Method for Truly Secure Communication

Author: Vlad Slyusar

***Abstract*—The purpose of this project is to develop a truly secure communication system for personal use. One-time-pad encryption (mathematically unbreakable) is utilized in combination with isolated endpoint hardware to mitigate a large number of potential attack vectors that plague most consumer devices. A large portion of the research focuses on such vulnerabilities, as well as evidence of their exploitation on a large scale. A unique design for a hardware security module-based solution is presented, implemented, and tested. The project also lays out a custom communication protocol for facilitating the creation of unique OTP keys for each new message and synchronizing keys on both ends of the communication channel. A fully functional, hardware and communication independent system is presented with a guarantee of absolute security of information transmitted over both WiFi and SMS public channels. The security limitations and some of the remaining vulnerabilities of the implemented system are also explored, and a number of recommended improvements for future iterations are provided.**

I. Introduction

Despite rapidly accelerating progress within the computing and communications industries, truly secure means of communication remain inaccessible to the majority of people in the world. While strong encryption techniques exist and are utilized within a large number of applications, such measures serve the purpose of simply providing security during transmission across the web. Even with perfect security during transmission, the endpoint hardware/software often remains highly vulnerable to numerous attack vectors. The secure information must undergo encryption/decryption at both ends of the channel, which is typically done on standard consumer devices running complex operating systems with numerous intentional and unintentional vulnerabilities. Consequently, government agencies or determined hackers can have access to the endpoint device without the owner's knowledge or consent. As a result, nearly all standard forms of communication must be approached with no expectation of privacy. A large amount of evidence for this is presented within the following section. Once some of the security limitations of common communication systems are presented, the architecture for one proposed solution is laid out. Although the overall system proposed still contains vulnerabilities of its own, it outlines one highly effective approach to significantly impede mass-surveillance techniques. For the proposed architecture, all necessary subsystems are further described in detail. The procedure for their integration into a complete project is provided along with overall test results. Following another stage of analysis of potential attack vectors against the implemented system, a number of possible improvements are outlined.

II. Research: Defining True Security

While encryption is already widely utilized for limitless applications worldwide, numerous security holes generally exist within its use. While generally considered the most secure aspect of the system, a cryptographic function itself may contain inherent vulnerabilities. It is not uncommon to hear about government agencies exerting their influence on popular encryption standards[1], with one particularly famous case being the backdoor within Dual_EC_DRBG algorithm as verified by files leaked by Edward Snowden[2]. The backdoor was implemented as a kleptographic weakness within the cryptographically secure pseudorandom number generator(CSPRNG) and only known and accessible by the NSA. The agency even paid out a sum of $10 million dollars to RSA Security to utilize Dual_EC_DRBG as the default within the RSA cryptographic library[3]. In fact, government agencies around the world dedicate significant resources to breaking encryption behind online communications. Perhaps the most well known are the Bullrun decryption program, operated by the US National Security Agency(NSA), as well as British Government Communication Headquarters(GCHQ) program codenamed Edgehill[4][5].Their methods include network exploitation, interdiction, secret relationships with industry, collaboration with other agencies, and advanced mathematical techniques[6]. The programs are extremely well funded with Bullrun alone operating on a 1/4 billion annual budget back in 2013[6]. Such programs fall under the overall umbrella of signals intelligence(SIGINT), which deals with interception of communication between people as well as general electronic signals. As encryption is utilized more and more, the incentives for breaking popular protocols greatly increase.

While the encryption function itself may be prone to vulnerabilities, a much more common and simpler tech-

nique for breaking encryption is through compromising the endpoint device. End-to-end encryption, especially with pre-exchanged keys, still poses a great challenge to even the best funded hackers. Many popular communication apps implement the feature with their own custom algorithms, often requiring in-person key exchange[7]. One of such apps personally utilized by the researchers is telegram, which currently hosts a public bounty of $300,000 for successfully cracking its encryption[8]. Although its unlikely that any government agency would attempt to claim the reward upon successfully breaking the protocol, the encryption may still remain secure simply due to the strong mathematical protections at its core. However, utilizing the application on nearly any popular communication device allows for a completely different attack without the need to ever compromise the encryption. Within a recent press release by WikiLeaks, some of the Central Intelligence Agency's documents reveal the immense extent of its surveillance capabilities[9]. Telegram is specifically mentioned, among numerous other apps, as one of the targets compromised through widespread malware developed by Engineering Development Group(EDG) for both Android and iPhone. Per the press release, the group is officially responsible for development, testing, and operational support of all backdoors, exploits, malicious payloads, trojans, viruses and other malware used by the CIA for covert operations world-wide. The security of applications such as telegram is easily circumvented via malware that can read messages before encryption is even applied. Any necessary information, perhaps matching certain keywords, can then be exfiltrated using the numerous communication channels available to the device. With actual evidence of the existence of such powerful players in the espionage game, even end-to-end encryption fails to provide a true measure of security.

Perhaps one solution would involve running custom firmware on an unlocked smartphone in the hopes of mitigating the effects of spyware developed for more popular source distributions. Alternatively, one might try to run a very thorough firewall on their computer, running an open source operating system, in the hopes of preventing such attacks. Unfortunately, even such extreme measures can be circumvented through hardware backdoors built into most consumer processors and SOCs. Perhaps the most famous is the Intel Management Engine:[10]

- Completely separate CPU with direct access to memory and BIOS flash
- Full access to TCP/IP stack with direct connection to Ethernet controller
- Proprietary code, signed with 2048-bit RSA key, cannot be audited
- Separate MAC address allows WakeOnLan apart from main CPU, even in S3 sleep mode
- Cannot be disabled on processors newer than Core2

This effectively means that every modern Intel CPU contains an additional processor that has complete access to all necessary systems to function as an ideal data exfiltration tool. Running closed-source encrypted code, the real functionality of the management engine is completely hidden from the main CPU and any kind of operating system the user could run. Through the use of a National Security Letter(NSL) with a Gag Order clause, Intel could have been forced to secretly give up the 2048-bit RSA key[11]. While little evidence exists, in the public domain, that this has actually happened, any party in possession of the private key can write custom firmware for the chip. While switching to a competitor may sound appealing, AMD processors include an ARM core with similar access under the name TrustZone[12]. With the necessary infrastructure already in place, there is a high probability that hardware level backdoors have been created and may already be utilized[13].

With such immense efforts on behalf of government agencies, and perhaps even private corporations, nearly all forms of communication on commercial devices should be treated as inherently compromised. A custom hardware and software solution must be developed if one seeks better privacy and security. Unfortunately, any large scale system will always remain potentially vulnerable to newly invented attacks. Hence the best approach is a system that works across distributed hardware platforms, limits external communication to a tightly controlled channel, and utilizes an unbreakable form of encryption. This is precisely the type of system implemented within this project.

III. Requirements

The research shows that there is a need for a more secure communication device that is designed to be open sourced so it can be replicated, and includes a strong form of encryption that is mathematically unbreakable. Using the above idea, the requirements for the project were determined as follows:

- Use OTP Encryption
- Allow storage of large OTP key
- Built using open source hardware
- Unencrypted data must never leave the device
- Have a screen to display messages
- Allow input with a standard keyboard

These requirements allow this project to realize its goal of creating a secure messaging platform that can

be used by anyone and provide a level of encryption that is known to have no backdoors, as well as be mathematically unbreakable.

IV. Overall Project Design

Because few open-source hardware security modules(HSMs) exist, a custom system was developed including hardware architecture as well as custom communication protocol. The system block diagram is provided in Figure 1.

