Towards Working with Small Atomic Functions

Alec Yasinsac*[1] and J. Todd McDonald**[2]

[1] Department of Computer Science
Florida State University, Tallahassee, FL
`yasinsac@cs.fsu.edu`
[2] Department of Electrical and Computer Engineering
Air Force Institute of Technology
`jmcdonal@afit.edu`

Abstract. Shannon's notion of entropy remains a benchmark reference for understanding information and data content of cryptosystems. Ideally, secure ciphers maintain high entropy between possible plaintext - ciphertext pairs. The one time pad, though perfectly secure in terms of entropy, remains impractical in most general cases due to key management issues. We discuss in this paper the similar notion of function entropy and examine its use on a small scale to provide perfect functional secrecy. We illustrate how such small units of composition can form the basis for obfuscating software transformations in a general, but highly constrained sense.

Key words: atomic functions, functional entropy, obfuscation, software protection

1 Introduction

In his seminal 1998 talk [1], Roger Needham lamented about the abdication of simplicity. He quoted Christopher Strachey to point out: "It is impossible to foresee the consequences of being clever". Indeed, clever solutions are rarely intuitive; otherwise they would be obvious rather than clever. Needham puts his own stamp on it by saying: "if you did something in a straightforward way it was more likely to be correct. The protocols devised in the early days were not straightforward. They relied on clever inferences and arguments to convince yourself that the goals had in fact been attained."

In this paper we propose a return to simplicity. Why do we believe it is possible to achieve simplicity when technology push and market pull demand greater functionality, at lower cost, over a wider audience, in order to extract a more exorbitant profit? Our foundational premise is that developing anything "general purpose" creates complexity. Conversely, leveraging properties of atomic functions provides a simple framework for protocol development.

* This work is funded in part by Department of Defense grant H98230-06-1-0232 and Army Research Office DAAD19-02-1-0235.

** The views expressed in this article are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government

1.1 Software Engineering

A major step forward in the computing field occurred when scientists realized that they could use computers to simplify computing process description. The conceptually simple notion involved writing programs to perform a well-defined function in a structured, but natural language that was more understandable to humans than was machine code, then mechanically translating the natural language description into an equivalent, machine executable version. The field of Software Engineering (SWE) continues to address approaches to understanding systems of such programs.

We may think of the SWE goal as trying to reduce *functional entropy.* SWE leverages abstraction and standardization to elevate algorithms to higher conceptual levels and to generate "look and feel" commonality that promotes understanding. Neither of these approaches themselves inherently reduces complexity. While abstraction reduces complexity for the higher end user, it does so at the cost of precision, injecting possible ambiguity and loss of nuance.

1.2 Binary Relations and Function Tables

While mathematicians are accustomed to working with algorithmic function representations, functions are more comprehensively (though less elegantly) represented as binary relations, i.e. as ordered pairs arranged as function tables. Indeed, it is because function tables can comprehensively and systematically define all functions that this representation best suits our interests. Additionally, it is important to note that a function table represents an atomic function. That is, while many algorithms may generate a given function table, every deterministic algorithm implements exactly one function table.

More precisely, we are interested in function tables as follows: for integers i, m, and n, and the set of integers $X = 0..2^n$, a function table $S_{n,m}$ is an ordered set of ordered pairs for $i = 1..2^n$, (x_i, y_j), where $x_i = i$ and $y_j \in X$. S^*_n is the set of all function tables of size of n input bits and m output bits.

This definition allows us to address classes of same-sized functions, with input and output lengths as the class-defining factors. Operating on such monolithic, yet general, functions simplifies many concepts and operations. For example, consider the function composition operation ($\circ$), defined in the normal way (i.e. $f \circ g = f \mid g = f(g(\cdot))$). In this monolithic environment, the composition operation naturally avoids the usual range-domain mapping issues. Thus we know that we can compose any two function tables x and y as long as $(x, y) \in S^*_n$ and we also know that S^*_n is closed under composition.

Much as symmetric data encryption technology generally derives its cryptographic strength from the exclusive-or (XOR) operation, methods on atomic functions derive entropy strength from function composition. Intuitively, since every function table is atomic and function table composition is a closed operation, every composition is atomic. These notions along with the ability to randomly select function tables provide the foundation for perfect function encryption.

