Secureduino: Implementing Secureboot on AVR Microcontrollers

Thomas Mahoney

+61 4 3979 5026
t.mahoney@me.com
mahoney@apple.com

Table of Contents

Tables and Figures

Introduction and Background

Microchip’s AVR collection of microcontrollers is commonly found in low-end and hobbyist equipment and has been the backbone of the ‘Arduino’ product range. These microcontrollers are also widely used in imbedded devices or Internet of Things products (Microchip Technology Inc., 2021). Common variants of the Arduino use either the Atmega2560 or Atmega368p microcontroller products. These have 32K to 64K of ROM and 2K to 8KB of RAM, with the Atmega368p being the more constrained device (Microchip Technology Inc., 2020). The devices have specific address space set aside for a bootloader in the upper 8K (or 4K, for the Atmega328p) of ROM.

Despite their widespread usage there is a corresponding lack of security in the devices. There is no built-in security mechanism excepting those designed to disallow in-system programming without the correct equipment. With this equipment, an attacker can simply reprogramme the device with a malicious payload.

Though this may not be of consequence in the hobbyist scenario at which the Arduino is targeted, many IoT and embedded devices are built on the same hardware and codebase. This project aims to implement Secureboot on the AVR Atmega2560 or Atmega328p microcontroller wherein the bootloader is able to cryptographically verify the firmware in ROM. In the hypothetical situation the project aims to address an attacker is able to freely overwrite the majority of the ROM but is not able to overwrite the bootloader; thus, the bootloader space becomes the root of trust.

After implementing a secure firmware signing method in the bootloader, several additional tasks are suggested:

- Secure Debugging. The bootloader cryptographically verifies a connection and provides RAM, ROM and register read and write utilities to an authenticated user
- Encrypted firmware that requires the bootloader to decrypt as well as verify the firmware image before execution
- Exploit mitigations such as Stack Canaries, ASLR and NX memory regions

These tasks will be discussed further on page 9 in *Statement of Completeness.*

Theory: Firmware Signing and Cryptographic Primitives

The author was free to choose an approach to this problem, provided the basic requirement of a cryptographically verified firmware image is met.

Digital signatures most often take the form of an x.509 Public Key Certificate. These certificates are dependent on the encryption algorithm and key size chose, but can often approach 1KB in size (Cooper, et al., 2008). In an environment where every byte is precious this occupies too much ROM and may not fit into RAM if required. Thus, the author chose to implement a firmware signature scheme without the overhead of the x.509 certificate format.

The Secureboot implementation hashes the payload firmware up to the bootloader section (BLS), leaving enough space for a cryptographically signed block containing the hash to be appended. In the final implementation this results in a loss of 1024 bits (128 bytes) of usable firmware space, resulting in a ROM usage overhead of 0.4% for the Atmega328p.

Initially the latest cryptographic primitives were considered to construct the Secureboot implementation. Elliptic Curve Diffie-Helman Signature was thought to be a good fit due to the lower number of bits required for the same level of encryption (resulting in lower RAM and stack usage); however, the author was unable to implement the curve in the small ROM space usable. Likewise, SHA3 was the first chosen hashing algorithm; the author was unable to implement the algorithm and unable to use an existing library that on the chosen platform.

The author developed a working solution based on a SHA1 hash of the firmware and an RSA signature with a 1024 bit key. Due to project constraints a pre-existing library was chosen rather than developing software from scratch. After reviewing a number of libraries exemplified RFC 8387 (RFC 8387: Practical Considerations and Implementation Experiences in Securing Smart Object Networks, 2018) and additional work such as an effort to port NaCL to AVR microcontrollers (Hutter & Schwabe, 2013), and a SHA-3 implementation by Markku-Juhani O. Saarinen (Saarinen, 2018). The library chosen for Secureboot implementation is that developed by Emile van der Laan (van der Laan, 2016), with a total bootloader size of 3374 of 4096 bytes available on the Atmega328p. The microcontroller is clocked at 20Mhz for the simulation

Implementation Approach and Usage Instructions

The board_simduino example provided in Jacob Gruber's simavr library (Gruber, 2020) provided for a simulated Arduino environment that required minimal modification to suit the needs of the project.

