscation tools that can be employed to protect the code of an agentlet from reverse engineering for some minimum time, in order to prevent reverse engineering and ensure that the obfuscation process itself was protected from tampering. Ensure that any progress an

attacker made in reverse engineering an old version of an agentlet conferred no useful information about reverse engineering a subsequent version of it.

Task 3: Extend agentlets to employ fine-grained monitoring for tamper detection, and to use mutually-suspicious agreement protocols to identify and exclude potentially subverted agentlets, thus providing a basis for a community of agentlets to cooperatively protect themselves from host-based attack. Perform periodic re-obfuscation of all agentlets often enough that an attacker would be unable to ever get enough information to interfere with the agentlets.

1.2 Accomplishments of SPMA Project

We carried out and reported on the first two tasks, as follows:

1.2.1 Agent division

The team built and demonstrated an application of IBM's freely available Aglets agent system technology (www.aglets.org) that divided a mobile agent into a group of agentlets that cooperated to produce the result of the original agent. The Self-Protecting Mobile Agent toolkit takes the binary of an existing mobile agent, and with the help of an input policy and a library of mobile agent helper functions, transforms it into an equivalent collection of cooperating agentlets.

The transformation of an existing agent into a collection of cooperating agentlets was achieved by exploiting the fact that agents in Aglets and other agents system have life cycles. The life cycles delineate creation, cloning, dispatching, and messaging phases in the agentlet. By augmenting and controlling the code that initiates and handles these phases, we created the means to change an agent into a collection of cooperating agentlets, as described in our Architecture Report [BMK+01].

1.2.2 Obfuscation

The team built a modular framework for manipulating Java bytecode, and wrote several powerful obfuscation operators for it. We 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, JBET (Java Binary Enhancement Tool), that supports the obfuscation techniques they describe and additional obfuscation techniques proposed by us. Our tool and its techniques were described in our Obfuscation Techniques Report, [BDM+01].

JBET is described in Appendix A.

1.3 Weaknesses in the SPMA Approach

To convince an agent creator to trust the system, we would have to provide an argument that the obfuscation was guaranteed unbreakable for some length of time, and that re-obfuscation would occur before the agentlets' protection could be defeated. The argument should include explicit assumptions about the resources an attacker might employ to

overcome the protection, and the scalability of the system's protection with respect to number and size of agentlets, number and power of execution nodes, and communications bandwidth. The analysis should be clear about what is assumed to be secret: a system that depends on keeping the method of obfuscation secret from the attacker has more vulnerability than one that assumes the attacker may know the obfuscation method, but not the particular keys or seeds used to select the particular obfuscation.

As we prepared to implement Task 3 of the original plan, we found several problems. The most serious problem was that we could not find a way to guarantee that any obfuscation method could resist deobfuscation for any specified minimum time. We also realized that there are several attacks against the agent system that give the attacker more time to deobfuscate and more ability to observe the agentlets than we had originally thought.

1.3.1 Breaking Obfuscation

We discovered a number of reasons to believe that obfuscation might not be an adequate protection mechanism for agentlets.

In order for our scheme to be certain to work, we had to be able to ensure that an attacker could not deobfuscate an agentlet for some safe period given assumptions on the computing power available to the attacker. Without a proof that the agentlets were safe, the whole protection methodology reduced to wishful thinking.

In an important paper at Crypto 2001, Barak et al. [BGI+01] discussed the impossibility of obfuscating programs. Their result showed that obfuscation was mathematically impossible, for a specific formulation of the problem, and raised the question of whether the methods we planned to use could work. Some theoretical claims by other researchers that certain obfuscation methods were NP-hard appeared to be based on incorrect reasoning [Schwab03].

We investigated the strength of our own obfuscation methods by building tools to deobfuscate code obfuscated by our most powerful methods, and discovered that it was disappointingly easy. Section 3 of this report describes these experiments. Further study of obfuscation methods proposed by others showed that a determined adversary could probably build similar deobfuscation methods for all such obfuscators.

Most proposed obfuscation, including the methods we proposed, defends against static analysis of the obfuscated program. When we considered dynamic analysis of programs, often used in practical reverse engineering, we found that most obfuscation methods provided little protection.

1.3.2 Rerun Attacks

Our initial agent-obfuscation scheme assumed that periodic re-obfuscation of the agentlets would prevent attackers from learning anything about the obfuscated program. Further consideration found some attacks on this scheme that we would have to provide defenses for.

Since a program is ultimately controlled by the hardware it runs on, and COTS virtual machines and simulators are readily available which can do things like save machine state to disk, it is possible to isolate a program and replay execution scenarios as many times as is necessary to understand the program or find its weaknesses. Obfuscation cannot prevent the program from being reverse engineered this way, however, it can prevent a malicious host from observing or predictably tampering with code and data in the running system.

Depending on the obfuscation used, the locations of the code and data the attacker is interested in could be different between versions. Assuming an attack could not happen within a specific time of obtaining an agentlet, the attacker would always have to study a version that is obfuscated differently than what they would eventually attack. If the attacker's goal is to subvert a running system, reverse engineering through rerun attacks would allow them to discover whatever vulnerabilities the system had. If their goal is to discover important algorithms, constants, or data that do not vary much, they could do that as well.

Although such rerun attacks were not judged fatal to our scheme, they represented an additional issue that the agentlet coordination protocol would have to cope with.

1.4 Revised Research Program

We discussed the problems with our initial plan with our Program Manager, and decided to redirect the SPMA project to focus primarily on obfuscation issues for the last year, rather than mobile agent issues, without changing the project name. This redirection was motivated by our realization that the mobile agent issues are tractable whereas the obfuscation issues have the potential to be show stoppers, and that not much work has been done to settle the big questions regarding obfuscation. Much of the community's wisdom on obfuscation is *ad hoc*. We believe that the most direct path to validating the Self-Protecting Mobile Agents architecture is therefore to focus on solid results regarding obfuscation strength when it is used as a defense against both static and dynamic analysis.

We chose to orient this project toward answering the question, "when should programs be obfuscated?" If a program creator asks whether to obfuscate a program for the purpose of protecting some specific secrets or behavior, we should be able to advise him or her, based on the type of attackers expected and the kind and length of protection desired. We and the program manager felt that answering this question would be a very valuable research contribution.

We identified three large unknowns with respect to obfuscation:

- How to measure and understand the strength of an obfuscation method.
- How much work can be imposed on the attacker, per unit of work by the defender.
- The possible variation between different applications of a single obfuscation. Put another way, if an attacker has deobfuscated a particular obfuscation of a program, how can we measure the advantage, if any, that this conveys to the

attacker when attempting to deobfuscate another obfuscation of the same program, or an obfuscation of another program by the same obfuscation tool.

We investigated these issues, conducted experiments, and present our results in this report.

2 New Problem Definition

Obfuscation transforms a program into another program that has equivalent behavior but which is harder to understand. This report investigates the strength of the protection provided by obfuscation.

Obfuscation has been proposed as the solution to problems such as protection of transient secrets in programs, protection of algorithms from use except in controlled ways, protecting protocols from spoofing, license management for software, temporary protection of digital watermarks in programs, software-based tamper resistance, and protection of mobile agents.

Many have said that "security by obscurity" is not a solution to a security problem. The requirement that an algorithm's secrecy is not required for the security of the cryptosystem is known as Kerckhoffs' Principle[1]. Commercial obfuscation tools will be purchased or otherwise obtained by attackers, reverse-engineered, and then all the users of that tool will be compromised if it required secrecy of the algorithms; a key is required for obfuscation as well.

Barak *et al.* [BGI+01] have shown that *ideal* obfuscation is "impossible." That is, they have shown that there is no obfuscation method that always yields an obfuscated program that reveals nothing about the original. Ideal obfuscation is more than is necessary for many useful applications: an obfuscation method that raises the cost of reverse engineering sufficiently would adequately deter attacks on low-to-moderate value programs, and delay attacks on high-value programs. For some applications, such as mobile agents, even a modest delay could be instrumental to a system's survival.

On the other hand, most currently available obfuscation practices provide protection that can be quickly and automatically broken. What we seek is a framework that allows us to reason about the strength of various kinds of obfuscation methods and the protection provided.

The problems of protection of transient secrets, algorithms, and other phenomena within a program exist because conventional wisdom says that no guarantee can be made about the execution of a program when nothing can be guaranteed about its environment; it may be altered, debugged, traced, lied to, or rerun. In short the problem is to restrict how a program can be used, while still allowing the uses that he creator wants. One proposed solution to this class of problems is software obfuscation. In general, software obfuscation refers to any technique for making software hard to understand or manipulate. The idea is that although attackers are able to change memory, trace

[1] Named after Auguste Kerckhoffs, 17th century author of *La Cryptographie Militare* (Military Cryptography).

execution, and otherwise manipulate the environment, they are not able to get any meaningful result from these techniques because they cannot determine what any particular byte of memory means, or what the program is doing at any particular point. The goal is to make the program so difficult to understand that the attacker will not make the changes needed to misuse the program, or extract the information he wants.

It should be noted that obfuscation should be seen as a proposed solution to a class of problems, not as a problem itself. One might propose a "general obfuscation problem", i.e. to find an obfuscation transform that is universally applicable. Such a transform would be able to protect any input given to it. The resulting obfuscated program would be such that the only useful thing to do with it would be to run it. Such transforms do not exist, as Barak *et al.* prove [BGI+01]. This does not necessarily mean that obfuscation is always useless, but it does mean that we must consider the security of each use case separately, carefully specifying what we wish to protect.

2.1 Overview of the Problem

We are considering obfuscation techniques that read in a program *P*, and an obfuscation policy *p*, then automatically generates a new, *obfuscated program OP*.

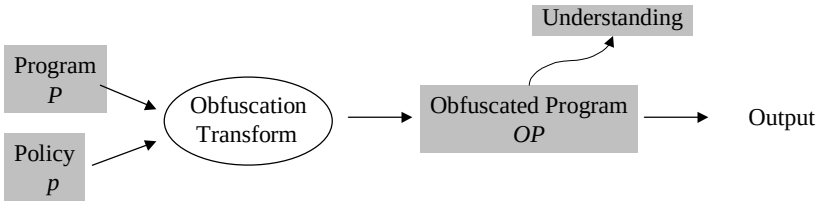

Figure 1

Figure 1 displays the concept: both the original program and the policy are fed into a transformation procedure that generates the obfuscated program. According to [BGI+01], after some period of time and expended effort, an attacker can gain some understanding of *OP*. It has been postulated that the program can run safely for a limited time. [Hohl98] In order to rely on obfuscation, we need to reason about how long the obfuscation can be trusted to protect the program.

The original SPMA design required 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.[2] If the interface is chosen reasonably, *OP* can be used wherever *P* can be used.

[2] For example, if *P* issues a write() call to output 100 bytes, *OP* in some cases could issue two write() calls

Property 2 asserts that knowledge of *p* is necessary for an observer to understand, without deobfuscating *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 deobfuscate 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 with acceptable performance, 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.[3]

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

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 it 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*.

---

of 50 bytes each.

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

The attacker has source code for *P*. Of course, if the objective is to obtain an algorithm or stored constant, there is no possible defense if the attacker has the source code. However, if the objective is to subvert a running system, the attacker must also understand *OP*. 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 can probably obtain all of the input data by 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 properly 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 deobfuscate *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 (deobfuscating) tools.

We investigated available freeware and commercial Java obfuscators and decompilers. The obfuscation methods described in our Obfuscation Techniques Report [BDM+01] are substantially more complex than most currently available. This comes with a cost: the output of most existing obfuscators are smaller than their input, and most produce output programs that execute at more or less the same speed as the original. JBET-based obfuscation attained substantially more obfuscation by abandoning this constraint.

2.1.1 Components of the Obfuscation Relationship

A **defender** Alice has a **program** P that she wants to distribute to one or more untrusted parties. She wants these parties to be able to run the program, but she has a set of security constraints as well, i.e. actions that she does not want attackers to be able to perform given OP. **Obfuscation** is the class of techniques wherein Alice applies a behavior-preserving transformation to P prior to distribution, in order to enforce her constraints.

This definition is extremely general, and we believe it accurately describes all the techniques that could reasonably be considered obfuscation. Alice's constraints may or may not be formal. For example, if P has a secret input, and the constraint is that the recipients of OP must not be able to determine that secret input, then the constraint is formal. If P has a secret algorithm that the recipients must not be able to "understand," then the constraint may depend on an informal definition of understanding. Our definition also allows for the domain of allowable programs P (for a given obfuscation technique) to be limited. For instance, Alice may only be interested in obfuscating circuits, or programs including some sort of annotation, or programs without while loops (i.e. primitive recursive programs). The definition also does not specify what other knowledge the attackers are given, i.e. whether or not they know some or all of the original program P, whether they know what obfuscation technique Alice is using, etc.

Because of the generality of this definition, it is useful to consider more restrictive frameworks for analyzing obfuscation. For instance Barak *et al.* only consider the situation where Alice's constraint is the confidentiality of certain data determined by the function P computes. This framework has the advantage that the criteria for success can be described very precisely in terms of complexity theory, and so it is possible to prove theorems about it.

The **attacker** can run the program and examine it, with the intent to extract a secret or modify the program's behavior.

Often we speak of **obfuscated data**: for instance, a shopping agent might know its originator's credit card number, and use it under certain circumstances, while making it

hard for an attacker to extract that number; we say the credit card number is obfuscated. In general "obfuscated data" is just another way of saying that an obfuscated program has an input that is given to it before it got to whoever is running it now, and the originator wants that input to remain confidential (except for whatever can be learned about it given black box access to the obfuscated program, of course).

The **secret** is what the defender is trying to hide: it may be a data value in the program, or what the program will do in given situations. Crypto keys, license status, credit card numbers, and bidding limits are examples of the first type. The second type includes algorithms used to calculate output, or behaviors of the program: for example, one might wish to discover what an obfuscated virus might do when triggered.

Some use-cases do not seek to protect a secret *per se*, but to enforce a more general constraint. For instance the copy protection of a game might want to prevent the game from being run without the CD it was distributed on being in the computer. It could try to enforce this by checking sub-channel data on the CD. An attacker could emulate the machine the game runs on, so the game might run a series of tests to determine if it is running under emulation. The goal of obfuscation in this case would be to prevent the attacker from running the game without the CD, given that the attacker does not have a perfect emulator or enough time to write one, not to protect any specific data. Of course this situation is very informal, but it is no less valid for that. The success of obfuscation in such a situation could possibly make a real difference to the game company's bottom line.

The defender creates the **unobfuscated**, or original form of the program. This is the representation that the program to be obfuscated is in before obfuscation techniques are applied. It should include all information that is needed to run the obfuscator; for instance, if annotations are required then they are part of the unobfuscated form of the program. In other words, we subsume any hand-done parts of the obfuscation into the act of writing the program. Note that some obfuscators may have special requirements on the unobfuscated from (if it's in Java, the use of reflection may be disallowed, etc).

The **Obfuscated form** is the representation of the program that is actually distributed to possible attackers. It is produced by running the unobfuscated form through an obfuscator.

The **execution environment** for a program can be described as an abstract machine. This may be a virtual machine environment such as the Java Virtual Machine, or a particular combination of hardware and operating system that supports the execution of the obfuscated form of a program, or a language interpreter such as Perl. Typically Alice will expect the non-attacking recipients of her program to use a specific execution environment (such as the Perl interpreter), or an environment that meets a specific specification (such as a Java Virtual Machine), but the attacker is of course free to use whatever environment he wants. He may even choose not to run the program at all, but to do some sort of inspection or static analysis on it instead.

The **behavior** of a program on a given input is a transcript of its interaction with the execution environment when it is run on that input. For example, if the program is represented as a Turing machine, the behavior is the output produced, along with the run time. If it is a Turing machine relative to an oracle, then the behavior also includes all the

oracle queries that were made. If the program is represented as a Java class, the behavior is all of the library calls made. The behavior of the program, then, is just the collection of all of its behaviors on specific inputs. When we say two programs have the same behavior, we allow for polynomially bounded change in runtimes. Note that an attacker can query the behavior for any given input, but in general cannot "know" the total behavior of the program; in fact, the secret to be protected is often determined by the total behavior.

2.1.2 Deobfuscation

Informally, **deobfuscation** is whatever the attacker is trying to do, i.e. if the attacker succeeds at deobfuscating the obfuscated form, then obfuscation has failed. Thus the task of deobfuscation is to discover the secret given the obfuscated form, or more generally, to violate the security constraints.

This definition may seem somewhat strange at first: if obfuscation is some sort of compilation, or translation process, then the natural inverse of obfuscation should be some sort of decompilation that should produce the original source code that was fed into the obfuscator. We will call the act of producing the unobfuscated form given the obfuscated one **source-recovery**. Using source-recovery as a definition for deobfuscation has several problems, however. First of all, it is never possible to fully recover the source code, because things like variable names are irrevocably lost. Thus in order to say anything formally you would have to specify exactly what data about the unobfuscated form should be recovered. It is difficult to make precise statements about source-recovery that correspond correctly with our intuitive ideas on the subject. The other problem with source-recovery as a standard for deobfuscation is that it addresses attacks against a particular obfuscator in general, not against that obfuscator being used to protect a specific type of secret. In other words source-recovery is too coarse: it cannot look at each use case separately, which is what we want to do.

