The Karacell 3 Cryptosystem

by Russell Leidich and Stuart Christmas
Copyright (c) 2013 Tigerspike Pte. Ltd.
All rights reserved.
http://karacell.info
Document version 10
Source code build 139
May 21, 2013

0. Background

Karacell 3 (hereinafter "Karacell") is a symmetric encryption algo designed with the express purpose of providing years if not decades of additional privacy as compared to existing popular standards, most notably AES, in scenarios potentially but not necessarily involving the existence of general purpose quantum computers, but with key sizes amenable to typical human memory limitations.

Karacell is not a replacement for key exchange algos such as RSA or Elliptic Curve. Moreover, if in fact its key strength per bit is as strong as the authors hypothesize, then the asymmetric keys involved in symmetric key exchange should be correspondingly longer, after adjusting for estimated cracking complexity.

For the record, Karacell 1, published in 2012, was the original embodiment. Karacell 2 was an experimental step on the way to Karacell (3). Karacell (3) represents the latest development, which incorporates the useful feedback that we received from the cryptographic community.

Prospectively, quantum computers, which stand to accelerate classical algorithms by up to a square root factor, appear to be a matter of engineering, and not a question of basic physics. Indeed, D-Wave of Canada has succeeded in producing a quantum computer which can solve a subset of combinatorial problems with revolutionary efficiency.

Quantum computers aside, it appears that our attention to this potential threat has made Karacell harder to crack on classical computers as well, given the same key size, as compared to AES. While compute time is difficult to estimate, it would appear, based on relative memory footprint, that the difference would be between 1 and 2 orders of magnitude for a key on the order of 100 bits, assuming dedicated hardware logic running the best known cracking algo in either case. But in the quantum case, on account of the empirically slow growth of entangled qubit population (especially in the absence of robust quantum error correction), there may be a spread of years or even decades between the implementation of the first AES cracker,

innovation lab Future-proof your Business
phoenix Multi-platform Technology
Visit www.tigerspike.com to find out more about the Innovation Lab and Phoenix, Tigerspike's tools for unlocking the power of Personal Media
Consulting | UI & UX Design | Multi-Platform Development | Licensing

and the first Karacell cracker – independent of the ensuing cracking time. Karacell, in its crudest sense, is a complexity arbitrage between the small fraction of switches dedicated to logic (as opposed to memory) in most networked machines, and the large (much closer to 1) such fraction in dedicated cracking machines, such that the performance of the latter, with respect to Karacell, is crippled by virtue of the large data tables involved, which are still small enough to fit in the CPU cache and thus do not significantly detract from performance in the course of encrypted communication.

In principle, any algo can be made more quantum resistant by simply increasing the key size. However, doing so may be impractical or even dangerous:

To begin with, long keys overtax human memory, so we invite the temptation to migrate to physical security chips as opposed to typed passwords. Chips can be lost or stolen much more easily. Longer keys also imply longer encryption time, which could result in increased latency for realtime communication systems, for example.

Worse, increasing key size without appropriately scaling other internal aspects of an algo can sometimes result in weakness as compared to shorter keys. However, "appropriate scaling" can be a complicated matter to resolve, involving cryptographic expertise that may not be available. Karacell, to the best of the authors' knowledge, scales itself safely and automatically within its specified key size limits with no such expertise or external intervention required.

It must be emphasized that no encryption algo has ever had its strength proven to any meaningful extent. It's possible that any given algo is much weaker than appears to be the case, simply due to a lack of ingenuity on the part of cryptologists. Thus it would be dangerous to advocate an algo which simply "looked hard", as opposed to one which is explicitly based on a known NP-complete problem. Karacell, for its part, is based on the Subset Sum problem (which is NP) as directly as possible while still meeting the requirements of practical encryption.

But encryption is not enough. With the historical perspective of the Arab Spring in mind, Karacell has been designed to operate in the venue of repressive regimes which seek to prevent encrypted communication, chiefly by means of packet filtering. As such, the Karacell file format has no discernible bitlane bias in its header, body, or footer, to the best of the authors' knowledge, nor any detectable association between 2 or more bits, absent knowledge of the key. (It does, however, have a minimum size, namely, that of its header, which is probably not a significant impediment to deniability. In the worst case, smaller junk files could be sent at random times, in order to achieve a Poisson distribution of file sizes, and thereby obscure the use of Karacell.) Under circumstances where *all* highly entropic packets are dropped for fear that they might be encrypted, one might implement a verbose aliasing system which maps dense binary to plausible uncompressed text, audio, or graphics streams, then converts them back to binary upon receipt. In any event, encryption deniability is imperative on the Internet today.

