Superpositional Computation Using Homomorphic Encryption Over the Integers

Alan Wolfe

Blizzard Entertainment

Abstract

This paper introduces an alternate usage case for homomorphic encryption over the integers, to operate on superpositional (ambiguously encoded) bits instead of obfuscated bits. The processing of these superpositional bits gives a superpositional result which can be interpreted for any possible input permutation without having to re-evaluate the circuit. This transforms an arbitrary logical circuit into a constant time decoding process. This differs from a lookup table in that the result can be further transformed by logical operations as desired. This technique likely applies to most other homomorphic encryption techniques and will also likely benefit from many other advancements in that active area of research. Computation time requirements are smaller than homomorphic encryption due to there being no security requirements, but are larger than evaluating a plain text circuit for a specific value.

1. Introduction

In recent years there has been much progress in homomorphic encryption which allows for cryptographically secure computation by performing logical operations on encrypted bits while they remain encrypted.

In this paper we take one homomorphic encryption algorithm and show how the bits may be made ambiguous such that they decrypt to different values based on which key is used to decrypt them.

This ability to make bits ambiguous puts them into a superpositional state which can be processed by digital circuits in the usual homomorphic fashion, resulting in a superpositional answer.

This superpositional answer can be decoded using the previously mentioned keys, giving a result which corresponds to the plain text input values associated with that key. This effectively transforms an arbitrary logical circuit into a constant time decoding process.

While it's true that a simple lookup table performs a similar function, a key difference is that our superpositional result can be processed by other digital logic on the fly so is a dynamic object, unlike a lookup table which is a static entity once created.

Also, as homomorphic encryption becomes more practical, those advancements are likely to also improve this technique to make it more efficient and more practical.

2. Homomorphic Encryption Over The Integers

In [TODO: cite he over integers], a homomoprhic encryption algorithm is defined as follows:
KeyGen: The key is an odd integer, chosen from some interval $p \in [2\eta-1, 2\eta)$.
Encrypt(p, m): To encrypt a bit $m \in \{0, 1\}$, set the ciphertext as an integer whose residue mod p has the same parity as the plaintext. Namely, set $c = pq + 2r + m$, where the integers q, r are chosen at random in some other prescribed intervals, such that 2r is smaller than p/2 in absolute value.

Decrypt(p, c): Output (c mod p) mod 2.

Besides the above operations, there are also the following homomorphic logical operations:
XOR(a,b): Given two encrypted bits a and b, homomorphic XOR is a + b
AND(a,b): Given two encrypted bits a and b, homomorphic AND is a * b

The residue of the modulus grows as you perform logical operations and eventually will become the size of the key itself resulting in the residue overflowing. When this happens, the calculations start giving the incorrect results, so the residue is often referred to as error.

Error grows additively when performing XOR and multiplicatively when performing AND, which makes AND a more expensive operation in general.

By choosing a sufficiently large key, you are able to know that you can process a logical circuit of a specified depth without accumulating enough error to give incorrect results. This is known as leveled homomorphic encryption.

Alternately, using Gentry's novel bootstrapping technique [TODO: cite gentry], the error of the bits can be decreased without having to decrypt and re-encrypt the bits. This is called fully homomorphic encryption with bootstrapping and reduces absolute error.

Another technique for realizing fully homomorphic encryption is to change the key when the error becomes too large. This is called key switching or modulus switching and reduces relative error (error relative to the size of the key). [TODO: cite]

In our method we are going to be using leveled homomorphic encryption.

3. Superpositional Computation: Our Construction

3.1 Leveled Homomorphic Computation

Since our technique has no security requirement, we can use the following simplified setup:

KeyGen: The key p is an integer (odd or even) large enough to support the depth of our circuit.

Encrypt(p, m): To encrypt a bit $m \in \{0, 1\}$, we set c such that ((c mod p) mod 2) = m, or c = pq + m, where p can be any integer we desire.

Decrypt(p, c): To decrypt an encrypted bit, we output (c mod p) mod 2.

Homomorphic XOR and AND operations remain the same.

3.2 Calculating a Single Ambiguous Bit

The obvious question is, how can we encrypt a bit c with two keys p_0 and p_1 such that when c is decrypted with p_0 it gives a 0, and when c is decrypted with p_1 it gives a 1?

