Binary code obfuscation through C++ template metaprogramming

Samuel Neves and Filipe Araujo

CISUC, Department of Informatics Engineering
University of Coimbra, Portugal
{sneves,filipius}@dei.uc.pt

Abstract. Defending programs against illegitimate use and tampering has become both a field of study and a large industry. Code obfuscation is one of several strategies to stop, or slow down, malicious attackers from gaining knowledge about the internal workings of a program.
Binary code obfuscation tools often come in two (sometimes overlapping) flavors. On the one hand there are "binary protectors", tools outside of the development chain that translate a compiled binary into another, less intelligible one. On the other hand there are software development kits that require a significant effort from the developer to ensure the program is adequately obfuscated.
In this paper, we present obfuscation methods that are easily integrated into the development chain of C++ programs, by using the compiler itself to perform the obfuscated code generation. This is accomplished by using advanced C++ techniques, such as operator overloading, template metaprogramming, expression templates, and more. We achieve obfuscated code featuring randomization, opaque predicates and data masking. We evaluate our obfuscating transformations in terms of potency, resilience, stealth, and cost.

1 Introduction

Today, the Internet is the major channel for software distribution. However, software available online may not only attracts legitimate users, but malicious agents as well (e.g., pirates, competitors, etc). To fight these agents vendors may resort to diverse mechanisms including legal suites, and technical measures. Technical measures consist of transformations that render the code harder or (ideally) impossible to reverse engineer. One particularly effective measure is to move critical areas of code out of the reach of the attacker, by e.g. offloading execution to a remote server. It is not, however, always possible to offload critical bits of an application elsewhere, leading to alternative "white-box" techniques to thwart analysis.

Obfuscation has been often used as such an anti-reverse engineering measure [10]. Formally, an obfuscating transform $\mathcal{O}$ is a function that takes in a program P and outputs a program $\mathcal{O}(P)$ with the same semantics, but somehow "harder" to understand. Although general obfuscators cannot exist [5], secure

obfuscators can exist for several important assumptions and classes of functions [6,20]. One widespread technique, used to thwart static analysis[1] of the output is to compress and encrypt a binary ("packing"), and append a decryption stub that when executed, rebuilds the original code in memory. Many variations of this idea have been developed and commercialized, but the rise of automated "unpackers" has forced the defensive side to use more aggressive means.

Current protections employ advanced code transformation methods that irreversibly mutate the original code into a hard-to-understand representation. One popular technique, "virtual machines" [32,23], converts actual binary code into custom generated bytecode, interpreted at runtime. The original code is lost, and an attacker is forced to reverse engineer the interpreter to understand the bytecode. This is a tedious and mostly manual process (cf. [26]), which raises the cost of reverse engineering. Code obfuscation may also have alternative positive applications. It may be used to increase code diversity at little cost, lowering the effectiveness of certain exploit techniques, cf. [24].

Obfuscators often come in two, sometimes overlapping, flavors: binary obfuscators, which transform the executable directly oblivious to the higher-level structures of the code, and source-based protectors, which rely on certain manually injected API calls and functions, that yield a protected binary. Both approaches have important disadvantages: binary obfuscation leaves significant trace in the binary code, whereas source-based protectors require considerable intervention. In this paper we strive for minimal developer intervention, while also trying to minimize possible mismatches between the original and the obfuscated program. To do this we resort to operator overloading and template metaprogramming of the ubiquitous C++ language.

Our work yields an obfuscator that makes sole use of the C++ compiler, and generates randomized obfuscated code using standard techniques, such as opaque predicates and dead code generation, and also our own code expansion technique. The results show that, albeit heavy on the compiler, our obfuscating transforms increase the difficulty of reverse-engineering a program.

Section 2 describes the very basics of the feature we abuse to achieve our goal, C++ template metaprogramming. Section 3 explains how to achieve randomized algorithms within template metaprograms. Section 4 describes our main obfuscating transforms that render regular into obfuscated code. Section 5 also describes how to obfuscate data, data being both the values one is working on, and static data such as integer constants and strings that may help an attacker. We evaluate our work in Section 6, and conclude with Section 7.

2 C++ template metaprogramming

[1] Static analysis refers to attempts to understand a program without running it, e.g., using a disassembler. When one is able to see the state of the program as it runs via, say, a debugger, one is performing dynamic analysis.

```
template<int N>
struct Factorial {
  static const int value = N * Factorial<N-1>::value;
};
template<>
struct Factorial<0> {
  static const int value = 1;
};
// Factorial<5>::value == 120
```

Fig. 1. C++ metaprogram capable of computing the factorial function.