All this is not to say that source-recovery is not a useful concept, only that we will not use it as our standard for attacker success. Intuitively, source-recovery does totally undo obfuscation, in the sense that if you can do source-recovery, then deobfuscation is just as easy as if the program were never obfuscated in the first place. Therefore if you have a use-case, and source-recovery does not imply deobfuscation in that use case, then there is something else going on other than just pure obfuscation. Perhaps the attacker's problem was impossible to begin with. For instance, a virus scanner that can determine if a program ever writes to the boot sector is impossible, whether the viruses are obfuscated or not.

2.1.3 Success

An attacker **succeeds** in deobfuscating a program by discovering the secret that the defender wishes to protect. In the case of a data value hidden in the program, this is straightforward to determine. If obfuscation is used to ensure that the attacker cannot find where to patch a program in order to change its behavior, as is often done to support licensing schemes, then the attacker can succeed without full understanding of the program. On the other hand, if obfuscation is used to hide what a program might do, then deobfuscation succeeds only if it can produce a complete account of the behavior of the

program. For example, an obfuscated virus might contain hidden time triggers or the ability to execute arbitrary commands sent from outside, and deobfuscation is only successful if it describes all such features.

Some writers on obfuscation define success at deobfuscation as producing exactly the original program. This is an unnecessarily strong requirement: many programs are trivially equivalent, say by substituting one equivalent opcode for another, and any member of this set of equivalent programs should be accepted. Another approach [CW00] defines deobfuscation success as determining which basic blocks in an obfuscated program are never reached.

2.1.4 Work factor

By analogy with cryptography, we would like to compare different methods of obfuscation by estimating the amount of **work** to deobfuscate an obfuscated program. The usual assumptions are that the attacker knows what algorithm was used to obfuscate a program (since otherwise one is relying on security through obscurity) and that the only unknowns are the parameters to obfuscation, which we have lumped into the policy.

We speak of a work *factor* to emphasize that the cost of deobfuscation is a relative measure that does not depend on CPU speed or implementation efficiency, but rather on the intrinsic properties of the obfuscation algorithm and the deobfuscation process.

If deobfuscation has a high enough work factor, i.e. can be made costly enough, it can be a useful method of information protection.

For SPMA, we desired to protect obfuscated agents for some "time," however measured. As we shall see in section 6.1.7, the Barak *et al.* paper [BGI+00] shows that this cannot be done in the general case.

2.2 Use cases

This section describes cases where systems may wish to make use of obfuscation and describes the problem to be solved from the point of view of the system implementer.

2.2.1 Mobile Agents

As previously stated, the original goal of the SPMA project was to produce tools for protecting mobile agents running on untrusted computers. Our proposal for a secure mobile agent system involved several components: obfuscation, running on multiple non-colluding hosts, and periodic movement among available hosts. Only the use of obfuscation is discussed here.

2.2.1.1 Problem

A developer wants to distribute a program that will run on an unstructured network of mobile agent hosts. The hosts are untrusted by the developer, but he still wants to ensure that the program operates correctly. The program may also have secret data to conceal. Interaction with the host system in a way other than as a processing resource and data

storage device (for example, accessing public data on the host) is unspecified. Protection against well-trained attackers possessing major resources is desired.

2.2.1.2 Attack

The attacker wants to either obtain secret data from a part of a mobile agent that is running on his host, or to modify the behavior of the mobile agent.

2.2.1.3 Use of Obfuscation

Obfuscation can make programs harder to understand, and make special-looking data, such as encryption keys or credit card numbers, more ordinary. This is particularly true when no user (or "plaintext") communication is involved. In addition, if the valuable duration of the data in the mobile agents is short (max bid for an auction that ends in 2 days), obfuscation increases the likelihood that its usefulness would expire before the attacker could learn it.

2.2.2 Standalone Software Copy Prevention

2.2.2.1 Problem

A developer distributes a software program with the intent of allowing only particular uses of the software. One desirable constraint might be the software should not function if installed on additional computers without intervention from the distributor. The software cannot require special hardware or "call home" except at installation. A variant is a time-limited version that refuses to run after a certain date.

2.2.2.2 Attack

The attacker wants to make a new program distribution that can run on any compatible computer and function unhindered. We will assume that the attacker cannot simply redistribute the installer; e.g. perhaps the installer sends system specific information to a network server and it supplies the program's license file. We will also assume that the attacker has one functional copy of the program.

2.2.2.3 Problems With Obfuscation

If the attacker has a working copy, obfuscation cannot completely prevent him from distributing it. As with the rerun attack described earlier, he could install the software in an emulated environment (e.g. Connectix Virtual PC or VMware) and distribute that without having to understand or even look at the obfuscated code.

A more elegant solution would be to trace the system configuration calls the program makes, and prepare a front-end to the program that reports the same system configuration data (hence making the system-dependent license usable on every machine), no matter what computer the program is subsequently executed on. The executable could also be modified such that the system calls that get the configuration data are replaced with static data.

2.2.2.4 Use of Obfuscation

Although obfuscation cannot absolutely prevent distributing copy protected software, a practical goal of copy protection is simply to raise the cost of copying it to the point that more people would rather buy it.

Emulation attacks make it impossible to achieve software-only copy protection in general, but emulation is often significantly slower than running a program directly. This means that in order to fool a program into thinking it is running in its original environment, one must either make many complex changes to the OS to systematically lie to programs it runs (which would be a rather large development project) or accept the cost of emulation (which can be significant).

An approach hackers often use is to try to find the right instruction to modify in a program to disable or bypass its protection scheme. Obfuscation may be used to make that task more difficult by concealing conceal one or more pieces of tamper checking code as described in Horne *et al.* [HMST01]. Delaying the defeat of a protection scheme in this way may generate enough sales to compensate for the cost of developing the obfuscator.

Smart cards used in e-commerce applications compute a crypto function of their inputs and a secret personalization on the card, which can be used to verify that the card was present. The software on the card and the card reading software could be obfuscated to raise the cost of attacks on the system, although memory constraints on smart cards may limit the usage to obfuscation methods that do not drastically increase the size of the code.

2.2.3 Viruses

2.2.3.1 Problem

A developer prepares a virus with some spreading mechanism and payload (search for financial data, delete data, etc.). He wants to conceal the spreading mechanism and more importantly, the payload, from analysis. Viruses will invariably be analyzed at some point, so general “analysis resistance” is useful to them.

2.2.3.2 Attack

A network security engineer discovers a host misbehaving, and guesses that a malicious program was installed. He wants to detect the presence of the virus on other hosts, remove the virus, and/or discover what the virus payload will do and when.

2.2.3.3 Use of Obfuscation

Obfuscation attempts to remove structural information from a program, making analysis of the program more difficult. This is especially true when the information being concealed is only valuable for a short time, as is usually the case with viruses. Even poor obfuscation will slow the attacker down and allow the virus to spread further before virus scanners are updated to deal with it.

2.2.4 Algorithm Hiding

2.2.4.1 Problem

A developer wants to distribute a program that implements some algorithm. He does not want others to discover that algorithm. Commercial software is usually distributed in executable form only, so that competitors cannot easily determine how it functions in order to make a compatible product. However, good automated reverse engineering tools exist that produce more readable results with executables from common compilers.

The attacker can do two things to obtain the technology. We could assume that a competitor would know a good deal about what it does and probably some of its input/output characteristics. One option is to use slicers and profilers to isolate the algorithm, then extract it, and use it in their own program without having to know how it works.

The other option is to learn how it works in order to replicate the technology. An important feature of this use case is that human understanding is the goal.. The task of "understanding" an algorithm cannot be defined mathematically and cannot be done by a computer. Therefore this use case cannot be fully analyzed mathematically, and lower bounds cannot be put on the difficulty of deobfuscation.

As mentioned above, tools (decompilers in particular) can help a person deobfuscate by translating the program into a more readable form. A partial analysis of this use case can be made by analyzing the extent such decompilers can recover data about the source code. Formally, if **O** is the obfuscator, **d** is a function on source code, and **D** is a deobfuscator then we want **d(p) = d(D(O(p)))** for some set of programs **p**. One could then investigate for which functions **d** a corresponding deobfuscator **D** exists. The "closer" **d** is to being the identity, the worse the obfuscator **O** is. One might hope to prove that for a particular obfuscator such functions **d** are limited to a class that would not be very useful to a deobfuscator, however this would not constitute a proof that it is hard to understand a secret algorithm given the obfuscated code.

2.2.4.2 Use of Obfuscation

A running joke is that obfuscation is the natural state of programs because analyzing a program without prior knowledge of its internals is so difficult. Potentially, obfuscation could make program analysis even more difficult.

2.2.4.3 Problems With Obfuscation

Because a human will perform this analysis, it is impossible to guarantee any kind of time or complexity bound on the analysis required for a particular obfuscation technique. Obfuscation for algorithm hiding does not require the attacker to solve a new kind of problem, as other uses of obfuscation might. The problem the attacker faced if obfuscation was not used was program understanding. The problem the attacker faces when obfuscation is used is also program understanding.

However, if the attacker's goal is to simply use the algorithm and not understand how it works, obfuscation does present the attacker with a new problem. Good programming

practices result in highly modular, portable, structured code, which makes it fairly trivial to extract and reuse. Among other things, obfuscators can create code and data dependencies to blur the lines between modules. The attacker is now faced with the genuine problem of extracting the algorithm where before it was trivial.

2.2.5 Summary

These use cases were selected because we know them to be common uses of obfuscation, not because obfuscation is well suited to those tasks. When determining whether to use obfuscation in a specific project, it is important to consider several factors specific to the project, including:

- the valuable lifetime of the data or algorithm to be obscured,
- an attacker's likely goal,
- the type of analysis the attacker is likely to use, and
- whether obfuscation can be combined with other techniques to increase the total system security.

Mobile agents (assuming time-assured obfuscation is available) are a good example of obfuscation used as part of a security system.

3 Our Experiments

Before redirecting SPMA to study obfuscation, we planned to build a useful self-protecting mobile agent tool. This tool would have consisted of an agent-splitting component and an obfuscation component. During the development of the obfuscation tool, we became concerned that the obfuscation techniques would provide little protection, and developed a deobfuscator to test that hypothesis. This section describes the obfuscation and deobfuscation software we built.

3.1 Obfuscator

We implemented an obfuscation tool that worked on Java binaries. To discuss the obfuscation tool, we will divide its processing into several categories: code and data storage, control flow, and interfacing with the system environment. Because of the restrictions of the Java environment, we have focused on breaking down semantically rich Java structures such as method invocation, virtual method dispatch, exception handling, data representation, garbage collection, and object structures. A program obfuscated using our JBET-based obfuscator looks entirely different from the original when viewed in terms of these structures because the high-level, self-describing structures found in the Java class files are synthesized using lower-level primitives. The lower-level structures then use flattened control flow (similar to, but independently of, that proposed by Wang's group [WHK+00]), and a variety of obscure data representation approaches. We use lightweight (and weak) techniques (e.g., offsets, value rotation, x*n mod 1, register splitting, table replacement, XOR with various constants) for obscuring heavily used but low-level temporary variables such as loop indices that must be quickly

manipulated. We could use stronger, more computationally intensive techniques (e.g., DES, parse tree representations, Chinese Remainder Theorem, permutations, parameterization) for obscuring longer-lived, semantically more important data structures of a program. Whenever possible, libraries were obfuscated along with the input program, in the same way had the libraries been submitted along with the input program. Most of the development time went into reimplementation of language features (but in an obfuscated way): method call stacks, virtual tables, object memory layouts, etc. It was only after the obfuscation tool was implemented that we realized it was easy to reverse. What follows is a summary of the feature set we implemented.

3.1.1 Java Binary Manipulation

The JBET core performs low-level Java binary manipulations. For complex code transformations such as obfuscation, a higher-level approach is desirable.

Because manipulating stack-based instructions directly is complicated, we explored two internal representations of Java binary code to use for implementing the obfuscator. The Java verifier allows stack slots to be treated as variables, because the stack configuration must always be the same at any given instruction no matter which control flow path led to that point. Our first representation was three-register code, which was dropped because it required manipulation of register indices, and not all Java instructions have sensible three-register equivalents.

The second representation was a directed acyclic graph (DAG), where the vertices represent Java instructions in an almost one-to-one fashion. The only instructions not translated are those that only manipulate stack elements (for example, `dup` and `pop`). Edges in the DAG represent data flow; each edge joins a producer node and a user node (e.g. an integer add node has two edges pointing to the integers to be added). Nodes may have shared edges, but cycles are not allowed as that would mean that a node needed to use its own value in its computation. Some nodes (such as constants and global variable references) have no producers. The DAG representation of code has many advantages, particularly making it easy to substitute parts of an expression.

3.1.2 Code Storage

Our obfuscator avoids storing obfuscated methods in the normal Java way, as an isolated code block with a method name and descriptor, to avoid giving this information to a potential attacker. Instead, all the code comprising the obfuscated methods is collected together, randomly ordered at the basic block level (our control flow implementation was constructed to support this arbitrary ordering), and placed into one Java method (we call this an “output method”). Because Java limits methods to 65536 bytes of instructions or less, even trivial programs (once obfuscated) often exceed this limit, making several output methods necessary. All of the “fixed code” such as the global exception handler is stored this way also. Each output method then has extra code to support jumping to basic blocks located in other output methods, but that is transparent to all other parts of the output program. Then, the final program has multiple methods, but they do not mean anything as the blocks were randomly ordered before being divideded into output methods. The entry point of the program (usually main, but different for applets) is coded as an “internal stub”, described in section 3.1.9.

3.1.3 Data Storage and Object System

Our implementation uses "semi-raw" memory blocks to store fields and array elements. Since Java is a type-safe language, using byte arrays would require us to perform all operations using only those allowed on individual bytes. As that would incur so much of a performance loss that testing would be inconvenient, we made a compromise between byte arrays and structured storage: arrays of primitive types. We believe that using arrays of primitive types gives only marginal information to the attacker, since they can be cast from one to another. That is, the primitive types of the original data do not have to be the primitive types of the obfuscated data; an `int` can be mapped to two `floats` for example.

Each original object is replaced with generic storage using this Java declaration:

```
class Memory {
  int[]    I;
  Object[] L;
  long[]   J;
  float[]  F;
  double[] D;
  Memory[] N;
}
```

Each class then stores its fields (including the virtual tables) in an instance of `Memory`. Multiple virtual tables would be needed if the Java class had interfaces. The last array is included for convenience and to reduce cast operations in generated code (all application objects are instances of `Memory`, and so can be accessed without casting). Each class with virtual functions stores one or more virtual tables, and an integer to identify the runtime type. We implemented a general multiple inheritance mechanism and so (unlike Java), the interfaces of a class were treated as additional superclasses by our simulated class/object system. Each constructor was responsible for storing the appropriate virtual tables (this turned out to be a critical weakness, as it allowed parts of constructors to be recognizable as such) into the Memory instances that would represent the new object.

Our architecture supports multiple obfuscated versions of a single user-defined class, but we did not implement that. Other data structures, such as those for implementing method call stacks, use instances of `Memory` also, possibly making analysis more difficult.

For a user-defined type not needing to be passed to a system library (such as a "Document" container or RSA private key), it has no Java class representation, only the "emulated" class as instances of Memory as described. User-defined types that need to be passable to library methods have an "internal stub", detailed later, but are otherwise the same (i.e. no direct Java class file representation). This means that an attacker would first have to discover the method invocation protocol before calling obfuscated methods.

3.1.4 Control Flow

As described in our Obfuscation Techniques Report [BDM+01], we removed all control flow instructions, replacing them with data-driven jumps. In Java bytecode, this means a `tableswitch` instruction with a large array of targets. Each target block is based on a basic block from the original program. The switch statements allowed us to use "basic block addressing", where each basic block in the obfuscated program could be jumped to

in a generic way. This mechanism was used for exception handling and virtual methods as well: the virtual tables contained block addresses for the entry points of the methods, and the exception records (discussed later) contained a block address for the exception handler.

3.1.5 Method Calls

The method call stack we implemented uses the `Memory` class for storage allocation. The Java local variables are stored directly in the call stack object. At each applicable method call, a new call stack "object" is created and added to a linked list of outer call stacks (Not a Java "LinkedList" object, but a link field (e.g. Memory::N[2]) in the simulated call stack object). Then, the method arguments and return address are stored in the new local variable array and control passes to the first block of the method. For a virtual method call, the caller examines the virtual tables to determine the block address to jump to; otherwise the jump address is hardcoded. When the method returns, it reads the jump address from the call stack and transfers control there. If there was a return value, it is stored in the caller's local variable array. It was necessary to store the local variables in arrays instead of Java local variables (which would have been much faster) because of the need to implement method calls without using Java's method call mechanism. It also allowed the multiple methods (mentioned in the Code Storage section) to be more transparent, as they would access the local variables from an array passed in as an argument.

3.1.6 Runtime Type Information

A limited amount of runtime type information was always placed in the output program, because many Java programs use the `instanceof` and `checkcast` instructions, or exception handling (the search for the non-local handler in Java requires `instanceof` checks). Each class was assigned an integer, the product of its private identifier and the identifiers for all superclasses and interfaces. The Java checkcast and instanceof operators were replaced with code that emulated the behavior of those, using the virtual tables and class identifiers stored in the Memory instances representing the objects. No attempt was made to use inter-method dataflow analysis to reduce the need for checkcast and instanceof operations.

