ITERATED TRANSFORMATIONS AND QUANTITATIVE METRICS FOR SOFTWARE PROTECTION

Mariusz H. Jakubowski, Chit W. (Nick) Saw and Ramarathnam Venkatesan
Microsoft Research, Redmond, WA, U.S.A.

Keywords: Software protection, Tamper-resistance, Obfuscation, Security metrics.

Abstract: This paper describes a new framework for design, implementation and evaluation of software-protection schemes. Our approach is based on the paradigm of *iterated protection*, which repeats and combines simple transformations to build up complexity and security. Based on ideas from the field of complex systems, iterated protection is intended as an element of a comprehensive obfuscation and tamper-resistance system, but not as a full-fledged, standalone solution. Our techniques can (and should) be combined with previously proposed approaches, strengthening overall protection.

A long-term goal of this work is to create protection methods amenable to analysis or estimation of security in practice. As a step towards this, we present security evaluation via *metrics* computed over transformed code. Indicating the difficulty of real-life reverse engineering and tampering, such metrics offer one approach to move away from ad hoc, poorly analyzable approaches to protection.

1 INTRODUCTION

Open systems, such as PCs and mobile devices, have long suffered from malicious tampering and reverse engineering by hackers. To facilitate more secure code execution, researchers have devised and implemented many approaches that complicate observation and modification of software. These include obfuscation (Collberg et al., 1997; Collberg et al., 1998b), anti-tampering measures (Aucsmith, 1996; Horne et al., 2001; Jacob et al., 2007), data hiding (Collberg et al., 1998a; El-khalil and Keromytis, 2004), and other protective transformations (Wang et al., 2000; Anckaert et al., 2004; Tan et al., 2006; Anckaert et al., 2007a). Such techniques have served well in various contexts, but few have offered a practical security analysis to estimate how long a protected program will remain uncracked in practice. While solutions exist under certain engineering assumptions (Dedić et al., 2007), a current open problem is to develop practical protection techniques that support a realistic security evaluation.

In this paper, we propose a new general approach to devise transformations that protect code while enabling practical assessment of security via quantitative metrics (Anckaert et al., 2007b). The central idea is to combine and iterate simple transformations, such as injection of opaque predicates (Collberg et al., 1998b), oblivious hashing (Chen et al., 2002; Jacob et al., 2007), and control-flow transformations (Collberg et al., 1997). These transformations may be far simpler than traditionally applied techniques, and need not create much obfuscation or tamper-resistance on their own; the main idea is to build up complexity by repeated application and recombination of simple operations, creating a cascading effect. Particularly simple yet useful transformations include injection of inert "chaff" code, as well as conversion of variable references to be performed via newly created pointers. Indeed, rather than relying on well known transformations, iterated protection may facilitate controllable security and simpler tool implementation by repeatedly applying straightforward primitives.

Our approach was inspired by the field of *complex systems*, which studies how simple transformation rules affect the state of abstract systems over time. For example, *cellular automata (CA)* such as the Game of Life (Wolfram, 2002) are represented as arrays or grids of cells, each in a particular state (e.g., discrete binary 0 or 1), and updated in discrete time steps. A function called an *update rule* is applied to each neighborhood (e.g., a 3x3 square) in the grid to yield the state of the center cell after the next time step. Some surprisingly simple CA can perform universal computation via emulation of Turing ma-

M. Jakubowski M., W. (Nick) Saw C. and Venkatesan R. (2009).
ITERATED TRANSFORMATIONS AND QUANTITATIVE METRICS FOR SOFTWARE PROTECTION.
In *Proceedings of the International Conference on Security and Cryptography*, pages 359-368
DOI: 10.5220/0002220103590368

chines. In essence, iteration of very simple functions over binary arrays can lead to arbitrary (or *emergent*) behavior of the system. By analogy, iteration of simple transformations can lead to similar complexity in software code.

In general, complex systems cannot be "short-cut" to predict states at arbitrary future times; the system must actually be run to determine what happens. Thus, we may not be able to model or predict the effects of iterating simple transformations over software. In terms of security analysis, we do not typically attempt to predict the outcomes of iterated transformations; instead, we evaluate security via *metrics* computed over the final transformed code. In particular, we use metrics from a study on quantitative evaluation of obfuscation (Anckaert et al., 2007b), along with additional metrics devised for analysis of iterated and recombined transforms. Via heuristics, experiments and analysis, such estimates of complexity can be associated with actual security. A spectrum of various metrics offers a means of evaluating complexity and security in terms of real-life tampering and reverse engineering.

