# Bit Influence Disparity in Binary Addition

Kevin Burk

June 18, 2018

**Abstract**

When adding two binary numbers, the least significant input bits have the potential to trigger cascading carries, influencing every bit of the output, while the most significant input bits will only ever influence the most significant bit of the output. This bit influence disparity seems to conflict with the cryptographic idea that a number can be completely obscured by addition with a uniform random value, though in fact the uniform randomness of the mask does ensure this property. In this paper, I investigate what information, if any, is leaked via bit influence disparity when obscuring a number by addition with a random mask. I also discuss a method of equalizing all input bits' influence with carry wraparound, and demonstrate that it is undesirable and unnecessary.

## 1 Introduction

In cryptography, we often find ourselves facing a simple but fundamental problem: We have two $n$-bit strings. One is sensitive data. One is random bits. How can we use the random bits to obscure the sensitive data so that it is meaningless to other observers, but still recoverable by anyone in possession of the random bits?

The first common approach is to XOR the two together. This completely obscures the sensitive bits, provided the random bits were sufficiently random, but they are still recoverable. XOR is its own inverse, so anyone who knows the random bits can repeat the operation and recover the original data.

The other common approach is to treat the two strings as numbers and add them modulo $2^n$. Again, they are obscured, but recoverable - in this case via subtraction.

In this paper, I examine the "intuitive security" of both of these approaches, with an emphasis on how, in the complicated case of addition, intuition can be misleading.

### 1.1 The Intuitive Security of XOR

Given a bit string $d$ and a random mask $m$ of the same length, the value $e = d \oplus m$ is a simple but effective encryption. Provided that each bit of $m$ has an equal

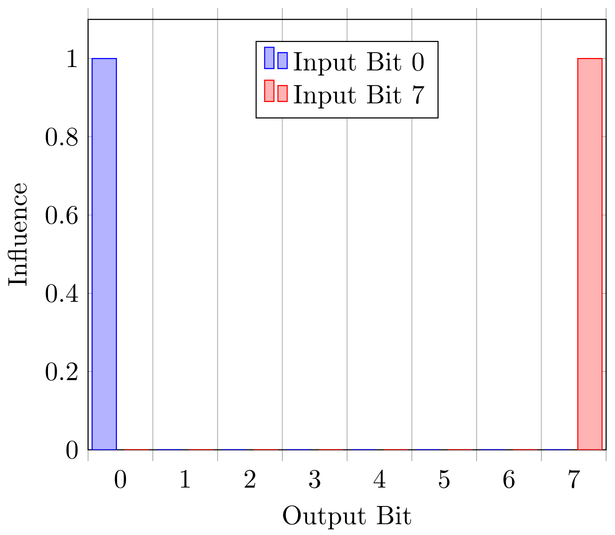

Figure 1: The probability of changing each output bit when toggling a specific input bit during an eight-bit XOR operation. Each input bit affects exactly one bit of the output.

probability of being 0 or 1, $e$ reveals nothing about $d$ to an observer that doesn't know $m$. It is possible that any bit in $e$ is the same as the original bit in $d$ (that is, the corresponding bit in $m$ was 0, so it was left unchanged), but it is equally likely that that bit in $m$ was 1, and that the bit in $e$ is the opposite of the original.[1]

It is also obvious that each input bit affects exactly one output bit, as shown in Figure 1. This is a nice property, as we don't have to worry about any input bit having a disproportionate influence on the output.

## 1.2 The Intuitive Security of Addition

Binary addition can be used in the same way. Given $n$-bit string $d$ and random mask $m$ of the same length, interpret them as integers in the range $[0, 2^n - 1]$ and compute $e = d + m \mod 2^n$. As long as $m$ was selected uniformly at random from all possible values, then $e$ is uniformly distributed as well. All possible values of $d$ appear equally likely to an observer that sees $e$ but does not know $m$.

This conception of addition - thinking of it as an operation in an additive group - gives us a sense that it is secure. When thinking about it on the level of individual bits, however, this is no longer the case; in fact, the intuition is that it is less secure.