2 Perfect Functional Entropy

Like data entropy, we can measure function entropy by analyzing random selection. This means that we must define a finite population and show that our obfuscation is indistinguishable from a random (unbiased) selection from that population. Defining a suitable finite population of functions represented as programs is problematic, because programs and functions can take so many different forms. While circuits are less obtuse than programs, they share some many of the same complexities. However, the ultimate functional representations, function tables, offer several positive "selection" properties.

Unlike programs or circuits, tabular function representation allows us to systematically capture and enumerate all functions of a given input/output size, as we illustrate below. Function enumeration ensures that we can make a random function selection [2] and that we can show indistinguishability between our obfuscation and a randomly selected function; the result is an encrypted function.

2.1 Function Table Illustrations

Function tables reflect atomic functionality. Though many algorithms may implement a given functionality (hereafter termed "operation") there is only one function table for that operation. We illustrate this notion by looking at small input-output sized operations; in fact, we start with the smallest possible: one bit input, one bit output operations. We do this because this function size clearly illustrates how we define a comprehensive finite operation population, in this case $S^*_{1,1}$.

There are exactly four single bit input-output operation function tables that reflect the following semantic transformations:

1. Preserve the input bit
2. Flip the input bit
3. Flip 1, preserve 0
4. Flip 0, preserve 1

Table 1 enumerates one-bit operation function tables. As we mentioned earlier, many (in fact infinitely many) operations could generate each function table, e.g. the operation "$or(x, 0)$" also generates $S1$. We selected the listed operations as representative to add clarity to the table. We emphasize that these four function tables are distinct and they comprehensively capture all possible one-bit operations. Given an efficient program that computes a one-bit function, it is easy to determine which function table the program implements: simply exercise the program once with input 0 and once with input 1. However, it is not as clear whether or not we can infer anything more if, for example, we know that the executing program is actually a composite function. It turns out, knowing there is a composition may allow us to glean some information about the two composed operations.

Table 1. Function Tables for All 1-Bit Functions

ID/circuit	operation	Semantics	x	f(x)
S1/p1	and(x,1)	preserve either	0	0
			1	1
S2/p2	xor(x,1)	flip either	0	1
			1	0
S3/p3	and(x,0)	flip 1, preserve 0	0	0
			1	0
S4/p4	or(x,1)	preserve 1, flip 0	0	1
			1	1

We know that any one-bit function composition (say $p \mid q$) must produce another one bit function (say f) that must also reflect one of the four given function tables. Specifically, $p, q \in S_1^*$ and $(f = p \mid q) \Rightarrow f \in S_1^*$. Table 2 contains all sixteen possible one-bit function compositions that correspond to the four, one-bit function tables of Table 1 (i.e. $p1$ and $q1$ implement $S1$, $p2$ and $q2$ implement $S2$, etc.).

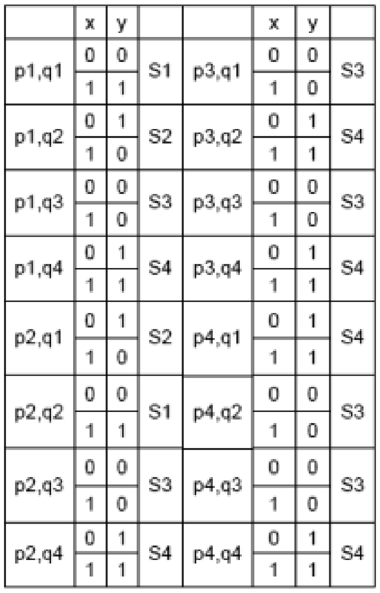
Let's say that we want to deliver a function that computes p to an adversary, but we want to protect p itself from that adversary. We can randomly select a second function that [atomically] computes one of the four one-bit function tables (and call it q), compose it with p in order to mask p and generate the corresponding function table. For example, from Table 2 we see that if we desire the adversary to execute $p3$ (which corresponds to function table $S3$) and we randomly select a q that computes $S2$ ($q2$), the composition $p3 \mid q2$ computes $S4$. Thus, if we send function $p4$ to the adversary, the adversary cannot determine whether we desire to compute $p1 \mid q4$, $p2 \mid q4$, $p3 \mid q2$, $p3 \mid q4$, $p4 \mid q1$, or $p4 \mid q4$, since each of these compositions compute $S4$.