1. Security Assumptions and Limitations

Keys with most significant bits in positions 120 through 511 bits are supported, with the latter having strength equivalent to 511 bits (most significant bit of 510).

Karacell, and specifically the reference source code indicated on the title page, assumes that the sending and receiving ends of a transaction are physically secure. This is not a trivial consideration, and involves protection against cold boot attacks, memory residue attacks, timing analysis attacks, radio emission snooping, etc. These issues are important with regards to any algo, and are beyond the scope of this paper. Timing analysis in particular could be done by a man-in-the-middle without access to either terminal. However, because Karacell was designed with deep pipelining and parallelism in mind, such analysis would likely only be possible in very constrained circumstances which could be easily thwarted by coarse time quantization.

Authentication is not provided. Instead, Karacell (as distinct from Karacell 1) includes *modular* hash support. The current reference source code provides for no hash, an LMD7 hash, or an LMD8 hash. (These are discussed in detail at http://leidich-message-digest.blogspot.com .) Tigerspike is happy to consider adding support for other hashes in the future, based on developer feedback and the availability of open source code. In any event, the hash of a Karacell file is equivalent to the xor of the hashes of all of its 4KiB blocks (with the last block being as small as 0B and as large as 4KiB). (Chained hashes are disallowed because they serialize execution.) For vastly better security at a minor performance cost, this xor of hashes is then encrypted before being appended to the file.

One could reasonably ask why hash support is necessary at all, when a program could simply append its own hash to the plaintext prior to encryption. While this is certainly a viable strategy, it foregoes the cache efficiency that can be had if a block is simultaneously crypted (encrypted or decrypted) and hashed.

Partly in support of the hash footer, Karacell contains a file header which will be presented in detail later in this document. The header is the only region of the file *not* covered by the hash, and is therefore open to random undetectable modification. (This design decision was made in the interest of minimizing cryption latency, especially in the case of microtransactions.) However, to the best of our knowledge, no such modification could result in either a secrecy violation or a security hole such as a buffer overflow. At worst, a hacker might replace a strong hash with a weak one, or none at all. But in that case, a mechanism is provided by which the caller can see than an unacceptable level of authentication has been provided, and take appropriate action, perhaps including the rejection of the decrypted payload.

2. The Karacell Kernel

Most of the Karacell reference code is devoted to matters actually unrelated to Karacell itself: OS calls, hash computation, random

number management, etc. A more succinct representation of the cryption process is found in mathematica.txt, which simulates demo.c, and can be run directly by cutting and pasting the commands sandwiched between the dashed lines. The Wolfram Mathematica Kernel functionality is all you need.

Karacell for its part has a kernel as well. This small piece of code is where cryption occurs on byte-granular blocks of at most 4KiB – potentially all in parallel. When sender and receiver have a preexisting understanding of the data structures expected, and if authentication is not a requirement, then it's possible to create a private, simple, and fast implementation with neither header nor hash. We refer to such a scheme as "Realtime Karacell", which is a subset of the cryptosystem described herein.

2.1 Cryption Overview

Cryption consists of (1) generating a xor mask, never to be reused anywhere, ever, then (2) xoring said mask to the plaintext or ciphertext.

The xor mask is a function of the key, the initial value ("IV", never to be reused with the same key), and the block number. The key is called the “master key” in the source code, which is a vestige of Karacell 1 terminology.

2.2 The Karacell Table

At the heart of Karacell is the Karacell Table (“K”), which consists of 2^16 bits, exactly half of which being 0s, rearranged by a true random number generator (TRNG). K is too long to reproduce in this document; see KARACELL_TABLE in the source.

