Specter-AAL: Theory, Performance & Impact on PKI (RSA) Cryptanalysis

Introduction

Specter-AAL (Specter – Abstract Arithmetics Library) is an open-source arbitrary-precision arithmetic library written in C. Its key purpose is to enable computations with integers of virtually *unlimited* size and precision on any hardware platform. Unlike conventional big-integer libraries that are optimized for powerful 32-bit or 64-bit processors, Specter-AAL is designed to run even on very low-end devices (e.g. 8-bit microcontrollers) with minimal resources. The project's tagline, *"towards breaking cryptography with a calculator,"* emphasizes its potential in cryptographic contexts: by empowering even simple devices to handle huge numbers, Specter-AAL can assist in tasks like RSA encryption, decryption, and even cryptanalysis (e.g. attempting to crack RSA by factoring large moduli) on platforms traditionally far too limited for such tasks.

In this report, we provide a deep technical analysis of Specter-AAL's methodology and mathematical underpinnings. We compare its approach and performance with standard big-integer libraries (such as GNU GMP) and examine how its design contributes to cryptographic computing and RSA cracking efforts. We will present graduate-level mathematical theory to explain *how and why* Specter-AAL works, including formal reasoning about the correctness of its algorithms, and discuss its role in security toolchains.

Core Idea: "String Mathematics" for Arbitrary Precision

Representation: Specter-AAL introduces a novel approach called **"String Mathematics."** In essence, it represents large integers as strings of decimal digits (each digit stored in a char byte) and performs arithmetic on these numbers digit-by-digit, *emulating the way a human would do calculations on paper*. For example, the number one billion (1,000,000,000) would be internally stored as the character array ['1','0','0','0','0','0','0','0','0','0']. Each byte holds a single decimal digit (0–9), and arithmetic operations iterate over these digits. This is fundamentally different from typical arbitrary-precision libraries that store numbers in binary "limbs" (e.g. 32-bit or 64-bit words representing portions of the number in base 2^{32} or 2^{64}). Specter-AAL's decimal string representation is *hardware-agnostic*: it does not rely on the machine's word size or endian conventions for correctness. In fact, it treats the minimum addressable unit (1 byte = 8 bits) as a container for a single digit, avoiding any dependency on larger native integer types. This design makes the library *portable across architectures* (8-bit, 16-bit, 32-bit, etc.) without any special case code or assembly optimization for specific CPUs.

Addition Algorithm: The core algorithms in Specter-AAL mirror the standard "grade-school" algorithms taught for manual arithmetic. To add two large numbers, Specter-AAL adds corresponding digits one by one, from the least significant digit (units place) to the most significant, propagating a carry. Formally, if we have two n-digit numbers (in base $B=10$) represented as sequences of digits $x = (x_{n-1}x_{n-2}\dots x_0)$ and $y = (y_{n-1}y_{n-2}\dots y_0)$ (where x_0 and y_0 are the least significant digits), the algorithm computes their sum $s = x+y$ as follows:

1. Initialize carry $c = 0$.
2. For each position i from 0 to $n-1$:
    - Compute $r_i = (x_i + y_i + c) \mod 10$ (the result digit at position i).
    - Update carry $c = \lfloor (x_i + y_i + c)/10 \rfloor$ (either 0 or 1, since $x_i+y_i+c \le 9+9+1=19$).
3. After the final digit, if $c=1$, append an extra digit $r_n = 1$ at the next position (overflow carry).

The result s is then given by the digits $(r_n r_{n-1}\dots r_0)$. This is exactly the usual addition procedure. A simple inductive **proof of correctness** can be given: as a loop invariant, after processing the i-th digit (for $i=0,1,\dots,n-1$), the partial result $R_i = \sum_{j=0}^{i} r_j 10^j$ equals the sum of the partial inputs $\sum_{j=0}^{i} x_j 10^j + \sum_{j=0}^{i} y_j 10^j$, and a carry c (0 or 1) is held such that $\sum_{j=0}^{i} x_j10^j + \sum_{j=0}^{i} y_j10^j = R_i + c\cdot 10^{i+1}$. This invariant is true initially ($i=-1$, with $R_{-1}=0$ and $c=0$) and is maintained at each step by the way r_i and c are computed. By induction, after the final digit $i=n-1$, we have $x+y = R_{n-1} + c\cdot 10^n$, and the algorithm adds the carry c as a new digit r_n, yielding the exact sum $R_{n-1} + c\cdot 10^n = \sum_{j=0}^{n} r_j 10^j = s$. Thus, the digit-by-digit method produces the correct sum. This is a well-known fact in elementary number theory – the standard multi-digit addition algorithm is provably correct for any base. The *advantage* of Specter-AAL's implementation is that it operates using only small, constant-size data (individual bytes for digits and a tiny constant-time operation for each digit), so it can handle arbitrarily large numbers so long as there is memory to hold the digit array. There is no fixed upper limit on the magnitude of numbers aside from available memory – adding one more digit extends the range by a factor of 10.

