Publ. Math. Debrecen
Supplementum **100** (2022), 617–637
DOI: 10.5486/PMD.2022.Suppl.5

Control flow obfuscation with irreducible loops and self-modifying code

By Gregory Morse, Midya Alqaradaghi and Tamás Kozsik

Abstract. This paper considers an obfuscation scheme designed around runtime generated self-modifying code and irreducible loops, both of which are notoriously difficult to reason about. This leads to a mechanism for both source and binary code that increase resistance to static analysis as well as dynamic analysis. By making use of the fact that static analysis of self-modifying code has limits in decidability as per Rice's theorem, and that transformation of irreducible loops to reducible ones using techniques such as node-splitting or introduction of variables can have a quadratic complexity increase on the size of the control-flow graph. Our construction looks at turning an algorithm into the lowest abstraction level with multi-input logic gates, to evaluate the most generic perspective on the scheme though it is applicable to any control-flow graph. A final benefit is that although complicated and naturally inter-related, the different ideas could be applied separately.

1. Introduction

Protection mechanisms in recent times have focused on virtualization mechanisms as a form of securing software from prying eyes of reverse engineers. Some

Mathematics Subject Classification: 68N01, 68R10.
Key words and phrases: x86, x86-64, Assembly language, self-modifying code, obfuscation, software protection, control-flow graph, irreducible loops.
The research of Gregory Morse and Tamás Kozsik has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications). Midya Alqaradaghi has been supported by the Stipendium Hungaricum program.

of the more successful implementations such as VMProtect have been largely defeated [16]. Other protections focus on anti-tamper capabilities and tend to rely on encryption such as Denuvo have similarly been defeated [17].

One area of protection which has not been extensively studied is the use of self-modifying code (SMC). Introducing it into binaries is possible with some special memory access modification whether in a static binary file or in memory created on-the-fly at runtime through operating system dependent procedures and based on the processor's memory access enforcement mechanisms. We will study an approach where loop transformations obfuscate the protected code, and dynamically generated SMC provides an effective solution to allow decisions in the transformed loops.

The idea that dynamically generated SMC has a blueprint that is used to generate custom "stamped" SMC on the fly has also not been studied. We will show that a graph algorithm such as depth-first search (DFS) based identification of strongly-connected components (SCCs) and topological sorting can be implemented by a SMC control-flow graph (CFG) representing the actual graph being queried. The ideas allow for an optimization geared towards simplification of the stack and are generalizable to largely any graph algorithm. The algorithms mentioned are those that make up an efficient linear Boolean 2-satisfiability (2-SAT) instance solver [2].

As for CFG obfuscation, the hardest structure for most static analysis tools to properly understand semantically is an irreducible loop (IL), which is, informally speaking, a loop with multiple entries. Nesting of irreducible loops can cause quadratic behavior where the loop nesting forest is constructed [22], and the only resolution to structuring them into an abstract syntax tree (AST) involves introduction of variables and conditional expressions [30]. Although almost linear time algorithms exist for building loop nesting forests of irreducible loops by various strategies which combine them, translation to the AST does not necessarily allow for the same reductions as identification.

This study will take a look at specifically a case study where every control flow construct, be it a conditional or loop, will be further embedded or converted into an irreducible loop nest designed to cause quadratic behavior in identification. These loops however will be largely fictitious by using opaque predicates allowing the original looping behavior, or a simple single iteration. The 2-SAT solver will be used to determine the exit conditions for all of the loops in question, and given that it uses dynamic SMC, will be beyond the scope of any state-of-the-art static analysis tools. The function which will be protected will be a security critical function such as a white-box AES or RSA implementation. Performance

measurements and a look at how powerful tools such as IDA Pro and Ghidra disassemblers and decompilers will process them will be presented under Windows and Linux on the modern x86 and x86-64 platforms.

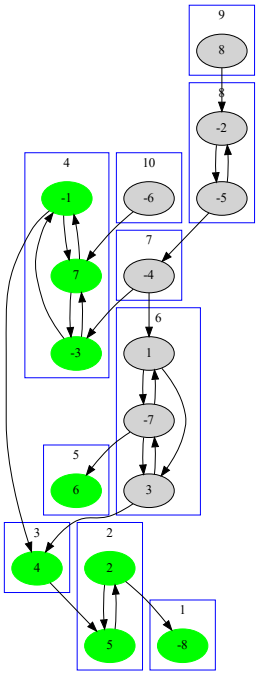
2. Background

A Boolean satisfiability (SAT) formula consists of a set of n variables $X = \{x_1, ..., x_n\}$ and a set of clauses $C = \{C_1, ..., C_n\}$ where $\forall i \in [1..n] C_i \subseteq (X \cup \{\neg x, \forall x \in X\})$ using the variables in X or their logical negation. Two-SAT (2-SAT) requires that $\forall C_i \in C, |C_i| <= 2$ restricting each clause to two variables. We consider the conjunctive normal form (CNF) which is a conjunction of clauses. Disjunctive normal form (DNF) are trivially solved but the representation has potentially $\mathcal{O}(2^n)$ clauses. Unrestricted k-SAT where there are k variables per clause is easily proven reducible to three variable per clause 3-SAT and the 3-SAT decision problem is non-deterministic polynomial-time (NP)-complete.

The 2-SAT decision problem asks whether a 2-SAT formula has a solution or not. This is a non-deterministic logarithmic space (NL)-complete problem. NL is a subset of polynomial-time (P), and it can be solved in linear time using a logical implication graph and Tarjan's strongly connected component (SCC) algorithm [2] making it economical at runtime in a protection mechanism. If a solution can be efficiently found with this method by using reverse topological sorted order, one can efficiently determine it. A clause with zero variables indicates an unsatisfiable equation. A clause with one variable has a forced value which can be replaced in all other equations or considered to be a clause with the same variable twice. Regardless it need not effect the implication graph or could be a trivial self-loop. A clause with two distinct variables $\{p, q\}$ yields two logical implications $(p \vee q) \equiv (\neg p \implies q) \wedge (\neg q \implies p)$. An example formula, its implications and visual depiction of its implication graph, SCCs and an identified solution is given in Figure 1.