The rest of this paper is structured as follows. In Section 2, we provide more explanation and background on the iteration approach. A list of some practical protective transformations is found in Section 3. Section 4 describes a metric-based approach towards evaluating the security of iterated and recombined transformations. A tool implementation and experimental results are the topic of Section 5. We provide a final assessment and future directions in Section 6.

2 ITERATED PROTECTION

We propose *iterated protection* as a general framework for design, analysis, and implementation of software-protection techniques. This methodology involves the iterated application and recombination of various obfuscating transformations (or *primitives*) over code, with the output of each successive transformation serving as input to the next. Via this strategy, even simple and easy to implement primitives can be cascaded to yield effective obfuscation.

As an example, the technique of oblivious hashing (OH) (Chen et al., 2002; Jacob et al., 2007) can serve as a tamper-resistance primitive. A single OH transformation injects code to hash a program's runtime state (i.e., data and control flow), thus ascertaining execution integrity of the original code. Applying OH again to the transformed program protects both the original program and the first OH round. In general, each new OH round verifies the integrity of both the original program and all previous OH rounds.

To increase security further, arbitrary other primitives can be combined and iterated with the OH rounds. For ease of design and implementation, such primitives can be quite simple; e.g., conversion of variable references to pointer references, and even source-to-source translation among different code dialects or languages. Via iteration, the interaction of simple primitives can achieve the effect of far more complex obfuscation operators.

An important general principle of the iteration approach is usage of transformations that appear to be "weak" or not particularly obfuscating. It is not necessary to eliminate all weaknesses from each transformation operator; instead, we rely on the iterated, combined effect of multiple operators to augment one another's security, essentially "filling in" both known and unknown holes. To verify this, overall security can be measured by quantitative metrics, as we discuss in detail later.

2.1 Related Areas

2.1.1 Cryptography

Iterated protection is related to the concept of *rounds* in cryptographic schemes such as hash functions and block ciphers (Menezes et al., 1996). Often chosen heuristically to resist known and unknown attacks, the number of rounds determines security and efficiency. Each individual round may be easily breakable, and a small number of iterated rounds can usually be attacked successfully. However, once the number of rounds becomes large enough, an algorithm may survive in practical use for many years, despite improved cryptanalysis and more powerful computing systems.

In this spirit, any single obfuscation method can be treated as a round of an obfuscation algorithm. The individual techniques may be very simple and not particularly secure when used alone, but allow us to bootstrap to a desired security level when applied iteratively. Much like in cryptography, iteration of obfuscation primitives can achieve the "confusion" and "diffusion" effects necessary for security evaluation via quantitative metrics or heuristic arguments.

The analogy between round-based cryptography and iterated obfuscation is not perfect, mainly because constructions like hash functions and block ciphers are highly specialized. In contrast, obfuscation should be able to operate on universal programs, making analysis and even heuristic arguments difficult or impossible. However, this also leads to more extensive possibilities for obfuscation, especially for spe-

cific purposes.

2.1.2 Complex Systems

Another related area is the field of *complex systems* (Wolfram, 2002), which studies aggregate behavior of systems of states controlled by iterative *evolution rules*. Such rules are essentially functions whose inputs are sets of states at time t and whose outputs are individual states at time $t+1$. A main theme is the frequent emergence of complex, essentially unpredictable behaviors over time in large systems of simple agents governed by simple rules. Such *emergent behavior* occurs in a variety of natural and abstract scenarios, such as weather, vehicle traffic, economic markets, cellular automata and software systems.

A program to be protected can be considered as a system of states (e.g., program statements, variables, and objects), with protection primitives serving as evolution rules. Emergent program structure and behavior can arise as a result of applying simple primitives iteratively. In such a program, we may observe characteristics that are not easily explained in terms of either the original program or the simple nature of each individual obfuscation round. For example, if an obfuscation primitive performs code outlining (i.e., transforming code sections into separate functions), call graphs of arbitrary shapes and properties can arise via iteration.