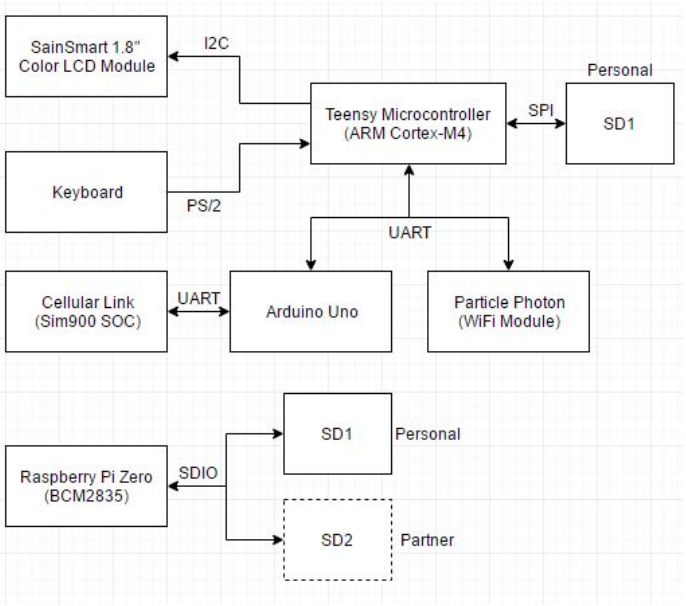

Figure 1. System Block Diagram

The project is designed over a distributed set of hardware and software components. The main controller for the HSM does not support a True Random Number Generator, so an extra device is employed (The Raspberry Pi Zero) to utilize its hardware random number generator. The Teensy is left with the rest of the software tasks, each of which were written in a layered fashion. On the bottom is the hardware interfacing layer. This includes the UART and SPI drivers. The layer above these belongs to the SD card driver. Interface functions were written to abstract the SPI communication needed to read and write the OTP key from the SD Card. Above the SD Card driver are the encryption and decryption functions, which hold the plaintext and encrypted data. These functions only know the location of the OTP. They call into the SD Card driver which retrieves the OTP from the SD Card. The top layer is the application layer. This layer accepts input on the keyboard and prints out to the LCD screen. The application layer also polls the UART channel directly to look for new messages to be received from the other device.

The device functions in the following fashion. First a large random key is generated on the Raspberry Pi zero from the thermal entropy of its CPU and stored on two MicroSD cards. Figure 2 shows the hex dump of the data from one of the SD Cards. Each SD card is then connected to its own communication system based around a Teensy microcontroller. Once a large single password is pre-shared, small portions can be used as individual keys for each message. A further section describes the exact protocol for handling the sliding window and synchronizing start and end bits for each key. To obtain any portion of the key, the SD card is initialized and a block read command is issued. All initialization and control commands are performed using SPI communication mode using a custom driver that was developed to access blocks of SD card memory directly, independent of any filesystem. The key is then used for either encryption or decryption operation. A keyboard and LCD display are used for interaction with the user. A PS/2 keyboard sends character commands to the Teensy, which updates the LCD display in real time. The backspace character is also programmed to allow modifications to the message. The page up key activates the send command, which performs an XOR of the entered message with a corresponding number of the next unused key bits. By performing an XOR of the message with a randomly generated key of equal length, and always using a new key, the system implements One-Time-Pad(OTP) encryption. Although quite simple at its core, this form of encryption is mathematically unbreakable, as the transmitted message contains zero information about the original[14]. As long as the keys are kept secure, OTP encryption is safe against cryptographic attacks with limitless computing power and effectively provides storage that is not limited by time[15]. The encrypted version of the message, combined with a custom header, is then sent to an external module over a serial communication channel. This is the only mode of communication for the secure system with the external world. For reception, messages are read over UART from the external device and then decrypted using the XOR operation with the corresponding key before being displayed on screen for the end user to read. Message header information contains the expected next starting bit of the key thus allowing synchronization and preventing reuse of any portions of the key. Because the core project provides significant isolation from the outside world, and enforces strict cryptography, the name Hardware Security Module(HSM) is appropriate. Many more details on each subsystem of the HSM are provided in the further sections.

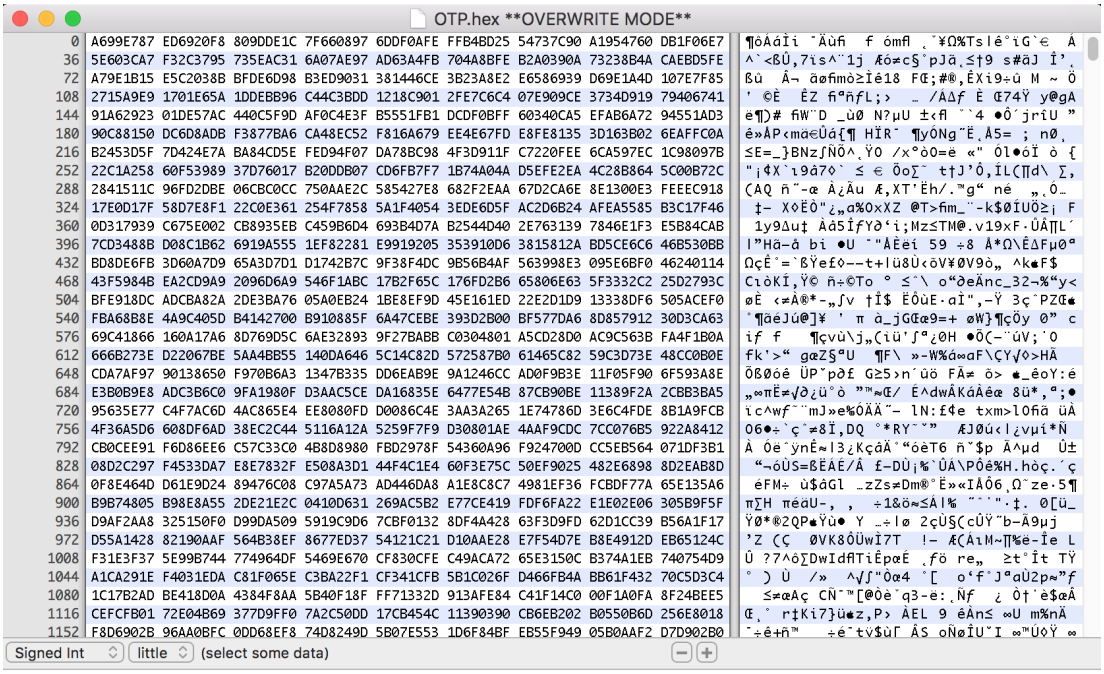

Figure 2. One Time Pad HEX Dump

V. SD Card Communication

The core functionality of the HSM is realized by the storage of the OTP password onto a MicroSD. Devices communicate with the SD in a master and slave fashion. Most MicroSD Card devices support communication over the SPI bus as well as SDIO/MMC. [16]On the HSM side, SPI mode was used for simplicity of implementation of a custom driver for accessing raw data directly, without having to rely on a driver limited to some specific filesystem. This allowed for a simpler implementation of the custom driver necessary to access raw data from the SD without relying on a filesystem-specific driver. This allows the OTP to be stored on unused sectors of a card that also contains a standard filesystem such as FAT32. As a result, such an SD may be usable within a standard computer while the key remains invisible as it can be stored in non-indexed sectors of the card, or encoded into arbitrary files using various stenography techniques.

A. Raspberry Pi Zero

The HSM gets its random OTP password from the hardware random number generator on the Raspberry Pi Zero. The Pi Zero generates the OTP and stores it to the SD Card. Multiple methods for communicating with the SD Card from the Raspberry Pi Zero were explored. Originally the design included using the MMC interface on the Raspberry Pi Zero to communicate to the SD Card. The second MMC interface required a kernel patch to the Raspbian OS that the Pi Zero runs[17]. After patching the kernel, network communication of the the Raspberry Pi Zero did not function impeding communication with the device.Rather than spend more time on the kernel, the SPI bus was selected as the new communication medium.

