The Superdiversifier: Peephole Individualization for Software Protection

Matthas Jacob[1], Mariusz H. Jakubowski[2], Prasad Naldurg[3],
Chit Wei (Nick) Saw[2], and Ramarathnam Venkatesan[2,3]

[1] Nokia
[2] Microsoft Research
[3] Microsoft Research India

Abstract. We present a new approach to individualize programs at the machine- and byte-code levels. Our *superdiversification* methodology is based on the compiler technique of superoptimization, which performs a brute-force search over all possible short instruction sequences to find minimum-size implementations of desired functions. Superdiversification also searches for equivalent code sequences, but we guide the search by restricting the allowed instructions and operands to control the types of generated code. Our goal is not necessarily the shortest or most optimal code sequence, but an individualized sequence identified by a secret key or other means, as determined by user-specified criteria. Also, our search is not limited to commodity instruction sets, but can work over arbitrary byte-codes designed for software randomization and protection. Applications include patch obfuscation to complicate reverse engineering and exploit creation, as well as binary diversification to frustrate malicious code tampering. We believe that this approach can serve as a useful element of a comprehensive software-protection system.

1 Introduction

One element of a comprehensive program-security toolbox is *code individualization*, which enables software diversity as a defense against various attacks. When different users receive individualized copies of the same sensitive application, a break on one system may not work on others, ideally forcing crackers to expend the same effort on breaking each copy. Individualization also helps to alleviate the so-called *monoculture problem* [15], which involves quick propagation of malicious programs on networks of equally vulnerable systems. As in biology and fault-tolerant systems, diversity can be an effective generic technique to guard against known and unknown threats.

Software individualization uses a number of techniques to modify code at various levels without changing its semantics [1, 2, 8]. For example, at an algorithmic level, a bubble-sort function can be replaced by quicksort or heapsort. At a syntactic level, a tool can inject inert “chaff” code, reorder basic blocks by flipping branch conditions, and replace instructions with operationally equivalent sequences. Such measures may be implemented via compilers and transformation tools.

In this paper, we focus on individualization at the instruction or byte-code level. Our main idea is search for a large number of instruction sequences that are semantically equivalent to an input code fragment. To accomplish this, we leverage and extend the compiler technique of superoptimization [4, 19, 21], which performs brute-force searches for minimum-size code fragments that implement a given simple function. Our work thus proposes a novel application of superoptimization towards software security, including areas such as exploit prevention and tamper-resistance, and leading towards code obfuscation.

1.1 Software Protection

Since the early days of the IBM PC and 8-bit home computers, software protection has been a never-ending battle between protectors and crackers. While no secure generic solution has been demonstrated in practice, various techniques of tamper-resistance and obfuscation have helped to delay reverse engineering and cracking of sensitive code and data [3, 7, 9, 10, 11, 18, 24, 27]. Among such approaches are code checksums or hashes to prevent easy patching, anti-debugging and anti-disassembly measures to hinder attack tools, and various obfuscation measures to complicate control- and data-flow analysis. For increased security, modern protection tools typically combine a large variety of such features.

Recent theoretical work [5, 16, 20, 28] suggests that only limited obfuscation is possible in general. However, such modeling has been generally distant from practice, particularly given the large variety of possible application and attack scenarios. We note that most published work on real-world software protection does not provide a cryptographic model or formally analyzable security. Even in systems whose breaking reduces to solving some difficult problem, it is often possible to "go outside the model" to produce efficient attacks. A system with guaranteed or estimated tamper-resistance measured in days, weeks or months may be formally insecure, yet quite sufficient for various business applications. Thus, presently deployed protection strategies generally rely not on theoretical models, but rather on heuristic analysis, quality of engineering, and penetration testing on software protected by combined techniques. In this context, individualization serves as an element of a comprehensive protection strategy or model [12].

1.2 Superoptimization and Software Diversity

Superoptimization was introduced by Massalin as a means of minimizing size of compiled code [21]. Though software security was not a stated goal of his original work, his description hints at some obfuscation and individualization inherent in superoptimized code. Massalin wrote that "startling programs have been generated, many of them engaging in convoluted bit-fiddling bearing little resemblance to the source programs which defined the functions" [21]. One of his examples is a sign function superoptimized to four 680x0 instructions, where "like a typical superoptimized program, the logic is really convoluted" [21] and warrants the detailed explanation he provides. Such code sequences often contain clever tricks that appear to be human-generated, yet were found automatically by simply enumerating all possible instructions up to a certain length. In essence, brute-force search has substituted for human cleverness.

Motivated by the above observations, we build upon superoptimization to develop a diversification toolkit. In particular, we output not only the shortest instruction sequence, but any sequences that implement the input function. Additionally, we typically use a secret key and other user-specified parameters to guide the brute-force search and control the nature of generated code (e.g., by changing the set of instructions and operands over which the search is performed). Our search may also use other heuristics, such as instruction frequencies collected from real-life programs. For example, the search may prefer instruction combinations observed often in real code, deferring or omitting instruction sequences not commonly found in binaries. We also use other performance enhancements, such as limiting exhaustive enumeration of constant operands (e.g., restricting constants to small ranges or sets of values harvested from real-life software). We refer to our approach as *superdiversification* or *superindividualization.*

The rest of the paper is organized as follows. We outline our basic methodology of superdiversification in Section 2. Implementation details and performance results are the topic of Section 3. In Section 4, we show several applications of our basic methodology. Finally, Section 5 summarizes our approach with directions for future work.

