University of Arizona, Department of Computer Science

CSc 620 — Security Through Obscurity — Final Project

Christian Collberg

March 1, 2002

1 Introduction

In this handout I will list some possible project ideas. Keep in mind that they are only suggestions — I'd be really happy if you came up with your own ideas.

This project is worth 50% of your final grade.

Here is the schedule for the rest of the semester:

Fri, March 1: List of projects (this document) handed out.

Tue, March 5: Come see me in my office to discuss your project selection. Bring with you:

- The idea that you want to work on!
- A *mission statement document*. This is an outline of exactly what you expect to accomplish. It should describe the purpose of the project and the "deliverables" that you intend to produce (code and/or documentation and/or a research paper, etc.).
- A schedule for the rest of the semester, describing project milestones.
- A list of problems that you expect that you will need to solve in order to run a successful project.

On the last page is an example of the type of document that I would like to see. The more detailed you make this document, the better it is.

Wed, March 6: Come see me in my office to hand in a *software design document*. This is similar to the mission statement, but focuses more on the design of the software artifact itself: class hierarchy, package hierarchy, classes, algorithms, UML diagrams[1], team organization; whatever you feel is necessary to describe your design and how you will go about implementing it. Having to write this document will force you and your teammate to sit down and discuss exactly who will do what, and how it will be done.

Tue, March 19: Every team makes an initial presentation to the class of the project they have chosen. Mission statements, design documents, etc. are distributed.

Thu, March 21: Every team comes to talk to me to describe the state of their project. At this time, you should also submit a first draft of the documentation. Essentially, this is a combination of the mission statement (what we intend to do), the design document (how we're doing it), rewritten in "documentation style." The documentation should be about four-five pages at this stage.

Tue, April 2: Every team comes to talk to me to describe the state of their project. A second draft of the documentation is handed in.

[1] No, not really.

Tue, April 16: Every team comes to talk to me to show me their partially working code.

Tue, April 23: Every team comes to talk to me to show me their fully working code. The last week before the end of classes should be used to fix minor bugs and finish up the documentation.

Tue, April 30: Final project presentations. On this date all deliverables (LaTeX documentation, source code, etc.) must be handed in. I'm expecting about ten pages of documentation in total. And 10,000 lines of code.[2]

2 Project Descriptions

Below I have outlined some possible projects. Keep in mind that implementation is only a small part of any research project — the reason that you are implementing an algorithm is to allow you to examine that implementation and learn something new. Therefore, an important part of your project will be measurement and evaluation. For example, if you implement a watermarking algorithm you will want to study:

1. How resilient is it against various transformations? Does it survive the the obfuscations in SandMark? Does it survive being optimized by BLOAT? If the mark *is* destroyed, how much slower/larger is the de-watermarked program?
2. What happens if I watermark the same program twice with the same algorithm?
3. How stealthy is the watermark? What is the right metric for measuring this?
4. What is the overhead of the algorithm? How much larger/slower is the watermarked program compared to the original one?

If you are in the PhD program you should consider picking a project that — should it be successful — could result in a publication. I'd be more than happy to supervise a follow-on independent study/summer job where we polish up the project and write it up for submission to a conference.

Project I: Echo Hiding

Invent your own software watermarking method based on the the "echo hiding" idea used in audio watermarking.

Project II: Venkatesan's Algorithm

Implement and study Venkatesan's software watermarking algorithm. How well does it work for Java bytecode? What's the overhead? How can it be attacked?

Ideally, you should implement both the algorithm and an attack, but this may be too much to ask. The algorithm is very poorly described, so you will probably have to "invent" important parts of it yourself.

Reference: Venkatesan, *A Graph Theoretic Approach to Software Watermarking.*

[2] Yes, really.

Project III: Athalla's Algorithm

Implement and study Athalla's tamper-proofing algorithm. How well does it work for Java bytecode? What's the overhead? How stealthy can it be made?

Reference: Chang, *Protecting Software Code by Guards.*

Project IV: Stern's Algorithm

Implement and study Stern's software watermarking algorithm. How well does it work for Java bytecode? What's the overhead? How stealthy can it be made?

Ideally, you should implement both the algorithm and an attack.

Reference: Stern, *Robust Object Watermarking: Application to Code.*

Project V: Potkonjak's Algorithm

Implement and study Potkonjak's software watermarking algorithm which embeds a watermark in the solution to a register allocation graph coloring problem. How well does this idea work for Java bytecode?

Reference: Qu, *Fingerprinting IPs Using Constraint-Addition: Approach and Graph Coloring Case Study.*

Project VI: Splitting boolean variables

Implement the "splitting booleans" code obfuscation. This is slightly tricky, since Java bytecode implements booleans as integers. Hence, in a prepass you need to identify integer variables that are actually used as booleans.

Reference: Collberg, *Breaking Abstractions and Unstructuring Data Structures.*

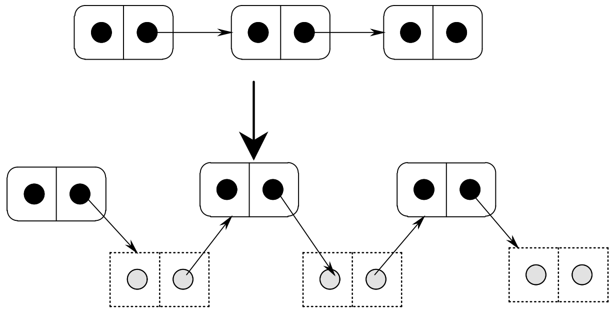
Project VII: Array obfuscations

Implement every possible obfuscation on arrays that you can think of. These could include merging, splitting, flattening, folding, etc.