The Pi Zero supports the two chip select SPI bus needed to write the OTP password to the SD Cards.[18] The initial SPI driver development for the Teensy was used as a base for the SPI driver of the Pi Zero. The SD Card waits for an initialization sequence before it reads or writes data from its memory.[19] Once the SD Card responds with the sequence `0xFF 0x01 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF` the initialization is complete. When connected to the Teensy, the SD Card was able to respond properly, and when connected with the Pi Zero the SD Card responded with `0xFF` instead of the expected `0x01` as seen in Figure 3.Checking the data transmission from the SD Card to the Pi Zero with an Oscilloscope showed large amounts of noise, corresponding to the Chip Select One (CS1) line of the SPI Bus. This noise was greatly reduced by using the CS2 line but reducing the noise did not resolve the communication errors.

The SD Card Breakout Board schematic[20] showed that a level shifting IC TXB0104 was shifting the voltage for use with the SD Card. The TXB0104 datasheet shows that the VCCB input determines the logic level used as input from the master.[21]On the Pi Zero, this was

```
SPI>] r:8 [ 0x40 0 0 0 0 0x95 r:8 ]
/CS DISABLED
READ: 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF
/CS ENABLED
WRITE: 0x40
WRITE: 0x00
WRITE: 0x00
WRITE: 0x00
WRITE: 0x00
WRITE: 0x95
READ: 0xFF 0x01 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF
/CS DISABLED
SPI>
```

Figure 3. SD Card SPI Example

connected to 5V, even though the Pi Zero uses 3.3V logic on its pins Changing the circuit to connect VCCB to 3.3V on the Pi Zero resolved the initialization failure the SD Card.

The SpiDev library used with the Raspberry Pi Zero brought some limitations of its own. The library always de-asserted the CS line at the end of a transfer.[22] There was no method to transfer data without asserting the CS line. The SD Card needs 80 clock cycles of the Serial Clock line to synchronize before it accepts some commands.[19]

The difficulties with the SPI Bus on the Pi Zero lead to the discovery of the full utility of a popular Unix utility `dd`. Using `dd`, binary data could be transfered from one device to another. Using a USB to MicroSD card adapter and the command `sudo dd if=/dev/hwrng of=/dev/sda bs=8192` the data from hardware random number generator was written directly to the blocks of the MicroSD Card. This was a time consuming process running at a rate of 120kB per second as seen in Figure 4. At this rate, the entire 16GB card would take about 39 hours to be filled with OTP data. 120kB provides the user with

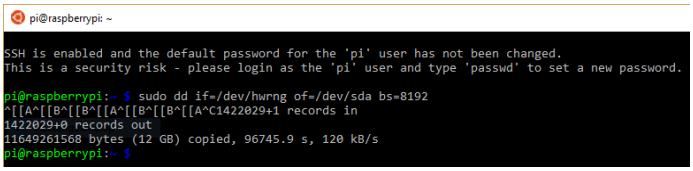

Figure 4. Hardware RNG Rate

750 text messages consisting of 160 characters, each one byte in size. Thus each second the generation was ran increased the total number of text messages the user can transmit by approximately 750 full length texts.

B. Teensy

The Teensy, like the Pi Zero, needed direct access to the SD Card for core functions of the HSM to be realized. Using the Teensy hardware SPI interface, a custom SD driver was written for initialization and block read/write commands. The commands reference sheetserved as the guide for implementation[23]. Many of the commands sent to the SD Card are sent with no expectation of data reception during the transmission. The SPI driver for the Teensy consumes the buffer where the transmit command is, by overwriting the bytes with the data received during the transmit. The function `void sendSPICommand(const byte * command, const int length)` was written to support transmitting these commands, as it creates a new temporary buffer that is removed from RAM at the end of the function.

VI. HSM Implementation

The HSM consumes data from a MicroSD Card for each corresponding password, and external communication link for message reception. The HSM also takes user input directly, rather than from another external device. A PS\2 Keyboard and LCD screen provide the user with an interface to compose and read messages sent to and from the device. (For absolute defense against RF-based attacks, a custom keyboard PCB must be used with identical trace lengths for each key, the LCD must be swapped for an E-Ink display, and the physical wires connecting HSM to the internet gateway must be replaced with a custom solution where data exfiltration is fully prevented even if both HSM and gateway contain HW backdoors controlled by the same adversary).

A. PS\2 Keyboard

Interface with the keyboard was performed using a library created for use with the Teensy.[24] The power, common, and single data line are connected from the keyboard to the Teensy. The library abstracts interpreting the keyboard input from the user. Repeatedly calling the `keyboard.available()` function polled the keyboard driver for more input. The characters returned were printed to the LCD screen and stored to a buffer so they could be encrypted, on demand, by the user. The library also provides constants to detect additional keys. These constants are used to map the `Page Up` and `Page Down` keys to send and receive message respectively.

B. LCD Screen

As with the keyboard, the LCD Screen interface uses a library. The Adafruit-ST7735-Library abstracts most

of the work of printing text to the LCD Screen into simple functions.[25] The basic functions did not include support for a ”backspace” command hence backspace support was added by moving the text cursor back to the previous character and then covering the character with a rectangle the color of the background as seen in Figure 5. The buffer location that held the data from the keyboard is overwritten by a `NULL` to prevent that character from accidentally becoming part of the encrypted message.

```
int newX = tft.getCursorX();
int newY = tft.getCursorY();
if(newX!=0 || newY!=0)
{
   if(newX==0)
   {
      newY=newY-8;;
      newX=150;
   }
   else
   {
      newX=newX-6;
   }
   tft.setCursor(newX, newY);
   tft.fillRect(newX, newY, 6, 8, ST7735_BLACK);
   inputIndex--;
   inputMessage[inputIndex]='\0';
```

Figure 5. Backspace Handling

VII. Encryption\Decryption Process

The encryption and decryption process begins at the command of the user. Pressing the `Page Up` key on the keyboard begins the encryption process of the data on the screen. This key press triggers a call to the `encryptMessage()` function, which performs the task of moving the data from the buffer to the UART port. First the block at address zero of the SD Card is read. This block contains the information of the most recently used portion of the OTP password. The ending location and the length of the input buffer from the keyboard are used to determine the number of OTP blocks that need to be read back from the SD Card. More than one read is needed if the length of the message passes over the boundary of a 512 byte SD Card block. The OTP password is aligned to the start of the input buffer, and the function `oneTimePad()` performs the bitwise exclusive or (XOR) operation on the message. The encrypted message is then appended to the plain text header and sent over UART to the communication channel. After the transmission, the header information is updated and written back to address zero of the SD Card.

The decryption process is performed in a similar fashion to the encryption process. The major difference is that the decryption can not happen at just any time. The user must wait for a message to be sent back to them from another user. When a new message is received from the communication channel over UART, the LCD Screen displays a `Message Received...` message to indicate to the user that a new message has arrived. Upon pressing the `Page Down` key, the Teensy begins the decryption process of the new message. The first seven bytes of the new message are broken down to determine the location of the OTP password on the SD Card that was used to encrypt the message. As happens with the encryption, the decryption function retrieves the OTP data from the SD Card and aligns it to the beginning of the received encrypted message. After the XOR operation is performed, the decrypted message is printed to the LCD Screen. Once any portion of the OTP key is used on sender/receiver side, the corresponding data on SD card is overwritten at least 42 times by random data (pseudorandom is fine for this) effectively eliminating any trace of the original message from physical existence (encrypted data transmitted over the public web contains exactly 0 information about each message, thus is not susceptible to brute force attacks even with a hypothetical perfect Quantum Computer the size of the Observable Universe utilizing over 10^{80} quantum bits.)