Subtraction and Other Operations: Similarly, subtraction in Specter-AAL is implemented via the usual borrow method (subtract each digit with borrow

propagation). If the result goes negative, a borrow of 1 (equivalent to 10 in that digit's place value) is taken from the next higher digit. The correctness of the subtraction algorithm is also ensured by an inductive argument analogous to addition (each step maintains the correct difference with a borrow that is either 0 or 1). In principle, all basic arithmetic operations can be implemented in this character-by-character framework – multiplication by summing partial products, division via long division, etc. Indeed, the project's goal is to support *"all kinds of numbers and virtually all operations"* on even a small microcontroller. As of the latest update, Specter-AAL is in an early alpha stage: the fundamental infrastructure is in place, but only addition and subtraction are fully implemented at the time of writing. Multiplication, division, modular arithmetic, and other operations are planned for future releases. Nonetheless, we can discuss how these would work in theory under Specter-AAL's methodology:

- **Multiplication:** Specter-AAL will use the long multiplication algorithm ("grade-school multiplication"). If x and y are represented as digit sequences, their product can be computed by multiplying each digit of x by each digit of y and accumulating the results at the appropriate shifted positions. Formally, one can express the product as $x \times y = \sum_{i=0}^{n-1}\sum_{j=0}^{m-1} x_i y_j , 10^{,i+j}$ for an n-digit x and m-digit y. The algorithm will iterate over i and j, compute $x_i y_j$ (a single-digit times single-digit multiplication, which fits in at most two digits), and add it to the partial result at offset $i+j$. This approach is $O(N^2)$ in time complexity for N-digit numbers, which is the same as the complexity of the classical algorithm. Although $O(N^2)$ is not asymptotically optimal, it is conceptually simple and requires only basic operations (small multiplications and additions), making it suitable for constrained hardware. The *mathematical correctness* of the long multiplication algorithm follows from the distributive law of multiplication over addition – it can be formally proven by double induction on the number of digits that the procedure yields the correct product, since it effectively computes the above summation formula term by term.
- **Division:** A similar logic applies to division (via long division or repeated subtraction). The algorithm would subtract multiples of the divisor from the dividend digit-by-digit, tracking the quotient digits. This is more involved to implement correctly (due to trial-and-error in choosing each quotient digit), but it's a well-defined procedure that does not fundamentally require hardware large-integer support beyond what Specter's framework provides.

- **Modular Arithmetic:** Operations like modular addition, multiplication, and exponentiation (crucial for RSA encryption/decryption) can be built on top of these basic operations. For instance, modular multiplication can be done by performing the multiplication and then dividing by the modulus (keeping the remainder). Specter-AAL's ability to handle huge integers means it can support RSA key sizes (1024-bit, 2048-bit, etc.) on any platform, given enough memory.

Memory and Precision: In Specter-AAL's paradigm, *precision is limited only by memory*. Because each digit takes one byte, an N-digit decimal number consumes N bytes of memory. There is a slight space overhead here: storing numbers in base 10 is less compact than base 2^{32} or 2^{64} as used by other libraries. For instance, a 2048-bit RSA modulus (about 616 decimal digits) would require 616 bytes in Specter-AAL, whereas a binary 32-bit limb representation would use roughly 64 limbs (256 bytes) for the same number. This is a trade-off for the sake of simplicity and portability. However, the memory overhead is linear and modest, and Specter-AAL is explicitly optimized for environments with *small RAM*. It avoids large, precomputed tables or complex data structures. In fact, the Specter approach boasts that it can perform operations on extremely large numbers using only a few megabytes of memory and minimal CPU involvement – demonstrations have shown it handling numbers with millions of digits using on the order of 5 MB of RAM and only a few percent of CPU time. This low CPU utilization is a result of efficient C pointer manipulation and the fact that the algorithm's workload is primarily simple additions on bytes (which modern CPUs handle easily, often faster than memory access). The *time* taken for an operation in Specter-AAL grows linearly with the number of digits for addition/subtraction, and (once implemented) roughly quadratic for multiplication. In practice, this means that on extremely large inputs, Specter-AAL's methods will be slower than advanced algorithms – but the library opts to prioritize guaranteed correctness and universality on any hardware over chasing maximum speed on high-end systems.