Initially the base simavr simulator was used for prototyping. Although this provided for more flexibility in the form of additional run-time switches, the simulator appeared to exhibit a bug in which constants were not loaded from a compiled ELF file but were loaded correctly from the same file in an Intel formatted hex file. After failing to get a simple 'Hello World' to print out to a serial terminal using the ELF input format, hex files were used for further development. This allowed for better compatibility with board_simduino and so further development proceeded using that codebase.

The simduino code was modified minimally; the code attempts to read the base address of the loaded intel hex file to establish the initial value of the program counter and sets the microcontroller model based on this model. The code was changed such that it was forced to a specific microcontroller model for testing, and the starting program counter could be overridden. As the code did not appear to allow for the configuration of the BOOTRST fuse that configures the starting PC on the real device (Microchip Technology Inc., 2020), this was a workaround that was used during testing. The codebase was renamed 'secureduino' to better reflect the intended functionality.

Initial development was conducted with the Atmega2560 as the target microcontroller to take advantage of the larger available bootloader area. However, stability issues forced a change to the Atmeg328p with half the available space. Standard C code that would run without issue on the simulated Atmega328p core would crash the Atmega2560 core; assembly code and USART register configurations that appeared legal for both devices would crash, reset or hang the Atmega2560 but run smoothly on the Atmega328p. The project changed target to the Atmega328p very late in the development process as the smaller code space was preferred to the debugging unrelated to the project code.

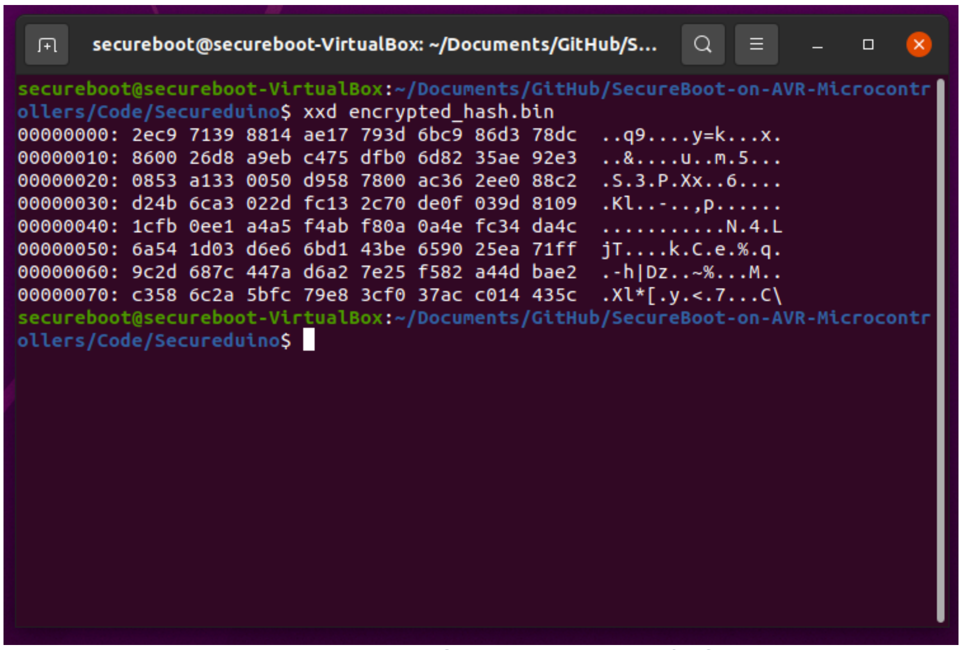
The codebase is comprised of three sections. A payload, which can be any legal code written for the Atmega328p; the bootloader, the key deliverable of the project; and a mimally modified copy of Gruber's board_simduino code. The project makefile compiles all three codebases and dependencies, calculates the SHA1 hash of the payload, signs it with the project's private key using OpenSSL and then appends the key to the payload at address 0x6F80. Thus, the last 128 bytes of the standard ROM area (before the bootloader ROM) is the signed hash of the previous 0x6F80 bytes. This is demonstrated below, in Figure 1, Figure 2 and Figure 3.

Compilation Instructions

The project can be run by changing into the project directory and running *make* followed by *make simulate.* Secureduino will output directions to connected to the simulated Arduino using picocom.