2 Superdiversification

Like a superoptimizer, our code individualizer accepts a sequence of instructions as input and attempts to generate equivalent code sequences up to a given length, by exhaustive enumeration. A secret key and other user-specified parameters may further determine characteristics of these code fragments and order of generation, as explained in detail later. Each candidate sequence is tested for equivalence to the input sequence. The test proceeds in two phases – a quick, probabilistic equivalence check, which rejects sequences that do not agree on random inputs, and Boolean equivalence checking of formulas when the sequences agree on a threshold number of random inputs:

– During the most common, quick phase of the test, the sequence is first executed or simulated on one or more random sample inputs; if the output is incorrect, the sequence is quickly discarded.
– If the candidate sequence generates correct outputs, a Boolean test is used to verify that the sequence is actually equivalent to the input sequence. For this, we encode instructions as Boolean formulas in conjunctive normal form (CNF) and use a SAT solver to test equivalence, as described later.

2.1 Generation of Code Sequences

In our basic approach, we enumerate a large number of candidate code sequences to test for equivalence with an input sequence. In particular, superdiversification performs one or more of the following actions:

– Restrict the set of instructions and operands over which the search is performed. For example, if we wish to generate a sign function that uses only addition, subtraction and negation, we simply search over just these three instructions. This has an effect similar to switching from brute-force to heuristically-guided search.
– Guide the search based on empirical data about probabilities of instruction occurrences in real-life programs. When choosing the next instruction in a candidate sequence, our enumeration procedure prefers instructions that are likely to appear, as determined by a table of instruction frequencies harvested from actual binaries. This is intended both to speed up the search and produce diversified code that blends in with target applications.
– Use various optimizations and pruning techniques to cut down on the time required for search. For example, a user-specified parameter determines whether or not to search over instruction pairs and longer sequences that have never been observed in actual binaries. In addition, to prevent brute-force enumeration of constant operands (e.g., 64-bit), we collect or harvest constants from real-life programs and search only over these. Alternately, we may use random (key-based) sampling to choose constants, as well as to enumerate instruction sequences in general. With such sampling, the search will not be exhaustive, but may still yield usefully individualized sequences.

Given an input code sequence of length m, we generate all code sequences up to a specified maximum length n, where $n \geq m$. For instruction selection, we use a subset of the processor's instruction set. The generated sequence uses the same inputs and modifies the same outputs as the input sequence. In addition, the generated sequence may also use a limited set of constant operands, restricted to commonly occurring values (e.g., 0, 1, -1), or alternately drawn from a set of constants harvested from real-life programs. We may also choose to reject generated sequences containing obviously redundant instructions that amount to no-ops unlikely to occur in actual binaries (e.g., `mov r1, r1`, or `add r1, 0`). To find such instructions, we can run the superdiversifier on a no-op input sequence, and ask it to find all equivalent sequences of length 1.

In order to improve search time, as well as to produce sequences that blend in more closely with actual programs, we can use a table containing empirical data about instruction frequencies. The table data are pre-computed from a sample set of actual binaries, where f_{jk} is the frequency with which instruction i_k is observed to follow instruction i_j in the sample set. The code generator will pick the next instruction in the generated sequence in order of decreasing frequency, based on the most recent instruction generated. This will naturally favor sequences that appear to blend in with compiled code in real binaries, thus making it more difficult to distinguish between unmodified compiler-generated code and code produced by the superdiversifier in an individualized binary.

We may further guide the search by randomly re-ordering the enumeration based on a secret key, or by eliminating instructions – either randomly or using heuristic information based on knowledge about the input sequence.

2.2 Practical Issues

In both superdiversification and superoptimization, the search space for code generation is exponential in the number of instructions in the sequence. Since superoptimization discards all instruction sequences longer than the current minimum-length equivalent sequence, its brute-force search often may be quite fast. In contrast, superdiversification searches over sequences of all lengths (within a user-specified range), and is thus inherently slower. However, even two decades ago, Massalin [21] was able to find superoptimized sequences of 10 or more instructions. Since computing systems today are several orders of magnitude more powerful, we expect that superdiversification can still be quite effective, particularly with the use of optimizations and pruning mechanisms.

Our approach is useful for generating various types of code, as determined by parameters. Using empirical instruction frequencies, as mentioned earlier, we can produce diversified code that does not stand out from the target application, helping to improve security. Alternately, we may produce unusual code by preferring instructions unlikely to appear (i.e., using the inverse of our instruction-frequency table). In general, parameterized diversification is a powerful enhancement over superoptimization in terms of security.

2.3 Testing for Equivalence of Code Sequences

To determine whether two machine-code fragments compute the same function, we first execute each sequence using one or more sets of randomly generated inputs, and compare their corresponding outputs. If their outputs are equal, we consider the sequences to be potentially equivalent and convert the instructions to Boolean formulas in conjunctive normal form (CNF). For each instruction, we assign distinct Boolean variables representing each bit of its inputs and outputs. The output variables are the result of the Boolean function performed by the instruction over its inputs. For example, consider the generic instruction

```
and r1, r2
```

which performs a bitwise-AND on registers `r1` and `r2`, placing the result in register `r1`. Let the variable x represent `r1` and y represent `r2`. Then we can represent the function for the `and` instruction by

$$F(x, y) = (x \wedge y)$$

Using static single assignment (SSA) notation such that each variable is assigned no more than once, we can rewrite this as the relation

$$x1 \leftrightarrow (x0 \wedge y0)$$

Note that we would have one such relation for each data bit. For example, for a w-bit data width, we would have w relations each representing the effect of the `and` instruction on each bit of the output. Since all variables are unique, we can simply combine them in a single conjunction as follows:

$$\varphi = \bigwedge_{i=1}^{w} (x1_i \leftrightarrow (x0_i \wedge y0_i))$$

To ensure that the resulting formula is in CNF, each term in the conjunction needs to be converted to CNF. We apply the Tseitin transformation [25] to the above expression, resulting in

$$\varphi = \bigwedge_{i=1}^{w} ((\neg x1_i \vee x0_i) \wedge (\neg x1_i \vee y0_i) \wedge (x1_i \vee \neg x0_i \vee \neg y0_i))$$

