An Update on the Sparkle Suite

Christof Beierle¹,², Alex Biryukov¹, Luan Cardoso dos Santos¹,
Johann Großschädli¹, Léo Perrin³, Aleksei Udovenko¹,
Vesselin Velichkov⁴ and Qingju Wang¹

¹ DCS and SnT, University of Luxembourg, Luxembourg ({first-name.last-name}@uni.lu)
² Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
(christof.beierle@rub.de)
³ Inria, France (leo.perrin@inria.fr)
⁴ University of Edinburgh, U.K. (vvelichk@ed.ac.uk)

sparklegrupp@googlegroups.com

1 The State of the Sparkle Suite

Intended use case

The Sparkle cipher suite contains several ciphers providing Authenticated Encryption with Associated Data (AEAD) and hash functions. These algorithms are all based on the Sparkle permutations that operate on 256, 384, or 512 bits. These algorithms are all intended to be both very fast and very small in software, especially on microcontrollers. More precisely, we aimed for a high throughput while retaining as low a RAM consumption as possible, and as low a code size as possible as well. Since all functionalities rely on the Sparkle permutations, it is possible to implement both AEAD and hashing in a very small package.

In terms of security, this cipher suite offers various trade-offs, from a 120-bit secure AEAD that uses only 256 bits for its internal state, all the way up to 248 bits of security for a larger instance.

Possible 3rd Round Tweaks

We do not plan to make any substantial change to our cipher suite should it be selected for the next round of the competition.

2 On the Security of the Sparkle Suite

Like many second-round candidates, our paper presenting this cipher suite along with its security analysis was accepted in the dedicated special issue of ToSC [BBC⁺20]. Nevertheless, we can add the following comments to this analysis.

A Detailed Analysis of Alzette

The security of our algorithms hinges on the properties of the Sparkle permutations. In turn, these all rely on the same component, namely the Alzette ARX-box. It is a 64-bit permutation parameterized by a 32-bit round constant, and whose properties are at the heart of our security arguments against usual cryptanalysis techniques. Our detailed analysis of this component was published at CRYPTO 2020 [BBdS⁺20], along with the
design of two new families of primitives based on this component. Having another fully-reviewed paper on a crucial design component of SPARKLE further strengthens the trust in our algorithms.

Because of this work, we have also been able to slightly improve the differential bound of Alzette. More precisely, we now have that the probability of the best 7-round differential trails is equal to $2^{-26}$, which improves upon our previous result which only stated that this probability was at most $2^{-24}$.

**Protection Against Attacks that Recently Emerged**

Like ours, other submissions rely on cryptographic permutations. In fact, the permutations used by two candidates to this standardization effort have recently been the target of full-round analysis, namely SPOOK [BBB+20] and Gimli [BKL+17]. However, the SPARKLE permutations are safe from such attacks. In the case of SPOOK, the attacks in [DHL+20] leveraged a strong similarity between the operations applied to different parts of the internal state along with some unforeseen interactions between the linear layer and the round constants, which ultimately allowed limited birthday attacks. In SPARKLE, similar operations are applied on each branch in parallel as well (namely, Alzette), but these are differentiated using dense and independent 32-bit constants. Thanks to these constants we are not worried about limited birthday attacks.

Like SPARKLE, Gimli relies on a two staged diffusion process along with a wide S-box (96-bit, which is similar to the 64-bit size of Alzette). However, the diffusion in Gimli is slow due to the structure of the linear layer, which allows efficient guess-and-determine algorithms capable of constructing symmetric inputs that are mapped to equally symmetric outputs [GLNP+20]. The Feistel structure and the strong (almost-)MDS permutations used to construct the linear layer of each SPARKLE instance give us confidence that SPARKLE is safe from such attacks.

## 3 On the Efficiency of our Cipher Suite

The updated specification we sent to NIST for the second round of evaluation was accompanied by reference and optimized C implementations of Schwaemm and Esch, and included also assembler implementations of the SPARKLE permutations for 8-bit AVR and 32-bit ARM Cortex-M3 microcontrollers. The submitted assembler code for 8-bit AVR is aimed at small (binary) code size, which means we did not apply certain optimization techniques that increase performance at the expense of a massive increase in code size, e.g. loop unrolling. Instead, to minimize code size, we implemented the permutation in a looped and parameterized fashion so that both the number of branches and the number of steps can be passed as parameter to the permutation function. In this way, one and the same assembler function of SPARKLE can be used for all instances of Schwaemm and Esch. On the other hand, the assembler implementations for the 32-bit ARM architecture can be characterized as “balanced,” which means the goal was to achieve a good trade-off between code size and execution time. This concretely means that we fully unrolled the branch-loop, but not the step-loop, i.e. the number of steps has to be passed as a parameter to the function. Consequently, our ARM implementation of SPARKLE consists of three separate assembler functions, namely one with four branches (SPARKLE256), one with six branches (SPARKLE384), and one with eight branches (SPARKLE512).

Earlier this year we developed new reference and optimized C implementations of Schwaemm and Esch, the latter of which is significantly faster than the optimized C implementation submitted to NIST last year. This new implementation merges the $p$ and rate-whitening functions and comes with a number of other tweaks to improve performance. Though this new implementation is not available to the public yet, we published already
some preliminary implementation results in [BBC⁺20]. For example, the new optimized C implementation of ❋SWAEMM256-128 requires 118917 clock cycles to encrypt 1536 bytes of data on an ARM Cortex-M3 microcontroller (see [BBC⁺20, Table 14]), which is 13759 cycles less than the 132876 cycles reported in [BBdS⁺19, Table 5.4] for the old optimized C implementation. We plan to submit the new reference and optimized C code to NIST in early October 2020, and we will also publicly announce their availability on the official NIST LWC mailing list.