3.1.7 Exception Handling

Our obfuscator supports Java exception handling by replacing the throw and catch mechanism with our method call mechanism. When an exception is thrown to a handler in the same method, it is coded as an ordinary jump – removing most knowledge that an exception was involved from the program. Otherwise, the call stack is augmented with exception handler records (implemented as instances of Memory) analogous to those stored in the Java class files (and maintained by the JVM for normal Java program execution). The exception handler list is updated at the start of each basic block inside a try range that might need non-local exception handling. Code that is not in a try range is still subject to exception handlers from outer methods, exactly as in Java. When a non-local exception is thrown, control is transferred to the global exception handler (stored as part of the obfuscated program). The global handler searches the exception records of the

current call stack until a type matches. Then, a jump is made (using the switch construct) to the exception handler block indicated by the exception record. The thrown exception (an instance of Memory, same as any other internal object) is stored in the handler's local variable array. We realize that the exception handler search is not good for obfuscation purposes; however, non-local exception handling requires type information at runtime. Java programs can be designed so they do not need non-local exception handling, so this is not a fatal problem, however it does require careful design.

3.1.8 External Stubs

A Java application will often store references to library objects, such as `FileInputStream`, in its own classes. When an object is returned from a library, a stub is created (an instance of class `Memory` with proper virtual table and class identifier setup) that redirects calls (and untransforms arguments) to the original object. This provides a window into the implementation of obfuscated data storage, but that is unavoidable, as the application will have to use library objects. Our use of a type-system conversion for library objects allows them to be accessed as their interface type is accessed within the obfuscated program, but the true type of the library object is not accessible. We believe that this is not a concern as many library implementation types are private, or restricted in some other way, and do not add public functions beyond those declared in the interface type. Note that an external stub is entirely implemented in the simulated class system produced by the obfuscator; it has no class file. It is implemented with instances of Memory in the same way that user-allocated objects of the same type would be. The obfuscation tool automatically determined the classes accessed by the application, and which of those could be "imported" and considered part of the input program to be obfuscated. Those imported classes may need stubs for interaction with the rest of the library. Non-polymorphic classes such as Strings usually do not need stubs; a new object is constructed and that one is passed to the library instead.

For example, java.util.Vector and java.lang.String were "imported" and considered part of the application for obfuscation purposes. This means that the application contained obfuscated code for the functions in String, as well as a "deobfuscation" interface so Strings could be passed to standard library methods.

3.1.9 Internal stubs

Another problem we encountered was passing application objects to library methods. For example, the library method `PrintStream.println(Object)` requires an `Object` reference that it will call `Object.toString()` on. When a library call is made, a stub is created (in the Java object hierarchy) of the type of the argument, which will make calls using whatever internal protocol is used by that object. The stub itself is a real Java class, with a name, superclass, interfaces, and methods. The methods of the stub class contain a small amount of code responsible for setting up the simulated call stacks (section 3.1.5) and jumping to the appropriate basic block. Note that this provides a window into the obfuscated data storage, but that is unavoidable, because the application needs to pass data (including object references) to the library in order to have interactivity.

The types of stubs that will be needed for this purpose are determined at compile time. As with external stubs, the library can only access methods declared in the declared argument type and not any only present in the runtime type, because the library is actually accessing instances of the stub class, which only have what the interface type declares. This is only a limitation if the library uses reflection to call methods or examine fields outside the interface of the argument type.

For example, the application passes an object of type UserDefinedInputStream to a library method declared void f (InputStream o). Then, the obfuscation tool will generate an internal stub with the InputStream interface that will call obfuscated methods in UserDefinedInputStream. The library only sees the stub, the same one that will be used for other user-defined InputStream subclasses, so the obfuscated type remains concealed. Another common internal stub is that for "main".

3.1.10 Java Obfuscation Demonstrations

At the OASIS PI meeting at Hilton Head and the summer PI meeting in Santa Rosa in 2002, we demonstrated our JBET obfuscation tool. We showed obfuscation of both a simple program (that just counts to 10 and prints the results) and a more complex, computationally intensive program (DES). Our tool translated compiled Java bytecodes into obfuscated bytecodes. The demonstration showed the costs of obfuscation (an increase in size of about tenfold, and a runtime slowdown that depended on the application, but ranged from fourfold to twentyfold). Additionally, we showed the JVM bytecodes generated by our tool.

3.2 Deobfuscator

We developed a "deobfuscator" (actually more of an analysis assistant) for our obfuscation tool. It worked by searching for patterns in the input program, and running selected parts of the program. It was largely successful, in that it was able to determine method entry points, the structure of the class hierarchy, which methods were constructors, etc. The DAG representation of Java bytecode developed for the obfuscator was extremely useful for deobfuscation as well, representing particular variable writes as operation trees.

Generating the DAG representation could be the first stage of decompiler implementation, where accurate and readable source code could be printed for each basic block. In Java, because of the type system and verifier, there is no way to hide the boundaries of basic blocks with arbitrary jumps.

3.2.1 Dynamic Analysis

The deobfuscator runs the `<clinit>` (static initialization, run when a Java class is loaded) of the program to retrieve the virtual tables and jump tables, so the basic blocks of the obfuscated program can be viewed as a uniform set instead of basic blocks in methods. If the deobfuscator were more complete, it would also run constructors to determine which class the constructor was for (by the virtual table it stored).

3.2.2 Pattern Matching

The control flow leaving each block was determined by pattern matching on the DAG representation of the basic block. Since our obfuscator produces basic blocks that use data-driven jumps, there is a calculation in each obfuscated basic block that returns the block to jump to. Our obfuscator output only a fixed number of formats for this calculation, so the deobfuscator can match against those formats.

After simplifying the control flow leaving each basic block, the deobfuscator used the virtual tables (read from running the initializer) to determine which basic blocks were method entry points. Since no Java method (in the original program) can jump to code in other methods, this allowed complete determination of method composition. The virtual tables also associated methods with classes. Certain facts about the class hierarchy could also be determined from the virtual tables, such as superclass and interfaces. Our obfuscator preserved all class hierarchy information from the original program, in the form of the factorization of the class identifiers (section 3.1.3).

Instances of primitive classes should be easy to determine from the virtual tables, or from comparing multiple obfuscated programs, as they will have the same methods.

3.2.3 How deobfuscation could have been made more difficult

A better obfuscator could hinder many of the techniques used by our simple deobfuscator. As the basic block to method mapping would be one of the most useful things to the analyst, searching for common basic blocks and placing only one instance of that block in the output program would complicate the control flow calculations required (because that block would have twice as many successors as it once had), adding to the list needed by the pattern matcher.

The parts of the program that dynamic analysis is useful for (`clinit` and constructors) could require additional state, so that “run one block only” dynamic analysis won't work properly.

Our implementation of the `instanceof` operator used integer factorization, and never discarded information. We could have only used meaningful class identifiers if that information was necessary. A great deal of information about an object-oriented program is contained in the class hierarchy, because it not only reflects program behavior, but design and specification as well.

The various control flow tables could be initialized on demand, instead of all in one place run at program startup. This could make extracting virtual tables with dynamic analysis harder as the table itself would not be created (or stored in a common location), until an instance of its class was allocated. However, constructors would probably be the easiest type of method to search for completely unaided.

Derived classes could have private casting methods, so that they may only be cast into those base classes that are actually used in the original program. This would prevent the attacker from calling `java.lang.Object.toString()` on everything in the system once the obfuscation for `java.lang.Object` was discovered (unless the attacker also discovered the cast-to-`Object` protocol for that class)

Each of these counter-strategies to deobfuscation has, in turn, a counter-counter strategy.

3.3 Summary

Obfuscators are hard to implement. The amount of work required to get usable obfuscation is much greater than checking or fixing many classes of simple security flaws (buffer overflows, not checking input, string quoting, etc). In addition, the security gained is likely to be nonexistent to marginal. There are solutions to all of the use cases we presented that do not involve obfuscation (such as running the program that would have been obfuscated on a network server). Our obfuscation tool took over 2000 hours to develop, for no to very marginal security. Implementation of obfuscators will require source or binary editing, and compiler-like functionality, which are not trivial to implement either.

4 Difficulties in Implementing Obfuscation

Regardless of whether obfuscation can effectively provide any security, people will use it anyway. This section discusses problems with using obfuscation on ordinary programs. A recurring theme is “build for obfuscation”: programs that are written to be obfuscated from the start will get better results from obfuscation than existing programs. In this section “defender” refers to the developer (or distributor) of an obfuscated program and “attacker” refers to the user of such an obfuscated program who wants to change its behavior or extract data.

4.1 Programming Languages

The programming language the obfuscation tool works on essentially determines the obfuscation techniques that can be used. Several attributes of a language are relevant to obfuscation: the kind of type system, the kind of system environment presented, and the use of convenient intermediate forms (such as Java bytecode).

4.1.1 Type Systems: Abstractions and Patterns

Abstractions in a programming language cause patterns to appear in the binaries (A pattern merely being a portion of the binary that satisfies some predicate). As discussed in section 3, our deobfuscator worked with the patterns created by our obfuscator. The difference between abstractions in the documentation for the language (or libraries) and abstractions in the programming language is very important for those who intend to implement or use obfuscation tools. In order for an obfuscation tool to disguise an abstraction, it must be able to detect it in the input program. Control flow and low-level obfuscations may change the appearance of a pattern but not remove it entirely. For example, low level data obfuscations (those that operate at the assembly level) change the representation of virtual tables to use 4 integers for each entry, but there is still an abstract array in the obfuscated program. Static analysis defeating obfuscations work regardless of abstractions, but don’t remove them either. Data-structure obfuscations (those that operate on structures created by the compiler) may be able to remove or

disguise evidence of an abstraction from the program if it was known to the developers of the obfuscation tool. For example, our obfuscator was able to disguise the abstractions of classes and methods (as described in section 3). Had the control flow transform be performed without integration with the type system obfuscation, the classes would be clearly visible in the obfuscated program.

Consider bit flags. A C source program declares them as `int`, but they will never be added or divided, only manipulated with bitwise logical operators. Knowing that, an obfuscator could identify which integers are sets of bit flags. Then it could represent flag sets in another way that makes them appear to be ordinary integers instead (e.g. by using prime factors to represent each flag).

Note that we are using the term "type system" more generally than is common; it includes low-level language features as well. For example: `int` is considered a class, with the arithmetic and logical operators as non-virtual methods; Java's `invokevirtual` operation could be considered a method of `java.lang.Object`; the `throw` operation (in Java) could be considered a method of `Throwable`. We are not normally discussing the programmer's use of the type system, but rather the effects of typed data on execution environments. It is important to consider those low-level operations as they carry abstractions into the output program also.

The type system of the language determines what kind (and how much) of type information is available to the obfuscation tool. (We will assume that the obfuscation tool attempts to work around the language's type system constraints, perhaps by representing all polymorphic user types with simple arrays as our obfuscator did, so the deobfuscator does not get this benefit.) In a strongly typed language, the obfuscator has enough information to obfuscate different types of values in different ways, without needing general conversion routines.

The worst case for hidden abstractions is a weakly typed language: the obfuscator only knows about the "fundamental" types of the language (like atoms and lists), and not anything higher level at all (without guessing).

The programmer, and any other reader of the source code, probably does know the high-level use of each variable, but that information is not explicit in the program. If the same use patterns are preserved (which they almost certainly will be), then an analyst will see the purpose of each variable. Example: Cloakware's white box crypto [CEJ+02a, b] claims to protect the abstraction of a fixed-key block cipher by integrating the key into the crypto implementation. Without special obfuscation tool support for this, the block cipher (with key) abstraction will be carried into the output program. Programming languages that come with many abstractions are good for obfuscation, because there would be less need for the application developer to implement more "dangerous" (difficult to obfuscate automatically) abstractions.

4.1.1.1 Example: Obfuscated Rationals

Certain fundamental structures seem completely impossible to obfuscate (given black-box access and a few value pairs) Consider the rational numbers. With black box access, and one obfuscated representation of a nonzero value (e.g. the obfuscated representation of 2 "a8654h"), it is possible to obtain all desired values in the set by repeated calls to the

black box. For example, first ask the black box to divide a8654h/a8654h, and then ask the black box to add the value received to itself, obtaining the obfuscated representation for 2. Given any unknown obfuscated rational, it is possible to derive all integers, and by extension all rationals.

So a search for what the obfuscated value of some x is, or a search for the unobfuscated value of some obfuscated representation, will take unbounded but finite time. With the obfuscated rationals, the attacker need not even be told which operator is which. Consider an attacker with two different obfuscated values, and access to add, subtract, multiply, and divide. Assume that divide by zero is detectable. The attacker can "test" the unknown values with the unknown operators as follows (Call the two values A and B, and each unknown operator f):

1. Determine if either value is zero by testing each operator f(A,A). If f(A,A)=A for at least three operators (add, subtract, and multiply), then A=0.
2. If no zero was found in step 1, find a zero (and the subtraction operator) by applying f(A,A)=C with each function. Then repeat step 1 with C instead of A.
3. If not discovered in step 1, find the divide operator using the known zero.
4. Find 1 by dividing a nonzero value into itself.
5. Find multiply by finding f such that f(1,1)=1
6. Find add by finding f such that f(1,1)-1 = 1

Typical obfuscated programs will contain several fundamental structures of this type (such as integers, floating point numbers, etc.) that each provide a "window" into the rest of the obfuscated program.

4.1.2 The Problem of Merging Type Systems

The problem of merging type systems has to be addressed to design any obfuscation tool that masks complex data storage usage. An unobfuscated program has one type system, that of the programming language. The obfuscated version has at least two type systems: the obfuscated type system constructed by the obfuscator, and an unobfuscated type system that has to be used to interface with unobfuscated code. The former is necessary otherwise no data obfuscation is being done. The latter is necessary because all programs need to interact at least with the system environment if only to do I/O. One might argue that the I/O does not need to be in an unobfuscated form, however, based on the constraint that the obfuscated program function the same as the unobfuscated version, it eventually does.

Because it is necessary to interact with at least one unobfuscated component, the program needs to convert some data between its obfuscated and unobfuscated forms. Unobfuscated data provide a handle through which an observer can derive other information. Take for example the simple case of printing a single result to stdout. An attacker would first be able to observe where the result came from. A data-flow analysis of that location would reveal data dependencies (albeit obfuscated dependencies) and operations performed on it. A control-flow analysis would reveal the decisions that

guided the calculation. Whether or not that gives the attacker any useful information depends on his goal and the program, however, it should be obvious now that unobfuscated interfaces leak more information than simply what is passed through them. As a general rule, the fewer unobfuscated interfaces necessary to run, the fewer handles into the program an attacker will have.

The problem is compounded in polymorphic OO languages because at compile time, the type of an object at runtime can be vague, necessitating general conversion routines *in the program* for every type that could be the type passed into and out of an unobfuscated interface. Because of Java bytecode's high-level characteristics, it has additional restrictions, which we explain in section 4.1.5.

4.1.3 System Environment

The system environment presented by a programming language and its associated libraries are important limiting factors for the quality of obfuscation. Libraries that are linked in (either at compilation or runtime) give useful information to the attacker, the same as giving the attacker part of the source code to the program. Unless a custom obfuscation tool is used, the attacker can obtain the libraries and the obfuscation tool, to see what kind of output that tool produces for those libraries, and compare with the obfuscated program.

One might object given a keyed obfuscator. Presumably the attacker does not have the key used to obfuscate the program. It turns out that the amount of variation among different obfuscated programs produced by the same tool from the same input program is limited because the automated tool is incapable of "understanding" the program and is merely emulating it. (This is discussed further in section 5.) Prof. Andrew Appel's result described in section 6.1.10 also shows that the amount of variation in multiple obfuscations by the same tool is limited.

Developers wanting good obfuscation should forego the convenience of using existing libraries because of the initial analysis data they give to the attacker.

In terms of the handles described in 6.1.2, it is easier to reduce the number of handles in some environments more than in others. If a program can be completely statically linked, the only unobfuscated interfaces are the system calls. In Java, the "system calls" are interleaved with support classes in one giant standard class library. Section 4.1.2 discusses why these integrated standard libraries cannot be split into system calls and support functions, which would be desirable. Being able to disentangle system and support calls allows the support calls to be imported into the input program for obfuscation purposes, eliminating the need to treat it as an unobfuscated interface.

4.1.4 Language Feature Concerns

We have discussed how general features of a program and its environment affect high-level obfuscations, now we discuss specific language features. It turns out that many languages with characteristics that are good for data obfuscation also contain features that are bad for data obfuscation. Ideally, the writers of an obfuscation tool would provide a summary of the benefits and penalties of using certain language features. This section lists a few problematic features we observed in Java, which we believe are relevant in

principle to other languages as well. Note that implementing particular features manually is just as bad for obfuscation purposes as using those supplied with the languages (e.g. adding reflection to C++). It may be worse because obfuscation tools designed for that language may have implementations or warnings for some difficult to obfuscate features, but a feature added by the user will not be noticed (unless the user also augments or writes the obfuscation tool).