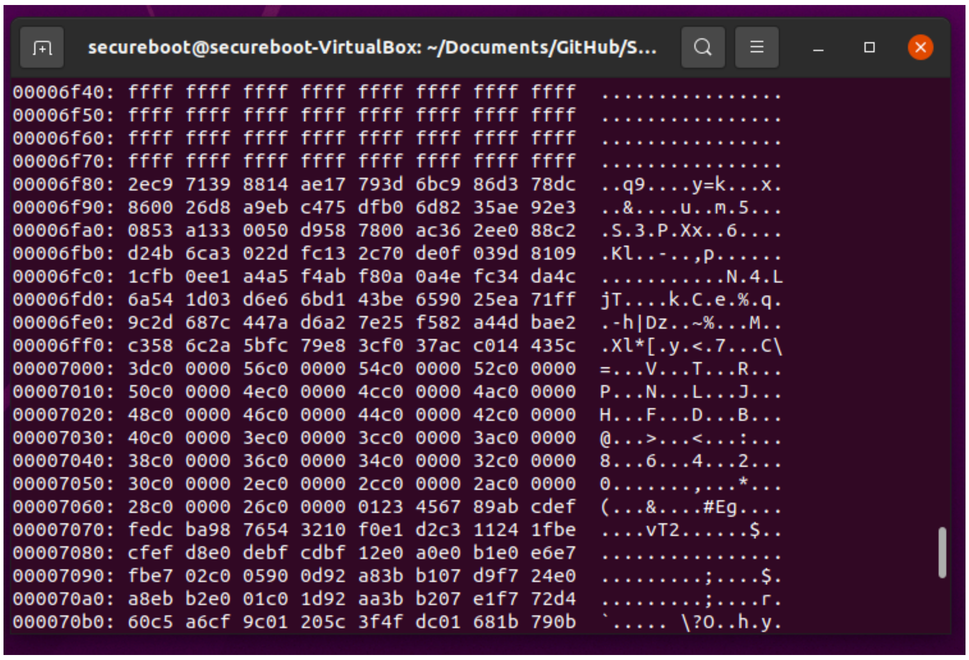
```
# Hex merging from https://www.avrfreaks.net/forum/merging-bootloader-hex-file-and-application-hex-file-one
# simduino doesn't like an Intel Hex file with multiple segments (it will only load the first)
# Below hack to get everything into a single .hex file with one segment. Leave BIN behind for debug
# We Compile to a .bin, hash, sign with our private key and concatenate the key on the end of the payload binary
binary: payload bootloader
    @cat ${payload}.hex | awk '/^:00000001FF/ == 0' > binary.hex
    @cat ${bootloader}.hex >> binary.hex
    @avr-objcopy --gap-fill 0xFF -I ihex -O binary binary.hex binary.bin
    @dd if=/dev/zero of=hash.bin  bs=1 count=108
    @dd if=binary.bin bs=1 count=${ROM_TOP} | openssl sha1 -binary >> hash.bin
    @cat hash.bin | openssl rsautl -sign -inkey keys/private.pem -raw > encrypted_hash.bin
    @dd conv=notrunc if=encrypted_hash.bin of=binary.bin bs=1 seek=${ROM_TOP}
    @avr-objcopy -I binary -O ihex binary.bin binary.hex
    @echo done
```

Figure 1: Hashing and Signing details in Project Makefile



```
secureboot@secureboot-VirtualBox:~/Documents/GitHub/SecureBoot-on-AVR-Microcontr
ollers/Code/Secureduino$ xxd encrypted_hash.bin
00000000: 2ec9 7139 8814 ae17 793d 6bc9 86d3 78dc  ..q9....y=k...x.
00000010: 8600 26d8 a9eb c475 dfb0 6d82 35ae 92e3  ..&....u..m.5...
00000020: 0853 a133 0050 d958 7800 ac36 2ee0 88c2  .S.3.P.Xx..6....
00000030: d24b 6ca3 022d fc13 2c70 de0f 039d 8109  .Kl..-..,p......
00000040: 1cfb 0ee1 a4a5 f4ab f80a 0a4e fc34 da4c  ...........N.4.L
00000050: 6a54 1d03 d6e6 6bd1 43be 6590 25ea 71ff  jT....k.C.e.%.q.
00000060: 9c2d 687c 447a d6a2 7e25 f582 a44d bae2  .-h|Dz..~%...M..
00000070: c358 6c2a 5bfc 79e8 3cf0 37ac c014 435c  .Xl*[.y.<.7...C\
secureboot@secureboot-VirtualBox:~/Documents/GitHub/SecureBoot-on-AVR-Microcontr
ollers/Code/Secureduino$ 
```