Consider the mechanics of carrying: the possibility of carrying means that

[1]This can be shown mathematically; see Section 3.1.

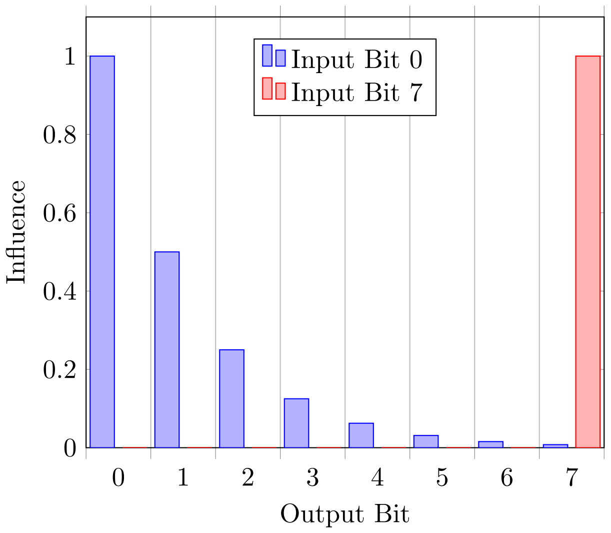

Figure 2: The probability of changing each output bit when toggling a specific input bit during eight-bit addition. Assuming all input bits are uniformly random gives the 50% carry rate shown here; see Section 5.1 for details.

one bit can “bleed over” and influence others, and each bit has a different potential to do so. Toggling the least significant bit of one of the inputs could trigger a carry, influencing the next least significant bit, potentially triggering another carry, and so on. But toggling the most significant bit of an input can only influence the most significant bit of the output.

This is illustrated in Figure 2. Toggling the most significant bit of an input will always change exactly one bit of the output, but toggling the least significant bit will change, on average, two output bits. This no longer has the satisfying property of equal influence that we saw with XOR. Suddenly, some bits seem twice as important as others.

## 2 Carry Wraparound

The instinctive reaction to this disparity is to do something about it, and the obvious way to equalize the influence of the higher bits is to let carries wrap around, back to the bits of lowest significance. In most architectures this can be done by finding every add instruction `x = add(a, b)` and inserting an add-with-carry instruction `x = adc(x, 0)` immediately after it.

A single add-with-carry instruction is sufficient. Since the original `add` instruction does not have the initial carry bit set, the first carry that occurred did so when both input bits were set and the carry flag was not, guaranteeing a zero in the output that can catch the carry bit that wraps around. Thus the `adc` instruction can never produce an overflow.

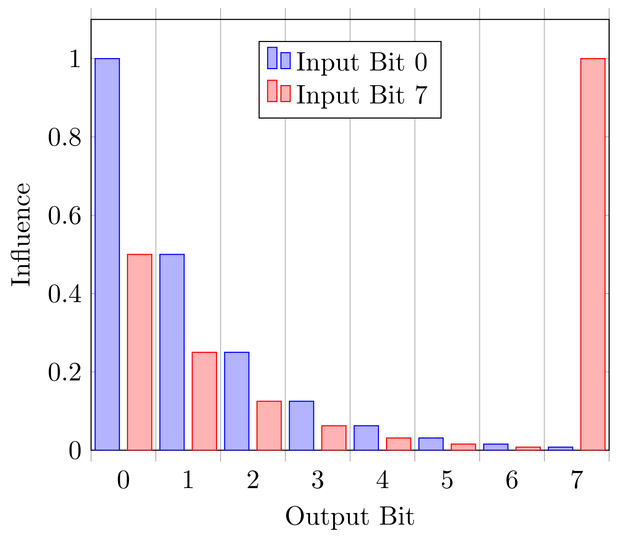

Figure 3: The probability of changing each output bit when toggling a specific input bit during eight-bit addition with carry wraparound.

As seen in Figure 3, the "problem" with addition is now solved. All input bits have the same expected influence on the output. But unfortunately, carry wraparound comes with multiple problems of its own.

## 2.1 Problems with Carry Wraparound

The obvious problem is performance. Addition just became twice as expensive on standard hardware. The prediction circuitry from a carry lookahead adder could likely be used to reduce this overhead significantly, but that would require custom hardware.