If our purpose is to protect p from the adversary, we mask p by randomly selecting q and composing p and q. This construction ensures that the adversary can only guess the intended program p with no better than 50% likelihood. We also note that to accomplish 50% likelihood, the adversary must compute all function tables S_n^*, which demands n-factorial computation.

We recognize that if we randomly select the masking operation q, we run the risk of also masking p's functionality. For example, if we utilize $q3$ as the masking function, the composition will always output zero regardless of p's functionality. Similarly, compositions with $q4$ will always return 1, preventing recovery of p's computation. Of course this limits the value of the construction.

On the other hand, if we only utilize $q1$ (preserve 0 and 1) or $q2$ (flip 1 and 0) as our masking operation, the adversary that either captures the input-output mappings or is in possession of the composition, can still not guess p with better than 50% likelihood, yet we can recover p's result if we receive the composition's

Table 2. Function Tables for Compositions



	x	y			x	y	
p1,q1	0	0	S1	p3,q1	0	0	S3
	1	1			1	0	
p1,q2	0	1	S2	p3,q2	0	1	S4
	1	0			1	1	
p1,q3	0	0	S3	p3,q3	0	0	S3
	1	0			1	0	
p1,q4	0	1	S4	p3,q4	0	1	S4
	1	1			1	1	
p2,q1	0	1	S2	p4,q1	0	1	S4
	1	0			1	1	
p2,q2	0	0	S1	p4,q2	0	0	S3
	1	1			1	0	
p2,q3	0	0	S3	p4,q3	0	0	S3
	1	0			1	0	
p2,q4	0	1	S4	p4,q4	0	1	S4
	1	1			1	1	

output and know the key. In the example above, if our intended operation is $p3$ and the masking function is $q2$, we send the adversary the corresponding (atomic) composition, $p4$. To recover the $p3$ result, feed the returned output into $q2$. This leads us to introduce three definitions.

Definition 1 (Function Preservation and Operation Invertibility). *The target operation (p) of a composed operation ($p' = \boldsymbol{O}(p) = p \mid q$ for some q) is preserved if and only if the masking operation (q) is invertible. The masking operation is invertible if and only if it is one-to-one, that is, if it maps each input to a distinct output.*

Definition 2 (Perfect Obfuscation). *An operation p' is a perfect obfuscation of p if and only if an adversary when given p' can compute p with no better than 50% probability.*

Definition 3 (Perfect Operational Obfuscation). *A function p' is a perfection functional obfuscation of p (notationally $\boldsymbol{O}(p)$) if and only if:*

1. *p' is a perfect obfuscation of p and*
2. *p' preserves the function p*

This illustrates our new computation model, where the obfuscator produces a function composition. If the composition preserves p, $\mathbf{O}(p)$ also produces the recovery operation.

Symbolically, Given q, $\exists q^{-1} \Rightarrow (p' = \mathbf{O}(p) = (p \mid q, q^{-1}))$, else $p' = \mathbf{O}(p) = p \mid q$ and p' does not preserve p. Algorithm 1 generates an encrypted one bit function. Perfect functional obfuscation is similarly as strong for protecting programs as Shannon's perfect secrecy [3] is for protecting data.

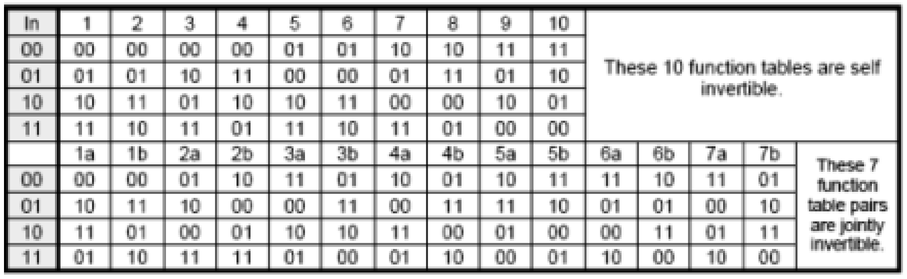
Algorithm 1 (A Perfect One-bit Functional Obfuscator).