It was discovered during the C++ standardization effort that it is possible to use the C++ template system to perform small computations [29]. It was later shown that C++'s template system is Turing-complete [31].

The basic mechanism by which C++ templates can be programmed is *template specialization*. The easiest way to describe it is by example. Consider Fig. 1. The value of `Factorial<N>::value` is defined recursively by instantiating `Factorial<N-1>`, and by specializing the implementation of the case $\texttt{N} = 0$, one avoids infinite recursion.

Template metaprogramming can also be used to implement higher-order functions [2]. A simple example is a function that, taking a function $f(x)$ and $g(x)$, returns $f(g(x))$. Fig. 2 illustrates this concept by creating a new function by composing two arbitrary functions passed as parameters (e.g., the `Factorial` function from Fig. 1).

```
template<template<int> class F, template<int> class G>
struct Compose {
    template<int N>
    struct apply {
        static const int value = F<G<N>::value>::value;
    };
};
template<int N>
using DoubleFactorial = Compose<Factorial, Factorial>::apply<N>;
// DoubleFactorial<3>::value == 720
```

Fig. 2. Metaprogram that returns the composition of two functions.

C++ template metaprogramming has since seen a myriad of practical uses. Expression templates [30] allow the elimination of unnecessary temporary variables by generating expression trees at compile time; entire embedded domain specific languages (EDSLs) have been devised with creative use of operator overloading and template metaprogramming [12,14]. In view of the popularity and utility of such techniques, the recent C++-11 standard [16] includes additional metaprogramming-oriented features, such as variadic templates, generalized constant expressions, static assertions, and user-defined literals.

3 Randomization

No matter how good a single transformation may be, if used repeatedly it loses its strength and becomes more prone to automated analysis [22]. The notion of "software diversity" was brought up as early as 1993 [8], as a measure of protection against malicious attacks. It is still an idea worth exploring, as shown by recent countermeasures to exploitation that rely on diversity of addresses [27] and code [24].

Performing randomization in the C++ template metaprogramming setting has difficulties similar to other pure functional languages. The first step is to define a pseudorandom generator. For simplicity, we employ a linear congruential generator, using leapfrogging [18] to split it in two when required (e.g., in recursive calls). Fig. 3 shows our simple generator.

```
template<u32 S, u32 A = 16807UL, u32 C = 0UL, u32 M = (1UL<<31)-1>
struct LinearGenerator {
  static const u32 state = ((u64)S * A + C) % M;
  static const u32 value = state;
  typedef LinearGenerator<state> next;

  struct Split { // Leapfrog
    typedef LinearGenerator<      state, A*A, 0, M> Gen1;
    typedef LinearGenerator<next::state, A*A, 0, M> Gen2;
  };
};
```

Fig. 3. Linear congruential generator to use with template metaprograms.

Code that receives a `LinearGenerator` as template argument can then use its `next` member as an argument to a subsequent recursive call; recursive calls with multiple arguments must split the generator to avoid repeating values in different paths.

To ensure that different expressions have different seeds, one can use the standard C macro `__LINE__` coupled with the common `__COUNTER__` macro as seed to the random generator. This, however, does not result in different results for different compilations of the same program. There appears to be no other alternative than to define a macro in the build script immediately before compilation (`__RAND__` in our case).

4 Obfuscating integer arithmetic

Integer arithmetic is, by and large, the most common operation that does not relate to control flow in most binaries [4]. In this section, we describe how we obfuscate integer operations by recursively dividing them into sequences of simpler operations that preferably are not algebraically compatible.

Our transformations are quite simple, and exploit several identities from boolean logic and two's complement integer arithmetic [7,33,3,17], such as:

$$a + b = (a \oplus b) + 2(a \wedge b);$$
$$a - b = (a \oplus b) - 2(\neg a \wedge b);$$
$$a \oplus b = a + b - 2(a \wedge b);$$
$$a \vee b = (a \oplus b) + (a \wedge b).$$