2.2 Security Modeling

Recent theoretical work (Barak et al., 2001; Goldwasser and Kalai, 2005) has put obfuscation on a formal foundation, and certain schemes (Lynn et al., 2004; Wee, 2005) have been shown secure in this framework. Earlier results on oblivious RAMs (Goldreich and Ostrovsky, 1996) involved a somewhat different obfuscation model based on randomizing memory-access patterns. However, such theoretical work has not yielded practical obfuscation schemes for typical real-life programs, where an important goal is often simply to provide a lower bound on breaking complexity (e.g., preventing hacks for a new game from appearing for at least a couple of weeks or months). In practice, “ad hoc” approaches, such as code encryption and integrity checks, are currently most popular.

For efficiency of implementation, commonly used cryptography is often not proved strictly secure in a formal model. Perceived security is based on design heuristics and long-term cryptanalysis by the research community, not on security proofs. Certain algorithms with provable security are known, but tend to be impractical and seldom used. Even such algorithms may fail when attacks violate their models or complexity assumptions turn out to be unfounded.

As with popular block ciphers and hash functions, a formal security proof may be neither known nor necessary for our iterative methods to be useful in practice. In addition to security metrics and heuristic arguments, practical security could be determined by releasing a system to be attacked by security researchers, both academic and commercial. In some sense, this could allow us to put obfuscation on a cryptographic foundation, at least in terms of establishing a new area of cryptanalysis devoted to obfuscation.

Formal security analysis of iterated protection may still be possible, at least for specific instances of transformations. However, given the current reality of software protection, a heuristic approach like that in practical cryptography may yield more immediate and useful results. In addition, we suggest that real-life security may be reasonably assessed through quantitative metrics (Anckaert et al., 2007b) computed over transformed code.

3 PROTECTIVE TRANSFORMATIONS

This section presents a number of transformations (or *protection operators*) suitable for iteration and recombination. Practical application of iterated protection involves mainly selecting sequences of operators and their parameters, including the number of iterations to be performed by each operator instance. Some of the operators described here are simple transformations derived from earlier work, while others are geared specifically towards the iterated-protection framework (and thus need not provide much protection or obfuscation when used standalone instead of iteratively). We present these operators in the context of our tool implementation, as described in Section 5.

The sections below group operators into several categories, based on the main intended purpose of the operators. Some functionality overlap exists among various operators, but this grouping helps to classify and organize different transformations.

3.1 Tamper-resistance Operators

These operators serve mainly to inject code that verifies runtime integrity of execution.

3.1.1 Oblivious Hashing

This operator is for injection of oblivious-hashing (OH) code (Chen et al., 2002; Jacob et al., 2007), including hash initialization, actual hashing, and hash verification. OH helps to provide tamper-resistance by verifying the integrity of both computations and control flow. The basic idea is to maintain special *hash variables* during runtime, updating these hashes upon every state change (e.g., after variable assignments and control-flow transfers). At chosen points in the program, hashes may be verified explicitly (e.g., by comparing against precomputed values) or implicitly (e.g., by using a hash to decrypt crucial data or code).

The implementation supports two main approaches to OH: (1) *Hash pre-computation*, which computes and stores "correct" hashes for a set of user-provided (or automatically generated) inputs that exercise all relevant code paths; and (2) the *code-replica* method, which creates individualized copies of basic blocks (or other code sections) and compares the hashes produced by independent execution of these redundant code sections.

```
int x = 123;

if (GetUserInput() > 10)
{
  x = x + 1;
}
else
{
  printf("Hello\n");
}
```

Figure 1: Sample code before application of OH.

As an example, Figure 1 shows sample C++ code before application of OH, and Figure 2 lists the same code after injection of one OH round. Figure 3 shows the code resulting from injection of two OH rounds. Note that with two OH rounds, hash variables of the second round are used to verify hash variables of the first round; i.e., the second OH round verifies both the original program variables and the first OH round.

3.1.2 State-change Verification

This tamper-checking operator injects code to verify the operation of individual instructions and small sets of instructions. For example, given an instruction that increments a variable by *N*, the difference between the new and old variable versions should be *N*; the operator injects code to check this explicitly at runtime. The intent is to introduce tamper-resistance at a low level without requiring specific inputs or code replicas, as with OH.

```
INITIALIZE_HASH(hash1);

int x = 123;
UPDATE_HASH(hash1, x);

if (GetUserInput() > 10)
{
  UPDATE_HASH(hash1, BRANCH_ID_1);
  x = x + 1;
  UPDATE_HASH(hash1, x);
}
else
{
  UPDATE_HASH(hash1, BRANCH_ID_2);
  printf("Hello\n");
}

VERIFY_HASH(hash1);
```

Figure 2: Sample code after one round of OH.