We repeat the above for each instruction in the sequence, encoding the operation performed by the instruction as a Boolean expression in CNF form. Thus, for a length-N instruction sequence, we would have N sub-formulas, and the entire sequence would be represented as

$$\Phi = \bigwedge_{i=1}^{N} \varphi_j$$

Let Φ' be the generated length-M sequence that we wish to test for equivalence against Φ.

$$\Phi' = \bigwedge_{i=1}^{M} \varphi'_j$$

We first ensure that the input states of both sequences are synchronized such that the same variables are used to represent inputs that are common to both sequences. In the above example, the initial values of `r1` and `r2` are represented by the variables $x0$ and $y0$ in both sequences. All other variables representing the intermediate and output states should be distinct and unique to each sequence. Let A be the set of variables representing the output state of Φ, and A' be the set of variables representing the output state of Φ'. Then we can define equivalence in terms of the output state as follows:

$$\Phi \equiv \Phi' \qquad \text{iff} \qquad \forall x \in A, y \in A' : A \leftrightarrow A'$$

or,

$$\Phi \not\equiv \Phi' \qquad \text{iff} \qquad \exists x \in A, y \in A' : A \not\leftrightarrow A'$$

For each bit variable x_i in A corresponding to the same bit in A' (represented by y_i), we generate the relation

$$z_i \leftrightarrow (x_i \not\leftrightarrow y_i)$$

This is equivalent to

$$z_i \leftrightarrow (x_i \oplus y_i)$$

Thus, for K output bits, we can state that

$$\Phi \not\equiv \Phi' \qquad \text{iff} \qquad \bigvee_{i=1}^{K} z_i = \top$$

In other words, $\Phi \equiv \Phi'$ iff the following CNF formula is *un*satisfiable:

$$(\Phi \wedge \Phi' \wedge (\bigvee_{i=1}^{K} z_i))$$

Generating the above formula and applying a SAT solver to it, we can thus determine whether two sequences are equivalent. If the SAT solver finds that the formula is satisfiable, then we have established that the sequences are *not* equivalent. Conversely, if the SAT solver determines that the formula is unsatisfiable, then the sequences are equivalent.

3 Implementation and Experimental Results

We implemented an initial prototype of the code individualizer in C++. This consists of a code-generation module (front-end) and a code verification module (back-end). We included only a subset of Intel x86 instructions (`mov`, `not`, `neg`, `xor`, `or`, `and`, `add`, `sub`, `inc`, `dec`, `cmp`, `shl`, `shr`, `sar`, `push`, `pop`), but the design allows future support for other instruction sets to be added. (In particular, we may use custom instruction sets geared towards individualization and obfuscation, not necessarily generality and performance.) For the verification phase, we perform a quick execution test on every sequence generated, followed by a Boolean test using the zChaff [26] SAT solver only if the quick test passes. With this approach, we were able to generate code sequences of up to 5 or 6 instructions in length in reasonable time over the full range of supported instructions. We ran all our tests on a Pentium 3.0 GHz CPU with 2 GB of RAM. Note that emulation would be needed to run the quick test on non-native instruction sets.

The code individualizer works as follows. Given an input machine-code fragment, the generator enumerates the next candidate code sequence. The verifier then compares the generated sequence against the original sequence. If it is determined that the two sequences are equivalent, the generated sequence is added to the list of equivalent sequences. The process is then repeated until the generator has generated all sequences.

While our approach generates code sequences that are exact matches with respect to the output states, in practice it is frequently sufficient to find a match over a subset of the output state. For example, in the Intel x86 instruction set, the `inc` and `add` instructions affect the carry flag differently. However, if we do not care about the value of the carry flag (i.e., it is not live) in the context of the original program, the instruction `inc r1` could be substituted with `add r1, 1` and vice versa. In order to allow for such possibilities, we made provisions in the code individualizer to designate flags and output variables that may be safely ignored during the verification phase. To store intermediate results, we also provided the ability to introduce free temporary registers not already in the input sequence into the generated sequence.

In general, modeling arbitrary memory accesses is a complex problem. In our current implementation, we limited the inputs and outputs to constants, registers, and stack memory variables only.

3.1 Sample Results and Discussion

We ran the code generator and verifier on a sample set of input sequences. In order to produce a greater variety of generated sequences, we chose to ignore the processor flags when comparing output states. Consequently, further program analysis to determine live flags would be necessary prior to considering any of the generated sequences for substitution in the original program.

We present results for some input sequences taken from actual programs. Consider a 3-instruction sequence, *seq1*, that swaps the contents of two registers, `r1` and `r2`, using `r3` as a temporary register[1]:

seq1:

```
mov r3, r1
mov r1, r2
mov r2, r3
```

Assuming `r3` is free and ignoring flag side-effects, the code individualizer was asked to generate all equivalent sequences up to length 5 using only the `mov`, `xor`, `push` and `pop` instructions. A total of 2426 equivalent sequences were found out of 8308224 generated in total. To minimize the correlation between generated equivalents and the input sequence, we filtered out sequences containing any of the original instructions in the input sequence. We also filtered out obvious no-op instructions. Table 1 shows a small sample of some of the more interesting sequences generated for different lengths n. The number in parentheses is the total number of equivalent sequences of length n that were found.

By selecting a different set of instructions for generation (e.g., using a secret key), we generate a completely different set of equivalent sequences for *seq1*. For example, Table 2 shows some of the equivalents we found using only arithmetic and logical operations.