Note that one can combine many of these identities together to obtain a more complex expression, which is not easily reduced using, say, constant folding or other standard compiler optimization methods. Certain recursions create cycles when used in certain orders, and such cycles must be broken by either recursion depth limits or randomization. Fig. 4 shows a single basic code transformation, which significantly increases the code length for an integer addition, rendering the binary output harder to understand. We can use the above identities to implement multiplication, division, and the remaining integer instructions, building up from basic logic and arithmetic operations.

```
template<typename T, typename RandGen>
struct ObfuscatedAdder {
  enum Adder_Method  {
    Adder_Method_0,
    // ...
    Adder_Method_Count
  };
  template<typename Gen, u32 MutationIndex> struct Mutation;

  template<typename Gen>
  struct Mutation<Gen, Adder_Method_0> {
    template<u32 N, typename D>
    struct Remaining {
      static inline T eval(const T a, const T b) {
        typedef Mutation<typename Gen::next, Gen::value%Adder_Method_Count> NextMutation;
        return NextMutation::template Remaining<N-1, D>::eval(a^b, 2*(a&b));
      }
    };

    template<typename D>
    struct Remaining<0, D> {
      static inline T eval(const T a, const T b) {
        return a ^ b; // no more carries to propagate, just xor
      }
    };
  };
  // ...
  static inline T eval(const T a, const T b) {
    typedef typename Mutation<RandGen::next, RandGen::value%Adder_Method_Count> M;
    return M::template Remaining<Bits<T>::value, NullType>::eval(a, b);
  }
};
```

Fig. 4. Randomized obfuscated adder generator.

Fig. 4 shows the basic structure for most implemented obfuscations. The class takes in an integer type and a random generator, and offers solely the `eval` function to the user. This function, in turn, uses the random generator to select one of many possible transformations, and jumps to its `eval` function. There,

that function can jump to any other `eval` function from any other mutation, again selected at random. The process terminates when enough operations have been performed (in this case, the number of bits of the input type).

To further confuse analysis tools, one can reimplement the addition of Fig 4 using MMX, SSE2, or floating-point CPU instructions, again increasing the cost of analysis of the code.

Operator overloading can be used to hide the ugly syntax, as illustrated in Fig. 5. With simple operator overloading, each operation is expanded, as shown in Fig. 5, but the *order* of operations remains the same as in the unobfuscated code. For example, the expression `a*b+c` will be evaluated as `t = a*b; t += c;`, whereas we might have wanted to execute the whole expression as a single multiply-and-add. To enable such operations, we use *expression templates* [30]. Expression templates allow us to manipulate the AST of an expression, and evaluate it any way we want. In the obfuscation context, we want to be able to change the order or combine subexpressions when possible.

```
struct uint32 {
  unsigned int x_;
  uint32(unsigned int x) : x_(x) {}
  uint32(const uint32 &oth) : x_(oth.x_) {}
  uint32 &operator=(const uint32 &oth) { x_ = oth.x_; return *this; }
  operator unsigned int() const { return x_; }
  // ...
};
// ...
static inline uint32 operator+(const uint32 &a, const uint32 &b) {
  return uint32(ObfuscatedAdder<unsigned int>::eval(a.x_, b.x_));
}
```

Fig. 5. Integer wrapper class for usage with obfuscation transformations.

4.1 Opaque predicates

One software complexity metric, due to McCabe [19], states that the more predicates ("ifs") a program has, the more complex it is. This immediately suggests a simple method to increase a program's complexity, without altering its semantics: add a large number of conditional operations within the program, whose result is known *a priori* to the obfuscator, but not the attacker. Arbitrary quantities of useless code can be inserted in the unused code paths. Such predicates are known in the literature as *opaque predicates* [11]. Some examples of (always true) opaques predicates are:

$a^2(a+1)^2 \bmod 4 = 0$;
$(a^3 - 3) \bmod 3 = 0$;
$a + b \geq a \oplus b$;
$7a^2 - 1 \neq b^2$.

```
template<typename T>
struct OpaquePredicate {
    template<size_t N> struct Predicate;

    template<size_t N>
    struct Predicate<0> {
        static inline bool always_true(const T a, const T b) {
                return (7*a*a - 1) != (b*b);
        }
    };

    static inline bool always_true(const T a, const T b) {
            return Predicate<0>::always_true(a,b);
    }
};
```

Fig. 6. Sample opaque predicate.

Fig. 6 shows the implementation of a single opaque predicate. The implementation looks needlessly complicated, but the generality is useful to implement many different variants of opaque predicates.

Combining arbitrary opaque predicates and useful code can be done fairly easily, as shown in Fig. 7. We make use of C++'s lambda functions to represent the correct code to be executed.

```
template<typename T, typename RandGen>
struct PredicatedExecute {
    template<typename Functor>
    static FORCEINLINE T eval(Functor f, const T a, const T b) {
        if(OpaquePredicate<T, RandGen>::always_true(a, b)) return f(a, b);
        else return BogusOperation<T, RandGen>::eval(a, b);
    }
};
// ...
typedef PredicatedExecute<T, LinearGenerator<__LINE__>> Doer;
Doer::eval( [](const T a, const T b) { return a + b; }, x, y);
```

Fig. 7. Template class wrapping opaque predicates, useless code, and useful code.