We consider a CFG $G = (V, E, h)$ to be a directed graph (digraph) with V being the set of vertices, $E \subseteq \{(x, y) \forall x \in V, \forall y \in V\}$ being a set of ordered pairs of predecessor and successor vertices, and a distinguished root node h which reaches every node in V so that $\forall v \in V, h \xrightarrow{*} v$. Each vertex is associated with a sequence of code. A vertex with a single out-edge is considered to be a sequence with its successor node. A vertex with two out-edges is a Boolean conditional, otherwise it contains more than two out-edges and is an n-way conditional. A DFS which is the basis of the SCC algorithm, classifies edges in a digraph between tree edges,

forward edges, cross edges and back edges based on their ancestor relationship in the tree induced by the DFS, usually computed based on their pre-order and post-order timestamps.



(a)

$$
\begin{aligned}
&(x_1 \vee x_4) \wedge (\neg x_2 \vee x_5) \wedge \ (x_3 \vee x_7) \ \wedge \\
&(x_2 \vee \neg x_5) \wedge (\neg x_8 \vee \neg x_2) \wedge \ (x_3 \vee \neg x_1) \ \wedge \\
&(x_4 \vee \neg x_3) \wedge \ (x_5 \vee \neg x_4) \ \wedge (\neg x_3 \vee \neg x_7) \wedge \\
&(x_6 \vee x_7) \ \wedge \ \ (x_1 \vee x_7) \ \ \wedge (\neg x_7 \vee \neg x_1) \\
&\equiv \\
&(\neg x_1 \implies x_4) \ \wedge \ (\neg x_4 \implies x_1) \ \wedge \\
&(x_2 \implies x_5) \ \ \wedge (\neg x_5 \implies \neg x_2) \wedge \\
&(\neg x_3 \implies x_7) \ \wedge \ (\neg x_7 \implies x_3) \ \wedge \\
&(\neg x_2 \implies \neg x_5) \wedge \ \ (x_5 \implies x_2) \ \ \wedge \\
&(x_8 \implies \neg x_2) \ \wedge \ (x_2 \implies \neg x_8) \wedge \\
&(\neg x_3 \implies \neg x_1) \wedge \ \ (x_1 \implies x_3) \ \ \wedge \\
&(\neg x_4 \implies \neg x_3) \wedge \ \ (x_3 \implies x_4) \ \ \wedge \\
&(\neg x_5 \implies \neg x_4) \wedge \ \ (x_4 \implies x_5) \ \ \wedge \\
&(x_3 \implies \neg x_7) \ \wedge \ (x_7 \implies \neg x_3) \wedge \\
&(\neg x_6 \implies x_7) \ \wedge \ (\neg x_7 \implies x_6) \wedge \\
&(\neg x_1 \implies x_7) \ \wedge \ (\neg x_7 \implies x_1) \wedge \\
&(x_7 \implies \neg x_1) \ \wedge \ (x_1 \implies \neg x_7)
\end{aligned}
$$

(b)

Figure 1. 2-SAT implication graph with a highlighted solution (a) for a satisfiable equation (b)

A loop $h_l \circlearrowleft l$ is not defined as a cycle but involves a strongly connected region and a back edge between a header h_l and latch l. A loop nesting forest (LNF) of a CFG is a forest whose trees represent the hierarchy of the loops nesting order. A general formal characterization can be given based on strong connectivity and dominance relationship [23]. A reducible CFG contains only reducible loops. For reducible CFGs, the LNF is unique but this is not the case with irreducible LNF which can have multiple representations based on the general LNF definition. We will use the Havlak LNF [14] which is an LNF which is a representation that

has a single header-node for each loop and is convenient for source and binary translation given it is practically universal that the abstract syntax tree (AST) in any given language will have a single header per loop and is efficiently computable in near-linear time [26]. An irreducible or multi-entry loop cannot be directly represented in the CFG of many languages such as Java or Python. An AST is a tree representation of the abstract syntactic structure of a program written in a formal language. For our purposes we focus on the control-flow elements of such a tree e.g., C keywords do, while, for, if, switch, return and goto as well as operators ? :, && and ||.

The state machine model is the simplest way to represent any CFG as an AST. It effectively flattens out the CFG by representing the state, or current executing vertex as a variable. A loop containing an n-way conditional is the only requirement as well as a variable with enough state space to uniquely represent any vertex $|V|$. This simple solution also produces code that is often very difficult to understand as it has done effectively no structuring with language elements. For more background regarding CFGs and ASTs, they are used most often in practice during the compilation process' syntax translation and code generation phases [1].

Structuring algorithms can also convert a CFG to an AST by maximizing use of language elements for loops, n-way conditionals and Boolean conditionals in some priority hierarchy. However, this is only possible in languages like C which have labels and a goto statement when dealing with constructs such as irreducible loops, among others. This is commonly done in tools like DCC [10] or its later REC [24], IDA Pro and Ghidra decompilers as well as lesser known ones. We looked specifically at the most state-of-the-art ones, Hex-Rays IDA Free 7.6, a cloud-based x64 decompiler [15] and the NSA open-source Ghidra v9.2.4 [13].

Finally re-structuring is also possible, but not without code duplication [27] or the introduction of state variables as were used in the DREAM decompiler [31] [30], and can eliminate irreducible loops entirely by constructing a new CFG with state variables and Boolean conditionals making use of those state variables. However, given the entries into a set of nested irreducible loops may be numerous, this can lead to a quadratic number of state variables and conditionals. Such code however can be easier to understand, and easier for formal verification tools to analyze as single-entry regions of a CFG require less complicated formal semantics.