Our experience in implementing obfuscators is limited to the Java language (and those features), but we have analyzed some other language features for this report.

4.1.4.1 Reflection

Reflection capabilities present a problem for obfuscation in several ways. Usually, the application will contain class or method names as strings. Some applications may access fields generically (i.e. through the `Class.getField()` method), requiring the reflection interface be able to access any data structure unobfuscated in the application, since it is not known in advance which runtime types may be used this way.

In an obfuscated program, the reflection mechanism will have to provide a uniform interface to all the differently-obfuscated data structures, essentially making it a "deobfuscation interface" within the application. If the attacker discovers the interface to ordinary reflection features, he can probably deobfuscate the program. If an application only makes limited use of reflection features, the obfuscated program's implementation of reflection only has to support a subset of the original language's features on a subset of data structures. However, there is no way to determine that an application using reflection generically only accesses a subset of classes (or methods, or fields), without requiring extra input from the user.

We believe the best way to support reflection in an obfuscator is to require input from the programmer, and not include any functionality unless specifically requested. Of course, if he requests a complete implementation then a complete interface will have to be provided, but many applications do not need that, and he was made aware of the consequences.

Some languages (such as Perl) have even more capable reflection features, allowing global variables to be searched through or accessed by name. If the program relies on that method of accessing variables, the obfuscator cannot remove the names.

4.1.4.2 Exception Handling

Non-local exception handling requires some runtime type information in order to determine which exception handler is most applicable. Even if the obfuscator implements exception handling independently of the language's implementation, it still must use 'instanceof' checks and go through the exception stack. If the application has a complicated hierarchy of classes used as exceptions (like the Java standard library), their virtual methods may be exposed through the exception handler implementation

4.1.4.3 Excessive use of Polymorphism

Polymorphic types, in general, are hard to obfuscate with the techniques we have discussed. However, using them is preferable to other means of achieving the same programming idiom, such as function pointers or switch statements, because of the information given to the obfuscator in the form of class declarations. Since polymorphic base classes are abstractions, the obfuscator will create patterns in the output program.

Deep class hierarchies (with base classes having many derived classes) are a problem for obfuscation, because a common interface to a lot of application data is exposed through the base class (or classes) of that hierarchy. In Java, the universal class java/lang/Object has a toString() virtual function, which is often used to print fairly verbose diagnostic information about the object. If an attacker can discover the cast-to-Object interface for several obfuscated classes (which will almost certainly exist), and the interface for calling Object::toString(), he would have a large window into the operation of the obfuscated program.

4.1.4.4 Arbitrary Casting

Some languages (notably C and C++) allow arbitrary type casts. Because arbitrary casting violates the type system, the obfuscated program will need to contain special case conversion routines used in case of an arbitrary cast, or prohibit them. For example, a C program to be obfuscated contains the following segment:

```
{
     float x;
     int *pi = (int *) &x;
     *pi = 3;
     printf (“%f\n”, x);
}
```

When this program is obfuscated with per-variable obfuscation techniques (such as those described in the Obfuscation Techniques Evaluation Report), the obfuscated version of the float variable x will have to support an additional operation (that normal float variables do not have): bitwise assignment from int. The per-variable obfuscation techniques are often unable to support all the normal operators while remaining obscure.

4.1.4.5 Templates

Some languages (e.g. C++ and Ada) support templates which benefit obfuscation because common idioms (e.g. container classes) can be specialized for certain types by the obfuscator, removing the need to use runtime type information to examine objects in a container as in Java.

4.1.4.6 Range Types

Range types (found in Ada) allow a programmer to specify that a variable may assume a range of integer values. The runtime environment checks those values for correctness when they are used. This facility is (if used properly) of great benefit to the obfuscator,

because many more representations of integers become useable if the range of values is limited. For example, several integers from 0-16 could be stored in one processor register.

4.1.4.7 Perl and other modern scripting languages

Perl has a combination of useful features which we believe are especially bad for obfuscation: weak type system, encouraged mixing code and data, and building and executing code at runtime. Although this section is specifically about Perl, it is not the only language with this kind of feature set; many modern scripting languages are similar.

Perl has a single type, 'scalar', that stores strings, numbers, and references. The string and number are indistinguishable; a program that stores a number into a scalar variable can access it later as a string or a number. The obfuscator would have to support this. As scalar variables are common, the attacker would discover the “get string value” protocol very quickly and be able to use this on everything (“get string value” also reports the type of reference, if any). In Java, it would be equivalent to discovering how to call `Object::toString()` for every class and primitive value type.

In Perl, objects are often represented using hashes, but this is usually unnecessary because only fixed strings are used as indices. But the language provides no 'struct' data structure, so hashes are used instead. A Perl obfuscator would have no choice but to represent all hashes the same way, even those that will only ever contain the same number of fixed-string entries. Even if only fixed strings are used, database modules and persistence mechanisms in Perl iterate over the contents of the hash tables, so they would require the fixed strings to be present anyway.

For example, if the obfuscator remapped a fixed-string hash to use arbitrary integers instead, and a persistence mechanism was used on those hashes (ex: if they were saved to disk), corrupted data would result as the fixed strings were actually needed when the saved hashes were reloaded by another program. Otherwise, the fixed strings remain in the program for the attacker to find.

Perl variables can be accessed by name, and the entire set of global variables can be searched through. In addition, Perl allows any ordinary values (strings, numbers, lists, file handles, or hashes) to become polymorphic objects at any time. These last two features practically prohibit using the 'information hiding' obfuscation techniques because since the program can use any number of data access methods on all variables, the obfuscator will have to support those general methods for all variables. This is even worse than the problems with reflection because there is no static type system to show that reflection methods are only used on some types (however a good type-resolution system might be able to help).

4.2 The Portability of Language Features

We already know that reflection is hard to support in an obfuscator, but what if the user builds reflection into a language that doesn't support it directly? Then the obfuscator would not necessarily be aware of it, and the complete interface would be present in the

obfuscated program. For example, consider a C++ program has a common base class with a reflection interface:

```
class Object {
      public:
      virtual ClassInfo *getClassInfo() const;
    }
```

The obfuscator would not be able to remove this function, because it has no way of knowing which runtime types it will be called for at runtime. So it will be common in every `Object` in the output program also. Then to access any data, the attacker only needs to decode the method call protocol for `getClassInfo`, and `ClassInfo`'s methods in order to access any data. The obfuscator will also not know about the universal class `Object`, so that abstraction will be carried into the output program. While the obfuscator could use alternate representations for derived classes when being used as derived classes, it would have to create a stub that uses the same protocols as `Object` when cast into an `Object`. These stubs may be apparent in the output program since they contain the same kind of virtual table as `Object` instances.

The white-box crypto proposed by Cloakware [CEJ+02a, CEJ+02b] is a solution to this type of problem also: encryption algorithms are fairly obvious, that is a technique “writing specifically for obfuscation” that can be used.

4.3 Networked Programs

An attacker with complete control over the program's environment, as is the case when the program runs entirely on the attacker's computer, will eventually gain whatever behavior changes or secret data he wants. The only counter to this is to split the program so that it does not run entirely on the attacker's computer. This is the client/server model, where the attacker only runs the user interface; any real functionality is done on a remote server, away from the attacker. The emphasis here is on “real functionality”; adding functions that simply contact a networked service to check license keys or anything similar provides little protection. Note that since both “efficient operation” and “real functionality” is involved, the choice of which sections of the program can run on the attacker's machine cannot be accurately determined automatically; it must be designed into the program. A program that attempted to determine which functions were computational might include user interface routines, making the client-server split inefficient.

4.4 Summary

At this point it should be clear that the best results from obfuscation will come from designing the program with the obfuscator in mind and avoiding language constructs that are hard to obfuscate.

To make obfuscation the most effective, a program should be written in a statically-typed object-oriented language with distinct system calls and support libraries. Additionally,

techniques that require an altered program (such as white-box cryptography) should be used whenever possible

5 Theories on Obfuscation

This section describes conclusions we reached about obfuscation.

5.1 Game theory

We analyzed the situation of obfuscator and deobfuscator in the language of elementary game theory. There are two ways to look at obfuscation situations.

5.1.1 Known deobfuscator

In this "game", we start by defining what the secret is; that is, what information the defender wishes to keep from the attacker. Then, the attacker then chooses a deobfuscation method, and announces it. Based on the information provided by the attacker, the defender is then free to choose any obfuscation method. The game continues with the defender obfuscating the secret with the previously chosen obfuscation method and passing the obfuscated information to the attacker. The attacker attempts to deobfuscate, using the previously announced deobfuscation mechanism.

This situation might correspond to the cases of virus attack, where sites are equipped with virus detection tools. A virus creator, knowing the characteristics of the tools that detected viruses at each site, could attempt to create a virus that couldn't be detected without updating the sites.

It is clear that it is always possible to construct an obfuscator that will defeat a chosen, fixed deobfuscator, so the defender (the one obfuscating information) can always win in this case.

5.1.2 Known obfuscator

This game is similar to the first; again, we start by defining what the secret is. But, in this game, the defender then chooses an obfuscation method, and announces it. Based on the information provided by the defender, the attacker is then free to choose any deobfuscation method. The game continues with the defender obfuscating the secret with the previously chosen obfuscation method and passing the obfuscated information to the attacker. The attacker attempts to deobfuscate, using the previously announced deobfuscation mechanism.

This situation is more like that contemplated for Self Protecting Mobile Agents. We have to assume that the attacker can find out the mechanism used to obfuscate mobile agents, and that the only security is in the set of random choices made during obfuscation; otherwise we are depending on security through obscurity. Given a fixed obfuscator, our practical experience shows that it is possible to create a mechanical obfuscator that will rapidly deobfuscate the information.

5.1.3 Solution

What we see from this analysis is that, if perfect information is available, whoever commits to a strategy (i.e. automated tool) first loses.

5.2 Recursion Theory vs. Complexity Theory

We contend that recursion-theoretic statements about obfuscation are sometimes not very meaningful from a pragmatic point of view. For example, we can define another game, in which the attacker (deobfuscator) picks a program and a set of axioms for mathematics, and publishes them. Then the obfuscator then produces two outputs, a real obfuscated program that computes the same function as the original program, and a fake one that does not. The attacker's job is then to figure out which is which, and prove it from his axioms. It is not hard to prove (prove that the set of indices of total functions is productive and use that) that the defender can always win this game. This result is not meaningful because the attacker's algorithm need not be anything close to polynomial, because people don't need to prove program equivalence in order to get at the secrets they typically want to get, and because even if for some bizarre reason the attacker did need a proof of program equivalence there is no reason they should limit themselves to a constant set of axioms.

Another recursion-theoretic result that can be applied to obfuscation is Rice's Theorem, described in section 6.1.1, which can be interpreted to say that all programs are obfuscated in the first place (if the secret is some property of the function that the program computes). This result is not meaningful because it does describe promise-problems, i.e. if you know that the program doesn't go into an infinite loop you can obviously run it and find out what it outputs. Rice's Theorem says you cannot find out what it outputs because you have no such *a priori* knowledge. There is a version of Rice's Theorem that does concern promise problems, and it says (more or less) that even if you do have a *priori knowledge* the best thing you can do with a program is run it. However the promise problem generalization of Rice's theorem still only deals with what is computable, not what is efficiently computable, so it is still not very meaningful for real-world obfuscation. You can formulate a complexity-theoretic version, but it is false, as proven in [BGI+00]

5.3 Automated Obfuscation is Emulation

We believe that all automated obfuscation is merely emulation; that is, the high-level structure of the program is preserved by the obfuscator. For example, consider a program that sorts a list with bubble sort, computes a greatest common denominator with Euclid's algorithm, and uses polymorphic objects to distinguish different kinds of network clients. Under this theory, an obfuscated version of the program will still use the same sort and greatest common denominator algorithm, and will still use polymorphic objects, although the low level implementation of each may be disguised. In essence, both programs carry out the computation "in the same way." An obfuscator could search for patterns in an input program to recognize particular algorithms and substitute alternate ones, but only for a finite number of algorithms. An obfuscated program preserving the structure of the

original is very useful for the attacker. All obfuscating transforms we implemented, as well as those in the Obfuscation Techniques Evaluation Report, satisfy our informal definition of emulation.

We made several attempts to formalize this concept of emulation (within the context of achieving a result about obfuscation), but none were successful. One of the more interesting is to consider the program as a “deterministic automaton”: a DFA, but with an infinite number of states. Each state of the automaton represents a complete memory and register configuration. The state just before a conditional jump will have two successors, for the condition met and not met. Representation of other control flow is similar. Note that if there no loops in the automaton, the program always halts. An “emulation” of a program would then have a deterministic automaton similar to that of the original program: it could be isomorphic, or a quotient.

Our “switchify” control-flow transform preserves the deterministic automaton structure of its input: only unreachable states are added. Our per-variable transforms also preserve structure; the new state graph is a quotient of the old. Our implementation of runtime type information and local variable storage added intermediate states to every operation and new unreachable states, but the structure of original states was otherwise unchanged.

The failure of this approach was that some trivial changes to the program could change the deterministic automata's structure, and some nontrivial changes would result in a similar deterministic automaton. Also, some nontrivial changes to the program could result in a “similar” deterministic automaton but seemed more difficult to reverse than our definition allowed for.

5.4 Static and Dynamic Analysis

Static analysis tools examine a program and attempt to produce higher-level explanations of the program's future behavior. In addition to traditional decompilers and disassemblers, we also include traditional data flow analyses (typically used to enable compiler optimizations), program slicing, and abstract interpretation in this category. Static analysis characterizes all possible executions of a program, with all possible inputs, with respect to some property of interest.

Dynamic analysis tools run the program being studied under a tracing environment that watches and records what the program actually does given chosen inputs, at any of various levels of detail; such traces can then be analyzed to discover future behavior and actual data operated on. Tracing virtual machine environments, subroutine call and system call tracers, test point insertion tools, data reference traces and counters, and packet and message tracing tools are commonly used for this purpose. Even if the computation of static control flow or block liveness for a program has been made intractable, dynamic analysis will observe the actual execution path taken by the program. Reverse engineers often use tracing to determine the general flow of a program, and then use static analysis to examine specific regions to see what they do.

To frustrate dynamic analysis, a program creator would need to arrange that a program took very different execution paths on different runs of the program, so that determining which basic blocks were ever invoked, and in which combinations, would require

unacceptable effort. To make the effort large enough to discourage reverse engineering would probably require an expansion of program size of many orders of magnitude, and is at the limit of practicality.

Wang et al [WHK+00] showed that there are obfuscation techniques that can foil static analysis. In order to deobfuscate those programs, dynamic analysis is necessary. This coincides with our experiments on obfuscation: static analysis alone is not enough. However, unless performed carefully, the results of dynamic analysis will only apply to the particular input(s) of the program specified in that run. To avoid this, dynamic analysis can be performed on sections of the program that do not depend on inputs. (This was a weakness in our obfuscator: the constructors could be partially evaluated to determine the virtual tables, even without inputs). Hybrid techniques, such as evaluating the program statements formally, may be of use in some scenarios. Had our constructors been designed to use the input before storing the virtual tables, running them and ignoring input-dependant statements would have provided the same information.

6 Related work

This section describes work by other researchers related to our problem. There are three major areas: obfuscation and deobfuscation research, reverse engineering, and cryptography.

Our approach to obfuscation, along with work of Hohl [Hohl98] and Wang *et al*. [WHK+00, W00, WDH+01], aims to delay an attacker. Other researchers are looking into whether one could develop an obfuscator that would prevent access entirely [BGI+01]. Other related research is in the categories of electronic commerce and mobile agent protection, computing with encrypted functions, and in practical obfuscation and reverse engineering and decompilation. These are described in section 6.1.

Section 6.2 describes reverse engineering, including both static and dynamic methods, some assisted by hardware.

People occasionally try to draw parallels between obfuscated programs and cryptographic functions, arguing that obfuscated programs have similarities to cryptographic functions and that the security of the obfuscated program should bear some resemblance to the security of cryptographic functions. In Section 6.3 we will see that it is the cryptographic functions whose security begins to resemble the security of obfuscated programs when the adversary can engage in probing attacks that are not as severe as what a reverse engineer can subject an obfuscated program to.

6.1 Obfuscation and Deobfuscation Research

Obfuscation has been studied as both a pragmatic discipline and a theoretical topic in mathematics. In order to prove that any obfuscation method works, we need to understand the limits that theory of computability places on what can be proved, and the limits that complexity theory enforces on the difficulty of the task.

6.1.1 Rice's Theorem

Rice's theorem says that given a non-trivial property *P* of Turing-acceptable languages, the problem of identifying the property in the language of some Turing machine is undecidable.

There are two types of questions that a deobfuscator could ask: (1) questions about the program's representation (ex: object or source code), and (2) questions about the function that it computes. Rice's theorem says that type 2 questions are undecidable; therefore no program could be guaranteed to answer it.

This means that if a deobfuscator has only black box access to a program, it is fundamentally limited in what it discover.

6.1.2 The Rice-Shapiro Theorem

The Rice-Shapiro theorem proves that non-trivial type 2 questions are not recursively enumerable (RE) or co-RE either, meaning that given a type 2 question there are cases where it is impossible to prove that the answer is correct, even if the answer is known. This result holds under any proof system; if a proof system is extended to add a proof for one case, there will always be another case it cannot prove. Consequently, general solutions to type 2 problems are not possible.