Dead code is generated randomly via expression trees, similarly to the methods used to construct expression templates. Trees are generated randomly up to a maximum depth, and use the same arguments of the correct predicate function.

5 Data obfuscation

It is often as important to hide the data being processed as it is to hide how it is being processed. Apart from masking the data being processed, obfuscating data also involves hiding strings, keys, and other data that may help an attacker to better understand the program. This section depicts both integer and string obfuscation.

```
template<char C, size_t I>
struct Pair {
  static const   char  first = C;
  static const size_t second = I;
};

template<template<typename> class BlockCipher, typename Key, typename T>
struct EncryptByte {
  static const u32 L = BlockCipher<Key>::block_length;
  typedef typename BlockCipher<Key>::template EncryptCtr<T::second / L> Block;
  static const char value = T::first ^ Block::template Byte<T::second % L>::value;
};

template<template<typename> class BlockCipher, typename Key, typename...T>
struct EncryptHelper {
  static const char value[sizeof...(T)];
};

template<template<typename> class BlockCipher, typename Key, typename...T>
const char EncryptHelper<BlockCipher, Key, T...>::value[sizeof...(T)] =  {
  EncryptByte<BlockCipher, Key, T>::value...
};

#define THRESHOLD 256
#define AT(L,I) (I < sizeof(L)) ? char(L[I]) : char('\0')
#define DECL(z, n, L) Pair<AT(L,n),n>,
#define ENCSTR(K0,K1,K2,K3,L) EncryptHelper<AES128, AESKey<K0,K1,K2,K3>, \
                BOOST_PP_REPEAT(THRESHOLD, DECL, L) DECL(0, THRESHOLD, L)>::value
#define SAFERSTR(L,K0,K1,K2,K3) DecryptCtr<AES128, AESKey<K0,K1,K2,K3>>\
                               (ENCSTR(K0,K1,K2,K3,L),THRESHOLD)
#define SAFESTR(L) SAFERSTR(L,__RAND__,__LINE__,__COUNTER__,0)

// std::cout << SAFESTR("AES-encrypted string") << std::endl;
```

Fig. 8. Compile-time string encryption using AES-128 in CTR mode.

5.1 Integer obfuscation

One method to represent obfuscated integers is to use alternative, less common, representations. One such representation was suggested by Zhu et al [34], relying on residue number systems (RNS). In this representation, an integer n is represented by its residue modulo a set of primes $n \bmod p_1, n \mod p_2, \ldots, n \mod p_n$, and arithmetic is performed element-wise modulo each prime.

5.2 String and constant obfuscation

Strings and integer constants often provide reverse engineers strong clues regarding the behavior of a program. Strings, in particular, often offer plenty of information of the high-level behavior of a program (e.g., error messages). We hide such data using strong encryption algorithms, namely the AES [1] (or, optionally, XTEA [21]) block cipher in counter mode.

Refer to Fig. 8. Integer constants are fairly easy to hide: given an integer, encryption is a simple matter of xoring against the block `EncryptCtr(K, IV)`. The key and IV here can be chosen in the same fashion as the seeds in Section 3.

However, string literals are harder to obfuscate, because even the most recent C++ standard does not allow for string literals as template parameters.

We can get around this by using preprocessor metaprogramming[2] and variadic templates. The downside of this approach is that more developer involvement is required, the maximum size for encryption is fixed, and every encrypted string will have this maximum size. Binaries with many small strings might see a significant jump in size due to this limitation.

6 Evaluation

Table 1. Average code size, compilation, and running time for the XTEA function.

	Code size (bytes)	Running time (CPU cycles)	Compile time (s)
Original XTEA	2012	400	2
Obfuscated XTEA	660803	54600	19
Increase/slowdown	328.4×	136.5×	9.5×

We have implemented the XTEA cipher [21], fully unrolled, and compiled obfuscated and unobfuscated versions. The compiler used was GCC 4.7.1, and the experiments were performed on an Intel Core i7 2630QM CPU. Table 1 shows the average overhead we obtained when using all the methods from Section 4[3]. While both the size and runtime of the obfuscated are quite higher than their original counterparts, both remain acceptable, provided the obfuscated code is not a bottleneck. In comparison, a simple Python XTEA implementation on the same machine runs in roughly 65000 cycles. The compilation time is significantly longer, but not unpractically so, considering the rather extense code sequence being obfuscated.