In contrast to superoptimization, the generated sequences were no more optimized in size and execution time than the original sequence, and in most cases resulted in longer, less efficient code. However, consistent with the objectives of superdiversification, each sequence produced by the code individualizer presents a different implementation of the same input function, and in some cases the implementation can be very different (and non-obvious) from the original code.

As another example, consider a sequence consisting of a `mov` followed by an `add` instruction. This sample was taken from a binary generated by the Microsoft Visual C++ compiler:

seq2:

```
mov r1, r2
add r1, 0x10
```

[1] For generality, we use r1, r2, etc., to denote the register operands in these examples rather than their x86 names.

Table 1. Sample Generated Equivalents For Sequence 1 (a)

$n = 3$ (4)	$n = 4$ (113)	$n = 5$ (2309)

`xor r1, r2`
`xor r2, r1`
`xor r1, r2`
	`push r1`
	`push r2`
	`pop r1`
	`pop r2`
		`xor r2, r1`
		`mov r3, r2`
		`mov r2, r1`
		`xor r3, r2`
		`mov r1, r3`

`push r2`
`mov r2, r1`
`pop r1`
	`push r1`
	`xor r1, r1`
	`xor r1, r2`
	`pop r2`
		`xor r2, r1`
		`push r2`
		`mov r2, r1`
		`pop r3`
		`xor r1, r3`

`mov r3, r2`
`mov r2, r1`
`mov r1, r3`
	`mov r3, r2`
	`push r1`
	`mov r1, r3`
	`pop r2`
		`xor r2, r3`
		`xor r3, r1`
		`xor r1, r2`
		`xor r2, r1`
		`xor r1, r3`

Table 2. Sample Generated Equivalents For Sequence 1 (b)

$n = 4$ (4)	$n = 5$ (176)

`add r1, r2`
`sub r2, r1`
`add r1, r2`
`neg r2`
	`and r3, r1`
	`or r3, r1`
	`sub r3, r2`
	`add r2, r3`
	`sub r1, r3`

`sub r1, r2`
`add r2, r1`
`sub r1, r2`
`neg r1`
	`sub r3, r3`
	`add r3, r2`
	`sub r3, r1`
	`add r1, r3`
	`sub r2, r3`

Feeding this to the individualizer and allowing it to utilize a small set of constants, Table 3 illustrates some of the equivalents we were able to generate:

Table 3. Sample Generated Equivalents For Sequence 2

$n = 2$ (1)	$n = 3$ (122)	$n = 4$ (13538)

`mov r1, 0x10`
`add r1, r2`
	`mov r1, 0x4`
	`shl r1, 0x2`
	`add r1, r2`
		`mov r1, 0xf`
		`xor r1, 0xffffffff`
		`sub r1, r2`
		`neg r1`

	`mov r1, 0x8`
	`add r1, r1`
	`add r1, r2`
		`mov r1, 0xf`
		`sub r1, 0x2`
		`add r1, r2`
		`add r1, 0x3`

Finally, we created the following 2-instruction input sequence to illustrate some additional types of sequences the individualizer generates. This sequence simply adds 1 to the value in register r1, then clears its most significant bit:

seq3:

```
add r1, 0x1
and r1, 0x7fffffff
```

Of the equivalent sequences generated by the individualizer, we observe groups of sequences that share certain common characteristics, which we attempt to classify in Table 4. Each sample is an example of a class of transforms bearing distinct features that we frequently observe in the generated results.

These categories of sequences were representative of what we observed when generating equivalents for a variety of input sequences. It appears that those belonging to Categories I and II are most "different" from the original, as might be measured by their edit distance.

We found that without explicitly filtering out sequences containing the original instructions, many of the longer-length generated sequences fell into Category V or Category VI, comprising simply the original sequence with the addition of inert "chaff" instructions. While these sequences do not constitute a transformed version of the original, they may still be useful from the point of view of diversity, since we can generate a large number of such sequences. However, they may simply be filtered out as mentioned above and discarded if we want to exclude these categories of sequences. A related class of transforms are those comprising previously generated sequences of shorter length, with the addition of chaff instructions. Similarly, these may be filtered out by excluding all sequences that contain any sub-sequence that has already been encountered.

3.2 Performance

As with superoptimization, the quick execution test is key to dramatically reducing the search time (by approximately a factor of 50 in our current implementation) by ruling out obviously non-equivalent sequences without having to perform the much slower Boolean test. This does, however, impose the requirement of either running on native hardware or under emulation, the latter of which would likely have performance implications.

We observed that the number of times the quick execution was run on each generated sequence can be a factor in whether or not the quick test yields false-positive matches, leading to additional Boolean test runs on non-equivalent sequences and hence reduced speed. Generally, the fewer output-bits that are altered by the sequence, the more execution test runs are required (over different random inputs) to reduce the number of false positives. Figures 1 and 2 illustrate the false-positive hit-rates and their impact on execution speed for different numbers of quick test runs.A relatively small number of quick-test runs on random input sets (about 4) is sufficient to reduce the rate of false positives

Table 4. Categories of Equivalent Sequence Transforms

Category	Characteristics	Examples

I. Corruption	Transformation or corruption of the original input, operation in the transformed domain, followed by uncorruption to yield the correct output.	`add r1, 0x1`
		`and r1, 0x7fffffff` ⟶ `shl r1, 0x1`
		`add r1, 0x2`
		`shr r1, 0x1`

II. Substitution	Substitution of one or more operations in the original sequence with a different operation that has the same effect.	`add r1, 0x1`
		`and r1, 0x7fffffff` ⟶ `add r1, 0x2`
		`sub r1, 0x1`
		`shl r1, 0x1`
		`shr r1, 0x1`