The more serious problem, though, is that addition with carry wraparound is no longer a permutation on its domain. Consider the eight-bit numbers $[0, 255]$ and some value $m$ in that range. As with normal addition, $0+m = m$. But with carry wraparound, $255 + m = m$ as well.[2] If addition with carry wraparound is used to encrypt $n$-bit strings, then the encryptions of 0 and $2^n - 1$ are identical. Such a scheme must either restrict its possible inputs or accept that decryption may fail. The former solution is a major devolution for a scheme that could previously accept any bit string of the desired length, and the latter is undesirable in any scheme. Addition with carry wraparound is not recommended.

[2]Provided that $m \neq 0$, which ensures that an overflow does occur.

## 3 Addition Reexamined

Given that this “fix” for a potential problem ended up causing obvious problems, let’s take a different approach. If we can show that the output of addition is uniformly random despite the varying contributions of its input bits then we can show that the intuition is faulty, and that no adjustments need be made.

We can do this with a more detailed model of addition - specifically, of the digital circuit with which addition is implemented.

### 3.1 Probabilistic Circuits

Instead of considering bits to be either exactly zero or exactly one, we can represent a bit as the probability that that bit is one. Zero and one retain their exact values, but we can represent random bits as well. A bit with value 0.6, for example, can be expected to result in one 60% of the time it is sampled.

We can then calculate the expected outputs of the standard logic gates as functions of the probabilities that their inputs are set:[3]

$$\begin{aligned} \neg p &= 1-p \\ p \wedge q &= pq \\ p \vee q &= p+q-pq \\ p \oplus q &= p+q-2pq \end{aligned}$$

With these equations we can start to make statements about the behavior of circuits. For example, it’s easy to show that XORing a value with a uniform random mask obscures that value completely:

$$\begin{aligned} m &= \tfrac{1}{2} \\ d \oplus m &= d+m-2dm \\ &= d+\tfrac{1}{2}-2d\tfrac{1}{2} \\ &= d+\tfrac{1}{2}-d \\ &= \tfrac{1}{2} \end{aligned}$$

### 3.2 Modeling an Adder

We can use the same approach to analyze binary addition. Since all correct adders will return the same result given the same inputs, it suffices to model the simplest circuit: the ripple carry adder.

A ripple carry adder consists of a chain of one-bit adders that operate serially. Each one-bit adder takes two input bits, $a$ and $b$, and an input carry bit, $c$. It calculates an output value, $v$, which is returned as part of the ripple carry

[3]See Section 5.2 for more on how these equations are derived.

```
#! /usr/bin/env ruby
a = (ARGV[0] || 0.5).to_f
b = (ARGV[1] || 0.5).to_f
c = 0

printf "a = %8.6f\nb = %8.6f\n\n", a, b
printf "    Value    Carry\n"
printf "-|--------|--------\n"

8.times do |i|
  v = a + b + c - 2*a*b - 2*a*c - 2*b*c + 4*a*b*c
  c = a*b + a*c + b*c - a*a*b*c - a*b*b*c - a*b*c*c + a*a*b*b*c*c
  printf "%i %8.6f %8.6f\n", i, v, c
end
```

Figure 4: Ruby code that simulates an eight-bit ripple carry adder. This program was used to generate the data plotted in Figures 5 and 6. For simplicity, this code assumes that all bits of one input have a probability $a$ of being set, and that the bits of the other all have a probability $b$ of being set.

adder's output, and an output carry bit, $c'$, which is passed to the next one-bit adder in the chain.

The output values can be found with simple circuits. The output value $v$ is set when an odd number of the three input bits were set; this can be found with XORs. The output carry bit $c'$ is set when at least two of the three input bits were set; this can be found with ANDs and ORs:

$$\begin{aligned} v &= a \oplus b \oplus c \\ c' &= (a \wedge b) \vee (a \wedge c) \vee (b \wedge c) \end{aligned}$$

Rewriting these boolean equations as their probabilistic versions gives, after a bit of algebraic simplification:

$$\begin{aligned} v &= a + b + c - 2ab - 2ac - 2bc + 4abc \\ c' &= ab + ac + bc - a^2bc - ab^2c - abc^2 + a^2b^2c^2 \end{aligned}$$

These equations can be used to simulate a ripple carry adder. See Figure 4 for a working example.