Formal Underpinnings: The theoretical foundation of Specter-AAL's "string math" approach lies in classical positional numeral systems. Any non-negative integer can be represented in base 10 (or any base B) as a finite sequence of digits. The operations of addition, subtraction, etc., are well-defined on these representations and have well-known algorithms that are taught in elementary arithmetic. These algorithms are in fact the basis of *multi-precision arithmetic* in computer science. Specter-AAL essentially implements these algorithms at the lowest common denominator of hardware (the byte), ensuring that nothing about the implementation assumes a particular machine word size. In doing so, it treats the computer similar to a human clerk performing arithmetic: the "proof of

correctness" for each operation reduces to the mathematical proof that these grade-school algorithms correctly implement the arithmetic operations on integers. Such proofs are standard – for example, the correctness of multi-digit addition with carry can be rigorously proved by induction on the number of digits, and similar inductive or algebraic proofs exist for multiplication and division. By adhering to these simple, proven algorithms, Specter-AAL's core is mathematically straightforward: it leverages the structure of $\mathbb{Z}$ (the integers) under addition and multiplication (an infinite abelian group under addition, and monoid under multiplication) and the fact that base-B representations and their digit-wise algorithms constitute an *algorithmic realization* of those algebraic operations. In summary, the core idea of Specter-AAL is to forsake any reliance on fixed-size machine arithmetic and instead operate directly on the digit sequences that represent numbers – a conceptually simple approach that scales to any size and is backed by the fundamental theory of positional notation.

Comparison to GMP and Conventional High-Performance Libraries

Specter-AAL's methodology stands in stark contrast to that of standard big-integer libraries like the GNU Multiple Precision Library (GMP). Here we compare them in terms of design philosophy, algorithms, and performance:

- **Hardware Exploitation vs. Hardware Independence:** GMP is engineered to exploit the full power of the underlying hardware. It uses the machine's word size as the basic unit (limb) of representation and frequently leverages assembly-optimized routines for specific CPU architectures (x86, ARM, etc.) to speed up multi-precision operations. This yields excellent performance on high-end processors, as GMP can perform computations using native 64-bit arithmetic and even vector instructions. However, it makes GMP *hardware-dependent*: for each supported architecture, GMP includes hand-tuned assembly code, and the performance (and even feasibility) of GMP on very small architectures (e.g. 8-bit microcontrollers) is poor or nonexistent. In fact, GMP assumes at least a reasonable C environment and has not been designed for tiny microcontrollers with a few kilobytes of RAM. Specter-AAL, on the other hand, is *hardware-agnostic*. It is written in pure ANSI C with no assembly, and it avoids assuming any specific endianness or word size. This means Specter-AAL can be compiled for virtually any platform – from a 64-bit server CPU down to a small 8-bit AVR chip – and it will function correctly and uniformly. The library's operations use simple C logic on bytes, making it truly portable. The trade-off is that Specter-AAL doesn't leverage, say, 64-bit arithmetic to speed up a 64-bit calculation; it will

treat that as 8 bytes and loop through them. Thus, on a PC, Specter's approach sacrifices some performance compared to a highly tuned library like GMP, but on a microcontroller, Specter-AAL may be one of the few viable options to handle big integers at all.