III. Instruction Re-ordering	Altering the order of the instructions where their order does not change the final result.	`add r1, 0x2`
		`sub r1, 0x1`
		`shl r1, 0x1`
		`shr r1, 0x1` ⟶ `sub r1, 0x1`
		`add r1, 0x2`
		`shl r1, 0x1`
		`shr r1, 0x1`

IV. Operand Re-ordering	Altering the order in which the operands appear in the sequence while preserving the final result.	`mov r1, r2`
		`add r1, 0x10` ⟶ `mov r1, 0x10`
		`add r1, r2`

V. Chaff Code	Addition of extra inert instructions or sequences of instructions into the original sequence (or a previously derived equivalent), such that they appear to be integral to the overall function but are in fact functionally irrelevant.	`mov r1, r2`
		`add r1, 0x10` ⟶ `mov r1, r2`
		`or r1, r2`
		`and r1, r2`
		`add r1, 0x10`

VI. Chaff (Obvious)	Addition of extra inert instructions or sequences of instructions into the original sequence (or a previously derived equivalent) that are clearly irrelevant.	`mov r1, r2`
		`add r1, 0x10` ⟶ `mov r1, r2`
		`mov r1, r1`
		`add r1, 0x10`

to the point where it no longer materially impacts the search. Note that in the case of Sequence 3, significantly increasing the number of test runs beyond this level does little to eliminate a residual number of false positives. In fact, with as

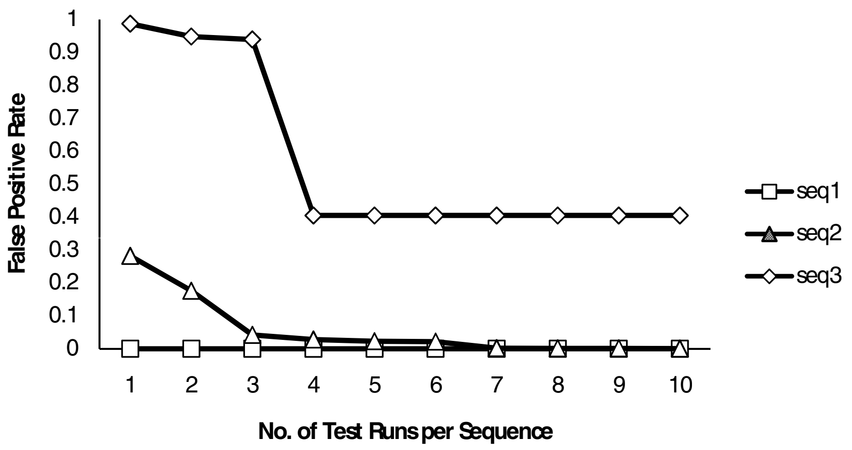

Fig. 1. Quick Execution Test False Positives

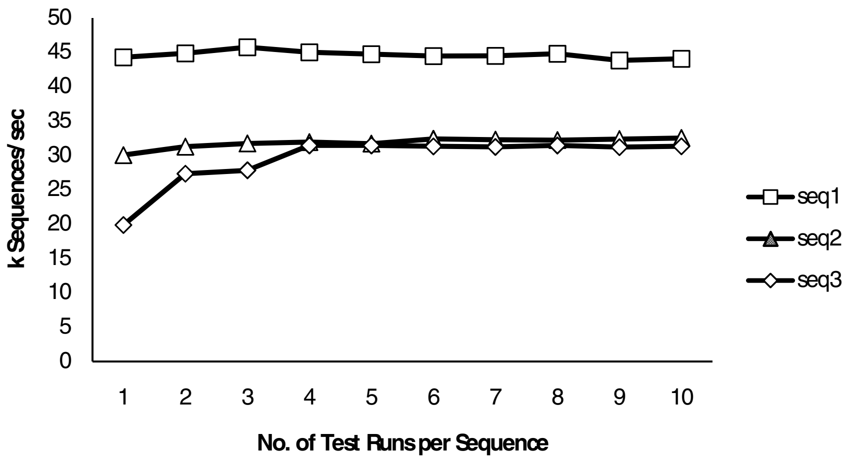

Fig. 2. Sequences Processed per Second

many as 1000 test runs, the same false-positive matches were present. This is due to the fact that for many input values, only a few bits of output are affected. In the case of Sequence 3, for example, consider the following false-positive match, where the instructions are simply interchanged:

```
and r1, 0x7fffffff
add r1, 0x1
```

It can be seen that for most initial values of `r1`, this sequence will yield the same result as the original sequence. It is only when we have input values of `0x7fffffff` or `0xffffffff` that the difference is exposed, and it is hard to achieve this even with many random test inputs. Compare this to Sequence 1 (swapping two registers), where for most randomly selected inputs, a large number of output bits are changed. In fact, we did not observe any false positives for this sequence, even with only a single run of the quick test.

Due to the size of the search space, the individualizer was practically limited to generating sequences of approximately 5 or 6 instructions long, depending on the number of inputs and outputs. Even so, it was able to find a diverse set of equivalents for a variety of input sequences. We expect to improve the performance in a future version with additional optimizations to the sequence-generation algorithm and better search-pruning strategies.

4 Applications

Our methodology fits well within the larger context of software individualization as a security mechanism. In particular, superdiversification is useful at the machine-instruction and byte-code levels by offering a systematic means of enumerating all possible customizations. Combined with higher-level methods, such as source-based control-flow transformations, our approach helps to complete a full-fledged diversification solution against attacks enabled by software uniformity and "monocultures." We briefly describe some specific scenarios that benefit from such defenses.