2.3 Xor Mask Formation

The xor masks are created by adding various *unique* rotations of K, using carry propagation in the normal integer manner. Specifically,

these are right rotations (toward the least significant bit). Each rotation of K is called a “ring” of K. The number of bits by which to rotate is called the “tumbler” of the ring, analogous to a rotating metal tumbler on a briefcase. Thus 2^16 tumblers exist. For a given key size -- ceil(log2(key)), to be precise -- the number of tumblers is fixed. In the source, this is accomplished via karacell_tumbler_idx_max_get(). This function uses an estimate of cracking complexity discussed in Section 7.3.1

For technical reasons, the number of tumblers is always even. For convenience, it's expressed as 2T, where T is thus half the number of tumblers. In practice, 2T is on [22, 106], corresponding to most-significant key bits of 120 through 511, in a roughly linear fashion.

The mask is thus the sum of the rotations of a large integer. A hacker can easily discover a portion of a mask by simply xoring a piece of known plaintext to the corresponding piece of ciphertext. Given the resulting mask fragment, the task is then to discover the rings which gave rise to the fragment. Given a correct set of rings, it would then be easy to produce the rest of the mask used to encrypt a given block, allowing the hacker to reveal more plaintext. Actually finding the key would be a harder problem, necessitating reversal of the oscillator discussed in Section 2.5, whose output state is substantially unknown even if the tumblers are discovered.

2.4. The Subset Sum Connection

Finding the rings which created the mask amounts to finding the right “few” addends from a set of 2^16 candidates. Finding a wrong set of rings which happen to add up correctly in the known mask regions is useless. Thus to the extent that rotating a single integer to produce many does not simplify the problem, then the process of finding the rings is at least as hard as the Subset Sum problem, as expounded on Wikipedia. Subset Sum is NP-complete, which means that the complexity of the problem scales superpolynomially with tumbler count, at least to the limit of the longest supported key size. Obviously

the rotational relation among the rings does simplify the problem, but to the best of the authors' knowledge, it remains NP-complete.

As mentioned above, no ring is used more than once in a given mask. The algo would be marginally faster in the absence of this provision, as it would eliminate the need for tumbler collision checking. However, it might be less secure, as Subset Sum does not allow the more general case of addends having coefficients other than 0 or 1, which is why Karacell behaves similarly.

It's possible, at certain particular rotations, that selecting a set of tumblers all within a “small” neighborhood, would result in what amounts to a single ring multiplied by a correspondingly small integer. However, for reasons relating to information conservation, the authors do not consider such cases to represent a reduction in complexity beyond what the rotation property already provides.

In the source, karacell_subblock_crypt() forms masks, given a tumbler set from karacell_tumbler_list_make().

2.5 Tumbler Set Construction

Tumbler sets are constructed from pseudorandom iteration of the key (X), the 256-bit IV (V), and the 64-bit block number (B). Tumblers are extracted from least to most significant bit from a 512-bit oscillator state, yielding up to 32 at a time. The same process is repeated as many times as necessary to generate the required number of unique tumblers. Collision filtering is done with a bitmap.

The initial oscillator state (Y) is given by:

Y=X xor (V<<256) xor (R(B)<<192)

where:

“<<” denotes shifting left.

innovation lab
Future-proof your Business
phoenix
Multi-platform Technology
Visit www.tigerspike.com to find out more about the Innovation Lab and Phoenix, Tigerspike's tools for unlocking the power of Personal Media
Consulting | UI & UX Design | Multi-Platform Development | Licensing

and

R(B) denotes the bitwise reverse (not inverse!) of the block number. (This makes for easy extension, if the block number were to someday cross 2^64, while avoiding interaction with the low bits of X, in the interest of more uniform entropy.)

This 512-bit state is then iterated 10 times using the following feedback function, embodied in karacell_marsaglia_iterate():

(A*(Y&((1<<256)-1)))+(Y>>256) -> Y

where:

A=(2^256)-0xE66FAD12.

This amounts to a Marsaglia oscillator, expounded on Wikipedia under "multiply with carry".

In particular, A was chosen such that:

P=(((2^255)*A)-1)

is a Sophie Germain prime, and also the period of Y. Because the period is prime, it's unaffected by the fact that we execute 10 iterations at a time. Because P is ~2^511, we have the aforementioned caveat that a 512-bit key is equivalent to a 511-bit key. Shorter keys are unaffected by this saturation property. The alternative would have been a much more awkward implementation.