3.2 Control-flow Operators

These transformations serve mainly to increase the complexity of control flow in target code.

3.2.1 Opaque Predicates

This is a traditional operator that injects code to compute predicates and corresponding branches that are either never or always taken (Collberg et al., 1998b). Alternately, both branch paths may be possible with varying probabilities. While the tool user knows this in advance, this property is difficult for other parties to deduce from the code. The effect is to add extra edges to the control-flow graphs (CFGs) of functions while only modestly impacting performance and code size.

3.2.2 Branch Transformations

This operator performs branch flattening, which moves one or more branch operations into a single dispatch block that performs the actual tests and jumps. The operator also implements branch diffusion, which arranges for a single branch to be scattered and merged with other branches. These operations are a form of control-flow flattening or obfuscation (Wang, 2000).

```
INITIALIZE_HASH(hash1);
INITIALIZE_HASH(hash2);

int x = 123;
UPDATE_HASH(hash1, x);
UPDATE_HASH(hash2, x);
UPDATE_HASH(hash2, hash1);

if (GetUserInput() > 10)
{
  UPDATE_HASH(hash1, BRANCH_ID_1);
  UPDATE_HASH(hash2, BRANCH_ID_1);
  UPDATE_HASH(hash2, hash1);
  x = x + 1;
  UPDATE_HASH(hash1, x);
  UPDATE_HASH(hash2, x);
  UPDATE_HASH(hash2, hash1);
}
else
{
  UPDATE_HASH(hash1, BRANCH_ID_2);
  UPDATE_HASH(hash2, BRANCH_ID_2);
  UPDATE_HASH(hash2, hash1);
  printf("Hello\n");
}

VERIFY_HASH(hash1);
VERIFY_HASH(hash2);
```

Figure 3: Sample code after two rounds of OH.

3.3 Generic Obfuscation Operators

These operators are geared towards various obfuscating transformations that increase the difficulty of understanding code.

3.3.1 Pointer Conversion

This transformation converts variable references to pointer references (by creating a new pointer for each variable and modifying variable references to use this pointer). This conversion can also transform function calls to be performed via function pointers. This is mainly a means of obfuscation via pointer indirection, and is effective especially when iterated and combined with operators that inject new variables.

```
int x = GetTickCount();
printf("%d\n", x);
```

Figure 4: Sample code before application of pointer conversion.

To illustrate, Figure 4 shows original sample C++ code, and Figure 5 lists the same code after one iteration of pointer conversion. Figure 6 shows the effects of two iterations. The latter two figures list the actual code output by the source-to-source transformation tool described in Section 5.

```
int * ptr_x_0;
int x;
ptr_x_0 = &x;
unsigned int tmp_151 =
  (* (unsigned int (__stdcall *)())
  &GetTickCount)();
int tmp_152 = (int) tmp_151;
*(int *) ptr_x_0 = tmp_152;
char * tmp_ptr_154 = (char *) "%d\n";
printf(tmp_ptr_154, * (int *) ptr_x_0);
```

Figure 5: Sample code after one iteration of pointer conversion (tool output).

```
int * ptr_x_2;
int ** ptr_ptr_x_0_1;
int * ptr_x_0;
int x;
ptr_ptr_x_0_1 = &ptr_x_0;
ptr_x_2 = &x;
*(int **) ptr_ptr_x_0_1 = ptr_x_2;
unsigned int tmp_151 =
  (* (unsigned int (__stdcall *)())
  &GetTickCount)();
int tmp_152 = (int) tmp_151;
int * tmp_ptr_159 = * (int **)
   ptr_ptr_x_0_1;
* (int *) tmp_ptr_159 = tmp_152;
char * tmp_ptr_154 = (char *) "%d\n";
int * tmp_ptr_160 = * (int **)
   ptr_ptr_x_0_1;
printf(tmp_ptr_154, * (int *)
   tmp_ptr_160);
```

Figure 6: Sample code after two iterations of pointer conversion (tool output).

3.3.2 Dataflow Stopping

This dataflow-obfuscation operator creates a copy of a variable at a random (or specified) point in a target function, overwriting the original variable and replacing all later references with the copy. This helps to hinder dataflow analysis.