It should be noted that although a general solution to a type 2 problem is not possible, the specific instances of these problems that occur in the real world are often solvable. For instance the Halting problem is not decidable, but given a real-world program it is often possible to prove whether or not it halts based on human understanding, which is not bound by algorithmic limitations.

This suggests that attackers seeking to answer type 2 questions about an obfuscated program would not be able to write general tools to do it, but would be able to sit down with a debugger and figure out how it works. This is indeed how deobfuscation is typically done. Consequently, the process of deobfuscating type 2 information in the real world is a human one, not a computational one, and therefore no meaningful bounds can be put on its difficulty. One can only estimate the difficulty by the time it takes real people to solve it.

It may seem contradictory that the question "what does the program output for a particular input?" is a type 2 problem, because the "solution" is to simply run the program on that input. Technically, the problem is unsolvable because the program may never halt, or it may halt after a very long time. This counterintuitive result suggests that sometimes we should consider the deobfuscation problem "solved" if we have a procedure that can solve it given that the obfuscated program is guaranteed to halt.

One might imagine that such a conditional solution is possible even though a general one is not. Barak *et al.* in [BGI+00] prove that this is not the case (except for trivial instances): in fact they prove that if P is a type 2 problem that is solvable with respect to some guarantee (i.e. it is a "promise problem") then P is solvable given only oracle access to the obfuscated program (oracle access to a program essentially means access to a black box which runs it for t steps; if the program terminates within t steps the output is given, if it does not terminate, the output is a special message indicating it). In other

words, obfuscation does not make P more difficult because solving P does not require access to the internals of the program.

Thus from a pure decidability point of view, all programs are obfuscated to begin with. Given program P that computes function f, anything that can be learned about f can be learned with only oracle access to P and the length of P. A natural question to ask is whether or not the previous sentence is still true if "learned" is replaced with "learned efficiently," which (very informally) is the main question that Barak *et al.* [BGI+00] ask and answer in the negative.

Type 1 questions may or may not be solvable in principle, but every obfuscation problem based on one we have encountered was solvable. For example, an obfuscator may add a copy prevention mechanism to a program. This mechanism may be parameterized by random input, and the obfuscator could be run several times on the input program. Then there are several versions of the obfuscated program that are distributed, with the hope that if the copy protection in one version is broken by hand, the work would have to be done all over again for another version. The associated type 1 deobfuscation problem is to write a program that automatically cracks any version of the copy-protected software.

Another example: there is a program p(x,y) known to Alice and Bob. Alice gives Bob an obfuscated version p'(x) which computes p(x,y0) for some fixed y0. Bob 's type 1 deobfuscation problem is to determine y0. These two problems share a common feature: the deobfuscator has some *a priori* knowledge of the obfuscator and the input program. In fact, every interesting type 1 deobfuscation problem will have this property. It is hard to imagine interesting questions that the deobfuscator could ask about an arbitrary obfuscated program except "What does it do and how does it do it?" and "Does it have characteristic X?" Most deobfuscation attempts will be done with some knowledge of the program and the secret being protected (e.g. remove the copy prevention, obtain the maximum bid from the auction mobile agent).

6.1.3 Cohen: Evasion and Mutation

Fred Cohen [Cohen92] was one of the first investigators of techniques to obfuscate programs. He proposed the notion of program evolution, where programs could protect and obfuscate themselves by either mutating themselves or by being mutated by another program into an equivalent program. Two programs are considered equivalent if, given identical input sequences, they produce identical output sequences. The paper focuses on proposing techniques and provides simple illustrative examples of the techniques.

The equivalence of two programs is undecidable as is the determination of whether one program can evolve from another. However the author notes that practical considerations may limit our ability to reach the levels of complexity of equivalent programs "required to eliminate concerted human attack, but we may succeed in increasing the complexity of automated attacks to a level where the time required for attack is sufficient to have noticeable performance impacts, even to a level where no attacker is able to design a strong enough attack to defeat more than a small number of evolutions."[4]

[4] In the "Techniques for Program Evolution" section of [Cohen92], see http://www.all.net/books/IP/evolve.html

More specifically, Cohen proposed the following obfuscations:

- Equivalent instruction sequences, i.e. replacing a sequence that adds 17 to some number with one that adds 20 and subtracts 3.
- Variable substitutions, i.e. altering the locations of memory storage areas to inhibit the static examination and analysis of parameters and altering memory references throughout a program without affecting program execution.
- Variable relocation
- Mangling control flow by adding jumps and subroutine calls
- Garbage insertion: Given an instruction sequence, inserting a meaningless independent sequence.
- Program encodings that are decoded just before execution, i.e. compression and encryption.
- Encoding the program for a different platform and using an interpreter to execute it.

He also suggested building redundancy and self-checking into these modified programs along with anti-debugging features and suggests that the techniques should be combined for best results.

Cohen reported performing some experiments; however no obfuscation tool appears to be available.

6.1.4 Collberg: Obfuscation and Watermarking

Collberg's team 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 [CTL97a, CTL98a, CTL98b, CT02].

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 using artificial data dependencies, aliasing parameters, etc. The opaque constructs include opaque predicates and making programs more parallel with multi-threading.

Collberg *et al*. have announced SandMark [SM03], a tool for software watermarking, tamper-proofing, and code obfuscation of Java bytecode. This tool was originally an implementation of the Collberg-Thomborson watermarking algorithm. Recently the scope of the tool has expanded significantly with additional watermarking and obfuscation techniques. The obfuscation techniques include [Col03]:

- A number of code obfuscations from Collberg, Thomborson, Low's "Breaking Abstractions and Unstructuring Data Structures" [CTL98b] and "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs" [CTL98a],
- An implementation of Paul Tyma's name obfuscation algorithm: Method for renaming identifiers of a computer program, US patent 6,102,966,
- Additional code obfuscations including (inlining, Boolean variable splitting (using XOR, parity, and equality splitting algorithms), class splitting, array obfuscations) and an obfuscation loop that selects and applies a sequence of obfuscations to a program,

and in the upcoming release:

- An opaque predicate library,
- Control flow obfuscations that rely on opaque predicates,
- A number of code reordering obfuscations, and a string obfuscator.

The toolbox provides many obfuscation techniques, but is not accompanied by an argument we can use to show resistance to deobfuscation for any minimum time.

6.1.5 Wang *et al*: Obfuscation

Wang *et al*. [WHK+00, W00, WDH+01] have implemented several control and data obfuscations in a source-to-source tool for C language programs. Of particular interest to this project is their technique for flattening control flow, and for exploiting the difficulty of alias analysis to prevent static analysis. They studied the performance and precision of the results of running static analysis tools, specifically 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, they discuss a survivability architecture and a number of other potential obfuscation techniques. These techniques include multi-threading, variable splitting and obfuscating procedure call and function call interfaces.

6.1.6 Hohl: Time Limited Black Box

[Hohl98] proposes using code and data obfuscation techniques to construct time-limited black box agents. He assumes that these agents can execute safely for a period of time, based on how much time-consuming reverse engineering is required for a host to know how to make any changes to an agent's code or state that could be useful to the host. During this period an attacker can, at most, make random changes to the mobile agent. Similarly, this technique protects secrets for a limited period of time, since a host must reverse-engineer an agent to know how to interpret its state.

Like all programs, obfuscated agents are subject to black box testing. [HR98] defines a protocol to detect such testing by using a trusted registry server to monitor agent execution.

6.1.7 Barak *et al*: Obfuscation is Impossible

Barak *et al.* [BGI+01] presented a paper on the theoretical limits of obfuscation techniques. Informally, they prove that all obfuscators leak some information, in fact there is a question whose answer is that all obfuscators must leak, or put another way, a completely secure obfuscator is impossible.

To be more specific, one must first define what “secure” means in the obfuscation context. One approach is to examine a single use case by specifying the secret to protect and defining "security" as that which makes the secret difficult to deobfuscate. The disadvantage of this approach is that it must be repeated for every problem.

An alternative would be to prove a very general security property of an obfuscator once and for all, and then use obfuscation as an opaque building block for other protocols, algorithms, and techniques. This second type of analysis is the sort used to describe the security of cryptographic methods; higher-level constructs such as pseudorandom number generators are built up and analyzed in terms of lower level ideas such as one-way functions. In trying to apply this sort of analysis to obfuscation, the “virtual black-box property” (VBB) seems to be the most natural and useful definition of security. Informally an obfuscator is VBB-secure if given white box access to a program, an attacker could derive no more information about what the program computes than they could with black box access. Barak *et al.* prove that no obfuscator is secure in this sense; they prove it for circuit obfuscators by describing a class of properties of programs (or circuits) that cannot be obfuscated. They also examine a particular use case and show that it is impossible to protect with obfuscation, whether the obfuscator is VBB-secure or not.

These results strongly suggest that analysis of obfuscation must be done on a use-case basis, because it is impossible for an obfuscator to be secure in every case. This paper does not prove, and should not be construed to say, that there are no use cases that can be secured with obfuscation; in fact it suggests several weaker criteria for obfuscator security that may be useful for certain cases. It only proves that no obfuscator is completely secure in the general case. The reason their conclusion does not preclude obfuscation for particular use-cases is that the information leaked may be of no value in that use-case.

One should note that the formalism used in [BGI+01] addresses only the situation where the obfuscator is seeking to hide information about the function that the program computes. In fact the VBB definition of security can be thought of as the strongest possible complexity theoretic definition of security in this situation. In this sense it is very similar to the definition of secure computation given in [MR91].

6.1.8 Sander/Tschudin/Cachin/Micali: 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 summarized here. In effect, these techniques obscure the operation of a program by cryptographic

techniques and in some cases by also distributing the program across multiple agents or to trusted servers.

The mobile agent paradigm can be viewed as a situation where several mutually distrusting parties want to perform a computation, while keeping certain parameters of that computation secret from each other. The problem is further constrained by the requirement that the computation take place off-line, i.e. once the agent is sent from one host to the next, no further communication is required between them to run it. In [GMW87] it is proven that without that constraint, and assuming that the majority of players are honest, then there is a cryptographic protocol that allows them to carry out their computation securely. This protocol does not rely on obfuscation in any way and is provably secure under reasonable assumptions. Since then a great deal of progress has been made towards a general solution to the mobile agent problem, using techniques based on oblivious transfer of secrets and encrypted circuits.

Sander and Tschudin [San98a, San98b, San98c] describe a model where a source launches a mobile agent that executes some program on an untrusted host (the agent receives inputs from the host), and then sends the results back to the sender. In their approach, the host executes a program that embodies an enciphered function. The host cannot decrypt the program to discover the original function. The only information exposed to the host is the result it computes and the inputs it provided.

The techniques of Sander and Tschudin are effective only for the evaluation of polynomial expressions, a very limited subset of agent algorithms. Sander, Young and Yung [SYY99] have developed a polynomial time (in circuit size) technique by which any circuit in NC^1 (functions with circuits that are logarithmic in the size of their inputs) can be evaluated.

Cachin *et al*. [CCK+00] developed a technique that, assuming the hardness of the decisional Diffie-Hellman problem, can be used to protect polynomial-size circuits, and they described how to support mobile agents that visit multiple hosts. These approaches do not allow interaction between the encrypted function and the executing host, i.e., the agent receives inputs from the host, but cannot provide clear-text results to the host. In particular [CCK+00] proves that if an agent can be expressed as a circuit, has a fixed list of hosts to visit, and does not need to provide output except to its originator when it returns, then there is a protocol for its execution that doesn't reveal any information to any of the parties that they would not already know. In a sense this is a generalization of oblivious transfer: The agent's input is a database and the agent is a query. The protocol allows a query to be answered without the database knowing what the query was.

The requirement of [CCK+00] that the agent not provide output to the hosts arises because it is impossible to prevent the hosts from simply rerunning the agent on different inputs in order to gain more information then they ought. In order to prevent such reruns, [ACC+01] adds a generic third party to the situation. The third party does not need to know anything about any particular agent and cannot learn anything about the agent computations without colluding with one of the participants. The addition of this third party allows rerun attacks to be eliminated, because running the agent requires communication with the third party, so the requirement that the hosts not receive output from the agent is lifted. Furthermore the agent can specify its next destination on the fly,

instead of needing a fixed list of hosts to visit. In principle [ACC+01] provides a solution to a subset of the problems that the SPMA project set out to solve. For example, the "comparison-shopping" agent can be made secure by it, although as the agents get bigger the protocol can get quite slow, and the method becomes impractical for medium or large agents. Also, any agent that requires frequent communication with the host (e.g. database searches) would be completely unsuitable, as the agent would have to contact a trusted server after each round of communication. The introduction of a trusted third party is similar to SPMA's multiple agentlet scheme; in fact, one could replace the third party with an agentlet that implemented the third party protocol and ran unencrypted on a separate host from the primary agent. Clearly then the two hosts running agentlets would have to collude in order to learn anything they weren't supposed to, because the second host is playing the role of the generic third party.

These techniques are limited to solving small functions and if a trusted server T is used, and the server crashes, the protocol stops: another trusted server cannot be used to complete the protocol. T must not collude with either the originator or the other hosts in order to protect everyone's interests.[5] The Cachin *et al*. model only considers a single interaction between the mobile agent and each host that it visits and a mobile agent only visits a host once, but simply visiting a host repeatedly (with a different version of the program) can simulate multiple interactions.

6.1.9 Cloakware

People at Cloakware have written several papers [CW00, CW01a, CW01b, CW01c] on obfuscation. The company uses the term "Tamper Resistant Software" to describe their approach. Their overall approach is similar to Wang *et al*: obfuscation is designed to eliminate any benefit of static analysis and force a difficult dynamic analysis to be performed. In particular, the problem of statically analyzing the control flow of the transformed program is showed to be reducible from the acceptance problem for Linear-Bounded Turing Machines (LBTM). Since the acceptance problem for LBTMs is PSPACE-complete, the problem of statically analyzing the transformed program's control flow graph is PSPACE-complete as well.

Cloakware's product obfuscates control flow by 1) applying transformations to sequential programs that flatten their control-flow structure, and 2) grouping the control-flow of the source program on a switch statement called a dispatcher, so that the targets of static jumps are determined dynamically. The dispatcher may be viewed as a deterministic finite-state automaton (DFA). Cloakware claims that once the program has been transformed in this manner, in order to obfuscate the program's flow control, it suffices to apply further obfuscation techniques to the dispatcher. Having already established that the problem of analyzing the dispatcher, named the REACHABILITY problem, is PSPACE-complete, the Cloakware paper proposes implanting another instance of this PSPACE-complete problem into the design of the dispatcher. Essentially, they take the natural instance of the problem that emerges from applying their transformation to the

[5] The authors want to limit what the originator and the hosts can learn about the inputs provided by the other hosts.

source program, and another instance of the problem that is the encoded form of an instance of the LBTM acceptance problem, and form a combined, merged instance of their REACHABILITY problem. They then argue that the resulting obfuscated program's resistance to deobfuscation of the program's control flow is guaranteed by the hardness of the implanted, merged instance of the problem. This portion of their argument appears to be intuitive, rather than rigorous, as they do not establish an argument regarding the inability of an attacker to separate the merged instance of REACHABILITY into the original two instances.

There is a second point upon which their argument is intuitive rather than rigorous. They argue that:

> *The redundancy of program components is the basic property to be checked to comprehend (or to optimize) a program. Therefore, it is highly reasonable to measure a resistance of obfuscated programs in terms of the complexity of redundancy checking for these programs.*

Their argument then continues by establishing that determining if a basic block of code, or a variable is redundant, is itself a PSPACE-hard problem. While the formal part of the argument is undoubtedly correct, it is an open question whether other techniques that do not directly attack the redundancy problem can be used to successfully perform static analysis of the program's control flow.

Next we consider Cloakware's complementary effort in obfuscation of data as opposed to code. They obfuscate data by a combination of several mathematical transforms, these are [NCJ01]: 1) polynomial transforms based on linear additive and multiplicative encodings; 2) residue transforms (essentially using Chinese Remainder Theorem to represent integers) and 3) “matrix or multi-linear” transforms using polynomials of several variables. Cloakware conducted a reverse engineering experiment on their data obfuscation using an outside organization. [NCJ01]. As a result of the experiment they claim that their data transform techniques alone can delay a knowledgeable insider for a month from successfully reverse engineering the transformed program. They define a successful reverse engineering of a program as reaching a level of program understanding sufficient to successfully alter the program's execution.

They report that their techniques increase a program's size by a factor of three to five times [NCJ01] (2-3 times in [NCJ+02]), and a slow down a program by about five to ten times [NCJ01] (4-5 times in [NCJ+02]) but point out that only part of the program may need to be obfuscated.

Cloakware’s commercial product may use some of the techniques described above. Sample code obfuscated with their product may be available only when reviewers sign an agreement. Because of this, the security of the scheme is questionable. Additionally, their results contradict both [Appel02] and [Schwab].