1. *Identify the function table entry from Table 1 corresponding to the target function and select the corresponding circuit (e.g. p1, p2, etc.)*
2. *Randomly select a masking function from $q1$-$q4$*
3. *Identify the function table entry ($S1$-$S4$) from Table 1 corresponding to the composition of the target function and the randomly selected circuit (e.g. p1, p2, etc.)*
4. *Select p' as the circuit that corresponds to the selected function table entry in step 3.*

2.2 Extending Perfect Functional Obfuscation to Larger Functions

Function table enumeration grows exponentially on the input-output size, so we only extend our preliminary exploration/illustration once, to 2-in, 2-out functions. There are 256 distinct operations with two bits input and two bits output. We show the 24 2-bit input/output function table entries for those that are invertible in Table 3.

Table 3. Invertible 2-bit by 2-bit Operations

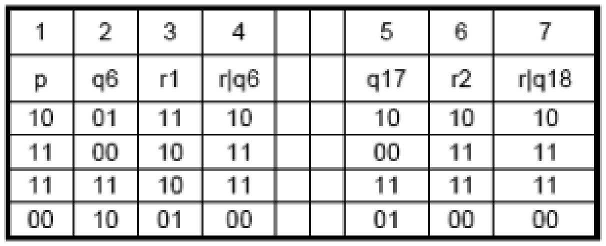


In	1	2	3	4	5	6	7	8	9	10	These 10 function tables are self invertible.
00	00	00	00	00	01	01	10	10	11	11	
01	01	01	10	11	00	00	01	11	01	10	
10	10	11	01	10	10	11	00	00	10	01	
11	11	10	11	01	11	10	11	01	00	00	

	1a	1b	2a	2b	3a	3b	4a	4b	5a	5b	6a	6b	7a	7b	These 7 function table pairs are jointly invertible.
00	00	00	01	10	11	01	10	01	10	11	11	10	11	01	
01	10	11	10	00	00	11	00	11	11	10	01	01	00	10	
10	11	01	00	01	10	10	11	00	01	00	00	11	01	11	
11	01	10	11	11	01	00	01	10	00	01	10	00	10	00	

As indicated in the table notes, function tables 1-10 are self invertible. That is, if you compose any 2-bit input/output operation p with e.g., $q6$ so that $r(\cdot) = q6(p(\cdot))$, for example, you can recover p's output by computing $q6(r(\cdot))$. On the other hand, tables 11-24 (marked $1a - 7b$) are pair-wise invertible, so for example, you may recover p's output from $r(\cdot) = q17(p(\cdot))$ by computing $q18(r(\cdot))$. We illustrate these examples in Table 4, where we select p (in column 1) from the non-invertible population. Column 2 contains $q6$, which is an arbitrary masking operation. Column 3 is p composed with $q6$; this is the module that we would send to a mobile host. Column 4 is r composed with $q6$, which illustrates that $q6$ is self invertible, i.e. that $q6(q6(p(\cdot)))) = p(\cdot)$. Columns 5 through 7 are the analogous demonstration with jointly invertible operations, $q17$ and $q18$, i.e. that $q18(q17(p(\cdot)))) = p(\cdot)$.

Table 4. Operation Obfuscation Examples



1	2	3	4			5	6	7
p	q6	r1	r|q6			q17	r2	r|q18
10	01	11	10			10	10	10
11	00	10	11			00	11	11
11	11	10	11			11	11	11
00	10	01	00			01	00	00

The fundamental point here is that the composed function $(q(p(\cdot)))$ is implemented by an *atomic function*, r. There is no "seam" between the composed functions that dynamic analysis or reverse engineering can discover. Why does that matter? It turns out that this construction resolves two classic security problems.