Figure 2: 1024 bit RSA Encrypted Hash of ROM



```
00006f40: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00006f50: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00006f60: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00006f70: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00006f80: 2ec9 7139 8814 ae17 793d 6bc9 86d3 78dc  ..q9....y=k...x.
00006f90: 8600 26d8 a9eb c475 dfb0 6d82 35ae 92e3  ..&....u..m.5...
00006fa0: 0853 a133 0050 d958 7800 ac36 2ee0 88c2  .S.3.P.Xx..6....
00006fb0: d24b 6ca3 022d fc13 2c70 de0f 039d 8109  .Kl..-..,p......
00006fc0: 1cfb 0ee1 a4a5 f4ab f80a 0a4e fc34 da4c  ...........N.4.L
00006fd0: 6a54 1d03 d6e6 6bd1 43be 6590 25ea 71ff  jT....k.C.e.%.q.
00006fe0: 9c2d 687c 447a d6a2 7e25 f582 a44d bae2  .-h|Dz..~%...M..
00006ff0: c358 6c2a 5bfc 79e8 3cf0 37ac c014 435c  .Xl*[.y.<.7...C\
00007000: 3dc0 0000 56c0 0000 54c0 0000 52c0 0000  =...V...T...R...
00007010: 50c0 0000 4ec0 0000 4cc0 0000 4ac0 0000  P...N...L...J...
00007020: 48c0 0000 46c0 0000 44c0 0000 42c0 0000  H...F...D...B...
00007030: 40c0 0000 3ec0 0000 3cc0 0000 3ac0 0000  @...>...<...:...
00007040: 38c0 0000 36c0 0000 34c0 0000 32c0 0000  8...6...4...2...
00007050: 30c0 0000 2ec0 0000 2cc0 0000 2ac0 0000  0.......,...*...
00007060: 28c0 0000 26c0 0000 0123 4567 89ab cdef  (...&....#Eg....
00007070: fedc ba98 7654 3210 f0e1 d2c3 1124 1fbe  ....vT2......$..
00007080: cfef d8e0 debf cdbf 12e0 a0e0 b1e0 e6e7  ................
00007090: fbe7 02c0 0590 0d92 a83b b107 d9f7 24e0  .........;....$.
000070a0: a8eb b2e0 01c0 1d92 aa3b b207 e1f7 72d4  .........;....r.
000070b0: 60c5 a6cf 9c01 205c 3f4f dc01 681b 790b  `..... \?O..h.y.
```

Figure 3: Encrypted Hash Inserted into Binary at the End of ROM (0x6F80)

The bootloader includes the project’s 1024 bit public key which is used to verify the signature, and compare the verifed hash value to that calculated during the bootloader’s hashing of the firmware ROM:

```
// 1024 bit RSA Public Key, Generated using OpenSSL
unsigned char public_key[]  =
{
    0xc9, 0x08, 0x12, 0xc5, 0xe4, 0x2c, 0x63, 0xc0, 0xe6, 0x88, 0xbb, 0xfa,
    0x88, 0x48, 0xa8, 0xf1, 0x87, 0x82, 0x5d, 0xfb, 0x85, 0xa1, 0x89, 0x4e,
    0xf1, 0xa1, 0xb0, 0x7a, 0x2e, 0x20, 0x65, 0x63, 0xdb, 0xb3, 0xf0, 0xc8,
    0xc6, 0xd1, 0x50, 0xbe, 0xa6, 0xb1, 0xb1, 0x8e, 0x31, 0x41, 0xfe, 0x7c,
    0x03, 0x0c, 0xcf, 0xad, 0x5a, 0xfc, 0x44, 0x8e, 0xf2, 0x06, 0xd8, 0xef,
    0x89, 0x74, 0x0c, 0x77, 0x4c, 0x2d, 0x55, 0x93, 0x93, 0x7e, 0xd4, 0xaf,
    0x84, 0x13, 0x4f, 0x32, 0x04, 0xf3, 0x44, 0x3f, 0x62, 0x55, 0xaa, 0x34,
    0xe2, 0xa2, 0xbc, 0x3e, 0xb5, 0xe2, 0x9d, 0xed, 0x1f, 0xb5, 0xe5, 0x77,
    0xfd, 0x84, 0x55, 0x4d, 0x7c, 0xef, 0xc4, 0x8b, 0x46, 0x5c, 0x4d, 0xc0,
    0x76, 0x49, 0xe1, 0x11, 0x89, 0x48, 0x4c, 0x2e, 0xfb, 0xc8, 0x98, 0x92,
    0xe0, 0x1a, 0x60, 0x78, 0xf1, 0xe2, 0xbb, 0x3f
};
```