The number of iterations between tumbler generations, namely 10, was not selected for its aesthetic appeal in a decimal world! To the contrary, it's after 10 iterations -- neither more nor less -- that the above oscillator reaches essentially maximum entropy. For a precise description of the analysis, please refer to:

http://leidich-message-digest.blogspot.com/2013/02/harder-than-diehard-scintillating.html

The number of unique tumblers which must be produced by the oscillator is a function of the most significant bit position of the key, as given in Section 7.3.

Finally, note the various forms of parallelism amenable to exploitation: (1) among different blocks, (2) among 2T rings, summable in O((2T)^0.5) steps, (3) among the additions of machine integers comprising the mask, which are amenable to SIMD acceleration and somewhat parallel carry propagation, (4) between the Marsaglia oscillator and the mask formation process, (5) between the aforementioned process and hashing, while the plaintext is cache-hot. Above all, xor masks can be fabricated before data arrival. (The PIPELINE build variable toggles such prefabrication, which offers no performance advantage as implemented, but could be tuned to do so.)

2.6 Karacell Kernel Summary

To recap: karacell_marsaglia_iterate(), karacell_tumbler_list_make(), and karacell_subblock_crypt() form the Karacell kernel. They implement a Marsaglia oscillator which selects rotations of an integer to serve as addends for the purpose of xor mask creation. A complete, albeit hashless, cryptosystem could be implemented therefrom.

3. IVs, Mask Reuse, and Replay Attacks

IVs may never be reused with the same key. Karacell makes no further requirements, though in practice, IV management for any algo is a delicate matter. Some suggestions are provided as follows:

The easiest way to adhere to this policy is to generate a new IV prior to each send attempt, whether or not a previous file is being resent. In order to avoid the substantial cost of TRNG invocation, the IV could be derived from a larger entropy pool previously seeded by a TRNG, for example, the 512 bits used in entropy_make() and entropy_iv_make(), neither of which being part of the Karacell specification. Careful

innovation lab
Future-proof your Business
phoenix
Multi-platform Technology
Visit www.tigerspike.com to find out more about the Innovation Lab and Phoenix, Tigerspike's tools for unlocking the power of Personal Media
Consulting | UI & UX Design | Multi-Platform Development | Licensing

attention must be paid to limit cycles, however.

This provision ensures that the probability of mask reuse, which would facilitate statistical attacks, is essentially 0. But it's not sufficient for the sender to ensure nonreuse upon encryption. *Incoming* encrypted files must be subject to the same policy.

In particular, absent some means by which to guarantee that neither sender nor receiver reuse an IV with the same key, it would be possible for an attacker to replay a captured Karacell file to a receiver. The result could be a repeated event, resulting in catastrophe.

In theory, the receiver could maintain an ever-expanding IV database in order to accomplish replay protection. This has obvious drawbacks. A simpler solution is to xor the entropy pools of sender and receiver, in order to form a new entropy pool from which future IVs are to be generated. If one of the peers suffers a power failure, then a new entropy pool can be generated, and the handshake repeated, indefinitely many times, without engendering a meaningful risk of reuse or replay, and supplanting the database with a simple notion of next-expected IV. Failure to meet such expectation would result in a new handshake.

This policy has the added advantage that it permits sender and receiver to anticipate future IVs. This facilitates xor mask prefabrication not only among blocks, but among files. In principle, cryption could be entirely buried in transmission latency, although hashing is still serialized on thereon. However, to the extent that anticipated IVs collide among various peers, file transmission failure could result, but this adds no more risk than an inherently unstable Internet connection already provides. It does, however, demand that resent files be encrypted with different IVs. If an IV arrives which was not anticipated, then the receiver should quarantine or reject the file, and assume that he missed an entropy pool handshake, then attempt to initiate the same, within the limitations of out-of-order arrival uncertainty and random exponential backoff against periodic failure events.

There is also the question of the size of the anticipation horizon, that is to say, the number of IVs which may be tolerably skipped due to packet loss or out-of-order arrival, beyond which a new entropy pool handshake is invoked. But this question is beyond the scope of this paper.

4. The Karacell Header

Karacell files formally begin with a header, although hashless operation is possible without one.

Its format is as follows, with each field described in turn below. Items marked “block0” belong to logical block 0 and are thus encrypted by a xor mask starting at the first byte of the mask.