VIII. Wi-Fi Communication Module

In order to support transmission of messages from the sender and receiver, a communication channels needs to be selected. One of the sample implementations within this project is a Wi-Fi Communication Module (WCM). The WCMs for each side of the channel are implemented using a Particle Photon and a Particle Core. Both of these devices use an ARM 32-bit Cortex™ M3 CPU.[26][27]The WCM is connected to the Internet using the Particle Cloud. The Particle Cloud accepts messages ”published” to a communication channel, and allows Particle devices like the Photon and the Core to subscribe to these publications. The Photon and the Core are automatically notified when a message is published. Figure6 shows an image of one of the modules used, a particle Core.

The Particle Cloud only allows data represented as ASCII text to be sent through the publish channel.[28] This presented a problem as the message, once encrypted, is comprised of all 8-bit hex values from `0x00` to `0xFF`. Many of these characters are not printable[29] and are sometimes called control characters. The Particle Cloud would not allow these characters to be transmitted. To work around this limitation the Core and the Photon

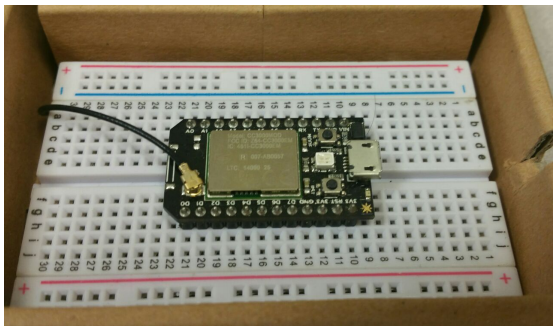

Figure 6. WiFi-Enabled Microcontroller used for External Communication

used the Base64 encoding. The encrypted message encoded in Base64 is then represented completely by printable characters as showing in Figure 7.[30]To save time on implementation, a Base64 library was used rather than created from scratch.[31] The library supported the encoding and decoding of the published data. The second

Table 1: The Base 64 Alphabet

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w	(pad)	=
15	P	32	g	49	x		
16	Q	33	h	50	y		

Figure 7. Base64 Characters

limitation of the Particle Cloud publish feature is the 255 byte per publish limit. The data is buffered and then chunked into 255 byte messages as it is pushed to the Particle Cloud. Figure 8 shows how the message in Figure 9 can be broken up into two events in the cloud.

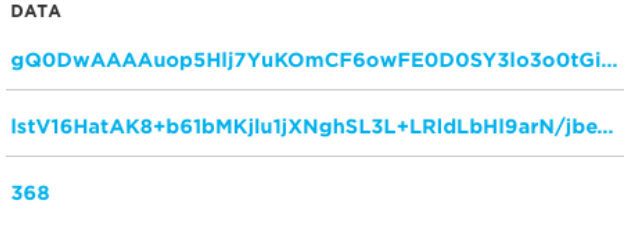

Figure 8. Particle Console Log of a Transmitted Message

The length 368 means that the full message is broken up by the Particle Photon. To tell the receiving node how much data to expect, the length of the total Base64 data is transmitted on a separate publish channel. The receiving node compares this length against the amount of data it received during the transfer. Wi-Fi connections do not guarantee the transmission of a data packet, and the Core and Photon are designed to clear there buffers and reset to a waiting flag for message state if a timeout of five seconds or longer occurs. Once the Core and the Photon have received all of the Base64 characters, the message is passed to the decode function of the Base64 library. The message, now in its transmitted form, is passed using a UART channel to the receiving HSM at 19200 baud.

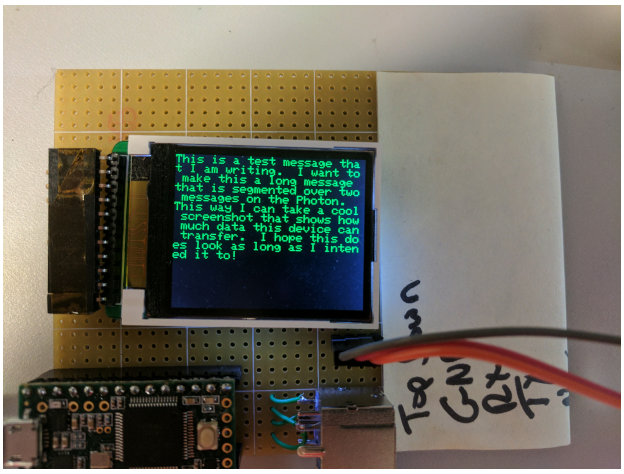

Figure 9. A long message that is converted to Base 64

IX. Development, Integration and Testing

Validation of the HSM was performed at all steps of the development process. Each component of the software was written individually, tested, then integrated into the system. The integrated components were tested to ensure that they worked together in a reliable and deterministic manner. Figure 10 shows the high level code flow of the execution path in the HSM.

A. SD Card Driver

The first developed component was the SD Card driver. In order to quickly test the SD Card driver, external devices were used to validate its performance. Using an SD Card writer, a test SD Card was filled with a known pattern `0x12`. The read function was validated when it returned the known pattern. A similar test was used for the write function. The SD Card reader was used to validate the expected written pattern. In the process of using a raw SPI driver to both initialize and control the SD card, a 4-channel oscilloscope provided extremely valuable information during initial testing. An image capture of the setup is provided in Figure11.

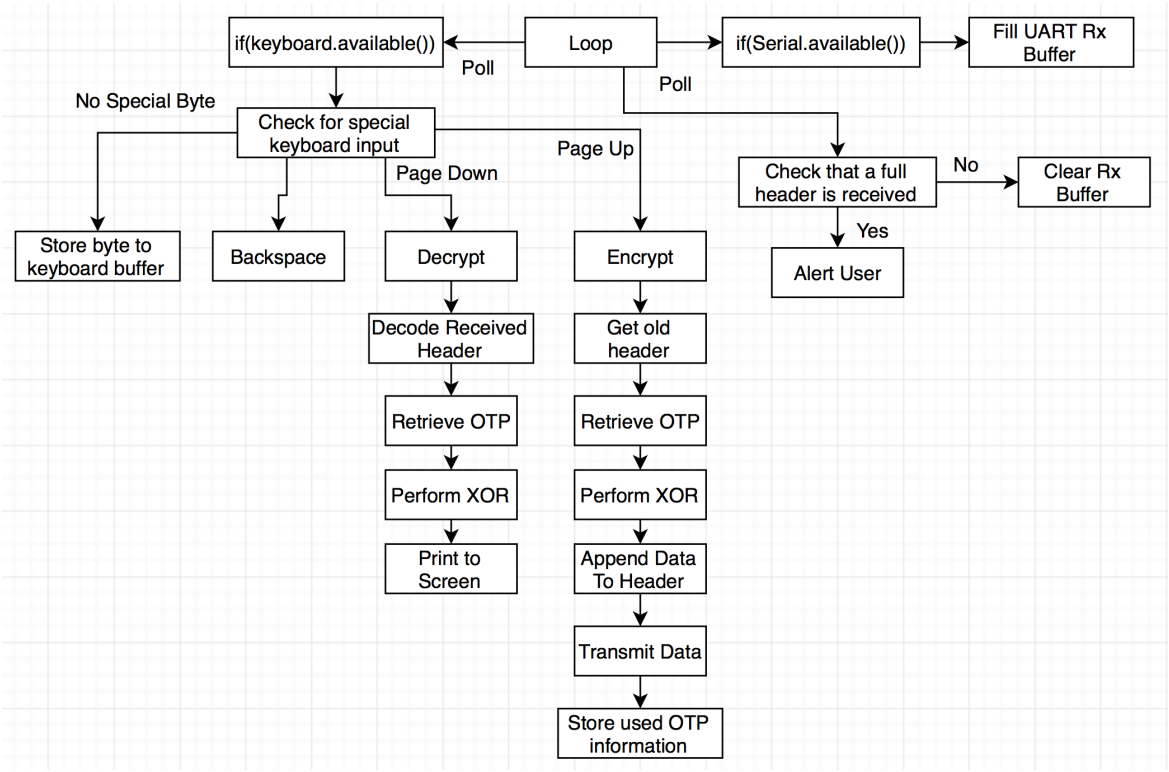

Figure 10. Program Flow

Figure 11. Debugging SPI Commands for SD Card Initialization

B. Message Header

The header was created to describe the encrypted data. It holds a single bit to detonate the start and the version. As seen in Figure 12, it contains 15 bits for the size, allowing 4kB messages to be transmitted. The third field is the location, which points to the specific bit of the 512 byte block in the SD Card making this field 12 bits wide. The last field is the address filed which is 28 bits wide to accommodate the entire size of the 16 GB SD Card used. The header functions were passed small data segments, and the produced header was verified to match the length and location of the passed data. If the first bit is a 0 then a longer version number can be used with a different header structure supporting larger a SD/USB/HDD.