Figure 4: 1024 bit RSA Public Key, Generated using OpenSSL

```
Sha_Init();

for (int i = 0; i < ROM_TOP; i+= 64)
{
    get512block(hash_block, i);
    Sha_Update(hash_block, sizeof(hash_block));
}

Sha_Final();
```

Figure 5: SHA1 Hashing of ROM

```
get1024block(signature_block, ROM_TOP);

rsa_decrypt(sizeof(public_key), signature_block, public_exponent, public_key, rsa_s, rsa_tmp);
```

Figure 6: RSA Verification of Signed Hash using Public Key

Whenever possible memory was manually managed. Buffers are created at run-time and re-used; large variables (such as the public key) that are required to be in RAM are loaded at startup by the compiler. Hand-written libraries such as those used to print to the UART are used in place of standard C libraries wherever possible to save on ROM space.

Testing and Performance

The provided code performs as expected. Making small changes in the payload result in a different hash and signature that are created correctly by the Makefile and decoded as expected by the bootloader. Modify the binary after it is compiled correctly causes the hash to fail and the bootloader does not run the firmware. However, extensive testing would be required before allowing this prototype code into production as no stringent testing regime or unit tests were developed for the project.

Performance was not measured on the simulated Arduino running at 20MHz although future work may involve measuring overhead using simavr's VCD support. Using the execution times published by (van der Laan) and confirmed in RFC 8387 (RFC 8387: Practical Considerations and Implementation Experiences in Securing Smart Object Networks):

Signature Stage	RAM Usage (Bytes)	Execution Time, Approx.
SHA1 Hash of ROM	120 Bytes	690 ms
1024 RSA Verification	512 Bytes	3,550 ms
Total	632 Bytes	4,240 ms

Table 1: Estimated Execution Overhead

Though the simulated Arduino environment cannot be considered an exact replica of the physical microcontroller, the above numbers appear accurate. There was no noticeable delay when booting the secureduino; indeed, the author assumed any delay was due to the busy-wait loops used in the UART code. This is a pleasant surprise and suggests the use of more secure cryptographic primitives is another avenue of future work to be explored. Likewise, the bootloader appears to have plenty of additional RAM that could be used for a larger key or with larger buffers to speed execution.

Statement of Completeness and Future Work

Although the basic functionality – secureboot on an AVR microcontroller – has been implemented, none of the bonus tasks have been attempted. The author lost a great deal of time attempting to debug issues that presented when simulating using the Atmega2560 core that were not present on the Atmega328p core. Switching back to the less capable core allowed the project to be completed.

Attack Surface

As presented the basic functionality of the project is present but may prove to have an unacceptably high attack surface. No analysis has been conducted of electromagnetic, power or timing side channels such as those outlined by (Liu, Großschädl, & Kizhvatov, 2010). Additionally, the project uses SHA1 which has been proven to be insecure in the SHAttered attack (Stevens, Bursztein, Karpman, & Albertini, 2017). Once calculated the hash is padded with leading zeros rather than an appropriate padding scheme such as PKCS#1 (Moriarty, Ed., et al., 2016), which is particularly vulnerable to attack with the exponent of 3 that has been chosen due to the speed of execution. Future work must consider these shortcomings before implementing the below additional features.

Secure Debugging

It would be advantageous to allow debugging of the firmware while in a production environment. This might be implemented by an extension of the existing bootloader, or by chaining to a trusted part of the firmware to allow the reading and writing of RAM, ROM and register values. Given the ~600 bytes free in the existing bootloader the author considers it feasible to include additional debugging features in the existing code. The user may be verified using the existing cryptographic primitives with no need to introduce additional overhead; consequently, much of the remaining code space can be used for additional debugging features.

Encrypted Firmware

This also could be implemented with the existing cryptographic primitives but would be better served with a symmetric (AES) or stream (LFSR) cipher that has the initial vector protected by the existing RSA implementation. In the imagined implementation, the encrypted code would start at halfway through the available ROM space (0x37C0 in the case of the Atmega328p) and be decrypted and copied 14k down by the bootloader, such that the first block aligns with 0x0000. Once complete, the bootloader will simply jump to 0x0000 as normal. This cuts the executable ROM space in half but allows for a fully cryptographically protected firmware image with minimal overhead on boot and no additional overhead once running. Care must be taken though to securely erase the firmware on device power down as this mechanism would be subject to power glitching attacks. Possible defenses include storing critical variables in RAM or only decrypting the blocks on an as-needed basis, trapping the microcontroller’s RJMP instruction to load the next block.