Offset (bytes)	Size (bytes)	Name
0	32	iv
32	8	block0.must_decrypt_to_zero
40	8	block0.size_following
48	2	block0.build_number_of_creator
50	2	block0.build_number_needed_to_decode
52	2	block0.reserved_zero
54	1	block0.hash_type
55	1	block0.hash_u32_count_minus_1

4.1 header.iv

Apparently, if not actually, 256 bits of true randomness. See Section 3 for details.

The size of the IV has been selected in order to ensure that mask

reuse remains undetectable, even in the regime of presently unimaginably large storage systems. (Reuse detected by a third party, as opposed to merely undetected reuse, is all that matters for statistical attack purposes.) But detection is not as easy as you might think:

http://leidich-message-digest.blogspot.com/2013/02/the-birthday-attack-myth.html

4.2 header.block0.must_decrypt_to_zero

Despite the name, this field does not compromise a xor mask which is used anywhere else. Nor does it provide any more useful data than any other known plaintext fragment would provide. It's merely a performance hint which suggests which key should be used to decrypt the file, with authentication resting entirely with the hash.

Obviously, this field must decrypt to 0. It's necessary because the receiver needs some means by which to rapidly determine the sender's identity with “good” probability (with the hash providing a definitive but comparatively latent answer), while maintaining the deniability requirement. Think of it as a hash of the sender's ID, such that if it's hacked in transit, the only cost is a failed transmission.

There is a small chance that 2 or more peers' keys will result in a 0 in this field, in which case the file must be treated as though lost in transit, in which case it could be resent using a new IV, thereby eventually resolving the conflict. Alternatively, the file could be decrypted with multiple keys, hoping for an eventual match on the hash.

4.3 header.block0.size_following

The number of bytes *after* this field. karacell_header_size_check() and karacell_header_decrypt() police against wraps or invalid values, while accounting for hash_u32_count_minus_1, both of which influencing the implied payload size. Modifying this field can make a hashed file

appear as though it has a weaker hash or no hash at all. However, that would result in an alert to the caller via hash_type_get(), likely culminating in the rejection of the file.

Additional protection against undetected modification of this field is dealt with in Section 6.

4.4. header.block0.build_number_of_creator

The build number of the Karacell release used to create this file. Useful for spreading the word about upgrade availability. One could hack this field to say that an upgrade was available when it actually wasn't, but so what?

4.5. header.block0.build_number_needed_to_decode

The minimum build number needed to decrypt the file. Must be at most build_number_of_creator, as policed by karacell_header_decrypt().

4.6 header.block0.reserved_zero

This field must be ignored by the receiver, except for bits which the receiver understands. It's not trustworthy, which future definitions must take into account. Performance hints would be a good use.

4.7 header.block0.hash_type

A code denoting the type of hash in use, if any. karacell_hash_xor_all_compare() tests the encrypted xor of all hashes for correctness in an abstract manner. But karacell_hash_type_get() allows the caller to determine whether or not the hash is sufficiently strong, so as to disallow the weakening or removal of the hash.

Defined values:

Code	Hash type
0	(Invalid)
1	(No hash)
2	LMD8
3	LMD7

4.8 header.block0.hash_u32_count_minus_1

This field is 1 less than the number of 32-bit integers in the hash. The only exception is that it must be 0 (meaning no hash) when hash_type is 1. It must be checked for consistency against hash_type. If hash_type is unrecognized, then the hash must be ignored (but its size still respected in payload size computation), which the caller can discover via karacell_hash_type_get().

5. Block Topology

Karacell blocks start with 0 at the base of header.block0. Block 1 is the first payload block. (The “payload” is the ciphertext which contains the message, but neither header nor footer.) All payload blocks except the last one must be 4KiB in size. The last one -- the so-called "tail block" handled by karacell_tail_crypt() -- must have a size on [0, 2^12] bytes, such that 0 is only permitted in the case that the payload is null.

Given that the last payload block -- which might be null -- is N. Then the xor of the hashes of blocks 1 through N (hash_xor_all) occupies block (N+1). Block 0 is not hashed.

6. Hash Computation

Hashing occurs in karacell_hash_get(), which then invokes the hash function implied by hash_type.