Despite the inherent difficulty in objectively evaluating obfuscating transformations, there are some metrics that can shed some light into their strength. Collberg et al. [9,10] proposed 4 general measures to classify obfuscations: potency, resilience, stealth, and cost.

Potency measures the complexity added by the obfuscation *in the eyes of a human reader*. Halstead [15] uses program size and diversity of operations as a proxy for complexity; McCabe [19] uses the complexity of the control-flow graph of the program. Our obfuscations increase both program size and control-flow graph complexity by about one or two orders of magnitude, which renders our obfuscated binaries quite unreadable for human readers.

Resilience is the resistance of the obfuscated program to automated analysis. Although difficult to assess, it is worth noting, however, that since our transformed outputs pass unscathed through an optimizing compiler, the bar for

[2] For brevity, we make use of the Boost Preprocessor library to implement repetition constructs.

[3] Maximum tree depth used for random dead code expressions: 5

generic deobfuscation is quite higher — most deobfuscators simply use compiler-like optimization passes to improve readability [13,25,28] — therefore special-purpose deobfuscators are required, increasing the cost for the attacker.

Stealth is the property of an obfuscation that is hard to spot by an attacker, human or otherwise [11]. Binary-level obfuscations that insert large quantities of useless code are often quite easy to spot. Our transformations, on the other hand, look very much like regular compiled code, because this is exactly what they are.

The cost metric represents the added cost, in terms of resources needed, to execute the obfuscated program. Since our transformations are a constant factor longer than the original (or, if inside a loop, a linear one in terms of execution cost), our transformations rank as "free" or "cheap" in Collberg's taxonomy [9].

7 Conclusions and Discussion

We have presented certain techniques and transformations, using C++ template metaprogramming and operator overloading, that enable the generation of obfuscated code from the original, clean, syntax tree. The idea of generating code at compile time in C++ is not new; metaprogramming has been used to build parsers [14], implement languages [12], and much more.

The idea of using the compiler itself to perform binary obfuscation, however, seems to be new. We have shown that not only it is feasible, but our methods are actually practical, provided that the amount of code to obfuscated is not prohibitively high. Further work will go into finding other, more complex, obfuscations and predicates, and also to try to break the statement barrier of expression templates.

Nevertheless and despite our previous results showing that compile-time obfuscation is feasible, there are some limitations inherent to C++: the obfuscation scope is quite limited and restricted to a single statement; only a few types that can be used as template parameters — e.g., floating-point types are not supported; finally compilation time might be a problem for large-scale projects. It may be advisable to obfuscate only small, critical pieces of code that need to be hidden.

Acknowledgments

This work has been supported by the project CMU-PT/RNQ/0015/2009, TRONE — Trustworthy and Resilient Operations in a Network Environment.

References

1. Federal information processing standards publication (FIPS 197). Advanced Encryption Standard (AES) (2001), `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`

2. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series). Addison-Wesley Professional (2004)
3. Arndt, J.: Matters Computational: Ideas, Algorithms, Source Code. Springer-Verlag New York, Inc., New York, NY, USA, 1st edn. (2010)
4. Bania, P.: Instructions frequency statistics (2011), http://piotrbania.com/all/articles/instr_stats.html
5. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im)possibility of obfuscating programs. J. ACM 59(2), 6:1–6:48 (May 2012), http://doi.acm.org/10.1145/2160158.2160159
6. Beaucamps, P., Filiol, E.: On the possibility of practically obfuscating programs towards a unified perspective of code protection. Journal in Computer Virology 3(1), 3–21 (2007)
7. Beeler, M., Gosper, R.W., Schroeppel, R.: HAKMEM. Tech. rep., Cambridge, MA, USA (1972), http://dspace.mit.edu/handle/1721.1/6086
8. Cohen, F.B.: Operating system protection through program evolution. Comput. Secur. 12(6), 565–584 (Oct 1993), http://dx.doi.org/10.1109/52.43044
9. Collberg, C., Thomborson, C., Low, D.: A Taxonomy of Obfuscating Transformations. Tech. Rep. 148, Department of Computer Science University of Auckland (July 1997), https://researchspace.auckland.ac.nz/handle/2292/3491
10. Collberg, C.S., Thomborson, C.D.: Watermarking, Tamper-Proofing, and Obfuscation-Tools for Software Protection. IEEE Trans. Software Eng. 28(8), 735–746 (2002)
11. Collberg, C.S., Thomborson, C.D., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: MacQueen, D.B., Cardelli, L. (eds.) POPL. pp. 184–196. ACM (1998)
12. Gil, J., Lenz, K.: Simple and safe SQL queries with C++ templates. Sci. Comput. Program. 75(7), 573–595 (2010)
13. Guillot, Y., Gazet, A.: Automatic binary deobfuscation. Journal in Computer Virology 6(3), 261–276 (2010)
14. de Guzman, J., Nuffer, D.: The Spirit Parser Library: Inline Parsing in C++. Dr. Dobb's Journal (September 2003), http://www.drdobbs.com/184401692
15. Halstead, M.H.: Elements of Software Science (Operating and programming systems series). Elsevier Science Inc., New York, NY, USA (1977)
16. ISO: ISO/IEC 14882:2011 Information technology — Programming languages — C++. International Organization for Standardization, Geneva, Switzerland (Feb 2012)
17. Knuth, D.E.: The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1. Addison-Wesley, Upper Saddle River, New Jersey, 1st edn. (2011)
18. L'Ecuyer, P.: Uniform Random Number Generation. Annals of Operations Research 53, 77–120 (1994), http://www.iro.umontreal.ca/~lecuyer/myftp/papers/tutaor.ps
19. McCabe, T.J.: A complexity measure. In: Proceedings of the 2nd international conference on Software engineering. pp. 407–. ICSE '76, IEEE Computer Society Press, Los Alamitos, CA, USA (1976), http://dl.acm.org/citation.cfm?id=800253.807712
20. Narayanan, S., Raghunathan, A., Venkatesan, R.: Obfuscating straight line arithmetic programs. In: Proceedings of the nineth ACM workshop on Digital rights management. pp. 47–58. DRM '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1655048.1655057

21. Needham, R.M., Wheeler, D.J.: TEA extensions. Tech. rep., University of Cambridge (Oct 1997), http://www.cix.co.uk/~klockstone/xtea.pdf
22. van Oorschot, P.C.: Revisiting Software Protection. In: Boyd, C., Mao, W. (eds.) ISC. Lecture Notes in Computer Science, vol. 2851, pp. 1–13. Springer (2003)
23. Oreans Software: Themida (2012), http://www.oreans.com/themida.php
24. Pappas, V., Polychronakis, M., Keromytis, A.D.: Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization. In: Proceedings of the 33rd IEEE Symposium on Security & Privacy (S & P). San Francisco, CA (May 2012), http://www.cs.columbia.edu/~vpappas/, To appear.
25. Raber, J., Laspe, E.: Deobfuscator: An Automated Approach to the Identification and Removal of Code Obfuscation. In: WCRE. pp. 275–276. IEEE Computer Society (2007)
26. Rolles, R.: Unpacking virtualization obfuscators. In: Proceedings of the 3rd USENIX conference on Offensive technologies. pp. 1–1. WOOT'09, USENIX Association, Berkeley, CA, USA (2009), http://dl.acm.org/citation.cfm?id=1855876.1855877
27. Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: Proceedings of the 11th ACM conference on Computer and communications security. pp. 298–307. CCS '04, ACM, New York, NY, USA (2004), http://doi.acm.org/10.1145/1030083.1030124
28. Spasojevic, B.: Using optimization algorithms for malware deobfuscation. Diploma thesis, UNIVERSITY OF ZAGREB — FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING (2010)
29. Unruh, E.: Prime number computation (1994), http://www.erwin-unruh.de/primorig.html, ANSI X3J16-94-0075/ISO WG21-462
30. Veldhuizen, T.: Expression Templates. C++ Report 7(5), 26–31 (June 1995)
31. Veldhuizen, T.L.: C++ templates are Turing-complete. Tech. rep. (2003), http://web.archive.org/web/20060208052020/http://osl.iu.edu/~tveldhui/papers/2003/turing.pdf
32. VMProtect Software: VMProtect (2012), http://vmpsoft.com/
33. Warren, H.S.: Hacker's Delight. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
34. Zhu, W., Thomborson, C., Wang, F.Y.: Applications of homomorphic functions to software obfuscation. In: Proceedings of the 2006 international conference on Intelligence and Security Informatics. pp. 152–153. WISI'06, Springer-Verlag, Berlin, Heidelberg (2006), http://dx.doi.org/10.1007/11734628_18