Running this simulation with various input bit probabilities, we can observe the probability of producing a carry at each one-bit addition. These probabilities are shown in Figure 5, and they seem to confirm that pessimistic intuition: these probabilities are not uniform, even when $a = b = \frac{1}{2}$. In fact, the most significant

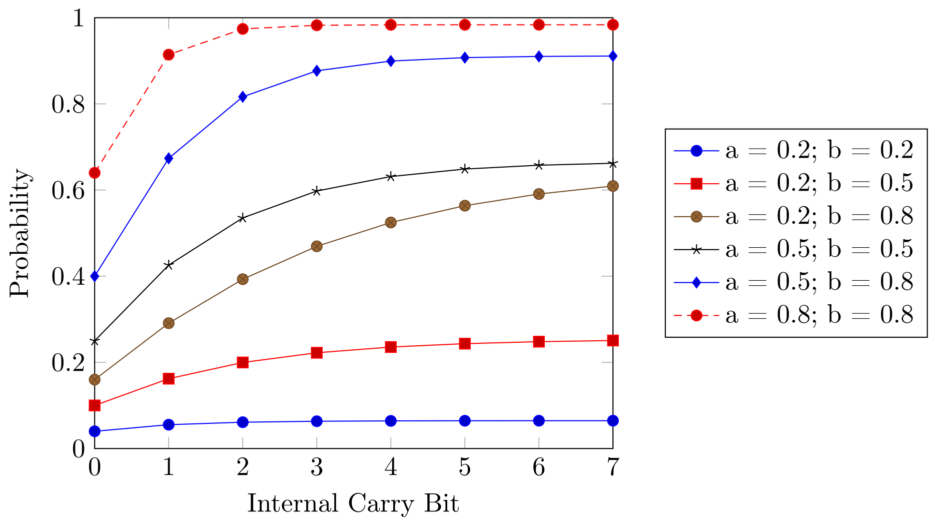

Figure 5: The probability of each internal carry bit being set during a ripple carry add, based on the probabilities on input bits $a$ and $b$ being set. Note that this is an increasing function for all values of $a$ and $b$.

bits are always the most likely to carry, regardless of the values of $a$ and $b$.[4]

Fortunately, these carry bits are only ever used internally by the adder.[5] To show a weakness in addition, we must show a weakness observable in the output bits, and these show a very different behavior.

Figure 6 shows the probability of getting a one at each output bit from variously distributed inputs. Here, as long as at least one input was uniformly random, the uniform distribution is preserved. It doesn't matter if a bit is being flipped with very high probability by a carry; if it is also being flipped with 50% probability by a bit from the random mask it is still completely obscured.

This property can also be derived from the equation for $v$:

$$\begin{aligned} b &= \tfrac{1}{2} \\ v &= a + b + c - 2ab - 2ac - 2bc + 4abc \\ &= a + \tfrac{1}{2} + c - 2a\tfrac{1}{2} - 2ac - 2\tfrac{1}{2}c + 4a\tfrac{1}{2}c \\ &= a + \tfrac{1}{2} + c - a - 2ac - c + 2ac \\ &= \tfrac{1}{2} \end{aligned}$$

Ultimately, the obscuring properties of XOR ensure that addition is secure despite the asymmetries introduced by carrying. And conveniently, an integer

[4]A few degenerate cases are not shown. If both $a$ and $b$ are one, a carry occurs at every step; if either $a$ or $b$ is zero then no carries occur.

[5]Or, in the case of the most significant carry bit, never used at all.