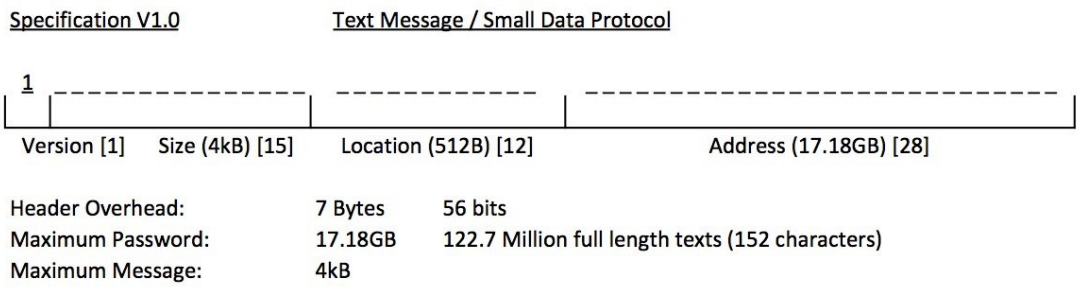

Figure 12. Custom Developed Protocol for Message Header, Version 2 must be designed for OTP keys above 16GB

C. LCD and Keyboard

The LCD and Keyboard libraries contained examples which displayed a known pattern on the LCD and passed the pressed keys to a UART connection. The pressed keys were then passed to the LCD print function, rather than a UART channel, to complete the integration of these two components. The encryption and decryption commands were added and the input buffer was passed to the encryption function rather than a hard coded character array.

D. External Communication

The HSM performs no formatting of the output data other than the addition of the header. To integrate the external communication channels, the devices were designed to accept the data as the HSM presented it. Figure 13 shows a sample message transmitted from the device. The encrypted data looks like garbage to the anyone who receives it. The header is visible in this transmission. The header `0x80 0x74 0x03 0xC0 0x00 0x00 0x02` can be deciphered as protocol version 1, 116 bytes in length, and starting at byte 60 of address 2.

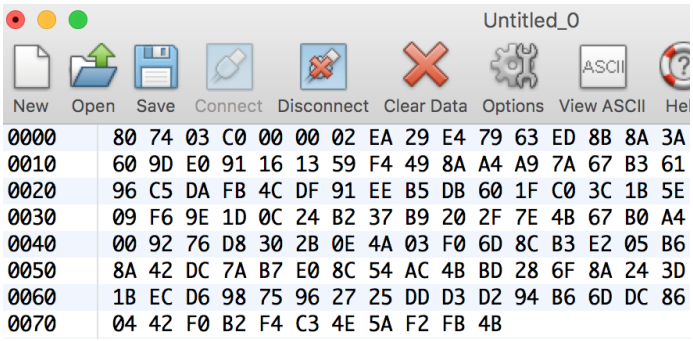

Figure 13. UART Transmission from the HSM

E. Final Integration Testing

To test the complete integrated HSM components, test messages were transmitted from the HSM to a PC over the communication channel. The PC then recoded the encrypted data it received. The HSM was powered off and back on to ensure that the message was cleared from all buffers. The PC then replayed the data it received back to the communication channel. The same text, that was transmitted in the beginning: `Testing`, was then verified and displayed on the screen. An image of the completed prototypes is provided in Figure14. The 3-pin header on each board is utilized for serial communication to the external communication module. Figure15 shows the wiring on the back side of one of the prototypes. The MicroSD interface was embedded within the LCD module as can be seen in Figure16 showing the back side of the screen.

F. Limitations and Improvements

While the device created supports a very strong method of encryption and mitigates a number of hardware vulnerabilities, there is still room for improvement in the design to increase the overall security.

G. OTP + Additional Encryption

The initial implementation of the HSM leaves the SD Card vulnerable to being stolen or lost and the OTP key being recovered. A first step towards better securing the secrets would be to encrypt the OTP itself (i.e. AES block cypher) before it is written to the SD Card. This would act as an additional layer of security for the key. If the SD Card were to become lost or stolen an attacker would need a means of breaking the standard encryption to get the OTP keys (this only impacts future messages, prior transmissions are still safe).

H. OTP Overwrite

Once a full 512 byte block (or a portion) of the OTP has been used for sending or receiving messages, it must be overwritten repeatedly with pseudo-random data to mask the initial OTP password. After enough rounds of