6.1.10 Appel

Prof. Andrew Appel of Princeton recently produced a nice result showing that “Deobfuscation is in NP” [Appel02]. He considered the complete deobfuscation problem i.e., positively determine the entire “source program” or a trivial equivalent from the

obfuscated binary. More formally, Appel assumes that the obfuscator is known, runs in polynomial time, and that the obfuscated programs are only polynomially slower then the source programs they were produced from. Further he assumes that the obfuscator is a deterministic algorithm (F) that takes a program to obfuscate (S) and a key (K). He then defines "complete deobfuscation" of an obfuscated program P as finding a source program that when obfuscated yields P. The algorithm is simply to nondeterministically guess all source programs, guess a key, compute P' = F(K, S), and verify that P' = P. This problem is clearly in NP because every element is polynomial.

What this result tells us is that in NP-time one can "reverse" the obfuscator. It does not necessarily mean that one can find the information wanted in NP-time, because that might be hard even without obfuscation. For instance, one might want to know whether a certain program is malware (e.g., "wipes the hard drive"). Malware writers may use obfuscation to make it more difficult to determine this. Appel's result says that in NP-time one can reverse the writers' obfuscator, but it does not say that one can determine if the program is malware in NP-time, because determining if a program is malware is undecidable whether it is obfuscated or not.

Since many program analysis problems are undecidable, or at least not NP, many authors have tried to embed these problems into their obfuscators in the hope that deobfuscation would inherit this hardness and also be outside of NP. Appel's result says that these arguments are necessarily erroneous. This error may arise from a backwards reduction (A is hard, B can be reduced to A therefore B is hard), or as above, because the deobfuscation problem under study was outside NP even for unobfuscated programs, or for other types of faulty reasoning.

Appel further states that,

> "In practice, it is my suspicion that program obfuscation will not provide strong security in practice because the resources and techniques available to attackers are so numerous and powerful: debuggers, simulators, test coverage tools, decompilers. Then, once the attacker has information about the algorithm F, it should be possible to make specialized execution-analysis tools tuned to F."

This point of view is consistent with our own. Also, in practice one may not need to completely deobfuscate a program in order to attack it, and one can typically break down the deobfuscation problem for a program into multiple smaller/simple deobfuscation problems and solve them piecemeal.

6.1.11 Ahpah and InterTrust

Ahpah Software produced a commercial Java decompiler and an obfuscator and once claimed that its obfuscator could not be reversed. More recently, their web site states that on further reflection, they believe that unbreakable obfuscation is impossible. We discussed their position via email with Paul Martino, formerly a principal of Ahpah, especially the statement "we did a lot of research on obfuscation and it's impossible" in their FAQ. Ahpah, InterTrust, and Princeton did a five-year research project on obfuscation, funded by a commercial organization. They chose similar obfuscation strategies to ours: making control flow dynamic, changing the representation of variables,

commuting control flow, etc. However, they ran into the same problem we did. Except for a few instances of irreversible information loss (variable names, variable extents, etc.), they could think of a way to reverse the transforms as quickly as they came up with the transforms themselves. As far as theoretical results go, they do not have any, because their employer was much more interested in the practical side of things. Another former member of this team stated informally that InterTrust felt that the hacker community would be able to break the best they could do within 24 months.

6.1.12 Schneier

Bruce Schneier's CRYPTO-GRAM articles often contain statements about trusted client software (which many uses of obfuscation aim to create) being impossible. The May 2000 CRYPTO-GRAM in particular, says, "Building a trusted client in software, and trying to limit the abilities of a user, on a general purpose computer is doomed to failure. For now, though, it provides a nice false sense of security." [Sch005] He also discusses Kerckhoffs' Principle (that a cryptosystem should not need a secret algorithm) in relation to other security schemes: "A corollary of Kerckhoffs' Principle is that the fewer secrets a system has, the more secure it is. If the loss of any one secret causes the system to break, then the system with fewer secrets is necessarily more secure." [Sch025]

6.1.13 Fraunhofer CCRG

Chenghui Luo at the former Fraunhofer Center for Research in Computer Graphics had an AFRL project for Rome Labs. Their web page claimed they can do "perfect" obfuscation and watermarking. What they mean by "perfect" is that they obfuscate the program and the libraries it calls (as our JBET obfuscator does also). Luo states, "It's hard to define a quantitative measure for the strength of obfuscation, and in our project, we didn't define it. The reason is that obfuscation is to remove or hide information, which falls in the 'security from obscurity' model, so it may be a wrong question to ask, based on a 'security from complexity' notion." [Luo02].

6.2 Reverse Engineering

The purpose of code obfuscation is to prevent reverse engineering. See Figure 2 for an illustration of the reverse engineering process.

A good deal of the research on reverse engineering does not consider obfuscated programs. This section describes some of the tools and techniques that do.

We partition these tools and techniques roughly into **static** and **dynamic** methods. However, real attempts at reverse engineering software whether performed in a lab or "in the wild" (e.g., by software license crackers [Acad]) involves the use of both types of techniques.

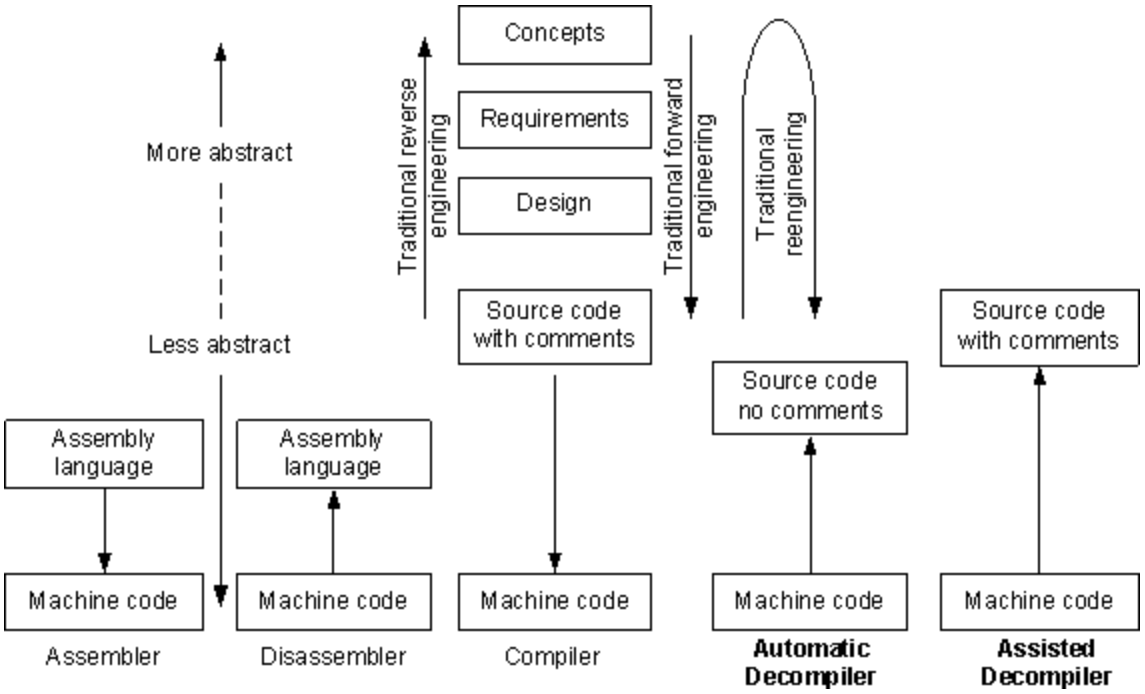

Figure 2: Mainstream software engineering and reverse engineering tasks [PTO-DR]

Figure 2 depicts the traditional breakdown of software (forward) engineering and reverse engineering tasks. We view decompilation and disassembly as the lowest forms of all software reverse engineering and when dealing with an obfuscated program these are the most difficult steps, especially when the goal of the reverse engineering is to accomplish the recovery of a secret or the bypassing of a control.

6.2.1 Static Reverse Engineering Methods

Static methods are essentially algorithmic methods: i.e., they can be modeled as applying an algorithm that produces useful information from the obfuscated program (e.g., a less obfuscated version of the program. Note that a standard binary is more obfuscated than the assembly source, which is more obfuscated than the high-level source.) The static methods used by the reverse engineer may be known to the obfuscator prior to performing the obfuscation.

In mainstream reverse engineering there are two primary classes of static methods, disassembly and decompilation, which are discussed below. Such methods can be abstractly modeled in complexity theory but much care must be taken in interpreting the results since a tool that only provides an approximate solution to a deobfuscation problem (e.g. detects 90% of the dead code) may be nearly as useful to an experienced reverse engineer as a perfect solution and problems that are theoretically difficult in the worst case may have efficient solutions in the average case, special cases, etc.

6.2.1.1 Disassemblers

Disassembling is the process of translating an executable program into its equivalent assembly representation. The greatest problem in disassembling is distinguishing code from data especially on architectures that can execute from data segments, store data in

code segments, or write to code segments. In principle this problem can occur in any architecture because a program can store another program as data and simulate it.

The problem occurs mainly when data is embedded inline with code. On variable-length instruction architectures, mistakenly disassembling inline data can cause instruction alignment problems in the disassembler, causing it to incorrectly disassemble the code that follows the data. Disassembly accuracy can be improved by making use of knowledge of compilers and libraries, but that method is not generally explored for obvious reasons. The best algorithms currently available essentially do a reachability analysis under the assumptions that only jump targets contain executable code and that all code that is executable is jumped to [Schwarz].

In theory, accurately disassembling a program in the general case is undecidable and hence cannot be fully automated for all programs. However even partial disassembly of a program is a great aid to the reverse engineer, who can combine partial disassembly (from static disassemblers) with disassembled execution traces generated by debuggers, logic analyzers or in-circuit emulators.

Most debuggers provide a simple disassembler and there are a number of stand alone disassembler products [PTO-DA], therefore these tools are readily available to potential adversaries of the developers of obfuscated programs.

6.2.1.2 Decompilers / Reverse Compilers

Since disassembly is required for decompilation, decompilation has all of the same issues in addition to the issues around identifying the nature and scope of control structures, and reconstructing non-primitive data types.

The main steps in decompilation are [PTO-DR]:

- Disassemble the program. This step is crucial because all other analyses depend on it. For example, the quality of a control-flow analysis depends on the quality of the raw control-flow information it gets.
- Perform semantic analysis to recover low-level data types such as integer variables, and to simplify the decoded instructions based on their semantics.
- Perform data-flow analysis to remove low-level aspects of the intermediate representation that do not exist in high-level languages (e.g., registers, condition codes, stack operations) and to reconstruct expressions.
- Perform control-flow analysis to recover the high-level control structures in each procedure.

Perform type analysis to recover high-level data types such as arrays and structures, classes, etc.

Architecture alone can make a program easy or hard to decompile. For example, Java bytecode is relatively easy to disassemble and is very object-aware. Class files are already structured into classes and methods, and the instruction set has single instructions to do complex, high-level things like call a virtual method. Consequently, a lot of the work involved with executing a Java program is done by the virtual machine. By

comparison, the work that can be done by a single instruction on a real processor is relatively small because of practical limitations. A C++ compiler for a real architecture has to synthesize all of its OO functionality from much lower-level primitives, which obviously requires more work to interpret as high-level phenomena.

Interestingly, many of our obfuscations attempt to make Java bytecode look more like C++ object code.

6.2.1.3 Specialty Tools and Techniques

Many tools and techniques have been developed by the research community to perform static analysis on object and source code. Such techniques include program slicing and alias analysis.

6.2.1.3.1 Program Slicing

The slice of a program with respect to a set of program elements S is the program elements that S is, or might be, code- or data- dependent on. Slicing gets rid of code and data that are irrelevant to S, effectively reducing the size of the program to analyze [WPS].

Extensive research has been done on slicing including work by Weiser [Weiser84], Tip [Tip95], Horwitz [KH02] and Reps [RT96, Reps98]. This research has led to the development of research slicing tools including The Wisconsin Program-Slicing Tool [WPS], Chopstick [CP], and commercial products such as CodeSurfer [CS03].

Slicing has also been used to extract functions of interest from executables [LV97].

6.2.1.3.2 Alias Analysis

Aliasing occurs in a program when multiple pointers reference the same memory location but are used separately. Given two pointer variables that refer to the same object, making a change through one changes the value that would be read through the other. Any analysis that considers data flow will be affected by knowledge that the pointers refer to a single object as opposed to two. As a result, being able to detect aliases has implications for many types of static analysis as well as for reverse engineering.

The problem of detecting aliases comes in several forms, whose solutions belong to different theoretical complexity classes [LR91, Deut94]. Programming mechanisms that create aliases include the following: reference formal parameters, single-level pointers, multiple-level pointers (i.e., pointers that point to pointers), and pointers to structures containing pointers. Alias analysis is used in software analysis tools such as JAAT [KOK+01] and Ajax [Ajax].

6.2.2 Dynamic Methods

Reverse engineers typically employ a combination of static and dynamic methods to analyze a program. The following summary of the reverse engineering of the Internet Worm from the famous “With Microscope and Tweezers” paper [ER89] demonstrates this.

The Internet was attacked on November 2, 1988 by a virus. In response a handful of teams across the country worked to reverse engineer the virus to discover how it worked, specifically, the vulnerabilities it exploited and how it propagated. The MIT team reported their findings in [ER89]. They conducted their analysis mainly by decompiling the virus rather than through black box testing. The team viewed the deobfuscation/reverse engineering task as:

- isolating a specimen of the virus in a form, which could be analyzed.
- "decompiling"[6] the virus, into a form that could be shown to reduce to the executable of the real thing, so that the higher level version could be interpreted.
- analyzing the strategies used by the virus, and the elements of its design in order to find weaknesses and methods of defeating it.

The first two steps were completed in less than two days, primarily by the efforts of the MIT team and people at Berkeley.

While the virus used a number of methods to obscure itself including preventing core dumps and erasing its argument list it was not really obfuscated nor was the actual virus very large. Its only real obfuscation was the XORing of strings.

It should be pointed out that the goal of the MIT and Berkeley teams was a complete understanding of the virus, not just understanding a single mechanism. When the goal is defeating copy protection [Gos85] the reverse engineer only needs to know about the copy protection mechanism; and providing effective defense through obfuscation is more difficult simply because the goal is smaller.

In the remainder of this section, we will discuss some of the dynamic analysis tools that are available. These tools all offer greater capability and better protection against anti-reverse engineering techniques than the tools used by MIT and Berkeley. The typical dynamic analysis tools available to the reverse engineer consist of debuggers and software emulators as well as logical analyzers and in circuit emulators.

6.2.2.1 Debuggers and Associated Tools

A debugger is a utility program that allows a reverse engineer to run a program while controlling its execution and examining the values of its variables. Many debuggers provide an execution history mechanism that at a minimum allows a reverse engineer to see a trace of previously executed instructions. Typically these traces are of limited length and are read only, i.e., the reverse engineer cannot roll back the state of the program, change an earlier state, and then resume execution.

EXDAMS [Bal69] was one of the first debuggers to provide a long execution trace facility. It was an interactive FORTRAN debugger developed in the late 60's. Under EXDAMS the program being debugged was first executed in its entirety and the full

[6] Decompiling was performed by first doing tool assisted disassembly, followed by with decompilation done by hand. The disassembly tools were simple disassembler (adb), an architecture manual, and the UNIX sources. However, even at the time (Nov. 1988) based on its experience with PC viruses, the National Computer Security Center felt that more sophisticated analysis tools must be developed.

execution history was saved. Then the program was "re-executed" through this trace. This re-execution could be backtracked any time using the trace. However, the reverse engineer could not change values of variables or registers.

INTERLISP [TM81] and the Cornell Program Synthesizer [TR81] provided execution traces with undo operations. These systems maintained a history list of operations while recording their side effects. They used bounded history lists: as new events occurred, the existing events on the list were aged, with oldest events "forgotten." The first complete execution trace solution was provided by the experimental Spyder debugger [ADS91]. By using such a tool the reverse engineer can stop the program at any point and examine past events that lead up to the current state, restart the program in a modified version of the previous state and observe the impact of the changes. Depending on how a debugger is implemented, a program can detect that it is running in a debugger and attempt counter measures.

6.2.2.2 Software Emulators

Simple software emulators are essentially instruction set interpreters for various processors or family of processors. More sophisticated emulators such as the Stanford SimOS/Embra [SimOS, Embra, RBD+97, WR96] emulate processors, caches, and memory systems of a set of processors. Some systems are toolkits for building emulators for different architectures [UC00, OG98]. For example, the New Jersey Machine-Code Toolkit [NJMCT, RF97]. This toolkit provides a specification language for describing the behavior of processors allowing a reverse engineer to create emulators for new systems. It is reasonable to assume that the reverse engineer of a program that we would obfuscate may have a high quality emulator. The emulations are typically not perfect; for example, providing the correct timing is difficult and detectable, so if the emulator is known to the obfuscator it may be possible to exploit such imperfections to improve the obfuscation.

6.2.2.3 Logic Analyzers

A logic analyzer is a physical test instrument used for developing, debugging, and maintaining digital systems. A logic analyzer can show the prior events that occurred at probe points when triggered by a predefined set of stimulus signals and subsequent events. Two well-known manufacturers of logic analyzers are Agilent/HP and Tektronix. Their systems support many popular processors.