Consider an application that collects and analyzes data at a remote host and then transmits a computed result to a central location. In our model, the p operation captures the data collection and analysis functionality, while q provides data privacy protection, say through encryption. Our composition construction $r = p \mid q$ hides p's functionality from the remote host with two fundamental properties:

1. Because r is an atomic operation, there is no seam to find that could divulge p's output.
2. Similarly, since r is an atomic operation, there is no systematic approach an adversary can undertake to divulge any processing detail, such as a key that q may use for data protection.

Both of these properties realize resolutions to classic computer security problems. In combination, they allow us to construct mobile code operating on a malicious host to accept input from that host, conduct a meaningful function on the input, and encrypt the output for transmission. Even with the malicious host in complete control of the computation, it cannot separate the encryption process from the functional process nor can it conduct any meaningful key-based cryptographic analysis. If q is an encryption process, p's output is returned by the corresponding decryption process $r \mid q^{-1}$, i.e., we compute the partial result $p(x)$ as: $p(x) = q^{-1}(r(x))$.

3 Perfect Program Encryption Process, Scope, and Limitations

The Barak program obfuscation impossibility result [4] is widely recognized as a condemnation of obfuscation techniques in general. Fortunately when needed, obfuscation is not performed in general, but in specific. We show that general obfuscators exist that provide perfect functional protection, though not without constraints. We address those constraints and our method's applicability in this section.

3.1 Function Performance and Size

Perfect function encryption requires that the dispatcher generate a function table for the composed function. From a processing standpoint, we generated a 32-bit (padded) input function table for the Data Encryption Standard on a standard desktop computer in about 24 hours. Each additional bit approximately doubles the computation time, but increased computation power and parallelism could substantially elevate the input size that we could compute.

While function table construction is computationally intensive, the execution code will be a computationally efficient table look-up. Thus, in our scheme the composition performance is very fast, however at the expense of storage demands. Function table storage size is exponential on input length. 32-bit functions form .5 Gigabyte function tables. One gigabyte flash memory is relatively inexpensive, so hand held devices could employ this technology for up to 40-bit computations. However, the cost/benefit advantage of moving large executables (mobile code) across a network is questionable. 24-bit computations with 2 Megabyte function tables may make more sense for network applications.

3.2 Adversarial Computational Capabilities and Limitations

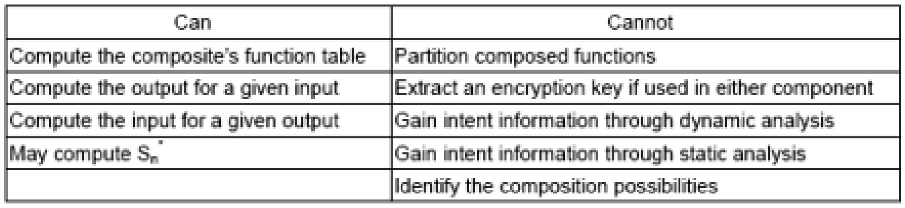
This is a particularly important section in this paper that captures a fundamental and confusing aspect of our method. In discussions with colleagues, confusion often arises because an adversary may be able reveal the input that produced a given output from the perfectly encrypted function. This seems to contradict the foundational notion of [*data*] encryption. The essential point here is that *function encryption does not intend to protect data confidentiality*; rather it protects against the adversary understanding the code's functional intent. Of course, because this protection ensures that there is no seam between the two processes embedded within the function and it prevents an adversary from revealing an embedded encryption key, perfect function encryption can protect partial result confidentiality, as we mention earlier.

We now identify several adversarial capabilities. Consider first, an adversary that maliciously possesses a device that contains perfectly encrypted function composition. Because the adversary possesses a copy of the composition and because our approach limits input size, given reasonable resources and time, the adversary can compute the composition's function table. Thus, the adversary knows the computational result for every input. Using this information, the adversary could introduce selected input to generate any desired output from the captured device. However, if the composition operation (q) is an encryption operation, this input-output mapping does not leak any information about the functional operation (p), so this attack would accomplish nothing more than blind disruption.

Additionally, if the adversary collected device output generated before the compromise, they could use the function table to determine the input for the corresponding collected output. Again, this illustrates that our result does not (and does not intend to) provide traditional data confidentiality. In fact, in mobile code applications, the host environment is expected to provide (thus to know) the data that is input into the device. Again for emphasis, we do not intend to hide the process input.