In addition to the “balanced” ARM assembly implementation of ❋SPARKLE contained in the second-round submission package, we also developed speed-optimized versions with fully unrolled loops. The speed-optimized version of ❋SPARKLE384 has an execution time of 781 clock cycles on a Cortex-M3 microcontroller, which is 160 cycles less than the 941 clock cycles of the balanced implementation. The execution times of these speed-optimized assembly implementation for ARM were already reported in [BBC⁺20], but the source code is not yet publicly available. We plan to submit the new assembly implementations together with the improved C implementations to NIST in early October. Table 1 shows a comparison of the new speed-optimized ❋SPARKLE384 with other relevant permutations, namely, the Ascon permutation, Gimli, and Xoodoo. This table also shows an important measurement artifact that is often not taken into account: On some devices, the flash memory used for program storage is clocked at a much lower frequency than the microcontroller core. This difference in clock frequencies requires the insertion of wait-states, that depend on program-size, layout, and cache. Results, when measured in cycles-per-byte, will tend to be better on a simulator, or boards with no wait states. The same effect might not be discernible when measuring performance as data over time, as boards with more waitstates usually are configured with a higher clock.

Table 1: Execution time of the four permutations as determined by simulation with Keil MicroVision using a generic Cortex-M3 device and measurement on Cortex-M3 development boards with 0, 2, and 5 flash wait states (values in parentheses are the performance penalties over the execution time on the VL Discovery board, which has 0 flash wait states).

<table>
<thead>
<tr>
<th>Permutation</th>
<th>Keil µVision (simulation)</th>
<th>VL Discovery 0 wait states</th>
<th>Nucleo-64 2 wait states</th>
<th>Arduino Due 5 wait states</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCON128a (8 rounds)</td>
<td>466</td>
<td>467</td>
<td>748 (1.60)</td>
<td>571 (1.22)</td>
</tr>
<tr>
<td>Gimli (24 rounds)</td>
<td>1041</td>
<td>1043</td>
<td>1656 (1.59)</td>
<td>1287 (1.23)</td>
</tr>
<tr>
<td>SPARKLE384 (7 steps)</td>
<td>781</td>
<td>782</td>
<td>1196 (1.53)</td>
<td>936 (1.20)</td>
</tr>
<tr>
<td>XOODOO (12 rounds)</td>
<td>657</td>
<td>659</td>
<td>1014 (1.54)</td>
<td>795 (1.21)</td>
</tr>
</tbody>
</table>

Another interesting benchmark for comparison of permutations is the throughput in cycles per byte, as permutations operate over differently sized states. Table 2 shows speed in cycles-per-byte, and in cycles-per-rate-byte, the latter offering a meaningful view on the number of cycles needed for processing each payload byte.

Table 2: Permutation performance in cycles-per-byte (cpb) and cycles-per-rate-byte (cprb) of the permutations of the main instances of the AEAD algorithms on an ARM Cortex-M3 microcontroller.

<table>
<thead>
<tr>
<th>Permutation</th>
<th>Size</th>
<th>cpb</th>
<th>cprb</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCON128a p8</td>
<td>320</td>
<td>11</td>
<td>29</td>
</tr>
<tr>
<td>Gimli</td>
<td>384</td>
<td>21</td>
<td>65</td>
</tr>
<tr>
<td>SPARKLE</td>
<td>384</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>XOODOO-r12</td>
<td>384</td>
<td>13</td>
<td>27</td>
</tr>
</tbody>
</table>
Third-Party Implementations and Evaluation

Besides the implementations mentioned above, there also exist a couple of third-party implementations. On the software-side, we refer to the implementation and benchmarking work of Rhys Weatherley, which is available online.\(^1\) They developed optimized C implementations of \textsc{Schwaemm} and \textsc{Esch}, and also an assembly implementation of \textsc{Sparkle} for 8-bit AVR microcontrollers. The latter is optimized for speed and, therefore, faster than our size-optimized implementation of \textsc{Sparkle}. We reviewed the source code of Weatherley’s AVR implementation and came to the conclusion that it is of high quality and leaves little room for improvement. Also, their C implementation is more aggressively optimized for performance than our own optimized C code; for example, they unrolled the branch-loop of the permutation. Even though this implementation secures \textsc{Schwaemm} one of the top spots in the results table for Cortex-M3, we remark that benchmarks collected with assembly implementation would be more accurate.

In general, the benchmarking results generated by Weatherley for 8-bit AVR, 32-bit ARM, and 32-bit ESP32 indicate that the algorithms of the \textsc{Sparkle} suite are consistently among the best performers. We also remark that, on AVR, our high-security AEAD instance \textsc{Schwaemm}\_256-256 outperforms most of the other candidates despite their lower security level.

Regarding hardware implementation, we refer to \cite{Col20}, which presents an FPGA design of our suite, both unprotected and protected against SCA. However, we believe that it is possible to obtain a better implementation of \textsc{Schwaemm}. Since \textsc{Schwaemm} is a classical ARX design, it is conceivable that its 32-bit integer additions would make \textsc{Sparkle} more costly to implement in hardware (and introduce a larger critical path delay) than an e.g. the permutation of \textsc{Ascon}, which does not use integer additions. However, the integer additions alone can not explain why, according to \cite[Table 4]{RCSD19}, \textsc{Schwaemm} reaches only relatively low clock frequency and throughput figures since e.g. \textsc{Comet-Cham} also performs integer additions, but its maximum clock frequencies are twice as high. We plan to collaborate with the authors of \cite{RCSD19} and try to improve the implementation described in \cite{Col20}.

References


\(^1\)\url{https://rweather.github.io/lightweight-crypto/index.html}. 