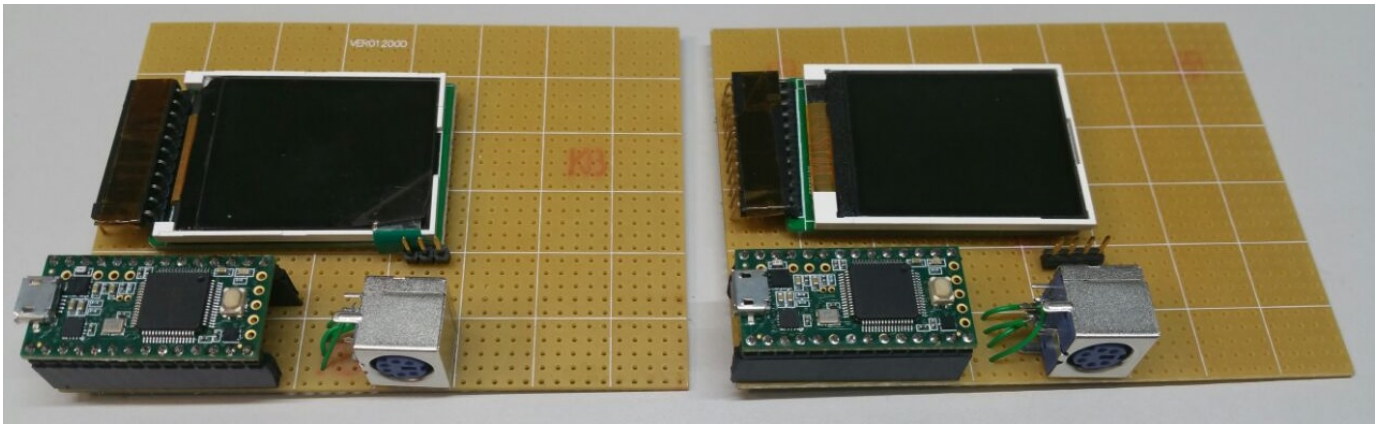

Figure 14. Completed Prototypes

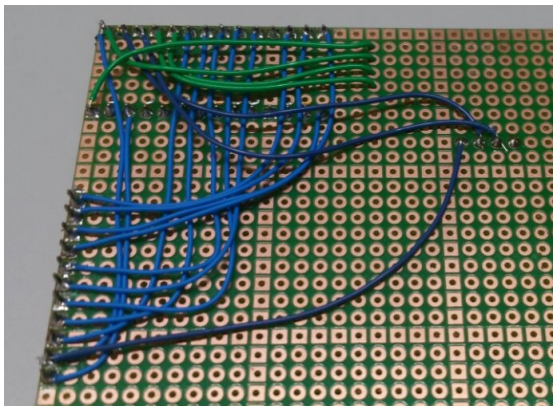

Figure 15. Wirinig/Interconnection on the Back

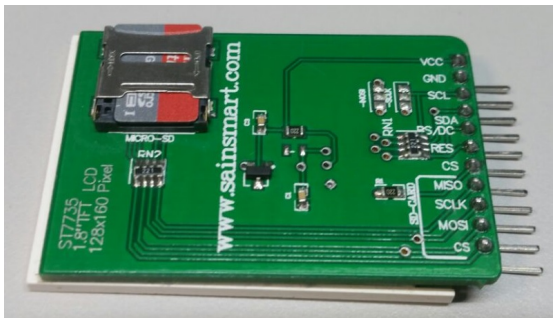

Figure 16. Reverse Side of LCD Module with SD Interface

over-writing it will become near impossible to read back the old OTP, that was once on the card. Normally, SD Cards employ a wear leveling technique which would render this operation useless. The HSM accesses the SD Card blocks directly, allowing it to target a specific location. To even further advance the abilities of this wipe, the final overwrite can be a OTP which will produce an expected message. This could be a message that would give adversaries enough information to think that they have discovered a secret. Effectively the just-used key could be overwritten by the XOR of received/transmitted message with a completely separate string specified by the end user. The original message which could have been captured during transmission would XOR with the corresponding key to the preselected alternative string. Thus the key itself could be safely given up upon forceful request, revealing seemingly a comprehensible message rather than random jumbled data.

I. E-Ink Display

The HSM currently displays its data on an LCD Screen. LCD Screens need to be constantly refreshed to hold their display values. The leaves the HSM open to a remote side channel attack where an attacker can use a high powered RF Receiver to capture the data going to the LCD display[32]. Using statistical analysis, the attacker may be able to filter out the noise on the signals by comparing the many samples from the constant refreshes of the screen. With the noise filtered, the attacker may be able to determine the signals sent to the LCD screen and determine the data it held. This would possibly allow the message to leak before encryption can be applied. E-ink displays may help partially mitigate the problem by only refreshing the display on necessary changes, however the signal lines may still be susceptible to such attacks.

X. Maintenance

With a fully implemented system, the final phase of the project is the maintenance phase, where bugs are found and updates are released to resolve functional and security issues. The evolution of the protocol is important for adding larger password block support (over 16GB) and improving security as novel attack vectors are discovered/published.

XI. Extra

A. Communication over SMS

In addition to supporting Wi-Fi, a communication channel over SMS using the GSM network was developed. This is to show the versatility of the HSM, and protocol, showcasing its ability to adapt its core functions to be used in many different ways. The GSM communication functions in a similar fashion to the Particle Cloud communication. The SMS messages only support passing of printable text characters. A test text message in Figure 18 sent to a phone showed some missing characters in the reception. The transmitted message was `THIS IS A TEST`. Base64 library was again used to convert the binary data to a printable format that SMS can transmit. SMS messages are also limited to 160 characters per message. This limited the length of the message that could be sent to about 140 characters.

The SMS protocol has to account for the overhead of the Base64 conversion. A future improvement would be to allow segmented SMS messages in a similar fashion as the WiFi communication. The data is passed to an Arduino over UART, which then performs the encoding and transmits the message to a SIM900 modem, connected to the T-Mobile network. The SIM900 is a GSM enabled IC which requires a SIM card registered on a GSM network to operate. Two identical modems were used, one for each endpoint. To provide further isolation, the interface to the modem was handled by an Arduino rather than the Teensy directly. Figure 17 shows both pairs of hardware used to for this portion. The phone

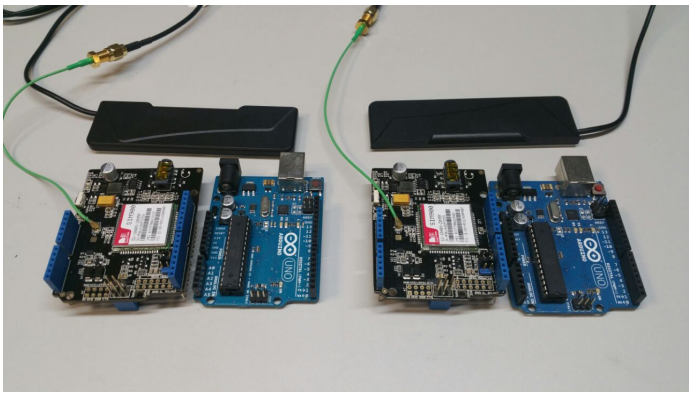

Figure 17. 2x Sim900 Modem and Arduino Microcontrollers used for Text Communication over the Cell Network

number that the GSM module is sending the SMS to is hard coded into the Arduino. After the message is sent out, another SIM900 then receives the SMS message and passes it to the Arduino on the other end. The Arduino performs the decoding of the Base64 data and passes the binary data to the HSM over another UART connection. This allows for pin compatible implementation of both forms of communication simply by connecting the HSM to either of the external devices. WiFi can be utilized when available and for large messages, while the SMS device allows communication from remote portions of the globe, within cellphone network coverage.

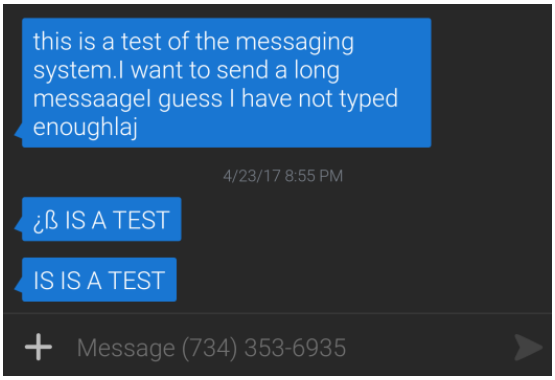

Figure 18. SMS Message Missing Characters

B. Validating true randomness of generated key

The One-Time-Pad encryption algorithm provides absolute security only if the utilized keys are truly random. For this reason, a pseudo random number generator(PRNG) was avoided altogether. While a number of high-bandwidth commercial TRNGs are available, the majority were outside the scope of a small project budget. High energy laser-based beam splitters and Geiger counter solutions were deemed infeasible for personal device use. Software-defined radio solutions were explored, where random data is generated from atmospheric noise picked up by the receiver[33]. While feasible in principle, the throughput would be highly limited and intentional manipulation of RF during key generation would remain a potential problem that could compromise the overall entropy of generated password.

The most viable solution was to rely on thermal noise inherently present on analog components within a circuit. Fortunately, the Broadcom BCM2835 chip on the raspberry pi zero contains a hardware random number generator module, ready to use; However, the device itself cannot be trusted to behave according to specification. Intentional modification may compromise the generator and return a bitstream that contains an inherent pattern which could later compromise the entire secure system.