Control-flow obfuscation methods have used different methods in the past. A popular method is the Dynamic Dispatcher Model of Wang, et al. [28] and Chow, et al. [9]. Another method which has similarity to our proposed method

uses opaque predicates [20]. More recently an obfuscation method using control-flow flattening was introduced [19] and another by hiding the control flow information in the stack [3] which was further extended to use SMC [4]. There is also a known method which obfuscates control flow at the function level [5]. Our contribution is the use of dynamically generated SMC to generate opaque predicates, and the use of irreducible loops to increase complexity.

3. SMC 2-SAT

Rather than considering the 2-SAT solving algorithm and graph separately, we can integrate them so that the vertices and edges of the implication graph become vertices and edges of a CFG. The code associated with each vertex is based on the SCC algorithm and a reverse topological sort. Rather than going into too much detail, we describe the transformation process as the pseudo-code is particularly difficult to understand without assembly language or a SMC meta-language. We outline a scheme which has been implemented in x86 and x86-64 assembly on Windows and Linux. The normal execution stack is replaced by a stack representing the return address into the stack of nodes for the graph algorithm. When a node starts processing, its own first instruction is changed to a return instruction. When a node completes processing, it marks an address to its topological sorted solution check in the prior topological sort node code. For convenience and to allow reuse, callback functions specified by lambda (anonymous) functions can be used to enumerate edges. Another lambda callback can be used as each solution variable is identified, to allow a data structure like a set to be used if checking for unsatisfiability.

To make this scheme generalized for reusability, we consider two processes, one for runtime initialization of the solver and the other for its execution as seen in Figure 2.

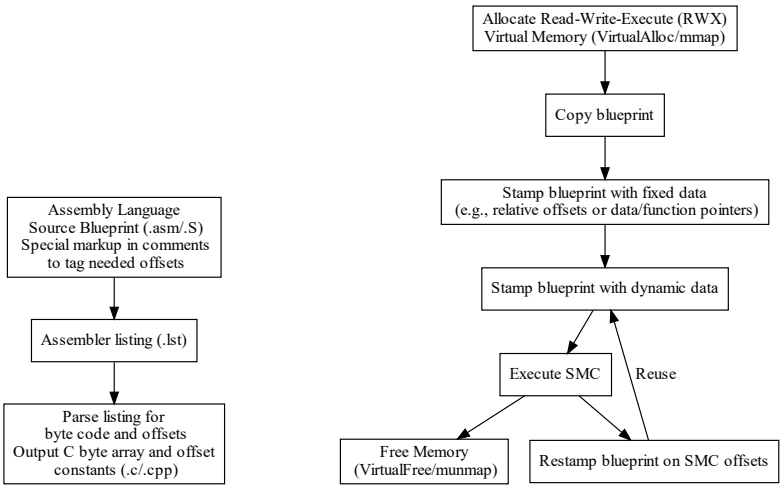


Figure 2. A method for fast generation of a blueprint (a) and blueprint stamping (b)

3.1. Generating 2-SAT formulae. A 2-SAT formula can be generated from a solution set of Boolean variables which have been pre-assigned certain literals from $\{T, F\}$. A procedure to do this merely needs to construct a tree of logical implications based on the desired values of the variables, and the choices for implications can be made randomly. Any remaining variables and their negations are individual trees in the forest. These create a set of implications required so that the solution of the formula yields the correct desired variable values. The next stage can generate a set of any number of optional variables that are not necessary and have a free choice of value, and which with their negations are added as trees to the forest. Finally, implications can randomly be added to the forest such that no loops are created which would imply unsatisfiability.

4. Converting (Cryptographic) Algorithms to CFG

The transformation can largely take the algorithm down to its description in Boolean logic as the hardware would implement it. This lower-level description

is more flexible though can result in a very large CFG given it does not benefit from practical reductions like machine word sizes. Yet the model is flexible, so abstractions of higher-level functions or machine word sizes can be reintroduced. The fundamental operations will be generalizations of Boolean logic multiple input combinators: conjunction (AND), disjunction (OR), exclusive disjunction (XOR) and its complement (XNOR).

Definition 1.

$$\mathrm{AND(X)} = \bigwedge_{\mathrm{x}\in\mathrm{X}} \mathrm{x}, \quad \mathrm{OR(X)} = \bigvee_{\mathrm{x}\in\mathrm{X}} \mathrm{x}, \quad \mathrm{XOR(X)} = \bigoplus_{\mathrm{x}\in\mathrm{X}} \mathrm{x}, \quad \mathrm{XNOR(X)} = \neg \bigoplus_{\mathrm{x}\in\mathrm{X}} \mathrm{x}.$$

Such a multi-input representation can be readily decomposed into two-input combinators requiring one less than the number of inputs, either sequentially or in a tree for maximal parallelization. Although the complements of conjunction and disjunction (NAND and NOR) are both sufficient on their own to cover all Boolean logic, having functional completeness, to reduce the size of the graph and make it more intuitive, such a set can be chosen. XNOR (or NAND or NOR) can provide logical negation when applied to a single input. Now we can define basic common operations that will be used in many algorithms.

The mutually exclusive selection (SEL) operation can be implemented as a sum of products where X is a set of ordered pairs with the selector s and value x such that one and only one pair can have a true selector.

Definition 2. $\mathrm{SEL}(X) = \bigoplus_{s,x\in X} s \wedge x.$

Notice in this context that XOR is replaceable by OR. Binary selection is a special case represented by $\mathrm{BSEL(s, x_0, x_1)} = \mathrm{s} \wedge \mathrm{x_0} \oplus \neg\mathrm{s} \wedge \mathrm{x_1}$ or $X = \{(s, x_0), (\neg s, x_1)\}$.