Hashes are always computed on blocks of size 4KiB, such that all

smaller blocks including null blocks are padded high with 0s prior to hashing. (Accelerated hashing could occur in the event of a short block, but this not currently optimized in the source.)

The hash seeds are of a size known to karacell_header_make() and karacell_header_decrypt(). For a given block, the hash seeds are formed in exactly the same manner as the cryption mask, except that K is first rotated by 2^15 bits (half way). Critically, there is no carry propagation from the 2^15 cryption mask bits, into the next 2^15 bits which are dedicated to hash seed formation; the additions could be performed in parallel. Hence the distinction between TUMBLER_BIAS_CRYPT (0) and TUMBLER_BIAS_HASH (2^15) in the source. Only as many hash seed bits are generated as are required for a particular hash algo.

Block 0 is not hashed, and must therefore be presumed to be compromised. Blocks 1 through N are hashed prior to encryption and after decryption. The resulting hash_xor_all becomes block (N+1), which is then encrypted in the same manner as every other block, with one exception:

The exception is that Y, defined in Section 2.5, is xored with the size of block N (tail_size, on [0, 2^12]) before being submitted to karacell_tumbler_list_make(). (This is accomplished in karacell_tail_crypt() by xoring it to the key, calling karacell_tumbler_list_make(), then restoring the key.) This is done so that the cryption mask of hash_xor_all is hypersensitive to the number of bytes in block N. It's already sensitive to N itself, on account of the dependence of Y on the block number (N+1). However, but for the aforementioned xor, it would not be sensitive to the size of block N, as size_following could be hacked in its low bits, which would otherwise go undetected.

This problem could be eliminated by hashing block 0. However, this is not done, as it would be expensive to do so relative to the tiny size involved and would not eliminate the possibility of a malformed header.

Upon decryption, the same encrypted hash_xor_all is recomputed and compared. The caller must take care to call karacell_hash_type_get() in order to guard against the case in which a hash evaluates correctly, but is not considered to be sufficiently strong for the application at hand.

7. Attack Methodology

7.1 Popular Algos

In 1972, Horowitz and Sanhi produced an efficient algo for the solution of Subset Sum. It runs in O(2^0.5N) for N *possible* addends. Since then, several accelerated approaches have been discovered for special cases. In 2013, Bernstein, Jeffery, Lange, and Meurer produced a quantum computer algo which runs in roughly O(2^0.24N) -- better than a square root factor improvement. Either method would be sufficient to attack Karacell.

7.2 Sparse Horowitz and Sanhi

However, there is a faster approach specifically tailored to Karacell, which we call "Sparse Horowitz and Sanhi". It runs in O((2^16 nCr T)/(2T nCr T)) for 2T tumblers. We have been unable to produce a plausible quantum acceleration. Faster approaches are solicited.

It works like this, as illustrated in the video on the website:

1. Identify a fragment of a xor mask of "useful" size, and the number of rings which were used to create it. Given a partially known plaintext, and some knowledge of key size, this is easily done.

2. Find the sums of all possible combinations of T rings of K -- even though 2T rings are ultimately involved. This costs O(2^16 nCr T) but is totally parallel. (In practice, there is a tradeoff to be struck between memory contention and switches wasted on redundant copies of K.)