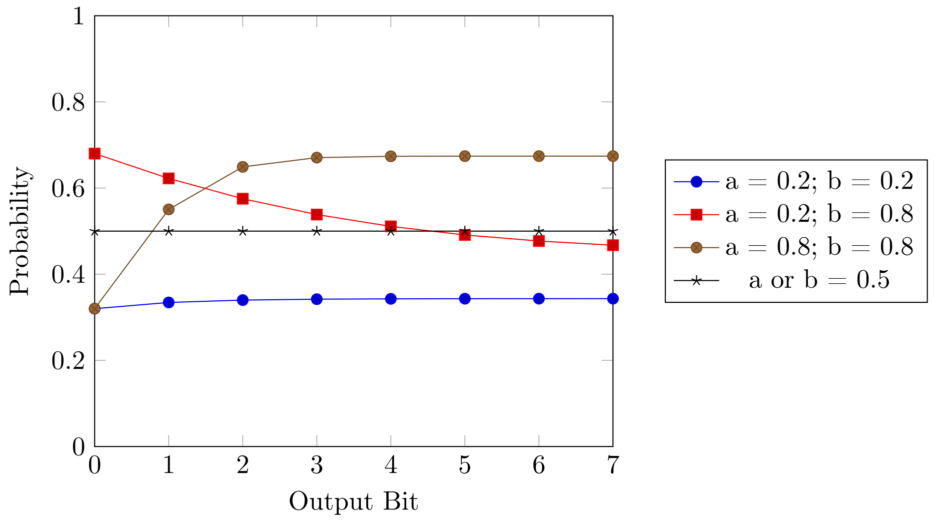

Figure 6: The probability of each output bit $v$ being set after addition, based on the probabilities of input bits $a$ and $b$ being set. If either $a$ or $b$ is set with probability $\frac{1}{2}$ then the probability of $v$ being set is also exactly $\frac{1}{2}$; only one line is shown for this case.

selected uniformly at random modulo $2^n$ will have each bit set with exactly 50% probability. The intuitive security of the additive group interpretation from Section 1.2 and the bitwise security calculated above are equivalent.

## 4 Conclusions

Binary addition, with the asymmetries introduced by carrying, lacks the simple, intuitive security of XOR. A mathematical treatment, however, shows that there are no issues in practice; in fact, attempts to fix such perceived problems, as seen with carry wraparound, can introduce more severe flaws than those they were intended to solve.

Addition, then, provides an excellent example of how intuitions can fail us. While they may serve as useful launching points, they should always be rigorously verified.

## 5 Notes

### 5.1 On the Influence Graphs

The influence graphs in Section 1 show the probability that flipping a specified input bit will result in a flip of each output bit.

Since each each output bit is calculated as the parity of the input bits,[6] flipping an input bit is guaranteed to change the corresponding output bit, hence the probability of 100%.

If flipping an input bit triggers a carry, the next output bit is guaranteed to be flipped by the same logic, but this needs to be multiplied by the probability of carrying. For these graphs, I assume that all input bits are uniformly distributed, and calculate the carry probability of 50% as follows:

For the first bit, toggling an input bit changes the output carry bit exactly when the other input bit is set; if the other input bit was not set, then there is no way a carry could occur. The uniform input distribution gives this case a 50% probability.

For all subsequent bits, a cascading carry bit will trigger a subsequent carry when exactly one of the input bits is set; if neither are set a carry cannot happen, and if both are set the carry will happen regardless. Again, this case occurs with 50% probability.

### 5.2 On the Gate Equations

The probabilistic equations for the logic gates from Section 3.1 can be derived from the probabilities of receiving each input combination.

| $p$ | $q$ | Probability | Simplified |
|---|---|---|---|
| T | T | $pq$ | $pq$ |
| T | F | $p(1-q)$ | $p-pq$ |
| F | T | $(1-p)q$ | $q-pq$ |
| F | F | $(1-p)(1-q)$ | $1-p-q+pq$ |

Since all of these cases are mutually exclusive and completely cover the input space, we can simply sum the probabilities of the cases in which the gate should return true. For example, XOR should return true when exactly one of its inputs is true, so:

$$\begin{aligned} p \oplus q &= p(1-q) + (1-p)q \\ &= (p-pq) + (q-pq) \\ &= p+q-2pq \end{aligned}$$

[6]This is true for both XOR and addition.