4.1 Signature-Based Attacks

Certain malicious software attempts to patch binaries by scanning for particular byte patterns within executables and replacing these sequences with attack code. For example, patching tools modify DRM and game binaries to eliminate security measures like license checking and binding to a CD/DVD. Other attack tools will even scan anti-virus binaries and other security software to alter and disable their protection, allowing infiltration by rootkits, adware, bot drivers and other malware. By eliminating signatures from customized copies of security-sensitive binaries, our techniques can hinder such attacks, especially when superdiversification is used alongside higher-level individualization.

Both malware and legitimate security software have used certain individualization techniques to defeat signature-based detection [22]. Polymorphic viruses, for example, may use variable code encryption and instruction replacement based on a small library of equivalents, while metamorphic code may transform itself to an intermediate representation (and back to native instructions) for broader diversity [29]. However, no technique is likely to be very strong when used alone, and a comprehensive individualization solution should combine various approaches. In this context, our technique fits well as a means of generating nontrivial code equivalents that appear to require human cleverness.

Our methodology could potentially be used to "normalize" code sections, or transform them into a standardized form to enable easier signature-based searches. However, we would normally apply superdiversification over code blocks chosen via a secret key. In addition, since we typically iterate the process, the next block to be individualized may overlap with other blocks, potentially including ones that have already been diversified. Other individualization tactics could be used to move code blocks and inject "chaff" code, further complicating any attempts at comparing and matching code sections. Thus, "code normalization" is unlikely to be effective, particularly if we employ a combination of individualizing transforms.

4.2 Patch Obfuscation

A compelling application of our technique is patch obfuscation. As mentioned earlier, in order to make reverse engineering of binaries harder, software providers employ a variety of code-transformation techniques. This can make the code difficult to reverse-engineer in its entirety, but may have little effect on the difficulty of comparing two similar binaries. When a security patch is released, tools such as BinDiff [23] and EBDS [13] can quickly discover differences between patched and unpatched versions of software, easily pinpointing vulnerable code fragments addressed by the patch. BinDiff represents a class of graph-based matching tools that diff two binaries via various heuristics on basic blocks and control-flow graphs (CFGs). As an advancement over simple byte-based diffing, such tools are robust to minor or syntactic transformations on control- and data-flow paths. Recent work [6] has even shown that it's possible to automate the process of finding inputs that exercise vulnerabilities in unpatched binaries.

In this context, we claim that our superdiversification method can delay or prevent reverse-engineering by maximizing the differences between patched and unpatched binaries. In conjunction with source-level and other individualization methods, our approach can provide different semantically equivalent patches to vulnerabilities. In addition, a patch update can include replacements for other code fragments that are not related to the vulnerability being fixed. In fact, most of the protection against sophisticated patch diffing can stem from individualizing unrelated parts of the binary, as well as adding inert "chaff" code and temporary corruption of variables. A cracker attempting to reverse-engineer the code using a graph-based matching tool will find many differences between the original and patched code. A determined cracker may be able to narrow this down eventually, but the slowdown in analysis will translate to a slowdown in vulnerability exploitation and improve the utility of the patch. Colluding crackers will have an even harder time at comparing patterns. While it is difficult to make any formal arguments, we believe that this approach can provide a practical solution to the patch-reverse-engineering issue. As a bonus, we may even be able to improve the performance of some patched versions if our equivalent code sequences are smaller in size than the original.

As a simple example, consider the following basic block:

```
51              push     ecx
8B 45 08        mov      eax,dword ptr [ebp+8]
8B 48 0C        mov      ecx,dword ptr [eax+0Ch]
```

```
81 F1 73 AE 28 3F  xor         ecx,3F28AE73h
89 4D FC           mov         dword ptr [ebp-4],ecx
8B 45 FC           mov         eax,dword ptr [ebp-4]
33 D2              xor         edx,edx
B9 07 00 00 00     mov         ecx,7
```

Using a sliding code window, we applied our superdiversification method over short sequences in this block. By using different keys and other parameters to vary the search, we can generate many different equivalent blocks. One example is below:

```
51                 push        ecx
FF 75 08           push        dword ptr [ebp+8]
58                 pop         eax
68 73 AE 28 3F     push        3F28AE73h
59                 pop         ecx
33 48 0C           xor         ecx,dword ptr [eax+0Ch]
51                 push        ecx
58                 pop         eax
50                 push        eax
8F 45 FC           pop         dword ptr [ebp-4]
03 D2              add         edx,edx
2B D2              sub         edx,edx
6A 07              push        7
59                 pop         ecx
```

The following is another individualized version of the original block, based on a different secret key:

```
51                 push        ecx
F7 D0              not         eax
FF 75 08           push        dword ptr [ebp+8]
58                 pop         eax
FF 70 0C           push        dword ptr [eax+0Ch]
59                 pop         ecx
81 F1 51 04 00 15  xor         ecx,15000451h
81 F1 22 AA 28 2A  xor         ecx,2A28AA22h
51                 push        ecx
8F 45 FC           pop         dword ptr [ebp-4]
FF 75 FC           push        dword ptr [ebp-4]
58                 pop         eax
42                 inc         edx
F7 D2              not         edx
42                 inc         edx
4A                 dec         edx
33 D2              xor         edx,edx
51                 push        ecx
```

```
59                  pop         ecx
6A 07               push        7
59                  pop         ecx
```

We then patched the original block by changing a few critical instructions:

```
51                  push        ecx
8B 45 08            mov         eax,dword ptr [ebp+8]
8B 48 10            mov         ecx,dword ptr [eax+10h]
81 F1 53 5D 3E 2C   xor         ecx,2C3E5D53h
89 4D FC            mov         dword ptr [ebp-4],ecx
8B 45 FC            mov         eax,dword ptr [ebp-4]
83 C0 64            add         eax,64h
83 C8 01            or          eax,1
33 D2               xor         edx,edx
B9 07 00 00 00      mov         ecx,7
```