A sample password was generated and tested for testing. The TRNG was run for one minute, producing 55.5 million random bits, stored under the filename 'random_gen_60s'. First test was a simple visual

inspection to determine if any patterns may exist within the bits represented graphically. The bitstream was piped into an image using the following set of linux commands: `sudo cat random_gen_60s | rawtoppm -rgb 1024 512 | pnmtopng > random-test1.png`. The image produced is provided in Figure19. Although no patterns can be

Figure 19. TRNG Data Represented as an Image

visually detected, the basic pseudorandom number generator was used in the same configuration as a baseline for comparison. As it turns out, the PRNG also produced no visually noticeable patterns, despite the high probability that a pattern exists, even if it is highly complex. The PRNG output represented as an image is provided in Figure20. A more thorough

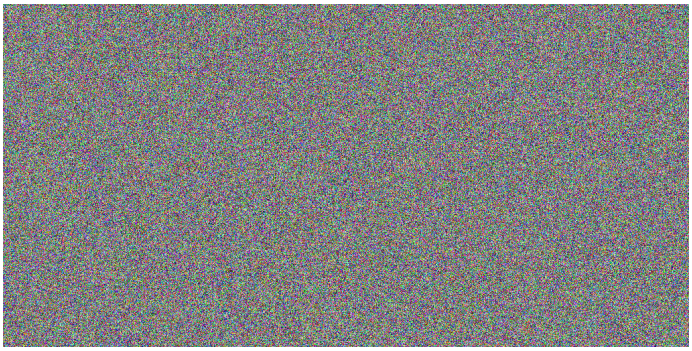

Figure 20. PRNG Data Represented as an Image

test is necessary to determine whether the TRNG data was truly generated from an unpredictable source. The US government standard utilized for approving cryptographic modules is the Federal Information Processing Standard(FIPS) Publication 140-2[34]. While official compliance testing is typically a highly expensive process, a free linux utility called rngtest, written by Henrique de Moraes Holschuh, was utilized for home verification[35]. The test was run with a block count of 10000(greater than file to ensure all bits are tested), with the bitstream from 'random_gen_60s' piped as an input. Per the test results, 2775 blocks passed the test and only 3 failed. With only 0.1% of the blocks failing, the results provide convincing evidence that the TRNG based on Raspberry Pi Zero truly works as intended.

```
user1@linuxbox1:~/Desktop$ sudo cat random_gen_60s | rngtest -c 10000
rngtest 4
Copyright (c) 2004 by Henrique de Moraes Holschuh
This is free software; see the source for copying conditions.  There is NO warra
nty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

rngtest: starting FIPS tests...
rngtest: entropy source drained
rngtest: bits received from input: 55574528
rngtest: FIPS 140-2 successes: 2775
rngtest: FIPS 140-2 failures: 3
rngtest: FIPS 140-2(2001-10-10) Monobit: 0
rngtest: FIPS 140-2(2001-10-10) Poker: 1
rngtest: FIPS 140-2(2001-10-10) Runs: 1
rngtest: FIPS 140-2(2001-10-10) Long run: 1
rngtest: FIPS 140-2(2001-10-10) Continuous run: 0
rngtest: input channel speed: (min=1666666666.667; avg=27821732598.898; max=0.00
0)bits/s
rngtest: FIPS tests speed: (min=12.623; avg=76.038; max=89.547)Mibits/s
rngtest: Program run time: 709016 microseconds
```

Figure 21. FIPS 140-2 Randomness Compliance Test with rngtool

C. Research: TEMPEST

Despite vigorous attempts to create a truly secure system, even the best efforts can only help mitigate the effect of mass-implemented surveillance. Protecting against directed espionage is a much more difficult task. While the 'Limitation and Improvements' section covered a number of techniques to make the system near perfectly secure, the end device may still be susceptible to radio frequency emanation attacks. Such attacks fall under the blanket term Tempest, which is an official NSA specification as well as NATO certification for resistance against emanation-based attacks[36][37]. While the term is most commonly associated with RF leakage, sound, vibrations, and optical signals can also be susceptible to spying through such means. Perhaps the most famous use of such techniques for serious espionage is Leon Theremin's bug often simply referred to as "The Thing"[38]. The device consisted of a resonant cavity microphone embedded into a wooden seal gifted to the US embassy by the USSR on August 4th, 1945 as a gesture of friendship coming from a group of Soviet children. After thorough inspection, it was hung in the US ambassador's lobby for 7 years before it was accidentally discovered by British radio operators. The entire device was comprised of an antenna and a small diaphragm contained inside a half cylinder. No batteries or active circuitry was used, yet it allowed the soviets to exfiltrate voice communications when the device was illuminated using a powerful RF beam around 800MHz. The reflection of the illumination beam was effectively modulated by the passive microphone inside the room[39].

Descriptive	Full	Intermediate	Tactical / Basic
SDIP-27 / 1NATO Standard	Level A	Level B	Level C
NATOLaboratory test Standards	AMSG-720B	AMSG-788A	AMSG-784
NATOZoning Standards	ZONE 0	ZONE 1	ZONE 2
USA NSTISSAM / 1-92 Standards	LEVEL I	LEVEL II	LEVEL III

Table I: Tempest Standards Publicly Available

Within the public domain, little progress has been available in the application of such technology. The basic principle gave rise to RFID communication, but the information leaked by Edward Snowden in 2013 provided a decent insight into the powerful advancements in espionage utilizing such technology[40]. A large number of unique tools are available within the NSA arsenal including devices that allow spying on remote video signals, or any other digital communication with an approach quite similar to that implemented by Theremin[41].

Furthermore, eavesdropping attacks are no longer limited to well funded government agencies. Markus Kuhn demonstrated the capability of remotely capturing a computer monitor using a relatively affordable directional antenna and wideband receiver[32].Because of high refresh rates of modern monitors(generally 60hz), many iterations of the same signal get transmitted over the wire. Although leakage during transmission is minimized, many samples of the same data can be utilized for noise rejection allowing significant amplification. While a partial solution, mentioned previously, was to use an e-ink display to reduce refresh rate, the problem is not fully eliminated. In order to mitigate emanation-based attacks, further techniques must be employed.

While most of the exact methods for preventing Tempest attacks remain classified, some shielding specifications are publicly available. Devices are sorted into three categories based on level protection requirements. The publicly available classification standards are provided in Table I. (NATO SDIP-27) is the strictest of the three and centers on the assumption that an attacker may have immediate access to the hardware. For ultimate security, this certification should be the goal of a proper HSM. Because of the incredibly high costs incurred not only designing to meet the specification but also for the testing itself, security against tempest-style attacks may be limited to devices built by well funded companies and governments.

XII. Conclusion

By using common off the self components and simple yet powerful encryption methods, it is shown that a robust communication system can be implemented. The design takes steps to ensure that even with knowledge of the methods, an attacker will be unable to successfully intercept a communication it secures. The components chosen can be sourced by anyone and integrated into a complete solution which can utilize a communication channel of the builders choice. Such a device can provide a strong level of protection for confidential message exchange between users in remote regions of the world. Unfortunately, as with nearly any physical system, the proposed device may still be susceptible to targeted attacks by a perpetrator with physical access to the endpoint to make modifications. Nonetheless, the implemented system provides a powerful level of defense against some of the most common, broadly implemented, surveillance techniques.

Appendix

A. Project implementation on FPGA

The proposed communication system is not only agnostic to hardware choice/variation but also allows for exotic hardware variants to communicate with regular CPU-based devices. The first step in designing an application specific integrated circuit(ASIC) generally starts with a prototype on a field programmable gate array(FPGA). A Spartan 3E FPGA from Xilinx was utilized as the hardware security module incorporating the specifications/header and designed for inter-operation with ARM/x86 devices. Each necessary subsystem was individually developed, tested, and then combined with others. The final system block diagram is provided in Figure 22.