- **Algorithmic Complexity:** The difference in algorithmic approach leads to different performance profiles. Specter-AAL, as described, uses the classical algorithms for multiplication, etc., which have quadratic time complexity in the number of digits ($O(n^2)$ for multiplication, $O(n)$ for addition of n-digit numbers). GMP and similar libraries employ more advanced algorithms to achieve better asymptotic complexity on large inputs. For example, GMP will switch to **Karatsuba multiplication** (with complexity about $O(n^{1.585})$) for moderately large numbers, then to **Toom-Cook** methods and ultimately to the **Schönhage–Strassen FFT-based multiplication** for extremely large numbers, achieving roughly $O(n \log n)$ time. These algorithms use divide-and-conquer and number-theoretic transforms (Fourier transforms) to multiply large operands much faster than the naive $O(n^2)$ method for huge n. However, they come with significant overhead: FFT-based multiplication, for instance, requires computing in complex number space or large integer modulo and allocating large auxiliary arrays (scratch space) proportional to the size of the inputs. On memory-constrained systems, these methods may be impractical. Specter-AAL explicitly avoids such approaches. Its philosophy is that *segmenting* the computation into chunks (as FFT multiplication does by splitting numbers into many parts) "does not always work without huge overhead in RAM and CPU utilization" on small devices. Thus, Specter sticks to the simpler $O(n^2)$ method which uses less extra space and simpler operations, trusting that for its intended use-cases (numbers that are large but still manageable on low-end hardware), this is acceptable. In summary, GMP minimizes *CPU time* at the expense of *complexity and memory*, whereas Specter-AAL minimizes complexity and *memory overhead*, at the expense of more CPU operations for very large sizes.
- **Performance in Practice:** On a modern desktop CPU, GMP's optimizations make it vastly faster than a naive implementation for large numbers. For example, multiplying two 10,000-digit (~33,000-bit) numbers might be hundreds of times faster with GMP (using FFT) than with a purely quadratic algorithm. Specter-AAL on such a platform would run significantly slower for such a task. **However, on an 8-bit microcontroller**, GMP either cannot run or would revert to extremely inefficient operations (since it lacks 8-bit assembly support and the

hardware itself cannot easily handle large words). Specter-AAL can run on an 8-bit CPU and will use the CPU's native capabilities to do the job digit by digit. In that environment, there is no competition from FFT or Karatsuba – those algorithms are not practical due to overhead, and the simple method may actually be *the fastest feasible method on that hardware*. Thus, Specter-AAL shines in constrained environments. It has been built with an emphasis on *low RAM usage and low CPU usage* per operation. Empirical tests by the authors show that adding or even multiplying very large numbers on a PC can be done with only a few percent of CPU usage (implying the operations complete quickly or at least do not tax the processor heavily) and memory usage roughly linear in the operand size (e.g. a few megabytes for millions of digits). This indicates that the implementation is efficient in terms of constant factors as well – using plain C and pointer arithmetic effectively. Meanwhile, GMP's approach might allocate larger buffers and use more CPU to achieve a faster runtime; the differences become a matter of scale and context.

- **Portability and Integration:** Another point of comparison is how these libraries integrate into projects. GMP is a large library with a specialized interface; it requires either static linking or dynamic linking, and its compiled size and memory footprint might be too high for small, embedded projects. Specter-AAL, being essentially just a C source (and shell/batch scripts for setup), is lightweight. It can be included directly in an embedded software project, and because it uses no assembly, it's easier to verify and maintain across platforms. The Specter-AAL documentation argues that libraries like GMP "reside on a compiler to be 'portable'" but are not truly universal because they rely on specific hardware features and extensive per-platform code. In contrast, Specter-AAL's codebase is unified – the same code runs everywhere, making it easier to ensure consistency and correctness across different deployments. This can be crucial in security-sensitive applications, where one needs to trust that the arithmetic behaves identically on all devices.

To summarize the comparison: **Specter-AAL VS GMP** is a classic example of simplicity and universality versus complexity and peak performance. Specter-AAL's string-based arithmetic has higher asymptotic cost, but it guarantees the ability to compute with big numbers "*anytime, anywhere*" even on devices with minimal resources. GMP's algorithms are mathematically sophisticated (employing number-theoretic optimizations developed over decades) and are unbeatable on large computations in a well-equipped machine, but that power comes with the assumption of a capable environment. Specter-AAL essentially

trades off some of that power to gain flexibility and *predictability* – its performance might be slower on a PC, but on an MCU it might be the only option, and one that still works within the tight memory/CPU budget.

From a mathematical standpoint, both approaches produce the same correct results. The difference lies in *how* they achieve it: GMP might use the convolution theorem (via FFT) to multiply polynomials (representing large numbers) quickly, whereas Specter-AAL sticks to the basic distributive calculation. GMP might unroll loops and use CPU carry flags directly in assembly, while Specter-AAL uses a C loop with explicit carry management in software. These choices affect not only performance but also factors like side-channel behavior (timing characteristics) and code complexity. Notably, the straightforward nature of Specter-AAL could be an advantage in security auditing—its correctness hinges on very basic, well-understood algorithms, reducing the risk of subtle bugs that sometimes occur in highly optimized code. Meanwhile, GMP's heavy optimization has occasionally led to platform-specific bugs or the need for careful maintenance as compilers and CPUs evolve. Thus, Specter-AAL complements the existing libraries by filling a niche where reliability, clarity, and portability are paramount, and ultimate speed is secondary.