It is not difficult to see which instructions were changed by simply comparing the original and the patched block. By applying superdiversification to the patched block, we can again generate many different equivalent blocks with more differences between them and the original than just the patched instructions, making it harder to isolate the patched code. One example of a generated block is shown below:

```
FF 75 08            push        dword ptr [ebp+8]
58                  pop         eax
51                  push        ecx
6A 01               push        1
59                  pop         ecx
83 E1 02            and         ecx,2
F7 D9               neg         ecx
F7 D9               neg         ecx
03 48 10            add         ecx,dword ptr [eax+10h]
41                  inc         ecx
49                  dec         ecx
81 F1 53 5D 3E 2C   xor         ecx,2C3E5D53h
8B C1               mov         eax,ecx
50                  push        eax
58                  pop         eax
50                  push        eax
8F 45 FC            pop         dword ptr [ebp-4]
83 F0 01            xor         eax,1
0B C0               or          eax,eax
83 F0 01            xor         eax,1
3B C0               cmp         eax,eax
3B C0               cmp         eax,eax
83 C0 64            add         eax,64h
```

```
48                  dec         eax
40                  inc         eax
83 C8 01            or          eax,1
3B C9               cmp         ecx,ecx
3B C9               cmp         ecx,ecx
3B C9               cmp         ecx,ecx
6A 01               push        1
5A                  pop         edx
83 E2 02            and         edx,2
6A 07               push        7
59                  pop         ecx
```

Below is another individualized version of the above block:

```
51                  push        ecx
FF 75 08            push        dword ptr [ebp+8]
58                  pop         eax
6A 01               push        1
59                  pop         ecx
FF 70 10            push        dword ptr [eax+10h]
59                  pop         ecx
81 F1 51 55 14 04   xor         ecx,4145551h
81 F1 02 08 2A 28   xor         ecx,282A0802h
6A 01               push        1
8F 45 FC            pop         dword ptr [ebp-4]
51                  push        ecx
8F 45 FC            pop         dword ptr [ebp-4]
FF 75 FC            push        dword ptr [ebp-4]
58                  pop         eax
F7 D8               neg         eax
F7 D8               neg         eax
0B C0               or          eax,eax
83 F0 01            xor         eax,1
83 F0 01            xor         eax,1
83 C0 44            add         eax,44h
83 C0 20            add         eax,20h
83 C8 01            or          eax,1
F7 DA               neg         edx
F7 DA               neg         edx
F7 DA               neg         edx
0B D2               or          edx,edx
42                  inc         edx
4A                  dec         edx
83 CA 01            or          edx,1
42                  inc         edx
81 CA 9E 1E 00 00   or          edx,1E9Eh
```

```
81 F2 9E 1E 00 00  xor         edx,1E9Eh
83 E2 03           and         edx,3
6A 07              push        7
59                 pop         ecx
```

Methods Against Graph-Diffing. As part of a comprehensive strategy against patch diffing, superdiversification addresses only one aspect of protection, namely the task of making basic blocks and small code sections appear different to data-based comparisons. We describe two additional techniques that help against graph-oriented diffing as well. These modify the patched binary's CFG by adding nodes and edges:

Code Outlining: Adding Nodes This technique moves sections of code into newly created functions, replacing the sections with calls to the functions. Such a process is used by compilers for certain optimizations, and has also found applications in software protection [17]. In the context of this paper, the new functions introduced by outlining serve as new nodes in the patched binary's CFG. We can iterate this procedure to outline code from already outlined functions, creating new patterns of edges and nodes in the CFG.

Chaff Control Flow: Adding Edges This method injects new control-flow transfers, such as branches, jumps, and calls, into the patched binary's CFG. To avoid interfering with the program's operation, the new transfers may never be executed, but should be protected via opaque predicates [11]. Such predicates themselves insert additional edges into the CFG.

In principle, these two techniques in combination suffice to convert the patched binary's CFG into a more complex CFG of arbitrary structure. The original CFG remains embedded in the new CFG. In a reasonable attack model, a graph-diffing tool may be required to find the original CFG in the new CFG. In the worst case, this reduces to solving the subgraph-isomorphism problem (NP-Complete), which is expensive for large graphs.

For this model to be useful in practice, certain implementation assumptions must be satisfied; for example, the new chaff edges and outlined functions should not be easily discernible. The design needs to consider the particular subgraph-isomorphism instances created for real-life programs, mainly to ensure that these are not easily solvable on average. In addition, both security and performance penalties will depend on the degree of outlining and chaff-edge insertion. Nonetheless, the approach takes initial steps towards a formal solution and avoids the need for ad hoc methodologies.

When used alone, neither superdiversification nor the above graph anti-diffing methodology suffices in general. For example, even if the CFGs of two binaries are very different, it may be possible to match up basic blocks by inspecting their contents. This is where superdiversification comes in: The numbers and types of instructions in basic blocks can be individualized to prevent easy block matching. Thus, superdiversification and graph anti-diffing techniques complement each other to create a more complete solution against patch diffing.

5 Summary and Future Work

We presented an initial version of our code individualizer, which we believe can be a useful tool for binary diversification. Another potential application is steganography, or data hiding via variably individualized code sequences [14]. Our technique also provides one possible practical implementation of code individualization assumed by certain software-protection models [12]. Given short input sequences, the individualizer is able to generate reasonably large numbers of equivalent sequences with a variety of different characteristics. Due to the exponentially large search space, performance of the tool is limited to generating sequences of only a few instructions in length over a small subset of opcodes. Further research into better search strategies and other optimizations (e.g., adaptive pruning based on input-sequence heuristics) will be undertaken to enable longer sequences and a larger set of instructions to be considered.