A table lookup is a mapping between a fixed sized m-bit input and n-bit output as $f(\{0,1\}^m) \to \{0,1\}^n$. This can be decomposed into single-bit outputs as $f(\{0,1\}^m) \to \{0,1\}$. There are two generic representations in logic and using logic synthesis to construct an optimal circuit is NP-hard based on the circuit SAT problem. The first representation uses two exponentiated via bit-shifting as $2^{\{0,1\}^m}$ giving selectors which can be combined as ordered pairs with a corresponding single-bit output of $f(\{0,1\}^m)$ and the selection operation used. Bit-shifting is implemented by starting with a constant of 1 and for each bit of the argument, computing conjunctions with the number and inverse of the bit and the number and bit itself. Alternatively each possible input $\{0,1\}^m$ can be turned into a circuit using AND and negations. The trade-off is between combinators and the number of inputs e.g., nodes versus edges.

Addition can be represented by sequentially chaining the basic equations $s_i = a_i \oplus b_i \oplus c_{i+1}$ and $c_0 = 0, c_{i+1} = a_i \wedge b_i \vee c_i \wedge (a_i \oplus b_i)$. Subtraction terms its carry as borrow and is not associative so $c_{i+1} = \neg a_i \wedge b_i \vee c_i \wedge \neg(a_i \oplus b_i)$. Combining these, we can choose dynamically between these two with a variable s indicating subtraction, instead of using selection, we have $c_{i+1} = (a_i \oplus s) \wedge b_i \vee c_i \wedge (a_i \oplus b_i \oplus s)$.

In fact this simple bit of operations makes constructing many more complicated algorithms straight-forward. A key advantage is that constant-time can be maintained. This representation is particularly useful for measuring efficiency of parallel implementations, as well as determining exact constants effectively algorithmic complexity. One more technique of using reusable subgraphs to reduce the graph size will be applied. The subgraphs have a single input edge representing its activation condition, some fixed number associated with how many times its execution is repeated, along with data input and output edges. This technique can be expanded to allow for conditional execution at the cost of maintaining constant-time.

4.1. AES/Rijndael. The Advanced Encryption Standard (AES) subset of Rijndael is standardized with three bit sizes, and referred to as AES-128, AES-192 and AES-256 based on the bit size of the key. It consists of a key expansion step which takes the key and generates a key schedule used in the rounds of encryption and decryption. The encryption has four main steps of byte substitution, shifting rows, mixing columns and adding the round key with the initial and final round performing only some of these steps. As this is a symmetrical algorithm, decryption does the reverse. At the core of the security are substitution (S)-boxes used in the substitution step that provides non-linearity using an invertible affine transformation. This multiplicative inverse of the inverse of the affine transformation is used for decryption.

The AES key expansion can be done using only XOR and table lookup operations. The encryption and decryption also only require XOR and table lookups, but additionally the column mixing can be efficiently implemented using a selection operation involved in the "xtime" function as outlined in the original Rijndael proposal [12]. For our graph, we used subgraphs to reuse S-box and inverse S-box table lookups. Identifying the table lookups would at minimum require identifying the bit-shifting circuit or DNF table lookup circuits as described above.

To strengthen the scheme against lookup table attacks, note that per our decomposition to single bit outputs above, an arbitrary circuit can be formed. Although the optimal one is NP-hard, one which introduces spurious additional inputs with no effect on the output, along with finding a deeper circuit between

the 2-level circuits from putting the equation in CNF and DNF forms is not as hard. This would make the lookup tables transparent as they would not have uniformity, nor the same timing characteristics nor the same number of inputs. Common terms of different parts of these equations from different lookup tables can be merged to further add complexity to analysis with first having to do equality (XOR) checks on the inputs. Although there are many options in this regard, there performance will likely be a performance penality.

4.2. RSA. Public-key encryption algorithms like Rivest-Shamir-Adleman (RSA) cryptosystem and even the elliptical curve digitial signature algorithm (ECDSA) rely on more computationally expensive operations like multliplication and so efficiency becomes a very high practical concern. There are a number of published techniques to reduce the number of operations. RSA relies on a secure encoding such as Public Key Cryptography Standard (PKCS) for encryption/decryption and its v1.5 for signatures with deterministic and probabilistic schemes both defined. Computationally, RSA relies upon modular exponentiation. Modular exponentiation can be done most efficiently by using the exponentiation by squaring method in $\mathcal{O}(Mn + Sn)$ where M and S are the complexity of modular multiplication and modular squaring respectively, and n is the bit size of the exponent. The number of multiplications can be further reduced using precomputation and windowing. The most optimal method makes use of addition chains, however no cryptographically useful algorithm is known to generate minimal chain, and only heuristics can be employed beyond a small bit size [11]. The multiplications themselves can be done with a divide-and-conquer algorithm like Karatsuba multiplication ($\mathcal{O}(n^{\log_2 3})$) or Toom-Cook ($\mathcal{O}(n^{\log_k 2k-1})$) down to a size where the constant coefficient of the algorithm makes traditional gradeschool multiplication ($\mathcal{O}(n^2)$) faster for n-bit integers. For very large bit-sized integers, the Fast Fourier Transform (FFT)-based Schonhage-Strassen method can be used in $\mathcal{O}(n \log n \log \log n)$ [7]. As gradeschool multiplication requires a sum of multiple inputs, an efficient implementation makes use of carry-save adders constructed in a Wallace tree. Squaring operations can be optimized to be twice as fast by using Karatsuba squaring and gradeschool squaring optimizations. Since modular multiplications are required to avoid exponential growth of subcomputations, one can make use of the Montgomery multiplication (MM) method which relies on Montgomery reductions (MR) and is only as complex as multiplication with a larger constant.