Reference: Collberg, *Breaking Abstractions and Unstructuring Data Structures.*

Project VIII: Class splitting obfuscations

There are two important object splitting techniques. The first splits each object into two, such that some fields belong to the first subpart, the remainder to the second subpart:



In the bytecode every `new`-operation turns into two `news` and every pointer operation (`getfield` and `setfield`) turns into two pointer operations.

The second splitting technique splits at the class level. A class C is broken into two classes C_1 and C_2, such that C_2 inherits from C_1. C_1 has fields and methods that only refer to themselves, whereas C_2 has fields and methods that can refer to themselves as well as fields and methods in C_1. Bytecode references to C will have to be replaced with references to C_2.

Implement these class splitting operations and any others you can think of.

There is some code in `sandmark.obfuscate.splitclass` but it never worked and probably won't be of much help to you.

Reference: Collberg, *Breaking Abstractions and Unstructuring Data Structures.*

Project IX: Cloakware

Implement the cloakware obfuscation, as described in their papers at `http://www.cloakware.com/resources/conference.html` and their patent #5,748,741, *Encoding Technique for Software and Hardware.*

Project X: Automatic degrading

Add an "obfuscation" to SandMark that implements automatic degrading of demo-programs. Companies that sell images off the web will often allow you to download a degraded (low-resolution) version for inspection. If you decide you like it you can then purchase the original image. The idea here is to do the same for software: Write a program that takes another program P as input, and produces P'. P' has the same functionality as P, but is much slower and/or uses more memory.

- This is quite a bit like obfuscation, but we *want* the program to be slower.
- Stealthiness is important. Just inserting null loops is no good; they can easily be found and removed. You'll want the program to be *uniformly* slower.
- Ideally, you don't want the program to be much larger than the original since that will increase download times.
- You will want to give the user a high degree of control over how much his program is downgraded.

Project XI: Tamperproofing by aliasing

Design a SandMark library function that will allow a program to die gracefully and imperceptibly when it detects that it has been tampered with. You will want the point of detection (where the program has figured out that it has been tampered with) to be far away in space and time from the point of failure (where the program dies.) One idea is to base your algorithm on the hardness of alias analysis: at the point of detection you change some pointer variable in such a way that millions of instructions later, an null-pointer exception is thrown.

Project XII: Generating integer opaque predicates

Design a SandMark library function that generates opaque predicates for use by other obfuscators. One idea is to write a Prolog program that encodes a set of simple mathematical truths and generates all possible true combinations of these as output. To make this practical you should limit the size of the predicate, to contain, say, no more than two variables and two small literal constants. The resulting statements should be coded into bytecode and stored in your library for future use.

Is your system powerful enough to generate $7y^2 - x^2 \neq 1$ and "the second bit of a squared number is always 0"?

Project XIII: Generating opaque predicates by aliasing

Implement a version of Collberg and Thomborson's technique for generating opaque predicates by aliasing. The real challenge here is to find appropriate places in the code to insert graph-manipulation code.

Reference: Collberg, *Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs.*

Project XIV: Encoding static data

Devise and implement a set of algorithms for obfuscating static data. For example, strings can be encoded as finite state machines, in tries, etc. Make sure that your algorithms are sufficiently randomized. Ideally, it should not be possible to discover how the strings are encoded by simple pattern-matching.

Reference: Collberg, *Breaking Abstractions and Unstructuring Data Structures.*

Project XV: Add a level of interpretation

Implement an algorithm that takes a Java bytecode method and turns it into a program in *another* bytecode language, and then replaces the original method with an interpreter for this language.

Reference: Collberg, *Breaking Abstractions and Unstructuring Data Structures.*

Example Mission Statement Document

Advanced Name Obfuscation

Bob and Alice

For our final project we would like to implement an advanced name obfuscator for Java. The basic idea behind this tool is ...
Our work will be based on the work described in a seminal paper by Charles, *Advanced Name Obfuscation by Zero-Knowledge Proofs*, published in the *Journal of Irreproducible Results.*
The reason we want to implement this algorithm is that, to the best of our knowledge, it has never been implemented before. We don't believe it is a very strong algorithm, and would like to prove this by implementing and evaluating it.
The implementation will be done within the SandMark framework. This will allow us to take advantage of the many excellent, well-documented libraries that are available within this tool.
We believe this will be a difficult project with many potential pitfalls. First of all, the paper is full of holes and so we need to work out a reasonable algorithm. This is our best shot so far:

```
for all classes c do
   ...
endfor
```

We expect to have to modify this algorithm as we understand the problem better. Secondly, it's not clear that we will actually be able to produce a complete implementation before the end of the semester. We therefore identify the following subtasks:

1. ...
2. ...
3. ...
4. Testing and evaluation.
5. Documentation.
6. Writing up the results in conference paper.

Item number 3 appears to be the most difficult one and we may only be able to manage a partial implementation. To evaluate the algorithm, we intend to ...
We propose the following schedule:

Tue, March 5: Mission statement handed in.

Wed, March 6: Software design document handed in.

Thu, March 7: Starting to code!

Tue, March 26: Initial project presentation to the rest of the class.

Tue, April 2: First milestone: Subtask number 1 completed.

Thu, March 26: Second milestone: Subtask number 2 completed. Meet with Collberg, first version of the documentation handed in.

Tue, April 23: Third milestone: Subtask number 3 completed. Testing, final debugging, and documentation commences!

Tue, April 30: Final project presentation. Documentation (LaTeX!) handed in. All sources committed to the CVS repository. Start cramming for 620 final.[a]

[a]No, not really.