An important extension for superdiversification is search over arbitrary instruction sets and byte-codes, not just instruction sets of commodity processors. Modern instruction sets, including x86, x64, and MSIL, are geared towards generality and performance, not individualization and obfuscation. On the other hand, superdiversification is free to use arbitrary custom byte codes designed specifically for randomization and protection. We may generate such byte-codes explicitly, allowing them to contain randomized operations (e.g., a single instruction to multiply by 3, rotate right by 2 places, and add 1). The nature of byte-code instructions we allow is heuristically determined to increase the chances of efficient and successful searches. For example, when superobfuscating a particular input function, we may vary the allowed instruction set based on the operations contained in that function. A custom byte-code compiler may transform our superobfuscated functions into mainstream source or native code.

References

1. Anckaert, B., Jakubowski, M., Venkatesan, R.: Proteus: Virtualization for diversified tamper-resistance. In: DRM 2006: Proceedings of the ACM Workshop on Digital Rights Management, pp. 47–58. ACM Press, New York (2006), doi:10.1145/1179509.1179521
2. Anckaert, B., De Sutter, B., De Bosschere, K.: Software piracy prevention through diversity. In: DRM 2004: Proceedings of the 4th ACM Workshop on Digital Rights Management, pp. 63–71. ACM Press, New York (2004)
3. Aucsmith, D.: Tamper resistant software: An implementation. In: Anderson, R. (ed.) IH 1996. LNCS, vol. 1174, pp. 317–333. Springer, Heidelberg (1996)
4. Bansal, S., Aiken, A.: Automatic generation of peephole superoptimizers. In: ASPLOS-XII: Proceedings of the 12th International Xonference on Architectural Support for Programming Languages and Operating Systems, pp. 394–403. ACM Press, New York (2006), doi:10.1145/1168857.1168906
5. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, Ke.: On the (im)possibility of obfuscating programs. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 1–18. Springer, Heidelberg (2001)

6. Brumley, D., Poosankam, P., Song, D., Zheng, J.: Automatic patch-based exploit generation is possible: Techniques and implications. In: Proceedings of the 2008 IEEE Security and Privacy Symposium (2008)
7. Chen, Y., Venkatesan, R., Cary, M., Pang, R., Sinha, S., Jakubowski, M.H.: Oblivious hashing: A stealthy software integrity verification primitive. In: Information Hiding (2002)
8. Cohen, F.: Operating system protection through program evolution (1992), `http://all.net/books/IP/evolve.html`
9. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, The University of Auckland, New Zealand (July 1997)
10. Collberg, C., Thomborson, C., Low, D.: Breaking abstractions and unstructuring data structures. In: International Conference on Computer Languages, pp. 28–38 (1998)
11. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Principles of Programming Languages, POPL 1998, pp. 184–196 (1998)
12. Dedic, N., Jakubowski, M.H., Venkatesan, R.: A graph game model for software tamper protection. In: 2007 Information Hiding Workshop (2007)
13. eEye Digital Security. eEye Binary Diffing Suite (2007), `http://research.eeye.com`
14. El-khalil, R., Keromytis, A.D.: Hydan: Hiding information in program binaries. In: López, J., Qing, S., Okamoto, E. (eds.) ICICS 2004. LNCS, vol. 3269, pp. 187–199. Springer, Heidelberg (2004)
15. Geer, D., Bace, R., Gutmann, P., Pfleeger, C.P., Quarterman, J.S., Schneier, B.: CyberInsecurity: The cost of monopoly–how the dominance of Microsoft's products poses a risk to security (2003), `http://www.ccianet.org/paperscyberinsecurity.pdf`
16. Goldwasser, S., Kalai, Y.T.: On the impossibility of obfuscation with auxiliary input. In: Proceedings of the 46th IEEE Symposium on Foundations of Computer Science (FOCS 2005) (2005)
17. Jacob, M., Jakubowski, M.H., Venkatesan, R.: Towards integral binary execution: Implementing oblivious hashing using overlapped instruction encodings. In: 2007 ACM Multimedia and Security Workshop, Dallas, TX (2007)
18. Jakubowski, M.H., Venkatesan, R.: Protecting digital goods using oblivious checking, US Patent No. 7,080,257, filed on August 30, 2000, granted on July 18 (2006)
19. Joshi, R., Nelson, G., Randall, K.: Denali: a goal-directed superoptimizer. In: PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, pp. 304–314. ACM Press, New York (2002)
20. Lynn, B., Prabhakaran, M., Sahai, A.: Positive results and techniques for obfuscation. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 20–39. Springer, Heidelberg (2004)
21. Massalin, H.: Superoptimizer: A look at the smallest program. In: ASPLOS-II: Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems, pp. 122–126. IEEE Computer Society Press, Los Alamitos (1987)
22. The Metasploit Project. Metasploit, `http://www.metasploit.com`
23. SABRE Security and Zynamics. Using SABRE BinDiff for malware analysis (2007), `http://www.sabresecurity.com/files/BinDiff_Malware.pdf`
24. Tan, G., Chen, Y., Jakubowski, M.H.: Delayed and controlled failures in tamper-resistant software. In: Proceedings of the 2006 Information Hiding Workshop (2006)

25. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Studies in Constructive Mathematics and Mathematical Logic, pp. 115–125 (1968)
26. Princeton University. zChaff, `http://www.princeton.edu/~chaff/zchaff.html`
27. Wang, C., Hill, J., Knight, J., Davidson, J.: Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia (December 2000)
28. Wee, H.: On obfuscating point functions. In: STOC 2005: Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing, pp. 523–532. ACM Press, New York (2005)
29. Wikipedia. Metamorphic code, `http://en.wikipedia.org`