Applications in Cryptography and RSA Cracking

Arbitrary-precision arithmetic is the backbone of most public-key cryptography. RSA relies on operations modulo a large composite number (the RSA modulus, typically 2048 bits or more) and requires generating and handling large prime numbers. Specter-AAL's ability to handle *huge* integers on any device opens up several important applications in IT security and cryptanalysis:

- **RSA Key Generation and Encryption on Low-End Devices:** RSA key generation involves choosing two large primes (e.g. 1024-bit each) and multiplying them to form the modulus N. It also requires computing the public and private exponents (involving modular inverses). These steps are impossible with fixed-size machine integers but straightforward with a big-integer library. With Specter-AAL, one could generate RSA keys on a microcontroller or other constrained system. For example, imagine a sensor or an IoT device that needs to create its own RSA key pair for secure communications – with Specter-AAL integrated, it can generate random large numbers, test for primality (via probable prime tests that rely on big arithmetic), and compute exponents, all in situ. Similarly, the device could perform RSA encryption and decryption (which require modular exponentiation with a large exponent and modulus). Although the operations would be slower than on a PC, they would be *feasible*. This can enhance security toolchains by enabling encryption in scenarios

where using a full cryptographic library isn't possible due to resource constraints. Specter-AAL essentially extends the reach of strong cryptography into the realm of very cheap hardware.

- **Distributed or IoT Cryptanalysis:** In the context of *RSA cracking* (i.e., attempting to factor RSA modulo or break RSA keys), one usually thinks of massive computational resources: clusters, GPUs, specialized hardware (or now, even quantum computers in the future). Specter-AAL proposes an intriguing angle: what if many low-power devices (even as simple as calculators or Arduinos) could be harnessed for number-theoretic computations? Because Specter-AAL lets such devices manipulate large numbers, it theoretically allows them to participate in attacks like trial division, Pollard's rho algorithm for factoring, Pollard's $p-1$ or elliptic curve factorization, and even the early stages of the General Number Field Sieve (GNFS) which involve arithmetic on large integers. For instance, **Pollard's Rho algorithm** for factoring a number N involves iterating a pseudorandom sequence modulo N and computing $\gcd$ of differences – all operations that require big integer support. A network of IoT devices, each running Specter-AAL, could split the workload by trying different random seeds or parameters. While a single microcontroller is *far slower* than a PC, the sheer number of microcontrollers (and their low cost) could make up for some of that gap in a massively parallel setting. This approach to cryptanalysis is unconventional, but it highlights Specter-AAL's contribution: it lowers the barrier to entry for participating in big-number computations. Cryptanalytic research often involves testing many possibilities or searching a huge space (e.g. trying many factorization paths or attacking multiple keys); with Specter-AAL, even cheap devices can join the effort.
- **RSA Vulnerability Scanning and Toolchain Integration:** Beyond full-fledged factoring, there are many practical attacks on RSA that require big-integer math. For example, a common vulnerability is **shared prime factors**: if two RSA moduli $N_1=pq$ and $N_2=p'r$ happen to share a prime factor (say $p = p'$), then $\gcd(N_1, N_2) = p$ reveals the factor and breaks both keys. Security auditors often take large collections of RSA public keys and compute pairwise GCDs to find such cases. This involves computing GCDs of enormous numbers (hundreds or thousands of bits) – a task requiring arbitrary precision. Specter-AAL could be used to implement such a GCD scan on an embedded system or within a custom security appliance. The **Euclidean algorithm** for GCD can be done with just subtraction and modulo operations; even if division (mod) isn't implemented directly, one can use subtraction iteratively (which is less

efficient but feasible with big integers). By using Specter-AAL, one could, for instance, build a firmware for a hardware security module that validates that a newly generated RSA key isn't trivial (e.g., checking it against a database of known weak keys or factors). Another example is **Fermat's factorization method**, which tries to express an RSA modulus N as a difference of two squares: one finds a and b such that $a^2 - b^2 = N$, which implies $N=(a-b)(a+b)$ gives the factors. Fermat's method requires computing successive squares and differences – again big-integer operations – and can be effective if the prime factors p and q of N are close to each other. Specter-AAL would allow a simple device to attempt Fermat's method on a given N (perhaps discovering a factor if p and q differ by only a small amount). This is relevant in cryptanalysis when keys are generated poorly (primes too close or following a pattern).