Exploit Mitigations

As the AVR series of microcontrollers have a Harvard architecture (Microchip Technology Inc., 2020) a number of existing mitigation techniques are not possible to implement; likewise, the small size of the memory space creates an hesitancy to waste bytes on low priority security features.

Assuming a malicious binary is written to the device the binary will have full control over the microcontroller and peripherals. ASLR is moot as a binary cannot be relocated or modified once programmed except by the bootloader (the root of trust), and NX features such as page flags or non-executable stack are unnecessary because nothing can execute from RAM. Stack Canaries may be beneficial to prevent return-oriented programming; however, these are typically implemented using a (known) random value and the Atmega2560 and Atmega328p have no true random generator; thus, any ‘random’ value may be deterministic. Any additional values on the stack will also negatively impact the available RAM of the device.

Conclusion

Though I initially struggled with simavr and repeatedly encountered obstacles I consider this a valuable learning experience. I’ve enjoyed working in C again as I feel I’ve been spoiled by high level languages features such as strong typing, flexible data manipulation and garbage collection. When working with machine learning applications it was not uncommon for my programs to use gigabytes of RAM – it was refreshing to count every byte again! I hope you will find the core component of the project fulfilled to your satisfaction and my analysis thorough. I am happy to answer any additional questions you might have.

Works Cited

Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., Polk, W., . . . Vigil Security. (2008, May). *Network Working Group Request for Comments 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.* Retrieved from Internet Engineering Task Force: https://tools.ietf.org/html/rfc5280

Gruber, J. (2020, October 5). *buserror / simavr: simavr is a lean, mean and hackable AVR simulator for linux & OSX*. Retrieved from GitHub: https://github.com/buserror/simavr

Hutter, M., & Schwabe, P. (2013). NaCl on 8-bit AVR Microcontrollers. *Progress in Cryptology – AFRICACRYPT 2013* (pp. 156-172). Cairo: Lecture Notes in Computer Science 7918, Springer-Verlag (2013).

Liu, Z., Großschädl, J., & Kizhvatov, I. (2010). Efficient and Side-Channel Resistant RSA Implementation for 8-bit AVR Microcontrollers. *Proceedings of the 1st International Workshop on the Security of the Internet of Things (SECIOT 2010).*

Microchip Technology Inc. (2020). *ATmega48A/PA/88A/PA/168A/PA/328/P megaAVR® Data Sheet.* Retrieved from Microchip: https://ww1.microchip.com/downloads/en/DeviceDoc/ATmega48A-PA-88A-PA-168A-PA-328-P-DS-DS40002061B.pdf

Microchip Technology Inc. (2021, April 21). *Microchip AVR-IoT*. Retrieved from AVR-IoT Development Boards: https://www.avr-iot.com

Moriarty, Ed., K., Kaliski, B., Jonsson, J., EMC Corporation, Verisign, Subset AB, & RSA. (2016, November). *RFC 8017: PKCS #1: RSA Cryptography Specifications Version 2.2.* Retrieved from Internet Engineering Task Force: https://tools.ietf.org/html/rfc8017

Saarinen, M.-J. O. (2018, October 22). *teserakt-io / sha3-avr: SHA-3 implementation for 8-bit AVR*. Retrieved from GitHub: https://github.com/teserakt-io/sha3-avr

Sethi, M., Keranen, A., Arkko, J., & Back, H. (2018, May). *RFC 8387: Practical Considerations and Implementation Experiences in Securing Smart Object Networks.* Retrieved from Internet Engineering Task Force: https://tools.ietf.org/html/draft-aks-lwig-crypto-sensors-01

Stevens, M., Bursztein, E., Karpman, P., & Albertini, A. (2017). The first collision for full SHA-1. *International Cryptology Conference - CRYPTO 2017* (pp. 570-596). Santa-Barbara: The International Association for Cryptologic Research.

van der Laan, E. (2016, January 16). *AVRCryptoLib*. Retrieved from AVRCryptoLib: http://www.emsign.nl

Appendix One: Code Listing

All code is included in the Secureduino folder in the attached compressed file