Finding the answer to that is equivalent to solving the equations below:

(c mod p_0) mod 2 = 0

(c mod p_1) mod 2 = 1

To make the equations easier to solve, at the cost of limiting ourselves to only finding a subset of all possible solutions, we removed the mod 2 to come up with these equations:

$c \bmod p_0 = 0$

$c \bmod p_1 = 1$

Since p_0 and p_1 can be any value that satisfies the equation, we can pick co-prime values for p_0 and p_1 and use the Chinese remainder theorem to solve for c.

As an example, if we chose 11 and 7 for p_0 and p_1, we are looking for the solution to these equations:

$c \bmod 11 = 0$

$c \bmod 7 = 1$

Where the answer is:

$c \equiv 22 + 77N$

$N \in Z$

So, our ambiguously encrypted bit c can take the value of 22. When we decode it using a key of 11, we see that it is 0, and when we decode it using a key of 7, we see that it is 1. We can also add any multiple of 77 to our encrypted bit c without affecting either the stored values or the error.

3.3 Calculating Multiple Ambiguous Bits

When a digital logic circuit takes in more than one bit of input, the equations get a little more complex. Instead of being able to plug in multiple independently ambiguous bits, you have to make all input permutations ambiguous.

For a circuit which takes two bits of input, the following equations must be solved:

$(c_0 \bmod p_0) \bmod 2 = 0$
$(c_0 \bmod p_1) \bmod 2 = 1$
$(c_0 \bmod p_2) \bmod 2 = 0$
$(c_0 \bmod p_3) \bmod 2 = 1$

$(c_1 \bmod p_0) \bmod 2 = 0$
$(c_1 \bmod p_1) \bmod 2 = 0$
$(c_1 \bmod p_2) \bmod 2 = 1$
$(c_1 \bmod p_3) \bmod 2 = 1$

You can once again drop the mod 2 to get the equations below.

$c_0 \bmod p_0 = 0$
$c_0 \bmod p_1 = 1$
$c_0 \bmod p_2 = 0$
$c_0 \bmod p_3 = 1$

$c_1 \bmod p_0 = 0$
$c_1 \bmod p_1 = 0$
$c_1 \bmod p_2 = 1$
$c_1 \bmod p_3 = 1$

Choosing any co-primes for the keys p_0,p_1,p_2,p_3 you can again use the Chinese remainder theorem to solve for each bit c_0, c_1 independently.

To generalize the process, for N input bits, you will need 2^N co-prime numbers to use for keys, and will need to solve 2^N simultaneous equations for each of the N input bits.

3.4 Superpositional Computation In Action

As a small demonstration of this technique, we will show AND and XOR circuits, which each take in two input bits.

Using the methods described in 3.3 we can come up with the following two superpositional bits as well as four keys.

c_0 = 10660
c_1 = 24090
p_0 = 10
p_1 = 11
p_2 = 13
p_3 = 17

Below is a table showing how the encoded bits decode when using the various key indices. Note that the bits are encoded such that the key index is also the specific input value of the two bit circuit. We have found this convention useful but it is not a requirement.

Key Index	Decrypted Bit 1	Decrypted Bit 2
0	(10660 mod 10) mod 2 = 0	(24090 mod 10) mod 2 = 0
1	(10660 mod 11) mod 2 = 1	(24090 mod 11) mod 2 = 0
2	(10660 mod 13) mod 2 = 0	(24090 mod 13) mod 2 = 1
3	(10660 mod 17) mod 2 = 1	(24090 mod 17) mod 2 = 1

We can do the below to perform our superpositional operations of XOR and AND:

c_xor: 10660 + 24090 = 34750
c_and: 10660 * 24090 = 256799400

Here are the results decoded to show that the operations were successful.

Key Index	Input	Decrypted XOR	Decrypted AND
0	0,0	(34750 mod 10) mod 2 = 0	(256799400 mod 10) mod 2 = 0
1	0,1	(34750 mod 11) mod 2 = 1	(256799400 mod 11) mod 2 = 0
2	1,0	(34750 mod 13) mod 2 = 1	(256799400 mod 13) mod 2 = 0
3	1,1	(34750 mod 17) mod 2 = 0	(256799400 mod 17) mod 2 = 1

For more complex examples, please see the supplemental material of this paper which consists of a small C++ program which implements some basic integer arithmetic circuits.