To increase the parallelization of the algorithm, addition can be done in a carry-free manner using a recoded binary signed digit (RBSD) encoding where

only converting back to regular binary form requires a sequential operation [21]. There is a constant cost associated with the initial conversions and recoding which must occur after each addition. There are also several techniques to different base representations of the exponent to also reduce the cost of exponentiation including non-adjacent form (NAF) and mutual opposite form (MOF) and their windowed forms which reduce the hamming weight of the exponent. However, using such techniques or sliding windows or even addition chains will gain performance at cost of not having a constant-time implementation. The Montgomery ladder approach can be employed to give constant-time, though compared to simple windowing, it is slower.

A CFG generated without reuse of subgraph regions will quickly become too large due to the exponential growth of multiplication as the bit size gets to realistic cryptographic sizes, so reuse of squaring, multiplication, MR, MM is necessary. For our implementation, we used Montgomery form, Montgomery and Karatsuba multiplication and squaring and window sizes to minimize products. We made use of subgraphs to reuse code for Karatsuba and Montgomery multiplication and squaring as well as Montgomery reduction.

4.3. Experimental Results. For the experiments, the AES and RSA algorithms were hand coded using the primitives we described to construct a large circuit CFG. The specific optimization strategies we mentioned were designed to keep the nodes and edges in the CFG to a minimum. We share sample code for both table lookup methods in Figure 3, the pseudo-code for all the arithmetic functions such as addition, negation, Montgomery multiplication, modular exponentiation contains much more logic, and omitted for sake of brevity.

The results in Tables 1 and 2 (where Arch. is the processor architecture, C. Ref is reference C implementation, Intrin. is intrinsics, Exec. is Executed, N. is nodes, E. is Edges) are showing AES encryption with a fixed randomly generated key, and the modular exponentiation of RSA decryption with a fixed randomly generated private key. The mentioned places where subgraphing was used to condense the graph size did provide useful space reductions and the results are as expected. AES of all bit sizes although far slower than hardware and software methods, performs potentially fast enough in certain scenarios. RSA on the other hand has its worse than quadratic complexity showing that it is basically going to be completely impractical both in terms of time and space after the 256-bit size. Machine word-reductions could easily increase the potential of the representation by order of 2^{32} or 2^{64} which could enable practical applications up to RSA-4096.

```
#2*2^n nodes, 2 predecessors per node
def decoder_shift_to_virtual(shift, node, g, op):
  num = constant_to_virtual(1)
  if len(shift) == 0: return num
  for i in range(len(shift)//BITSZ):
    num = multi_to_virtual([num, not_to_virtual(
        shift[i*BITSZ:(i+1)*BITSZ], node, g, op) *
        (len(num)//BITSZ)], and_to_virtual, node, g, op) +
      multi_to_virtual([num, shift[i*BITSZ:(i+1)*BITSZ] *
        (len(num)//BITSZ)], and_to_virtual, node, g, op)
  return num
#2^n nodes, n predecessors per node
def decoder_to_virtual(shift, node, g, op):
  return [x for y in [cmp_zero_to_virtual(multi_to_virtual(
      [constant_to_virtual(i), shift], xor_to_virtual,
        node, g, op), False, node, g, op)
    for i in range(1 << ((len(shift)//BITSZ)))] for x in y]
```

Figure 3. Python code representing two methods of virtualized translation to CFG for one-hot encoding, the primary operation of table lookup functionality. Here **BITSZ=1** for typical single bit Boolean representation, **BITSZ=2** for positive and negative bit representations as separate inputs/outputs, **multi_to_virtual** streams multiple bits to a single bit operation function, **not_to_virtual**, **and_to_virtual**, —**xor_to_virtual**— are logical NOT, AND and XOR operations, **constant_to_virtual** converts a numeric constant to its bit sequence using logical operations, **cmp_zero_to_virtual** does a comparison with zero using a multiple input XOR, **node** represents the next node counter, **shift** is a bit vector representing the shift distance, **g** is an adjacency list of the CFG and **op** is a mapping for node v where $v \to o, o \in \{\text{NOT}, \text{AND}, \text{OR}, \text{XOR or XNOR}\}$

AES	Arch.	C. Ref.	Intrin.	CFG	Nodes	Edges	Exec. N	Exec. E
128	x86	61μs	5μs	0.58s	51127	222683	134455	386203
	x86-64	49μs	3μs	0.65s				
192	x86	72μs	5μs	0.74s	61364	267320	161489	463381
	x86-64	58μs	4μs	0.82s				
256	x86	83μs	6μs	0.91s	71568	311924	188438	541018
	x86-64	67μs	4μs	0.89s				

Table 1. AES (1000 operations in succession)

RSA	Arch.	C gmp	CFG	Nodes	Edges	Exec. N	Exec. E
32	x86	851ms	4.90s	8882	16591	686447	1321378
	x86-64	707ms	5.03s				
64	x86	2.207ms	32.12s	15743	29925	4538657	8706840
	x86-64	1.015ms	33.06s				
128	x86	9.141ms	196.69s	34604	67583	27372136	52186545
	x86-64	7.640ms	201.02s				
256	x86	9.141ms	1171.36s	88121	180473	167967539	320612875
	x86-64	7.640ms	1228.89s				

Table 2. RSA/modular exponentiation (1000 operations in succession)

5. Irreducible loops

Irreducible loops are less intuitive than reducible loops and their rarity in C, a language whose syntax allows them is exceedingly rare in real-world applications [25]. When a process is being repeated, it is natural to start from its beginning, rather than conditionally start somewhere in the middle. Furthermore by embedding a set of conditions, static analysis will be unable to proceed with any reasonable guarantees per Rice's theorem as decidability prevents a static analyzer from having a set of conditions to even continuing scanning self-modifying code or what could be unreachable code [18]. There are some details such as the permissions of the code segment being loaded with read-only permission giving possible guarantees that no modifications occur to the code, though this is operating system dependent. Regardless, potentially unreachable code could be crafted such that it is not intended to be executed and deliberately hostile to static analysis, attempting to cause a static analyzer to run out of memory or get consumed

by gratuitious computations or have invalid, illegal or other instructions that are not representable in high-level languages.