3.4 Individualization Operators

These operators help primarily to diversify code, creating individualized copies that prevent easy reuse or retargeting of one particular break. This also allevi-

ates the software "monoculture" problem, where malicious programs work more or less equally well on most systems that run the same installed code.

3.4.1 Code Replication

This is a code-duplication operator that implements various methods to create redundant, individualized copies of code sections. This is useful for the code-replica approach of OH, as well as for other obfuscation and tamper-resistance operations.

3.4.2 Random Code Generation

This operator injects random expressions generated from a simple grammar. After recursive generation of a random parse tree, the operator generates code from the tree in a compiler-like manner. The main purpose is to hide existing program code in tightly integrated, randomized chaff code generated by this operator.

3.4.3 Chaff Code Generation

This code-injection operator inserts random expressions that corrupt and restore existing program variables, resulting in more thorough integration with target code. A variable is corrupted after an assignment (def) and restored prior to each reference (use). Corruption and restoration may occur at randomly selected locations between defs and uses.

Our current approach creates *shadow variables* to save correct values of corrupted program variables for restoration. An alternative is to corrupt variables reversibly and un-corrupt prior to uses. However, complex, unpredictable control flow complicates the task of matching up the corrupt and restore operations, unless the corruption is simple and generic (e.g., always the same operations). Nonetheless, either shadow variables or simple corruption/un-corruption may suffice if additional operators obfuscate the code output by this chaff generator.

For corruption, this operator uses assignments to random expressions produced by the random-code-generation operator. Via randomly built parse trees, such expressions may reference existing program variables, helping to integrate chaff code more securely. These expressions may read uninitialized variables, leading to compiler warnings; however, this is intentional and helps with obfuscation.

3.5 Non-Transformation Operators

These operators do not actually transform code, but perform other useful functionality. In our tool design, implementing various tasks is often best done simply by representing them as operators inserted at the desired positions in the transformation pipeline.

3.5.1 Source Generation

This operator generates source code from the tool intermediate representation (IR), but does not modify the IR. This can be used to generate source code at any point during processing, typically after all protection operators have finished. Since this operator transforms the IR instructions into source, obfuscation is naturally introduced into the output sources (in the same spirit as "obfuscation" due to compiling C++ into x86 code, for example). However, while this operator fits naturally as a "protection operator" in our implementation architecture (Section 5), it is not an obfuscation primitive per se.

3.5.2 Metric Evaluation

This operator provides functionality to compute complexity and security metrics over code. As with source generation, metric evaluation fits well into the architecture of our tool as just another operator, despite the fact that no transformations occur for metric computation. We describe metrics in the next section.

3.6 Other Operators

The above listing is meant to provide only a sampling of possible transformations. Depending on security goals and application contexts, the possibilities for other operators are virtually unlimited. We again emphasize that such operators may be nearly trivial and very easily implemented; the combined effect of iterating many such operators in different orders can create far more complexity than typical individual transformations.

4 COMPLEXITY EVALUATION VIA METRICS

In general, complex systems do not lend themselves to accurate prediction of future state. Such systems must be run forward or allowed to evolve, and state can be inspected at any time. Thus, instead of predictive modeling, we use *a posteriori metrics* to assess complexity and security in a quantitative manner. In other words, we evaluate various functions over code to quantify its properties in terms of complexity and security.

As a starting point, we use three specific metrics investigated in a quantitative study of obfusca-

tion (Anckaert et al., 2007b). These are used in a relative fashion; i.e., the metrics are computed over both original and transformed code, and the differences between these metric values serve as indicators of how much complexity was added by the transformations. The actual metrics are as follows:

- **Instruction Count.** This is simply the number of instructions in code, and serves as a very rough indicator of code complexity.
- **Cyclomatic number.** This is equal to $e - n + 2$, where e and n are the numbers of edges and nodes, respectively, in a function's control-flow graph (CFG). Intuitively, this indicates the number of decision points where control flow can take alternate paths.
- **Knot Count.** This is the number of crossings in a function's CFG, assuming the CFG is drawn in a specific manner; i.e., with nodes laid out linearly in order of address, and with edges all drawn on one side of the node list. Heuristically, this estimates the lack of typically expected structure in the CFG, as well as potential complexity of control-flow transfers in the CFG.