Conversely, while the adversary may be able to compute the composition's function table through black box analysis, this function table alone does not expose the composed operations. Since the composition is an atomic operation, there are no embedded hints for an adversary to detect (e.g. the seam and key we mentioned earlier). Additionally, the control flow of all atomic operations is identical (table lookup), so neither static nor dynamic code analysis can leak any meaningful functional details. Similarly, compiler optimization or other analysis can reflect no functional distinctions. Table 5 summarizes function encryption adversary capabilities and limitations.

Table 5. Adversary Capabilities and Limitations Given a [Composite] Encrypted Function



Can	Cannot
Compute the composite's function table	Partition composed functions
Compute the output for a given input	Extract an encryption key if used in either component
Compute the input for a given output	Gain intent information through dynamic analysis
May compute S_n^*	Gain intent information through static analysis
	Identify the composition possibilities

3.3 Generality

Composing atomic functions for function encryption is limited in scope to small input/output sized functions. However, our method is otherwise uniformly general in the sense that it protects any program of the given size; in fact, it garners its perfect strength from that generality. Our approach requires that we enumerate all functions of a given size without respect to any algorithms that implement those functions. Systematic enumeration ensures uniform distribution, essentially eliminating bias in the selection process.

3.4 Strength

As we demonstrated above, if an adversary uses an encrypted function to generate a function table and if they know that the evaluated function was created through composition, they may be able to gain information about the possible composed functions. To recognize this correlation, they must construct the potential composition function tables. Constructing all function tables in S_n^* is super exponential on n. For small n, this is not decisive, e.g. for $n = 16$, $n^n = 2^{64}$, but for thirty two functions (2^{190}), this is computationally infeasible. Additionally, once all tables are derived, constructing all possible compositions requires an additional n-factorial computations.

If we assume that an adversary is intent on conducting this analysis, as we described, even if an adversary constructs S_n^* and generates $S_n^* x S_n^*$ possible compositions, the process guarantees that the adversary cannot identify the functions used in the composition with greater than 50% chance. Thus, our program encryption provides absolute protection against even computationally unbounded adversaries.

Moreover, a watermark of any effective program obfuscation mechanism must protect even a program that prints its own source code [4, 5]. Since the obfuscation's output is always encrypted, our mechanism protects program intent for even this classic program.

4 Conclusion

There are theoretical and practical reasons for desiring to create programs with [apparent] high functional entropy. Though general program obfuscation in the virtual black box paradigm does not exist, we introduce a different computational model where general, though constrained obfuscation is possible. In fact, we show that provable, perfect obfuscation is possible and we introduce the term function encryption to describe its manifestation. To generate obfuscation metrics and proofs, we introduce the notion of functional entropy, and we show how our approach to function encryption is analogous to Shannon's perfect secrecy for data encryption. Though this method is only practical for functions with very small input and output sizes, its theoretical impact challenges the Barak result and suggests that for small functions, obfuscation is possible.

History shows that classic programs (and protocols) are less subject to failure than their more complex counterparts. We foreshadow an approach to narrow application variability, for example by restricting functions to a fixed input-output size. We illustrate this functionality utilizing atomic functions.

References

1. Needham, R. M.: Logic and Over-Simplification. Proc. of the Thirteenth Annual IEEE Symposium on Logic in Computer Science, 21-24 June (1998) 2–3
2. Goldreich, O., Goldwasser, S., Micali, S.: How to Construct Random Functions. Journal of the ACM (JACM), Vol. 33-4. (1986) 792–807

3. Shannon, C.E.: Communication Theory of Secrecy Systems. Bell System Technical Journal, Vol. 28-4. (1949) 656–715
4. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (Im)possibility of obfuscating programs. Proc. of the 21st Annual Int'l Cryptology Conference on Advances in Cryptology, Lecture Notes in Computer Science, Vol. 2139. Springer-Verlag (2001) pp. 1–18
5. Thompson, K.: Reflections on trusting trust. Communications of the ACM. **27**-8 (1984) 761-763