3.5 Determining Key Size

Using a key that is too small for the depth of a circuit results in the wrong answers when the error becomes too large, so we must be able to have a lower bound on the size of the keys to prevent residue overflow.

Looking at the fact that XOR is addition of encrypted bits and AND is multiplication of encrypted bits, so long as our encrypted bits themselves are non negative values, we know that both operations can either cause the result to be zero (in the case of ANDing against a constant 0 bit), or, the value (residue) will either stay the same or grow.

Using this information, we know that if we pass a constant 1 value in to the circuit (unencrypted) for all input bits, the output of the circuit will represent the largest possible residue that the circuit can have.

If we use this value as the lower bound for the keys, we can then just find the next N co-prime numbers larger than that value and use those as our keys.

While the circuit may operate correctly for smaller keys than found when using this method, we have found that this is a decent way to get a lower bound value which is fairly close to the true lower bound.

3.6 Extensions

In the above, we make ambiguous all possible permutations of input. However, if there are specific input permutations which are not desired (such as having a divisor of 0 in a division circuit), you can omit that permutation when solving the simultaneous equations for a specific superpositional bit.

Omitting permutations can make the superpositional bit encode to a smaller number due to having fewer modulus constraints to satisfy, which will in turn make your computation more efficient. This value effectively becomes a “don't care” value which makes for a more optimal result, much like the same concept in a Karnaugh map.

Another extension of this technique is that you can use unambiguous values in your circuits along side the ambiguous ones. That is, you can XOR or AND against unambiguous bits which have a literal 0 or 1 value.

TODO: cite the other paper that mentions this

For instance, for some encrypted, unknown or ambiguous bit c:

c AND 0 = 0

c AND 1 = c

c XOR 0 = c

c XOR 1 = !c

Since c is an integer, and the homomorphic AND is multiplication, and XOR is addition you can see that this is true, remembering that we are working in mod 2:

$c * 0 = 0$

$c * 1 = c$

$c + 0 = c$

$c + 1 = !c$

We have also found that when working with realistically sized circuits - such as the size of an unsigned division circuit - that taking the modulus of intermediary values with the LCM of the entire key set can keep the size of the answers down, which helps computational efficiency.

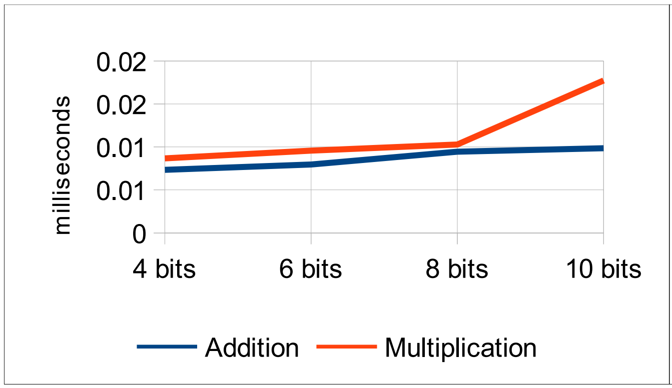
4. Performance Characteristics

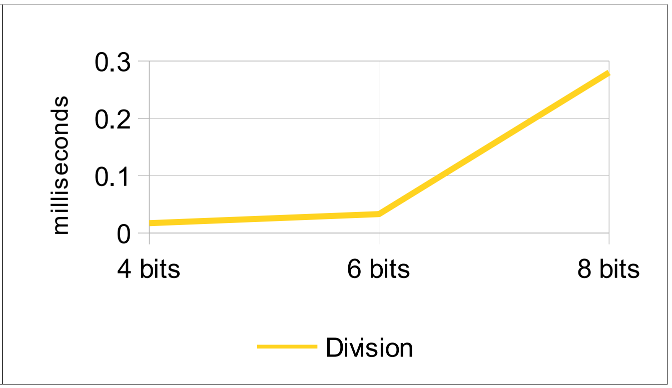
The performance characteristics of this technique are explored below for signed integer addition, multiplication and division circuits of various sized integers. The number of bits indicate the total number of bits input into the circuit, so represent double the size of the underlying integer type being operated on. Calculation times are affected both by the complexity of the digital circuit as well as the total number of input bits. The graphs show the mean of 30 samples.