In Figure 4, a simple example that highlights the different LNF definitions [23] is shown with its jungle (the depth-first spanning tree and original edges) and LNF. We compiled C++ code representing this with both an unstructured or hybrid structured representation versus a structured goto-free representation. The results on decompilation are given in Table 3. As our abstraction used C++ lambda functions for convenience and realism, some function pointer safety checks caused additional unstructured code.

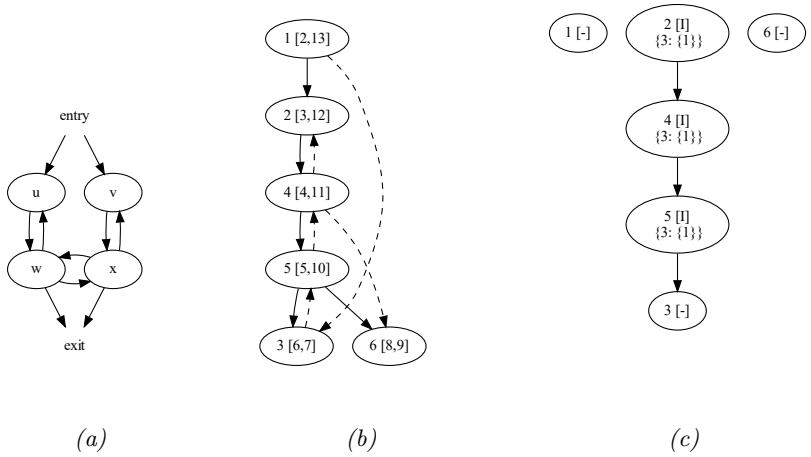


Figure 4. Simple Irreducible Loop CFG (a) with its Jungle (b) and Havlak LNF (c) (entry=1, u=2, v=3, w=4, x=5, exit=6)

Tool	Style	gotos	Loops	B. Conds.	S. Conds.	S. gotos
IDA Free	Unstructured C	6	2	11	6	3
IDA Free	Structured C	7	1	10	5	4
Ghidra	Unstructured C	8	2	11	6	5
Ghidra	Structured C	7	2	13	6	3

Table 3. Comparison of Irreducible Loop output in IDA Free and Ghidra (B. is Boolean, S. is safety and Cond. is conditional)

To appreciate the quadratic complexity of representing irreducible loops with structured programing, it is interesting to look at the worst case with a complete digraph $K_n = (V_{K_n}, E_{K_n}), |E_{K_n}| = n^2 - n$ when $n > 2$ which regardless of root

node choice is always a CFG. We can optionally assume that only the root node is connected to a special distinguished return or exit node if prefering to avoid the outermost infinite loop. We can augment with or ignore introduction of self-loops as they are trivial cases, though notably they induce post-test loops. In the Havlak LNF, there will be a single branch tree of loops with $n-1$ loop headers, and only the last node traversed in the DFS will be a non-header or self-loop. The entry node will head a reducible loop as no node can enter it which is not contained within it. Therefore a CFG can have a maximum of $n-2$ irreducible loops and requiring a triangular number sequence-based amount of multi-entry edges (MEE) or gotos to be represented without CFG modification.

Definition 3. $|\text{MEE}(K_n)| = \sum_{i=0..n-2} i = \frac{(n-1)(n-2)}{2}$

As K_n has a quadratic number of edges, we can also reduce the graph by removing extra back edges whose successor is not the inner-most non-header node, giving a DFS with $n-1$ tree edges, $n-1$ back edges plus all the multi-entry edges so that a graph $M_n = (V_{M_n}, E_{M_n}), V_{M_n} = v_i \forall i \in [1..n]$ will have maximized multi-entry edges while minimizing all others. We can formalize such a construction.

Definition 4. $E_{M_n} = \{(v_i, v_{i+1}) \forall i \in [1..n-1]\} \cup \{(v_n, v_i) \forall i \in [1..n-1]\} \cup \{(v_i, v_j) \forall i \in [1..n-2] \forall j \in [i+2..n]\}$,
$|E_{M_n}| = \frac{(n-1)(n-2)}{2} + 2(n-1) = \frac{(n-1)(n+2)}{2}, |\text{MEE}(M_n)| = |\text{MEE}(K_n)|$

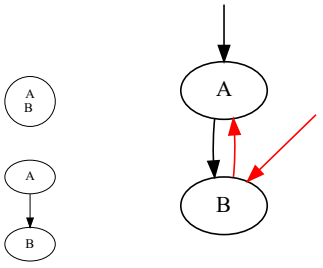
The DCC and its later REC decompiler algorithm uses the derived sequences of graphs based on intervals [10] or parenthesis theory. It does not align with this analysis perfectly as it makes practical assumptions that loop headers and/or the final conditional in a loop, termed a latch have Boolean conditionals which makes it not properly generalized. However, some modifications can be made to correct this oversight by node-splitting into a Boolean conditional and remaining conditional without code duplication or using n-way conditionals and gotos. Its results will therefore most accurately reflect the initial CFG.

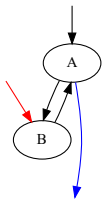
Removing irreducible loops via the node splitting technique [27] eliminates the multiple entries into a loop by effectively "splitting" which actually involves duplication of code. However, by processing a set of two other reduction rules in a priority order, it attempts to minimize the number of node splits. However this optimization will still require a quadratic amount of code duplication for each multi-entry edge where the base reduction cases of a single node and two-nodes can never have an irreducible loop.

The DREAM technique handles multiple entry and multiple exit regions by introducing state variables and related conditions. Every entry into a loop via

other than its header node requires a variable and conditionals for every node on a path between the loop header and the given entry node. The number of integer variables introduced is the number of irreducible loops $n-2$. The number of conditions (and variables if Booleans rather than integers were used) however will be based on the number of multi-entry edges. Although conditions can be combined into $n-2$ compound conditions, their presence is nevertheless required and compound conditions if based on short-circuit evaluation are still a series of Boolean conditionals.