We also use other metrics designed to capture various complexity properties of code. Some examples are as follows:

- **Number of Variables per Instruction.** This is computed simply as v/c, where v the number of variables in a function's symbol table, and c is the number of instructions in the function's intermediate representation. Intuitively, this measures the potential complexity of data handling within the function.
- **Variable Indirection.** This is measured as p/v, where p is the number of pointers in a function, and v is the total number of variables. This is designed to capture the complexity of using pointers to access data indirectly.
- **Operational Indirection.** This is computed as r/R, where r is the number of pointer references in a function, and R is the total number of variable references; i.e., this is the fraction of references performed through pointers.
- **Code Homogeneity.** This measures the uniformity or "local indistinguishability" of instruction sequences throughout functions. This could be computed via histograms or frequency tables of instructions in selected portions of code.
- **Dataflow Complexity.** This is a data-centric analog of CFG-complexity metrics like cyclomatic number and knot count. One means of measurement is the complexity of a graph where each node represents a variable, and a directed edge between variables exists if the first variable influences the value of the other variable. If such a graph is complete, all variables influence one another. Thus, the metric may compute how close the graph is to a complete graph, or how "random" the graph appears to be.

5 IMPLEMENTATION AND EXPERIMENTAL RESULTS

We have implemented a Phoenix-based (Microsoft Corporation, 2008) toolkit that protects high-level code by iterated transformations. As emphasized earlier, this toolkit should be used to complement other methods, not to create a standalone, all-inclusive obfuscation solution. However, the source-based transformations used by the tool may help to implement other techniques. Moreover, arbitrary protective transformations in other tools can often be iterated (or modified to make iteration possible and effective).

The current section describes the architecture and usage of the tool, followed by experimental results on several SPEC CPU2006 benchmarks.

5.1 Phoenix-based Implementation

The tool implementation relies on the following systems:

- Phoenix (Microsoft Corporation, 2008): A Microsoft compiler and analysis framework based on a common intermediate representation (IR). We use Phoenix mainly as a code-processing engine that reads input code, both source and binary, and passes this to our tools for processing.
- .NET and CLR (Common Language Runtime): A next-generation Microsoft development and runtime environment. The tool is implemented using C# in the .NET framework.

Assuming the availability of some means to process input source code, iterated obfuscation lends itself to straightforward implementation. Relying on Phoenix as a code-processing engine, our basic tool design is centered on the concept of *protection operators*, which serve as primitives for iteration and recombination. Such an operator is a class that implements some protective transformation, such as OH or conversion of variable references to pointer references. Typically, each operator is derived from an abstract base operator class, which encapsulates useful basic functionality common to all operators.

Table 1: Metrics for one round of pointer conversion.

Benchmark	Code Size	Cyclomatic No.	Knot Count	Variable Density	Indirection	Performance
401.bzip2	1.035	1.000	1.000	1.513	1.663	1.572
429.mcf	1.063	1.000	1.000	1.646	1.325	1.055
458.sjeng	1.025	1.000	1.000	1.309	1.780	1.267

Table 2: Metrics for 5 rounds of pointer conversion.

Benchmark	Code Size	Cyclomatic No.	Knot Count	Variable Density	Indirection	Performance
401.bzip2	4.043	1.000	1.000	4.095	3.214	8.550
429.mcf	5.363	1.000	1.000	4.624	2.186	2.938
458.sjeng	3.114	1.000	1.000	2.731	3.703	5.379

At runtime, the tool applies a sequence of protection operators to each input function, as specified by a user-created configuration file. This text file contains an ordered list of operators specified by name, along with parameters for each operator (e.g., number of iterations to perform and names of functions to obfuscate). The tool executes the operators in order, as listed in the configuration file. Alternately, a *secret key* and additional user input may select a randomized subset of operators, number of iterations, order of application, and other parameters.

Our current tool works as a Phoenix compiler backend (C2) plug-in, operating on input C++ source code. A compiler (Visual C++) parses input source code into a high-level intermediate form (CIL, or C Intermediate Language). Phoenix converts CIL into its own universal high-level intermediate representation (HIR). The Phoenix backend then passes each HIR function to our plug-in tool, which applies the transformations and passes the function back to Phoenix for further processing and eventual native-code generation. As described above, the tool uses configuration files to steer its operation.