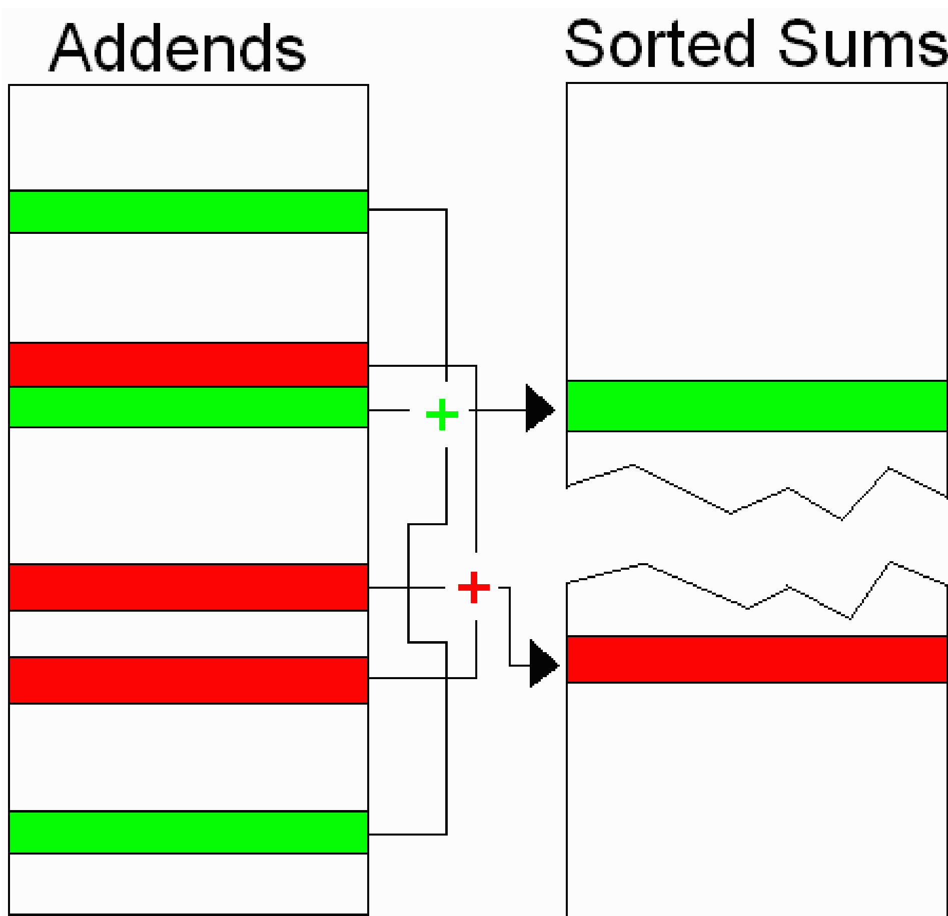
3. Sort the sums in a list. We assume this costs nothing. (In reality,

this is the most costly step, on account of random access to various levels of the cache hierarchy, some of which serialized on other transactions.)

4. Walk the list of sums from both ends, looking for a pair which sums to the mask fragment. We assume this process will end after walking just (1/(2T nCr T)) fraction of the list, on account of the multitude of ways in which pairs of T-sums could add up to the original mask.

5. Expand the xor mask by adding the rings beyond the ends of the mask fragment. Verify that decryption is sensible. If not, continue searching for a valid sum.

Illustration:



7.3.1 Tumbler Expansion Theory

Given B, the bit position of the most significant bit of the key, then we can use the Sparse Horowitz and Sahni complexity to estimate the number of rings, 2T, required such that the latter is a larger space than the key space. In other words, we can find the number of tumblers which will make it harder to reverse engineer the mask than to try every key. Subject to this constraint, we want to minimize 2T in order to minimize cryption latency. (For the sake of simplicity and better performance, odd tumbler counts are not supported.) karacell_tumbler_idx_max_get() accomplishes this task via a lookup table, which was generated according to the method presented in Section 7.3.2.

7.3.2 Tumbler Count Example

Assume that we have a key, X, whose most significant bit position is 315. Thus 2^315<=X<2^316. The Sparse Horowitz and Sahni complexity is such that:

floor(log2((2^16 nCr 60)/(60 nCr 30)))=315

and

floor(log2((2^16 nCr 62)/(62 nCr 31)))=324

so 2T=60 *might* be sufficient, but we can't know that without much more number crunching. So we round up to 2T=62, meaning that we need to generate 62 unique tumblers for the sake of producing each xor mask from block 0 through (N+1).

8. Future Prospects

Quantum computing, swarm behavior, self-assembly, particle physics, and asteroid mining are diverse fields which may eventually cooperate to yield a computer of unimaginable power. In the meantime, (classical) Moore's Law is alive and well, although it has demonstrably

migrated from the megahertz war to the ALU utilization war. Given its exponential nature, it would appear that cryptographic privacy is a linear function of key size, as opposed to a linear function of the key itself. There are fundamental physics limitations to computability, but those limits are far removed from state-of-the-art circuit fabrication, and do not necessarily involve atoms, whether or not entangled. With key sizes already pushing if not exceeding typical human memory, there is a market imperative to maxize the strength of each bit, yet without adding intolerable latency to communication. This is the purpose of Karacell.