So the idea to use irreducible loops, to add edges at the cost of quadratic behavior for eliminating gotos presents a new challenge to static analysis. We will refer to two schemes, the first a static compile-time scheme, and the second which is merely proposed would be dynamic and run-time based. Consider a simple sequence of statements. The minimal edges required to induce an irreducible loop is 2 edges, while transformation to a simulated loop can also be done as shown in Figure 5. Creating a loop requires that the first node is an empty injected node such that it contains only a condition, so it requires one node, an increase of three edges and SMC or a Boolean state variable. Notice that the successors and any condition of the second node are transfered to the first node.






```
A;
B;
```

(a)

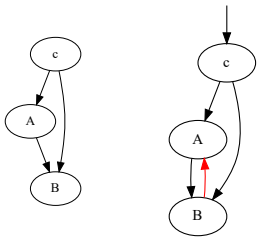
```
if (falseCondition1)
        goto labelB;
...
do {
        A;
labelB: B;
} while (falseCondition2);
```

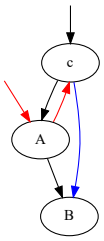
(b)

```
if (falseCondition1)
        goto labelB;
...
while (trueThenFalse) {
        A;
labelB: B;
}
```

(c)

Figure 5. Statement block (a) and with minimal edges to induce an IL (b) or with a simulated loop (c)






```
if (c) A;
B;
```

(a)

```
if (!c): goto labelB;
do {
        A;
labelB: B;
} while (falseCondition1);
```

(b)

```
if (falseCondition1)
        goto labelA;
...
while (c &&
        trueThenFalse){
labelA: A;
}
B;
```

(c)

Figure 6. One-sided conditional (a) with minimal edges to induce an IL (b) or with an extra edge (c)

Alternatively we consider a conditional and its minimal edges to induce an irreducible loop in Figure 6. Inducing irreducible loops on reducible loops is trivial as it only requires arbitrary edges from nodes outside the loop that are not reached from nodes in the loop to be added. Creating transformation primitives for two-way and n-way conditionals is straight-forward from simply extending this approach. Any false or true conditions are thus determined from separate outputs of the 2-SAT solver. Any conditions which change upon execution either use SMC or a Boolean state variable which is negated.

For our proposed dynamic scheme, a dependency ordering of a constant-time CFG can be used with multiple sets of Boolean conditions, which ensures that regardless of the execution path actually taken at runtime, at termination, all dependencies have executed one or more times and in proper order. The conditions should satisfy properties which limits a maximum duplicate execution of any vertex in the CFG. It should also tie the execution order of these conditions to the input to the CFG.

5.1. Security. The security goals of obfuscation are to increase the cost of an attacker trying to reverse engineer the code, whether to understand it, to bypass certain protection mechanisms or to extract security information. The attack cost is reduced when an attacker uses automated tools, and unsurprisingly most commonly obfuscation schemes are considered broken if they are suspectible to such tools as it indicates complete de-obfuscation knowledge. Manual attacks are much harder to prevent, but they are much more expensive as the attacker must spend significant time and energy to do binary code analysis and reverse engineering. The proposed scheme of using irreducible loops makes manual analysis difficult as binary code or decompiled representations become complex to analyze. Furthermore algorithms reprsented as CFGs are very hard to reverse engineer, as hard as Boolean circuit identification. The concepts presented are merely an example. A secure customization of a well known cryptographic algorithm would increase attack costs. Changing the SMC 2-SAT method for another constant Boolean conditions generator is possible. The irreducible loop generation can also be done in a non-deterministic or deterministic way with all sorts of various alternative patterns possible such as using n-way conditionals for loop headers or loop latch nodes.

As far as cryptographic indistinguishability is concerned, our proposed obfuscation scheme is indistinguishable from the underlying algorithm from a practical standpoint though not a theoretical one [8]. But the theoretical constructs are so impractical at this point, it is considered to remain as an open problem [29].

Both schemes are practically resistant to static analysis, as the presence of dynamic SMC makes a static analyzer unable to safely analyze past the point that it is detected, and the presence of irreducible loops makes a static analyzer unable to provide a simple representation of the original code. State-of-the-art tools do not perform a general enough analysis and are mostly geared towards non-obfuscated compiler-generated code.

The proposed dynamic scheme would increase resistance to differential fault analysis (DFA) as a fault would need to be generated at different times based on different input. However, a fault on the final execution of a vertex in the CFG is still an issue. It is resistant to differential computation analysis (DCA) and its subset differential power analysis (DPA) as the execution order changes based on the input.

To consider the resilience or success rate of the obfuscation scheme, we use the framework of Banescu, et al [6]. Namely that based computational power and time of an attacker, the difficulty in getting closer to the original CFG by a certain threshold, is greater than some overall amount of time deemed acceptable in

a given context. The authors specifically note that opaque predicates have only been defeated by automated tools in simple contrived scenarios. For static analysis attacks, the time would be based on the hardness of symbolically executing the dynamically generated SMC to obtain the predicates, which is beyond the capabilities of existing tools. In the dynamic scheme, however, the repeated operations would require exponential time without proving these operations as repeated and removing them to reduce it to the static scheme. Simple dynamic analysis would not provide a sufficient proof though it could have statistical success.

6. Conclusion

Although software development practices have moved away from SMC and irreducible loops, they continue to have importance to obfuscation and resistance against static analysis. In the context of SMC, there is little agreement over simple definitions such as loops or recursive code. There are also questions surrounding its time and Kolmogorov complexity.

We have proposed a dynamic scheme using a sequence of generated 2-SAT formulae, a CFG with irreducible loops and an execution path which changes based on input with controlled redundancy based on dependency analysis. This could provide a basis for even more sophisticated obfuscation.