5.2 Experimental Results

The tables in this section present experimental results from running the Phoenix-based tool on selected SPEC CPU2006 benchmarks (data compression, transportation scheduling, and chess). For each benchmark, we computed metrics on the SPEC source code both before and after sample sets of transformations; we then calculated the ratios of these values. The results indicate how the metrics change as a result of applying the transformations. A value of 1.0 indicates that the corresponding metric was unaffected; values greater than 1.0 indicate higher complexity. Values less than 1.0 show lower complexity, which may occur as a result of higher complexity in other metrics. To estimate overall complexity, the metrics should be interpreted in combination.

The metrics in the tables include code size (in terms of the number of IR instructions), cyclomatic number, knot count, variable density, and operational indirection. These metrics are cumulative over all benchmark source functions. In addition, the rightmost value in each table indicates the performance hit due to the transformations; e.g., a value of 1.5 indicates that the transformed code took 1.5 as much time in our tests.

As a first example, Table 1 shows the effect of applying a single round of pointer conversion. This increases the instruction count slightly, but does not affect any metrics related to the CFG. The extra pointer variables cause increases in the variable density and operation indirection. Finally, depending on the benchmark, the performance hit is variable. This shows that pointer conversion should sometimes be applied selectively, avoiding performance-critical variables such as loop indices in compression algorithms (401.bzip2).

Table 2 shows the results of applying 5 rounds of pointer conversion. Every round approximately doubles the number of variables, including new pointers to existing pointers from all earlier rounds. Thus, the effect on some metrics is exponential.

Table 3 shows the results of injecting 10 opaque predicates into each function. This also creates some additional non-pointer variables, increasing variable density but reducing operational indirection. The cyclomatic number and knot count also increase, since randomly injected opaque branches may straddle other branches. This comes at relatively little expense in code size and especially performance.

Table 4 adds a round of OH to 10 rounds of opaque predication. Since OH was applied to hash every relevant variable, including performance-critical loop indicates, the results are expensive in space and time. This shows that OH may need to be applied selectively.

Table 5 shows an example where iterating multiple rounds of several transformations results in a pro-

Table 3: Metrics for 10 rounds of opaque predication.

Benchmark	Code Size	Cyclomatic No.	Knot Count	Variable Density	Indirection	Performance
401.bzip2	1.191	1.580	1.899	1.277	0.712	1.036
429.mcf	1.397	2.137	4.241	1.525	0.672	1.019
458.sjeng	1.192	1.409	1.713	1.149	0.734	1.068

Table 4: Metrics for 10 rounds of opaque predication plus a round of OH.

Benchmark	Code Size	Cyclomatic No.	Knot Count	Variable Density	Indirection	Performance
401.bzip2	2.979	1.580	1.862	0.903	1.331	12.727
429.mcf	3.543	2.137	3.952	0.692	0.886	6.403
458.sjeng	3.337	1.409	1.707	0.719	1.587	12.360

Table 5: Metrics for 10 rounds of opaque predication, 2 rounds of OH, 3 rounds of random-code injection, and 3 rounds of pointer conversion.

Benchmark	Code Size	Cyclomatic No.	Knot Count	Variable Density	Indirection	Performance
401.bzip2	48.788	3.285	2.258	0.696	3.919	68.164
429.mcf	61.310	4.147	5.234	0.427	2.347	41.966
458.sjeng	59.147	2.883	2.032	0.517	4.784	80.683

tective code envelope that dwarfs the code size of the actual SPEC benchmarks. While this results in final code several dozen times larger and slower, such protection can still be used in areas where performance is not critical (e.g., typical DRM and license checks).

We note that all our experiments involved applying operators over the entire code of the selected benchmarks. Thus, effects on code size and performance are sometimes significant, especially when transformations impact performance-sensitive program elements. While the tables show such worst-case scenarios, typical usage may involve selective application of operators. For example, users may specify performance-critical variables and code sections where operators should limit or omit processing. Also, users may indicate which application sections should be protected, though transformations should be applied elsewhere as well (to avoid attracting attention to security-sensitive parts). Additional complexity and unpredictability are possible via a user-specified secret key used to select operators and parameters. Via these and other means, a balance between performance, code size and metric complexity may be achieved for different applications.

6 CONCLUSIONS AND FUTURE WORK

This paper presented a framework for design and implementation of software protection via iteration and recombination of simple primitives. As in complex systems, such a process can lead to cascading complexity and emergent behavior via the interaction of multiple transformations. The nature of individual transformations, as well as number of iterations and order of application, can make dramatic differences in the final output code. We demonstrated the use of quantitative metrics to evaluate the complexity and corresponding security of transformed code. Such an approach may be useful as part of a comprehensive software-protection system.