Logic analyzers provide the reverse engineer the ability to monitor a program without the program being able to detect the monitoring even if the obfuscator has knowledge that the monitoring will occur. The logic analyzer may be a stand-alone system or PC-based. Some common characteristics of interest to software reverse engineers:

- A good analyzer automatically disassembles, shows executed instructions and filters out unexecuted code fetches, trigger in instruction execution patterns, memory access patterns, register contents, data bus patterns.
- Analyzers provide large buffers for storing system activity that occur before and after a trigger.

- Acquire every bus cycle in real time without interfering with full-speed operation of the processor or bus; probing cannot be detected in software so static obfuscations cannot provide a defense. Only dynamic obfuscations can , which currently push the limit of what is feasible.
- Some systems correlate execution traces to high-level source code (if source provided by the user). This can be very handy if the reverse engineer is dealing with an executable with partially known source code, i.e., standard language library.

6.2.2.4 In-circuit Emulators

Unlike a logic analyzer, which attempts to passively monitor signals sent between devices, especially between processors and memory and I/O devices, an in-circuit emulator replaces a component of the system with a special device (sometimes called a pod). The pod is controlled by the emulator and provides data to it. Unlike a logical analyzer the emulator can stop execution and change the contents of the emulated processor registers. These devices are especially well suited for injecting faults for reverse engineering purposes. In-circuit emulators and related systems such as in-circuit debuggers offer the reverse engineer greater capability than a logic analyzer, but at some potential loss in stealth since the pod is not exactly the same as the device it is replacing, especially with respect to undocumented features and bugs that the device may have. Two well known manufacturers of these devices are Lauterbach and Microekintl.

The power of logic analyzers and in circuit emulators have long been recognized by the reverse engineering community and also by the maintainers of license cracking web sites. [Acad]

6.3 Cryptography

People sometimes draw analogies between the security of cryptosystems and the security of obfuscated programs. They both obscure an object through a keyed transform, however, their uses are very different and reverse engineering them is very different. An obfuscated program must retain the functionality of the original program, so attacking obfuscation can be done through interaction with the program and noticing patterns by white-box inspection. There is no interaction possible with an encrypted object; ideally it should be indistinguishable from random bits. Theoretically, if a cipher is good, the only possible attack is a brute-force attack on the key. Obfuscation transforms are not designed to be reversible, so finding the key yields almost nothing.

6.3.1 Exploiting Error Conditions

Real world cryptosystems can be less secure than their abstract models. For example the different error conditions of a real implementation occasionally yield information about the key, for example attacks on RSA PKCS #1 v1.5 by Bleichenbacher [Ble98], PKCS #1 v2.0 by Manger [Man01] and on Cipher Block Chaining modes by Vaudenay [Vau02]. These kinds of attacks use error conditions generated by implementations of decryption functions (or in specifications of how to securely use a cryptosystem) to slowly learn

about the private keys used by the functions. In the published results new mathematical models of the cryptosystems are developed that can be used to determine the private keys by applying typically large amounts of data collected by probing the system. Such attacks roughly correspond to a reverse engineer learning about an obfuscated program by manipulating the programs input parameters, calling the function, and examining the return values.

6.3.2 Power Analysis and Similar Attacks

Real cryptographic systems can leak information through electromagnetic radiation [Tempest], power consumption [CJR+99, KJJ99], or the time required to perform operations [Koc96]. While these attacks are usually performed in an otherwise black box setting, some of these attacks determine events that may be observed during the reverse engineering of some obfuscated program. An example of this relationship is that simple power analysis exposes information about the execution path of the function. Simple power analysis can expose:

- Sub-key bits of a DES key from the DES key schedule behavior and from the behavior of DES permutations
- Duration of string comparison operations, usefully for examining a secret bit by bit, and
- Data used by modular multipliers and by modular exponentiation operations. [KJJ99].

Timing attacks also exploit data dependent variations in control flow of cryptographic functions [Koc96]. Tempest attacks can reveal internal state or data flows of a function, for example traffic on a data bus,[7] which is not that different from a reverse engineer monitoring a data bus using a logic analyzer.

6.3.3 Fault Analysis Attacks

Other probing attacks against cryptosystems are the Differential Fault Analysis (DFA) attacks [BDL97, BS97, BMM00] including glitch attacks [AK96, AK97, SA02]. Such attacks involve interjecting faults into hardware or software that is performing a cryptographic function. These faults may attempt to manipulate crypto-variables, (e.g., transient register faults), or read bits of crypto-variables (e.g., detecting leakage currents), or alter the control flow of the function. Some published DFA attacks, like the error condition attacks of Section 6.3.1, result in new mathematical models of the cryptosystems that can be used to determine the private keys by applying data collected by probing the system, while other fault analysis attacks are a direct (or indirect) reading of critical memory locations.

Conventional models of cryptographic functions assume a protected space in which cryptographic functions are performed. The adversary can attack the system by passively

[7] Sometimes the signals from a bus are so strong that an ordinary AM radio receiver can be used to detect them [KJJ99].

monitoring physically insecure interfaces to these spaces and actively probing (i.e., sending data) only these interfaces.[8] For this general model cryptographers seek to establish that the valid cryptographic functions only have to perform a feasible number of operations relative to a value (called a security parameter) while an adversary that can do anything allowed by the model has to perform an infeasible number of operations to break the system. However, changing the model alters the adversary's cost relative to the conventional security parameter or introduces an alternative new cost parameter, (e.g., rather than the cost of breaking the system being exponentially related to the size of a cryptographic key, the cost may be linearly related to the size of the memory used by the function).

7 Our Conclusions

The major issue at stake in this research is whether automated obfuscation tools can produce obfuscated code that is resistant enough to analysis so that deobfuscation always requires significant manual analysis (or manual guidance of deobfuscation tools). If those techniques are sufficient to force a manual component to deobfuscation, they have provided a positive cost/benefit tradeoff between obfuscation and deobfuscation since the obfuscation techniques were 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. However, our experiments and much of the related work we examined lean in the other direction; that is that automated obfuscation will not be useful for protecting long-term secrets. Additionally, our experiments show that the attacker has to perform less computational work than the obfuscator.

It should be noted that security by obscurity (e.g. a secret obfuscation program) is not a solution as there are numerous ways the attackers could obtain the "secret" obfuscation algorithms, especially if the obfuscator is a commercial product.

It should be noted that security by obscurity (e.g. a secret obfuscation program) is not a solution as there are numerous ways the attackers may obtain the "secret" obfuscation program, especially if the obfuscator is a commercial product.

The SPMA scheme would work if (1) a lower bound on deobfuscation could be established, or (2) the obscurity of the obfuscation technique could be relied upon. Alternative 1 is not possible for obfuscation algorithms that blindly obfuscate all programs -- but the Barak, *et al.* paper suggests that perhaps some restricted class of programs could be obfuscated -- in any event, this is a hard path forward. Alternative 2 is really security through obscurity, which is known to be fragile. But as long as a user is willing to replace an obfuscation method with a new one each time it is detected that it has been broken, and the user is willing to accept some intrusions/compromise of some agent sessions on occasion, then this scheme could still be used.

[8] The classic notion of an insecure channel is a collection of the insecure interfaces.

Future arguments surrounding strength of obfuscation should attempt to incorporate the practical differences between idealized models of computation such as Turing machines and real finite space and time register machines such as Intel Pentiums. Again, we found that recursion theory leads to results not useful in practice, as it is primarily concerned with what is computable and what is not. In practice being able to compute something "most of the time" would be good enough for a deobfuscation process, especially when monitored by humans.

7.1 Don't Depend on Obfuscation for Security

First and foremost, we conclude that (at this time) there is no reason at all to depend on obfuscation for security. This is not to say that obfuscation should not be used, but high-value secrets must not be entrusted to it. For instance a game company might use obfuscation to prevent copyright infringement, but it would be very unwise for such a company to forecast its revenue assuming that the obfuscation will hold. They must assume that it will not, and plan accordingly (The history of commercial software greatly favors that it will not).

7.1.1 Argument From Theory

We make this conclusion for several reasons. The first reason is that we know of no theoretical result that suggests a deobfuscation problem is hard. Although we trust cryptography and have no proof that a strong cryptographic algorithm exists, we can prove that one exists assuming one-way functions exist. We have found nothing analogous to that result for obfuscation. There are also results to the contrary, such as Barak.

Of course some deobfuscation problems cannot be formalized at all, such as the secret algorithm use-case. These situations are even worse, from a trust point of view. Not only do we not have a lower bound, or a lower bound relative to a reasonable assumption, but we cannot hope to ever have one, because the problem: to "understand" the secret algorithm, cannot be defined in any formal way. The success criteria for these types of problems are fundamentally human, and not mathematical; therefore, their difficulty cannot be analyzed. You can never have assurance that tomorrow someone will not find a new way of looking at your code that allows him or her to "understand" how it works. Thus obfuscation for either formalizable or non-formalizable use cases must be regarded as "icing on the cake," and not an essential component for security.

7.1.2 Argument From History

Our second reason for concluding that obfuscation cannot be trusted is the real-world history of obfuscation. Software companies have been trying to use obfuscation (to prevent copyright infringement, to prevent competitors from creating a product that is compatible with theirs, to hide APIs, to hide algorithms) for years, and their schemes are broken routinely, and often very soon after release. We know of no obfuscation that has been actually deployed and withstood serious attack. This indicates that all of the techniques in current use are far from effective. If in fact there were a use-case where

obfuscation can actually be secure, then that obfuscator would need truly new ideas. Therefore no incremental improvement in obfuscator technology is useful.

7.2 Barak's Result is Very Strong

Our second conclusion regards purported proofs that obfuscators are secure. Barak's paper clearly proves that a general obfuscator does not exist. Therefore any claim that an obfuscator protects all the information not explicitly revealed by the program's behavior is false. Any claim that an obfuscator works in any use-case is false. Any claim of "drop-in" security, "just run it through our tool and it'll be secure" is false. Any attempt to analyze the hardness of deobfuscation must identify the secret that is being protected, i.e. it must choose a particular use-case (or a proper subset of use-cases). This provides a sort of litmus test for obfuscation-related snake oil: if someone makes general claims about obfuscation security, rather than about protecting a specific secret, then they are wrong. Note that one-round obfuscated program execution (e.g. Cachin [CCK+00]) is not general program execution, and examining claims about such schemes will necessarily be different.

Consequently we suggest that anyone seeking to write an obfuscator first lay out very clearly what secrets they wish to protect. It may be (probably is) the case that different types of secrets require entirely different types of obfuscator to protect them (if they can be protected at all).

7.3 Better Solutions Are Available

Our third conclusion is that many of the use cases that people suggest have better solutions than obfuscation.

One common use case is secure execution of mobile agents. These use cases can be solved via cryptographic means. Unfortunately the cost can be rather high except for small agents, but if it is not too high for what you want to do, then the cryptographic approach is clearly superior to obfuscation.

Because obfuscation is rather difficult to implement, untrustworthy, and has a history of being broken, we suggest that obfuscation not even be considered until all other possible approaches have been ruled out. It should be noted that there are several formulations of security problems that are impossible to solve, but by making slight modifications they become possible. We suggest that people considering obfuscation ask themselves if the formulation of their problem is too restrictive and eliminates from consideration security mechanisms that are effective. For example, the "comparison-shopping mobile agent" use case is not solvable in the usual mobile agent set up, where the computation takes place totally offline. However by adding a generic trusted third party it becomes possible. [CCK+00] Another example: software-only trusted clients are impossible [Sch005], but by adding the assumption of tamper-resistant hardware it becomes possible to hinder attackers who cannot attack the hardware.

7.4 Applicability beyond Java

We could devise analogues of many of our obfuscating transforms on Java programs for use on native languages similar to Java, such as C++. Assume that the native program obfuscator is implemented as a compiler back-end, or is a source-to-source obfuscator, so it can have easy access to structural information. Note that an ordinary C++ compiler performs some of our obfuscating transforms already, such as the replacement of class names with unique identifiers. Other control flow transforms, such as using table-computed jumps in place of processor conditional jump instructions, can be implemented in the same way as our obfuscator as all processors should have a jump-to-address instructions. Our call stack mechanism is considerably less revealing than Java's mechanism, but is probably equivalent to a stripped executable call stack. The local variable transforms could be implemented similarly, as our DAG code representation is similar to (at least) the compiler's parser, and possibly its internal representation (for performing optimization and code generation from).

7.5 Summary

We introduced three unknowns about obfuscation. All three remain unknown, especially when restricted to certain use cases, or by having non-general program execution models. Our experiments showed that “simple” obfuscation techniques are not secure, and they create less work for the attacker than the defender. The findings on patterns show that the variability able to be produced by obfuscation tools similar to ours is very limited.

The SPMA Project set out expecting obfuscation to be merely a software engineering problem. It turned out to be something much more difficult, if not impossible. None of the techniques we know of are sufficient to get any reasonable level of assurance that our secrets are secure. Furthermore, Barak’s result rules out general obfuscation, which is what we wanted to achieve. If obfuscation is to succeed, it must concentrate on particular use cases, and it must contain fundamentally new ideas. We have seen that obfuscation is more than an engineering problem, although it will be a difficult engineering problem as well: it is a mathematical problem. In order to come up with an effective obfuscator, researchers need some new insights that go beyond the techniques currently known.

8 Acknowledgements

Our thanks to our program managers Jay Lala and Doug Maughan, for their support. Thanks also to our former team members Lee Badger, Doug Kilpatrick, Steve Kiernan, and Larry Spector for their contributions. Thanks also to Andrew Appel, Ross Anderson, Stanley Chow, John Knight, Chenxi Wang, Christian Collberg, Paul Martino, and Chenghui Luo for stimulating discussions and ideas.

9 Bibliography

[Acad]	Academy of Reverse Engineering: ESSAYS 1-100, http://www.woodmann.com/fravia/aca100.htm [License cracking website, with many "how to" essays on reverse engineering].
[ACC+01]	J. Algesheimer, C. Cachin, J. Camenisch, and G. Karjoth. "Cryptographic Security for Mobile Code," in Proceedings of 2001 IEEE Symposium on Security and Privacy, Oakland, May 2001, pp. 2-11.
[Ada95]	Ada 95 Language Reference Manual, Intermetrics Inc., ISO/ICE 8652:1995.
[ADS91]	H. Agrawal, R. DeMillo, and E. Spafford. An Execution Backtracking Approach to Program Debugging. IEEE Software, pp. 21-26, 1991.
[AFK87]	M. Abadi, J. Feigenbaum, and J. Kilian, "On Hiding Information from an Oracle," in Proceedings 19th Annual ACM Symposium on Theory of Computing (STOC), 1987, pp. 195-203.
[Ajax]	Ajax Project web page, http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/roc/public/www/Ajax.html.
[AK96]	R. Anderson and M. Kuhn, Tamper Resistance - A Cautionary Note, in Proceedings of the Second USENIX Workshop on Electronic Commerce. (1996) pp. 1-11, 1996.
[AK97]	R. Anderson and M. Kuhn. Low Cost Attacks on Tamper Resistant Devices, in Security Protocols, 5th International Workshop, Paris, France, LNCS 1361 pp. 125-136, 1997.
[Appel02]	A. Appel, "Deobfuscation is in NP." Preprint available from http://www.cs.princeton.edu/~appel/papers/deobfus.pdf
[Ble98]	D. Bleichenbacher. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS#1, in Advances in Cryptology -- CRYPTO '98, LNCS 1462, pp. 1-12, 1998.
[BMK+01]	L. Badger, B. Matt, D. Kilpatrick, and L. Spector, "Self-Protecting Mobile Agents Architecture and Policy Specification," NAI Labs Technical Report 01-006, March 23, 2001.
[BDM+01]	L. Badger, L. D'Anna, B. Matt, A. Reisse, and T. Van Vleck, "Self-Protecting Mobile Agents Obfuscation Report," NAI Labs Technical Report 01-036, November 30, 2001, Revised January 2003.
[Bal69]	R. Balzer, "EXDAMS -- EXtendable Debugging And Monitoring System," in AFIPS 1969 Spring Joint Computer Conference, Vol. 34, AFIPS Press, May 1969, Proceedings SJCC, 1969, pp. 567-580.
[BDL97]	D. Boneh, R. A. DeMillo, and R. J. Lipton "On the Importance of Checking Cryptographic Protocols for Faults," in Advances in Cryptology

	--- EUROCRYPT '97 Proceedings, Springer-Verlag, 1997, pp. 37-51.
[BGI+01]	B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (Im)possibility of Obfuscating Programs," in Advances in Cryptology, Proceedings of Crypto'2001, Lecture Notes in Computer Science, Vol. 2139, pp. 1-18.
[BMM00]	I. Biehl, B. Meyer, and V. Müller: Differential Fault Attacks on Elliptic Curve Cryptosystems, in Advances in Cryptology, Proceedings of Crypto'2000: pp. 131-146.
[BS97]	E. Biham and A. Shamir: Differential Fault Analysis of Secret Key Cryptosystems, in Advances in Cryptology, Proceedings of Crypto'1997: pp. 513-525.
[CCK+00]	C. Cachin, J. Camenisch, J. Kilian, and J. Müller. "One-round Secure Computation and Secure Autonomous Mobile Agents," in Proceedings 27th International Colloquium on Automata, Languages and Programming (ICALP), Volume 1853 of Lecture Notes in Computer Science, pp. 512-523. 2000.
[CEJ+02a]	S. Chow, P. Eisen, H. Johnson, and P.C. van Oorschot, White-Box Cryptography and an AES Implementation, in Proceedings of SAC 2002 - 9th Annual Workshop on Selected Areas in Cryptography, August 2002.
[CEJ+02b]	S. Chow, P. Eisen, H. Johnson, and P.C. van Oorschot, A White-Box DES Implementation for DRM Applications, in Proceedings of ACM CCS-9 Workshop DRM 2002 - 2nd ACM Workshop on Digital Rights Management, November 2002.
[CG95]	C. Cifuentes and K. J. Gough, "Decompilation of Binary Programs," Software - Practice & Experience. Volume 25 (7), July 1995, pp. 811-829.
[CJR+99]	S. Chari, C. Jutla, J. Rao, and P. Rohatgi, Towards Sound Approaches to Counteract Power-Analysis Attacks, in Advances in Cryptology, Proceedings of Crypto 1999, pp. 398-412.
[Cohen92]	F. Cohen, "Operating System Protection Through Program Evolution," Computers and Security 1992. http://all.net/books/IP/evolve.html
[Coll03]	C. Collberg, "SandMark Algorithms," V3.1.1, January 28, 2003 http://cgi.cs.arizona.edu/~sandmark/SandMark3.1.1/smalgs.pdf
[CP]	"Chopstick Program Analysis and Visualization Tool," http://www-2.cs.cmu.edu/afs/cs/project/chopshop/pub/www/home.html
[CS03]	"CodeSurfer Program Analysis Tool," http://www.grammatech.com/products/codesurfer/overview.html
[CT02]	C. Collberg and J. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection," in IEEE Transactions on Software Engineering 28:8, pp. 735-746, August 2002. See also University of Arizona Technical Report 2000-0, February 2000.