- **Cryptographic Protocol Analysis:** Modern security protocols sometimes use big integers in various ways (not just RSA but also Diffie–Hellman key exchange, DSA, ECDSA on curves over large prime fields, etc.). While elliptic curve cryptography typically involves fixed-size 256-bit numbers (which could be handled without arbitrary precision if coded specifically), protocols may require handling of unusual sizes or operations (for instance, some privacy-preserving protocols use extremely large RSA moduli for homomorphic encryption or use multiparty computation with large integers). Specter-AAL can serve as a general big-number engine within custom security tools that run on limited hardware. For example, an offline password recovery tool might use big integers to test cryptographic hashes or derive keys; if one wanted to deploy such a tool on a portable device, Specter-AAL would be needed for any operations exceeding standard data sizes.
- **Educational and Research Use:** On the flip side of offensive cryptanalysis, Specter-AAL is useful in educational contexts to *demonstrate cryptographic algorithms on simple hardware*. A student or researcher could load Specter-AAL on a microcontroller and observe how RSA decryption works step by step, or measure how the performance scales with key size on a modest CPU. This can give insight into why certain key sizes are considered secure. For instance, if factoring a 256-bit RSA key takes a microcontroller only a few hours using trial division or Pollard's rho (which one could try with Specter-AAL), that illustrates why 256-bit RSA is utterly insecure. If a 512-bit key takes perhaps weeks on that device, still feasible, it shows why 512-bit was broken in the 1990s. And if a 1024-bit key doesn't finish factoring in any reasonable time, it

aligns with the current understanding that 1024-bit RSA is hard (though not impossible with large clusters). Thus, Specter-AAL can help concretely demonstrate cryptographic strength by enabling experiments on everyday hardware.

In all these applications, the **meaningful contribution** of Specter-AAL is that it *extends the capability of performing big-number computations to environments where it wasn't previously possible*. This can enhance security toolchains by allowing more flexibility in where and how cryptographic computations occur. For example, a security researcher could include Specter-AAL in a forensic toolkit on a USB thumb drive or a tiny single-board computer to analyze encrypted data or test keys on the fly, without needing a full laptop. It could also be integrated into custom hardware implants or devices that need to process encrypted communications – giving them the power to handle RSA/AES keys of arbitrary size.

Regarding RSA **cracking** specifically: while Specter-AAL does not magically reduce the theoretical complexity of factoring (which remains super-polynomial), it lowers the *practical barrier* to setting up experiments and participating in collaborative efforts. In cryptography, there's a notion of *"if a device as simple as X can break your crypto, then that crypto is too weak."* Specter-AAL helps illustrate such points. If we can show that a network of inexpensive chips running this library can factor a 512-bit RSA key in a short time, that reinforces the need for using larger keys. If someday optimizations or improved algorithms (maybe combined with Specter's approach) allow a calculator to break 768-bit RSA, that would be a significant cryptanalytic milestone. Thus, Specter-AAL is a facilitator for cryptanalysis: it provides the raw arithmetic layer upon which attacks can be built, even on small-scale hardware, thereby democratizing the ability to experiment with cryptanalysis. The project's focus on cryptography is evident from its documentation, and its development is guided by use cases in cryptographic research and engineering.

Technical Rationale and Theoretical Justification

Why does Specter-AAL "work" at a fundamental level? The answer lies in the solid bedrock of number theory and computer science that underpins multi-precision arithmetic. Specter-AAL's operations are essentially implementations of arithmetic in base 10 (though base 256 could equally be used with byte digits – base 10 is chosen to mimic human calculation). The correctness of these operations is guaranteed by the same reasoning that guarantees, for example, that any integer addition can be done by adding digits with carries. This is ultimately because a base-B representation is essentially a sum of powers of B: for a number X with digits $d_k d_{k-1}\dots d_0$ (where each d_i is in

$0,\dots,B-1$), we have the exact formula $X = \sum_{i=0}^{k} d_i B^i$. The algorithms used by Specter-AAL are doing nothing mysterious – they are applying the standard algorithms which correspond to the following mathematical equalities:

- *Addition:* $\sum_{i=0}^{n-1} x_i B^i + \sum_{i=0}^{n-1} y_i B^i = \sum_{i=0}^{n-1} (x_i+y_i) B^i$. If some (x_i+y_i) exceeds $B-1$, we write $(x_i+y_i) = c_{i}B + r_i$ with $0 \le r_i < B$ (here c_i is 0 or 1 for decimal addition, representing the carry). Then the sum becomes $\sum_{i=0}^{n-1} r_i B^i + \sum_{i=0}^{n-1} c_i B^{i+1}$. The algorithm's logic ensures that c_i becomes the carry added into the next digit's sum. By telescoping this process, all carries propagate, and we end up with $\sum_{i=0}^{n-1} r_i B^i + c_{n-1}B^n$, which is exactly $\sum_{i=0}^{n} r_i B^i$ (with $r_n = c_{n-1}$). This final sum equals the true arithmetic sum $X+Y$. This argument can be made rigorous by induction, as discussed earlier. The *existence* of a carry sequence c_i that makes this work is guaranteed by the division algorithm (any integer sum x_i+y_i+c can be uniquely decomposed as $B \cdot c_{i} + r_i$ with $0 \le r_i < B$). Thus, the algorithm will always produce a correct digit r_i and a valid carry c_i. Specter-AAL simply implements this on actual data structures, and its correctness is a direct consequence of this number-theoretic fact.
- *Multiplication:* The grade-school method relies on the distributive law: $(\sum_{i}x_i B^i)\cdot(\sum_{j}y_j B^j) = \sum_{i,j} x_i y_j B^{i+j}$. This double sum naturally groups into "partial products" $x_i y_j B^{i+j}$. The algorithm that multiplies each digit of one number by each digit of the other and accumulates the result in the proper shifted position is effectively computing this double sum. The correctness is ensured by the commutativity and distributivity of integer multiplication. Specter-AAL's eventual multiplication routine will thus be grounded in this formula. Formally, one could prove its correctness by double induction (first on the number of digits of x, then y), or by observing that multiplying by a single digit and adding (the standard approach of building the result one digit of y at a time) each produce correct intermediate results by the induction hypothesis. The result is the standard proof found in any elementary algebra text that the long multiplication algorithm yields the product. This is again a testament to relying on foundational arithmetic properties.
- *Modular arithmetic:* If implementing $a \bmod m$ (the remainder of a upon division by m), Specter-AAL would rely on either a division

algorithm or repetitive subtraction. The correctness of these approaches is guaranteed by the definition of the modulo operation in arithmetic: there is a unique quotient q and remainder r such that $a = q\cdot m + r$, $0 \le r < m$. An algorithm like long division is an effective procedure for finding q and r, known to be correct by induction on the number of digits as well. Specter-AAL doesn't reinvent any wheels here; it simply would carry out the same long division steps that one would do on paper (which are known to produce the correct quotient and remainder by the theory of integer division).

In summary, *the theoretical justification for Specter-AAL's functioning is the robust theory of positional representations and the classic algorithms for arithmetic*, which were proven correct long ago. Specter-AAL's innovation is not in inventing new mathematics, but in **applying the existing mathematics in a context (low-end hardware) where it is seldom used**. It treats the limitation of hardware (say an 8-bit CPU that can only directly add numbers up to 255) not as a barrier but as merely a constraint on the digit size. By choosing a digit size (base) that the hardware can handle – in the case of an 8-bit CPU, base 10 or base 256 are both handleable (base 256 fits in a byte without carry, base 10 fits with carry logic) – Specter-AAL bypasses the need for any larger primitive. It effectively *virtualizes* a big calculator inside a small one. This is akin to how one could, in theory, perform 64-bit arithmetic on a CPU that only has 8-bit registers by doing it in 8-bit chunks; Specter-AAL just generalizes this to arbitrary length and emphasizes decimal digits for broad compatibility. The developers explicitly note that an 8-bit ALU (Arithmetic Logic Unit) on a microcontroller "would not have the capacity" to directly compute with huge numbers, but by breaking the numbers into manageable segments, AAL can compute what the hardware alone cannot. The caveat they point out is that splitting into *large* segments (like using 32-bit chunks and advanced algorithms) complicates things on low-end systems. So they choose the smallest reasonable segment (the byte, or even nibble conceptually) to keep the algorithm simple.