For the timing samples we used the source code included in the supplemental material, compiled for release x64 using visual studio community 2013, using boost::multiprecision::cpp_int as the multi precision integer class. The machine used was an Intel Core i7-4810MQ CPU 2.8ghz with 16GB of ram. The full sample data is also available in the supplemental material.

Key Lower Bound

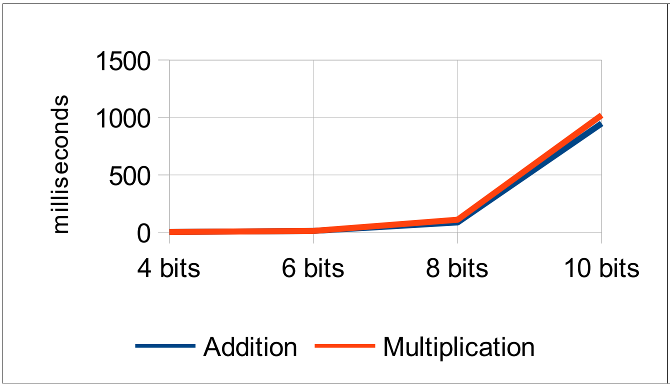
This is the time it takes to process the digital circuit passing a literal "1" value in for all input bits to calculate the lower bound on the keys needed. This calculates the lower bound value of the keys.

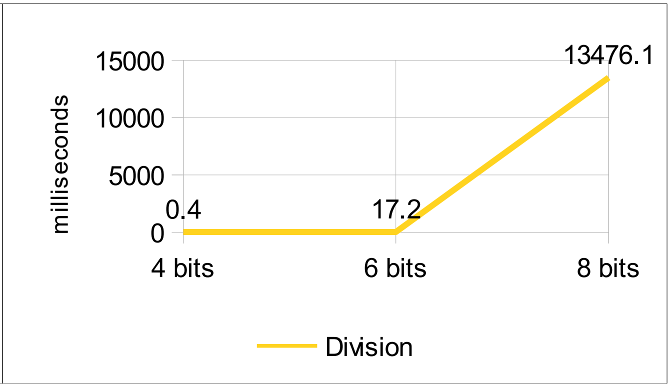




Key Generation

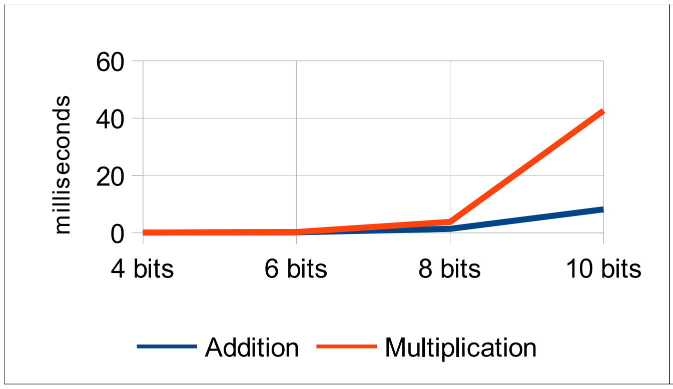
This is the time it takes to generate the 2^N coprime keys, as well as the N superpositional bit values.