Future work will involve designing and implementing additional protection operators, as well as analyzing their security and benefits. A main goal is to position this work in a formal context, including analysis that accurately estimates the practical resistance of our methods against hacker attacks. We also plan to investigate how iterated obfuscation can help other approaches currently under development, perhaps combining various standalone methods into a more systematic, comprehensive solution for software protection.

REFERENCES

Anckaert, B., Jakubowski, M. H., Venkatesan, R., and Bosschere, K. D. (2007a). Run-time randomization to mitigate tampering. In *2nd International Workshop on Security (IWSEC 2007)*, Nara, Japan.

Anckaert, B., Madou, M., De Sutter, B., De Bus, B., De Bosschere, K., and Preneel, B. (2007b). Program obfuscation: a quantitative approach. In *QoP '07: Proceedings of the 2007 ACM workshop on Quality of protection*, pages 15–20, New York, NY, USA. ACM.

Anckaert, B., Sutter, B. D., and Bosschere, K. D. (2004). Software piracy prevention through diversity. In *DRM '04: Proceedings of the 4th ACM Workshop on Digital Rights Management*, pages 63–71, New York, NY, USA. ACM Press.

Aucsmith, D. (1996). Tamper resistant software: An implementation. *Information Hiding, Lecture Notes in Computer Science*, 1174:317–333.

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., and Yang, K. (2001). On the (im)possibility of obfuscating programs. In *Electronic Colloquium on Computational Complexity*, volume 2139, pages 1–18.

Chen, Y., Venkatesan, R., Cary, M., Pang, R., Sinha, S., and Jakubowski, M. H. (2002). Oblivious hashing: A stealthy software integrity verification primitive. In *Information Hiding 2002*, Noordwijkerhout, The Netherlands.

Collberg, C., Thomborson, C., and Low, D. (1997). A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, The University of Auckland, New Zealand.

Collberg, C., Thomborson, C., and Low, D. (1998a). Breaking abstractions and unstructuring data structures. In *International Conference on Computer Languages*, pages 28–38.

Collberg, C., Thomborson, C., and Low, D. (1998b). Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages, POPL'98*, pages 184–196.

Dedić, N., Jakubowski, M. H., and Venkatesan, R. (2007). A graph game model for software tamper protection. In *Proceedings of the 2007 Information Hiding Workshop*.

El-khalil, R. and Keromytis, A. D. (2004). Hydan: Hiding information in program binaries. In *International Conf. on Information and Communications Security (ICICS)*.

Goldreich, O. and Ostrovsky, R. (1996). Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473.

Goldwasser, S. and Kalai, Y. T. (2005). On the impossibility of obfuscation with auxiliary input. In *FOCS '05: Proceedings of the 46th IEEE Symposium on Foundations of Computer Science*.

Horne, B., Matheson, L. R., Sheehan, C., and Tarjan, R. E. (2001). Dynamic self-checking techniques for improved tamper resistance. In *Digital Rights Management Workshop*, pages 141–159.

Jacob, M., Jakubowski, M. H., and Venkatesan, R. (2007). Towards integral binary execution: Implementing oblivious hashing using overlapped instruction encodings. In *2007 ACM Multimedia and Security Workshop, Dallas, TX*.

Lynn, B., Prabhakaran, M., and Sahai, A. (2004). Positive results and techniques for obfuscation. In *Eurocrypt '04*.

Menezes, A. J., Vanstone, S. A., and Oorschot, P. C. V. (1996). *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA.

Microsoft Corporation (2008). Phoenix compiler framework.

Tan, G., Chen, Y., and Jakubowski, M. H. (2006). Delayed and controlled failures in tamper-resistant software. In *Proceedings of the 2006 Information Hiding Workshop*.

Wang, C. (2000). *A Security Architecture for Survivability Mechanisms*. PhD thesis, University of Virginia.

Wang, C., Hill, J., Knight, J., and Davidson, J. (2000). Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia.

Wee, H. (2005). On obfuscating point functions. In *STOC '05: Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, pages 523–532, New York, NY, USA. ACM Press.

Wolfram, S. (2002). *A New Kind of Science*. Wolfram Media Inc., Champaign, IL, USA.