It's worth noting that while Specter-AAL currently uses base 10 digits, nothing in theory stops one from using base 256 (each byte holds 0–255). In fact, using base 256 would more fully utilize each byte's range and reduce the number of digits needed (roughly by a factor of 2.5, since $256 \approx 10^{2.408}$, one base-256 digit holds about 2.4 decimal digits worth of value). The choice of decimal is likely for human-readability and ease of debugging (and the conceptual tie to "like a human doing decimal math"). From a theoretical standpoint, base 256 arithmetic is just as valid and follows the same proofs of correctness. The implementation would be slightly different (the carry can be up to 255+255=510, so could be as high as 2, and each digit's addition would involve mod 256 and

carry = floor(sum/256)). Specter-AAL's documentation indicates that it sticks to char and uses the fact that C's char can act as a small integer. The upper 4 bits of the byte (if using only 0–9) are wasted, but the simplicity of coding and verifying decimal operations might have influenced this design choice. In any case, the approach is flexible, and the *abstract arithmetic* principles apply regardless of base. This is why they call it an **"Abstract Arithmetic Library"** – it abstracts the arithmetic away from the hardware details (bit-length of registers, etc.) and implements it in a way that could conceptually work on any abstract machine that can process bytes.

Conclusion

Specter-AAL represents a fundamentally different approach to big-number computation, prioritizing *universality and security* considerations over raw speed. Using "string mathematics" – performing arithmetic at the digit level – it achieves a level of hardware independence that is rare in this domain. We have seen that the mathematical theory backing Specter-AAL is sound and rooted in the basic properties of integer representations and algorithms. The library's methods can be rigorously proven correct using classical techniques (loop invariants, induction) on the operations in each base, giving confidence that even though it runs on arbitrary hardware, it will always produce reliable results.

Compared to highly optimized libraries like GMP, Specter-AAL trades asymptotic performance for portability and low resource usage. In scenarios with powerful hardware and extremely large operands, Specter-AAL is not meant to outperform GMP – rather, its niche is in scenarios where GMP either cannot run or is impractical. Our analysis showed that on microcontrollers or embedded systems, Specter-AAL opens the door to performing cryptographic computations (like RSA key generation, encryption, and even some forms of cryptanalysis) that were previously out of reach on such devices. In the context of RSA cracking, while Specter-AAL does not offer a new algorithmic shortcut to factoring, it enables creative new strategies – such as distributed attacks using low-end nodes – by equipping every node with the big-integer handling capability necessary for the job. It thereby enriches the cryptographic and security toolkit: for example, allowing security practitioners to test RSA implementations or vulnerabilities directly on the devices in question, or to integrate big-number operations into custom security firmware without relying on large external libraries.

In a broader sense, Specter-AAL underscores an important principle in cryptography and computer engineering: **the feasibility of an attack or computation is not solely a function of algorithmic complexity, but also of resource accessibility.** By drastically lowering the resource requirements to do

big-number math (running in 2 KB of RAM on an 8-bit chip, for instance), Specter-AAL shifts the landscape of who can do what computations. It brings the ability to handle thousand-bit numbers to devices as simple as a hobbyist's kit or a smart card. This democratization can have both positive and negative implications – it helps engineers implement strong cryptography everywhere (no excuse for using insecure short keys due to hardware limits), but it also means one must consider that even low-power adversaries might be capable of attacks that earlier would have required a supercomputer.

From a performance point of view, there is a clear boundary where one would switch from Specter-AAL to a library like GMP – likely when operating on desktops or servers and dealing with extremely large data (millions of digits) that demand the efficiency of FFT-based algorithms. However, Specter-AAL could still serve as a reference implementation or a fallback: its simple design might be easier to formally verify, which could be valuable for high-assurance systems (e.g. formally verified cryptographic software) where GMP's complexity is a hindrance. It could also complement GMP in a hybrid approach: use Specter's methods for sizes below a threshold or on platforms where assembly isn't available, and GMP's methods above that. Since Specter-AAL is GPL-licensed, it can be incorporated into open-source security projects and academic research freely.

In conclusion, Specter-AAL provides a meaningful and innovative contribution to the field of cryptographic computing. It builds on well-understood mathematics – ensuring its reliability through formal reasoning and proof – and presents those ideas in an accessible form for all hardware. By comparing it with established libraries, we see clearly the differences in philosophy: Specter-AAL is about *making big math universally accessible*, whereas GMP is about *making big math as fast as possible*. Both have their place. For RSA and other cryptographic applications, Specter-AAL expands the horizon of where such applications can run. It enables *"cryptography on a calculator"* in a literal sense. As the project matures (implementing multiplication, division, and perhaps optimized variants in the future), it could become a cornerstone for security applications in constrained environments and an object of study for algorithmic optimization on low-resource systems. The rigorous analysis above affirms that the project is grounded in solid theory and is poised to bridge the gap between theoretical cryptography and practical, everywhere-available computation.