De-obfuscation techniques will continue to get stronger against complex virtualization schemes. Inevitably, the power of dynamic SMC would ensure more advanced capabilities be developed in static analysis frameworks. Currently, they largely do not consider it, making it one very open practical technique which causes theoretical problems that have not yet received attention. In the future, it is hoped that static analysis tools have better methods for reasoning about SMC and a way to choose the optimal reduction strategy from an irreducible CFG to a reducible one.

References

[1] A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman, Compilers: Principles, Techniques, and Tools (2nd Edition), *Addison Wesley*, 2006.

[2] B. Aspvall, M. F. Plass and R. E. Tarjan, A linear-time algorithm for testing the truth of certain quantified boolean formulas, *Information Processing Letters* **8** (1979), 121–123.

[3] V. Balachandran and S. Emmanuel, Software code obfuscation by hiding control flow information in stack, In: 2011 IEEE International Workshop on Information Forensics and Security, 2011, 1–6.

[4] V. Balachandran and S. Emmanuel, Potent and stealthy control flow obfuscation by stack based self-modifying code, *IEEE Transactions on Information Forensics and Security* **8** (2013), 669–681.

[5] V. Balachandran, N. W. Keong and S. Emmanuel, Function level control flow obfuscation for software security, In: 2014 Eighth International Conference on Complex, Intelligent and Software Intensive Systems, 2014, 133–140.

[6] S. Banescu, M. Ochoa and A. Pretschner, A framework for measuring software obfuscation resilience against automated attacks, In: 2015 IEEE/ACM 1st International Workshop on Software Protection, 2015, 45–51.

[7] D. J. Bernstein, Multidigit multiplication for mathematicians, 2001, https://cr.yp.to/papers/m3.pdf.

[8] L. Buttyán and M. Horváth, The birth of cryptographic obfuscation – A survey, *IACR Cryptol. ePrint Arch.* (2018), https://eprint.iacr.org/2015/412.pdf.

[9] S. Chow, Y. Gu, H. Johnson and V. A. Zakharov, An approach to the obfuscation of control-flow of sequential computer programs, *Information Security* (2001), 144–155.

[10] C. Cifuentes, A structuring algorithm for decompilation, In: XIX Conferencia Latinoamericana de Inform'atica, 1993, 267–276.

[11] N. M. Clift, Calculating optimal addition chains, *Computing* **91** (2011), 265–284.

[12] J. Daemen and V. Rijmen, The Rijndael Block Cipher – AES Proposal: Rijndael, 1999.

[13] Ghidra 9.2.4, https://ghidra-sre.org/.

[14] P. Havlak, Nesting of reducible and irreducible loops, *ACM Trans. Program. Lang. Syst.* **19 4** (1997), 557–567.

[15] Hex Rays - IDA Free 7.6, https://hex-rays.com/ida-free/.

[16] A. Kalysch, J. Götzfried and T. Müller, VMAttack: Deobfuscating virtualization-based packed binaries, In: Proceedings of the 12th International Conference on Availability, Reliability and Security, 2017, 1–10.

[17] J. Karthik, P. P. Amritha and M. Sethumadhavan, Video Game DRM: Analysis and paradigm solution, In: 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT), 2020, 1–4.

[18] W. Landi, Undecidability of static analysis, *ACM Lett. Program. Lang. Syst.* **1** (1992), 323–337.

[19] T. László and Á. Kiss, Obfuscating C++ programs via control flow flattening, *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica* **30** (2009), 3–19.

[20] A. Majumdar and C. Thomborson, Manufacturing opaque predicates in distributed systems for code obfuscation, In: Proceedings of the 29th Australasian Computer Science Conference, 2006, 187–196.

[21] B. Parhami, Carry-free addition of recoded binary signed-digit numbers, *IEEE Transactions on Computers* **137** (1988), 1470–1476.

[22] G. Ramalingam, Identifying loops in almost linear time, *ACM Trans. Program. Lang. Syst.* **21** (1999), 175–188.

[23] G. Ramalingam, On loops, dominators, and dominance frontier, In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, 2000, 233–241.

[24] REC Studio 4 - Reverse Engineering Compiler, http://www.backerstreet.com/rec/rec.htm.

[25] J. Stanier and D. Watson, A study of irreducibility in C programs, *Software: Practice and Experience* **42** (2012), 117–130.

[26] W. Tao, M. Jian, Z. Wei and C. Yu, A new algorithm for identifying loops in decompilation, *Static Analysis* (2007), 170–183.

[27] S. Unger and F. Mueller, Handling irreducible loops: optimized node splitting versus DJ-graphs, *ACM Trans. Program. Lang. Syst.* **24** (2002), 299–333.

[28] C. Wang, J. Davidson, J. Hill and J. Knight, Protection of software-based survivability mechanisms, In: International Conference on Dependable Systems and Networks, 2001, 193–202.

[29] H. Xu, Y. Zhou, Y. Kang and M. R. Lyu, On secure and usable program obfuscation: A survey, *arXiv:1710.01139* (2017), https://arxiv.org/abs/1710.01139.

[30] K. Yakdan, S. Dechand, E. Gerhards-Padilla and M. Smith, Helping Johnny to analyze malware: A usability-optimized decompiler and malware analysis user study, In: IEEE Symposium on Security and Privacy (SP), 2016, 158–177.

[31] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla and M. Smith, No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations, In: 22nd Annual Network and Distributed System Security Symposium, 2015, 1–15.

GREGORY MORSE, MIDYA ALQARADAGHI, TAMÁS KOZSIK
EÖTVÖS LORÁND TUDOMÁNYEGYETEM / UNIVERSITY (ELTE)
DEPARTMENT OF PROGRAMMING LANGUAGES AND COMPILERS,
BUDAPEST, HUNGARY

E-mail: gregory.morse@live.com

E-mail: alqaradaghi.midya@inf.elte.hu

E-mail: kto@elte.hu