[CTL98a]	C. Collberg, J. Thomborson, and D. Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs," in IEEE International Conference on Computer Languages, May 1998.
[CTL98b]	C. Collberg, J. Thomborson, and D. Low, "Breaking Abstractions and Unstructuring Data Structures," in ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1998.
[CTL97a]	C. Collberg, J. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," University of Auckland Technical Report #170, July 1997.
[CW00]	"Preliminary Report on Optimizing Compilers and Code Transformations," Cloakware Corporation, June 2000. http://www.cloakware.com/resources/external.html
[CW01a]	"Preliminary Analysis of the Security of Data Flow Code Transformations," Cloakware Corporation, March 2001. http://www.cloakware.com/resources/external.html
[CW01b]	"Preliminary Analysis of the Security of Control Flow Code Transformations," Cloakware Corporation, April 2001. http://www.cloakware.com/resources/external.html
[CW01c]	"An Approach to the Obfuscation of Control Flow of Sequential Computer Programs," Cloakware Corporation, October 2001. http://www.cloakware.com/resources/external.html
[DCC]	C. Cifuentes, DCC C Decompiler Home Page, http://www.itee.uq.edu.au/~cristina/dcc.html.
[Deut94]	A. Deutsch, "Interprocedural May-Alias Analysis for Pointers: Beyond k-Limiting," in Proceedings SIGPLAN Conf. Programming Language Design and Implementation (PLDI '94), pp. 230-241, June 1994.
[Embra]	The Embra simulator web page, http://www-flash.stanford.edu/Embra/
[ER98]	M. Echin and J. Rochlis, "With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988," in Proceedings of 1989 IEEE Symposium on Research in Security and Privacy. May 1998.
[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
[FM99]	S. Funfrocken and F. Mattern. "Mobile Agents As An Architectural Concept For Internet-Based Distributed Applications - the WASP Project Approach". In Kommunikation in Verteilten Systemen (KiVS). Springer-

	Verlag, 1999.
[GMP+97]	L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2, in USENIX Symposium on Internet Technologies and Systems, Monterey, California, December 1997.
[GMW87]	O. Goldreich, S. Micali, and A. Wigderson. "How to Play any Mental Game," in 19th ACM Symposium on Theory of Computing, pp. 218-229, ACM Press, 1987.
[Gos85]	J. Gosler, Software Protection: Myth or Reality?, Advances in Cryptology, Proceedings of Crypto 1985, August 1985, pp. 140-157.
[H02]	S. Hada, "Zero-Knowledge and Code Obfuscation," Asiacrypt 2000 http://link.springer.de/link/service/series/0558/bibs/1976/19760443.htm
[HBC+99]	M. Hind, M. Burke, P. Carini, and J. Choi, "Inter-procedural Pointer Analysis," ACM Transactions on Programming Languages and Systems, Vol. 21, No. 4, July 1999, pp. 848-894.
[HMST01]	B. Horne, L. Matheson, C. Sheehan and R. Tarjan, "Dynamic Self-Checking Techniques for Improved Tamper Resistance," Proc. ACM Workshop on Security and Privacy in Digital Rights Management, Nov 2001. Also in LNCS 2320.
[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.
[HR98]	F. Hohl and K. Rothermel, "A Protocol Preventing Blackbox Tests of Mobile Agents," the 11th Fachtagung "Kommunikation in Verteilten Systemen" (KiVS'99). To appear.
[KH02]	S. Kumar and S. Horwitz, "Better Slicing of Programs with Jumps and Switches," in Proceedings of FASE 2002: Fundamental Approaches to Software Engineering, April 8-12, 2002.
[KJJ99]	P. Kocher, J. Jaffe, and B. Jun: Differential Power Analysis. Advances in Cryptology, Proceedings of Crypto 1999: pp. 388-397.
[Koc96]	P. Kocher: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems, in Advances in Cryptology, Proceedings of Crypto 1996: pp. 104-113.
[KOK+01]	T. Kamiya, F. Ohata, K. Kondou, S. Kusumoto, and K. Inoue: "Maintenance Support Tools for JAVA Programs: CCFinder and JAAT." ICSE 2001: pp. 837-838.
[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, pp. 93-103, Orlando, Florida, January 1991.

[Lou02]	C. Lou, private communication to Tom Van Vleck, E-mail December 31, 2002.
[LV97]	F. Lanubile and G. Visaggio, "Extracting Reusable Functions By Flow Graph-Based Program Slicing," in IEEE Transactions on Software Engineering, Vol. 23, No. 4, pp. 246-259, April 1997.
[MR91]	Silvio Micali and Phillip Rogaway: Secure Computation (Abstract). Advances in Cryptology - CRYPTO 1991: pp. 392-404, 1991.
[MS-ZM]	MacAfee -AVERT virus information library entry for W32/Zmist.gen http://vil.nai.com/vil/content/v_99382.htm
[NCJ01]	J. Nickerson, S. Chow and H. Johnson. "Tamper Resistant Software: Extending Trust into a Hostile Environment," ACM MultiMedia 2001, October 2001.
[NCJ+02]	J. Nickerson, S. Chow, H. Johnson, and Y. Gu. "The Encoder Solution to Implementing Tamper Resistant Software," Information Survivability Workshop ISW-2001/2002. March 2002.
[Nec97]	G. Necula. Proof-carrying code, in 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 106-119, New York, January 1997.
[NJMCT]	The New Jersey Machine-Code Toolkit web page http://www.eecs.harvard.edu/~nr/toolkit/
[OG98]	S. Onder and R. Gupta, "Automatic Generation of Microarchitecture Simulators," in the IEEE International Conference on Computer Languages (ICCL98), May 1998.
[Reps98]	Reps, T., "Program Analysis via Graph Reachability," Information and Software Technology 40, 11-12 (November/December 1998), pp. 701-726.
[PTO-AS]	Program-Transformation.Org Decompilation Application Specific Approach web page (including Java and .Net), http://www.program-transformation.org/twiki/bin/view /Transform/DecompilationApplicationSpecificApproach
[PTO-DA]	Program-Transformation.Org Companies Offering Disassemblers and tools for building disassemblers web page, http://www.program-transformation.org/twiki/bin/view /Transform/DecompilationDisassembly
[PTO-DC]	Program-Transformation.Org Companies Offering Decompilation Services web page, http://www.program-transformation.org/twiki/bin /view/Transform/CompaniesOfferingDecompilationServices
[PTO-DR]	Program-Transformation.Org Decompilation And Reverse Engineering web page, http://www.program-transformation.org/twiki/bin /view/Transform/DecompilationAndReverseEngineering.
[PW97]	T. Proebsting and S. Watterson. "Krakatoa: Decompilation in Java (Does Bytecode Reveal Source?)," in Proceedings of the Conference on Object-

	Oriented Technologies and Systems, Portland, Oregon, USA, June 1997, USENIX.
[RF97]	N. Ramsey and M. F. Fernandez. Specifying Representations of Machine Instructions. ACM Trans. Programming Languages and Systems, 19(3): pp. 492-524, May 1997.
[RT96]	T. Reps and T. Turnidge, "Program Specialization via Program Slicing," in Proceedings of the Dagstuhl Seminar on Partial Evaluation, (Schloss Dagstuhl, Wadern, Germany, February 12-16, 1996), LNCS, Vol. 1110, 1996, pp. 409-429.
[Rutgers]	"The Prolangs Analysis Framework (PAF)." Rutgers University. http://www.prolangs.rutgers.edu/public
[SA02]	S. Skorobogatov and R. Anderson, Optical Fault Induction Attacks, Workshop on Cryptographic Hardware and Embedded Systems 2002 (CHES 2002), August 13-15, 2002.
[Sch005]	Bruce Schneier, "CRYPTO-GRAM, May 15 2000" www.counterpane.com/crypto-gram-0005.html
[Sch025]	Bruce Schneier, "CRYPTO-GRAM, May 15 2002" www.counterpane.com/crypto-gram-0205.html
[Schwab03]	Steve Schwab, "The Embedded Hard Problem Fallacy," to appear.
[Schwarz]	B. Schwarz, S. Debray, G. Andrews; "Disassembly of Executable Code Revisited"; Proceedings of 2002 Working Conference on Reverse Engineering, pp 45-54; Oct 2002.
[SimOS]	The SimOS Simulator web page http://simos.stanford.edu/
[SM03]	SandMark Project home page. http://www.cs.arizona.edu/sandmark
[SML-C]	Software Migrations Ltd IBM 370 Decompilation to C Code http://www.smltd.com/Products.htm
[ST98a]	T. Sander and C. Tschudin. "On Software Protection via Function Hiding," in Proceedings of the Second Workshop on Information Hiding, Portland, Oregon, USA, 12 April 1998.
[ST98b]	T. Sander and C. Tschudin. Towards Mobile Cryptography, in Proceedings of the 1998 IEEE Symposium on Security and Privacy, Oakland, California, May 1998.
[ST98c]	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," in Proceedings 40th IEEE Symposium on Foundations of Computer Science (FOCS), 1999.

[Tempest]	The Complete, Unofficial TEMPEST Information Page, http://www.eskimo.com/~joelm/tempest.html
[Tip95]	F. Tip, "A Survey of Program Slicing Techniques," Journal of Programming Languages, 3(3): pp. 121-189, September 1995.
[TM81]	W. Teitelman and L. Masinter. The Interlisp Programming Environment. IEEE Computer, pp. 25-33, April 1981.
[TR81]	T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: A Syntax-directed Programming Environment. Communications of the ACM, 24(9): pp. 563-573, September 1981.
[UC00]	D. Ung and C. Cifuentes, Machine-Adaptable Dynamic Binary Translation, in Proceedings of the Workshop on Dynamic and Adaptive Compilation and Optimization, pp. 37-47, 2000.
[Vau02]	S. Vaudenay: Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ...., Advances in Cryptology EUROCRYPT'02, LNCS No. 2332, pp. 534-545, 2002.
[WLA+93]	R. Wahbe, S. Lucco, T. Anderson, and S. Graham, "Efficient Software-based Fault Isolation," in Proceedings ACM Symp. on Operating System Principles, December 1993, pp. 203-216.
[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. Http://www.cs.virginia.edu/ survive/pub/wangthesis.pdf.
[WDH+01]	C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of Software-based Survivability Mechanisms," in International Conference of Dependable Systems and Networks, Goteborg, Sweden (July, 2001).
[WHK+00]	C. Wang, J. Hill, J. Knight, and J. Davidson, "Software Tamper Resistance: Obstructing Static Analysis of Programs." Technical Report CS-2000-12, Department of Computer Science, University of Virginia, 2000.
[WPS]	The Wisconsin Program-Slicing Project web page, http://www.cs. wisc.edu/wpis/html/
[WR96]	E. Witchel and M. Rosenblum, Embra: Fast and Flexible Machine Simulation, in Proceedings of ACM SIGMETRICS '96: Conference on Measurement and Modeling of Computer Systems, 1996.
[WPS]	The Wisconsin Program-Slicing Project web page, http://www.cs. wisc.edu/wpis/html/

Appendix A: JBET

The Java Binary Enhancement Tool (JBET) is a general Java program analysis and manipulation tool. JBET uses a convenient internal representation of all the contents of Java binary (.class) files, for easy manipulation. For example, the code in each method is stored as a list of Instruction objects, which know about the operands of the Java instructions, how to emulate the instruction, etc. A directed acylic graph (DAG) representation of code is also available. Various manipulations are very easy to code, for example, a plugin that automatically turns field accesses into getter/setter method pairs is 152 lines of source code.

We tested the correctness of JBET by reading in and writing out thousands of Java binaries and ensuring that the generated files are identical, and by using JBET to transform itself and verifying that the program still ran. Our approach includes representation of the entire Java instruction set. JBET can insert new methods (or functions) into Java binary (.class) executable files, and also cause original program code to invoke our new functions; this ability allows us to augment the functionality of mobile agents (and Java programs in general) without requiring access to source code. This binary capability will significantly ease the technology transfer of our technology since the technology can be applied to fielded Java programs without having to first obtain their source code.

Design

Basic data structures

The JBET core uses several classes to represent the contents of a Java class file in an easily manipulatable form. Many transformations on Java programs are easy to code using these data structures.

ClassInfo

The `ClassInfo` data structure represents a single Java class (essentially, the contents of one .class file). It stores data such as name, access flags, a list of `MethodInfo`s for the methods, and a list of `FieldInfo`s for the fields.

MethodInfo

`MethodInfo` represents a single Java method. It stores the name, descriptor, containing `ClassInfo`, the exception specification for the method, and a list of Instructions representing the code for the method (if any). An equivalent to the Java class verifier is included, so that assembled code can be checked for errors that would cause the JVM to reject the class to be detected during processing. In addition, our verifier outputs the exact instruction causing the problem.

Instruction

Instruction objects represent one Java instruction, including opcode and operands. The Instruction class has a database of Java instructions, allowing it to be used in a "Java assembler" to ensure that all instructions have correct operands. The Instruction class also handles jump addresses internally, so programmers of JBET modules do not need to be concerned with aligning bytes and adjusting jump targets.

Graph-based representation

For nontrivial code transformations, a graph-based representation is desirable compared to a list of raw instructions. (We developed the graph-based representation while writing the obfuscator). In this representation, the code for a method is stored as a list of basic blocks, each containing graph nodes for Java operations. This frees the programmer from having to manage the Java operand stack while manipulating code. Because of the strict requirements on code layout enforced by the JVM, it is always possible to split a method into basic blocks. The programmer may use the graph-based and normal representations interchangeably, converting to one or the other depending on which is more desirable.

The graph representation stores Java instructions in a high-level form (almost like parse trees of source code). For example, an `invokevirtual` node contains a list of nodes for the arguments, whereas in the instruction list, the arguments would have merely been pushed on the stack by earlier instructions.

Plug-ins

JBET is designed to allow separate code (or graph) modification transformations to be plugged in, and to make it reasonably easy to continue development of advanced transformation techniques into the future. (If the on-disk layout of class files is changed, we can include support for that in an updated version, without affecting "most" plug-ins.)

The obfuscator and deobfuscator are two notable examples of plug-ins.

Use in Survivable Server Project

One use proposed for JBET technology is to rewrite parts of the Java standard library to include security checks that the Java security manager interface does not support (for example, access to files checked at every read). This use of JBET was part of the DARPA funded Survivable Server project, which applied multiple technologies to hardening a component of the Joint Battle Infosphere. This project, led by Teknowledge Corporation, began in July 2002 and ended in March 2003.

We created a plug-in for JBET that operated on a Java program to intercept calls to native methods invoke policy enforcing code before each such method call.

Packaging

We developed a portable build environment for JBET. (Only tested on Linux and Windows NT/XP, but should work on other Unix platforms with Java) Our. environment

requires only a few free and easily downloaded tools to build and run our software on any of these platforms.

To support Survivable Server and other projects, we organized JBET into separable packages, and removed all traces of obfuscation from the main core, which supports general analysis of Java class files. The obfuscation and deobfuscation operations are separately packaged and need not be shipped with the main JBET core.

In discussions with the DARPA Program Manager, it became evident that our obfuscation software could be used by attackers as well as by defenders. To prevent this, the Program Manager requested that we not distribute JBET as Open Source software or make it available except to the Government and NAI's trusted business partners. While this will reduce visible transfer of SPMA technology, we believe it is a prudent decision and will not materially affect our ability to transfer the technology to national defense applications. With the reorganization of the software, we can distribute the JBET core while holding back the obfuscation tool, forestalling embarrassing malicious use of the obfuscation tool.