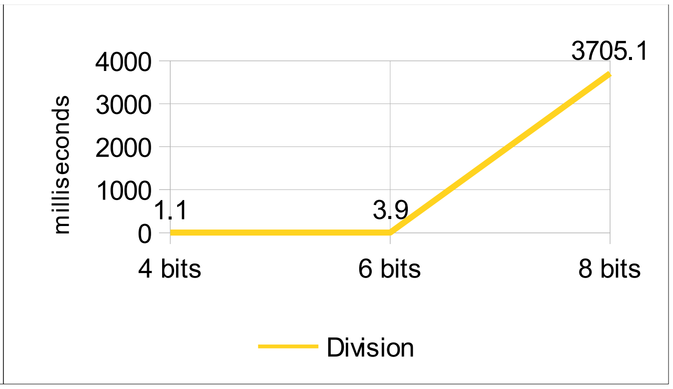




Superpositonal Operation

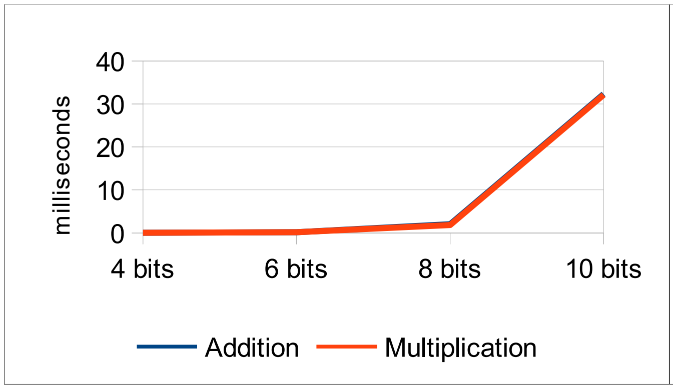
This is the time it takes to do the superpositional operation and get the superpositional result.

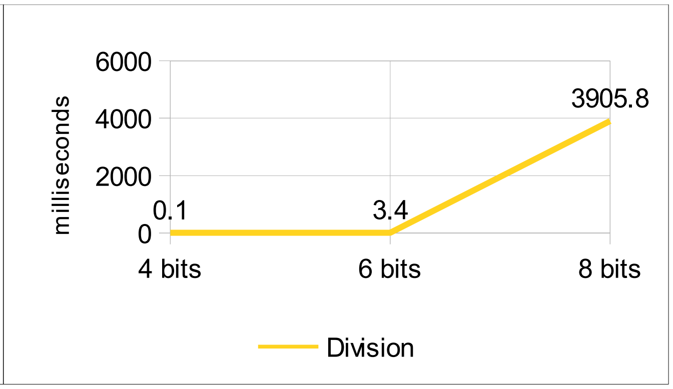




Decoding and Verification

This is the time it takes to loop through all possible inputs, decode those values from the superpositional result, and ensure that each one matches the expected value of the operation.





5. Future Work

One area to look into would be to see which other homomorphic encryption techniques also were able to support superpositional computation. I suspect that these two techniques are dual and one will always be present where the other is.

Another area to look into would be seeing about applying fully homomorphic techniques to superpositional computation to see whether they can apply. Boostrapping seems unlikely to work due to the fact that there is no single key to encrypt and then reduce error by. Modulus switching also seems unlikely to be useful since finding the modulus to switch to involves satisfying 2^N co-prime keys, so is likely to be a very large number in practice. Decoding every permutation and then re-encoding the permutations perhaps with new keys seems like it would be a very slow operation.

Many advancements in homomorphic encryption would also seem to benefit superpositional computation. Since homomorphic encryption is a strongly desired technology and an active area of research, I believe many more advancements will come out of it that should be attempted to be applied to superpositional computation as well.

Better ways to solve for a set of keys, or the superpositional bits would be good improvements to the technique. We looked at infinite co-prime sets to generate the co-prime list of keys, including Sylvester's sequence and Fermat numbers, but found that these numbers grew too large too quickly to be of practical use. Anything that can find smaller numbered solutions more quickly would be a good improvement.

We would also like to explore the numerical properties of the superpositional result of logic circuits to see if they give any hints as to the nature of the logic circuit which produced them. For instance, we have seen evidence that redundant logic gates may be able to be detected in the superpositional result [TODO: briefly explain?] but have yet to explore more deeply.

Lastly, even though this paper drops the security aspect of homomorphic encryption, it would be interesting to understand the usefulness and implications of superpositional computation in situations which do also desire security. It could perhaps be the basis of getting more computational bandwidth out of existing homomorphic encryption techniques when the same circuits must be run for multiple input values.

Index of Supplemental Materials

The included source code has a dependency on the boost 1.5.9 multiprecision library for multi precision ints, but tint.h can be modified to use a different multiprecision int instead.

TODO: this!

TODO: put perf data in as well?

References

TODO: this!