1) PS/2 Keyboard Input: Because the Basys2 board contains a PS/2 header, no external hardware is necessary to communicate with a keyboard. A standard PS/2 keyboard controller written by Scott Larson was integrated into the project[42]. The code utilizes latch-based debounce code for the 'clk' and 'data' lines both relying on the "debounce.vhd" module. Using a 50MHZ clock, a 5 microsecond delay is achieved by using an integer counter of size 8. The keyboard logic is handled within 'ps2_keyboard.vhd' which checks for activity on the bus, verifies signal parity, and outputs the corresponding message. This output must then be

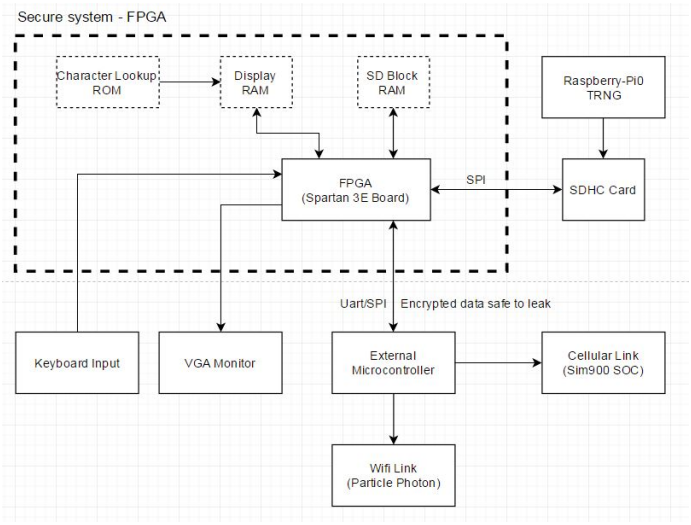

Figure 22. FPGA System Block Diagram

converted to ASCII format for compatibility with the rest of the project. The conversion is performed within 'ps2_keyboard_to_ascii.vhd' relying on large 'CASE' statements for direct conversion. Capslock, control, and shift buttons set register-based error codes which control the overall ASCII translation process. The entire keyboard controller is packaged up as a single component and integrated directly into the top module within the project. Figure 23 shows the final component port definition.

```
component ps2_keyboard_to_ascii
    port (
        clk          : IN  STD_LOGIC;                     --system clock
        ps2_clk      : IN  STD_LOGIC;                     --clock signal from PS2 keyboard
        ps2_data     : IN  STD_LOGIC;                     --data signal from PS2 keyboard
        ascii_new    : OUT STD_LOGIC;                     --output flag indicating new ASCII value
        ascii_code   : OUT STD_LOGIC_VECTOR(6 DOWNTO 0)   --ASCII value
    );
end component;
```

Figure 23. Keyboard Controller Module

2) VGA Display: Creating a responsive graphics controller for text output took a bit more work than just getting the keyboard input. Many different approaches were tested each encountering a number of different problems. The screen output had to be fully responsive to immediately reflect characters being typed on the keyboard. Finally, a piece of code written by Kevin Lindsey was tested producing satisfactory results[43]. Almost every approach tested utilized some form of ROM to store bit patterns corresponding to ASCII character codes, but this time a 7x3840 block of RAM was also used. The active state of ASCII characters mapping to positions on the screen can be read and written to in real time and at any address. Although the initial code served as a proof of concept of the functionality, significant changes were made to support numerous features required by this project. First the block RAM IP core had to be regenerated to comply with minimum version requirements within Xilinx 14.7. Because the pixel driver utilized the upper 5 bytes of y-counter and upper 7 bytes of x-counter for the RAM address, the effective line length represented in ram ended up 128 characters. Because only 80 could fit on the screen, a quick workaround was to increment 81st character position by 64 rather than just 1. Such an approach does waste some of the ram, but a more efficient solution would require restructuring the entire display driver. The overall interface of the VGA port specific to the Basys2 board is shown in Figure25. In this case, to give

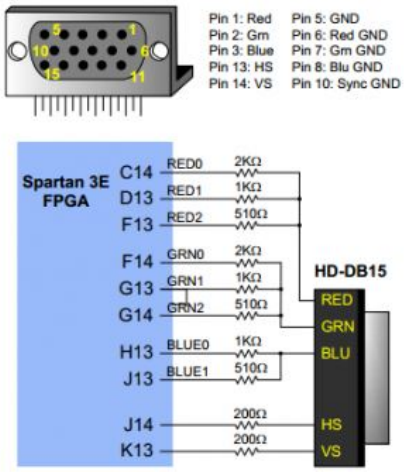

Figure 24. VGA Pin Definition and Basys2 Circuit [44]

the project a retro-style feeling, green color characters were used against a black background. Effectively only the 3 green lines were driven(F14,G13,G14) with the rest connected to ground. The main VGA driver is contained within 'vga_sync.vhd'. The display area as well as horizontal and vertical porches is hardcoded here as well. A small deviation from standard specification for 640x480 had to be made to better align text to the screen: the vertical back porch was set to 37 clock pulses instead of 31. The 50MHZ clock is divided in half using a simple register-based clock divider to provide a 25MHZ clock necessary to drive a display at such resolution. The horizontal and vertical sync pulses are generated from this clock and 10-bit pixel vectors are incremented as the display is scanned.

The font ROM implemented in 'font_rom.vhd' did not require any changes and was utilized to directly feed 'video_ram.vhd' with bit patterns corresponding to ASCII encoding, managed within 'font_generator.vhd'. Here the logic for supporting backspace and next line output was also implemented along with a custom control structure to switch between displaying typed

message and one received from an external device. The font generator and VGA signal driver are both combined within 'font_test.vhd' which controls the final RGB buffer used to drive physical pins. Figure25 shows the final interface of the screen driver module.

```
component font_test
   port (
      clk, reset: in std_logic;
      hsync, vsync: out std_logic; --horizontal and vertical sync pulses
      rom_write: in std_logic_vector(0 downto 0); --write character to display signal
      write_char: in std_logic_vector(6 downto 0); --character to write
      rgb: out std_logic_vector(7 downto 0); --physical signal driver for VGA screen
      main_send_cmd : out std_logic; --send characters from display for processing
      main_read_char : out std_logic_vector(6 downto 0); --pass character on screen at specific index
      main_read_size : out integer; --size of message
      main_read_char_num : in std_logic_vector(11 downto 0); --character index to read
      main_display_cmd : in std_logic; --display received message initiated by main controller
      main_display_size : in integer; --number of characters in received message
      main_display_char : in std_logic_vector(6 downto 0); --actual character to be displayed on screen, from received message
      main_display_char_num : in std_logic_vector(11 downto 0) --iterate over each character from received message
   );
end component;
```

Figure 25. VGA Screen Controller Module

3) Micro SDHC Card: Block Read and Write: Four separate approaches were tested with each one failing for different reasons. The most difficult portion was initialization of the card which requires a sequence of commands specific to the type of card. One of the libraries from XESS corp was finally successful, but required a number of custom VHDL packages to get it to work[45]. The initialization function was thus handled within 'SdCardCtrl.vhd' but the read and write functionality required further development. Within the SDHC protocol memory is addressed in blocks of 512 bytes, with 32 bits for the address to designate which block. This effectively allows the SPI protocol to be utilized with cards as large as 256GB in size. The I/O interface for the utilized module is provided in Figure26. Although testing was successful in reading and writing

```
sd_controller1: SdCardCtrl
   port map (
      clk_i => clk,
      reset_i => reset,
      rd_i => spi_read,
      wr_i => spi_write,
      continue_i => '0',
      addr_i => X"00000000",
      data_i => X"00",
      data_o => block_write,
      busy_o => open,--leds(1),
      hndShk_i => trigger,
      hndShk_o => open,--leds(0),
      error_o(7 downto 0) => open,
      error_o(15 downto 8) => open,
      cs_bo => cs,
      sclk_o => sclk,
      mosi_o => mosi,
      miso_i => miso
   );
```

Figure 26. SPI SD controller Module

to SD, as verified by outputting individual blocks to an LED array, further difficulty was encountered when handling both an initial loop as well as further access. It was noticed that the SD would exhibit sporadic behavior after reset, which seems to be a limitation of the utilized module. As a workaround, the 512 byte ram was pre-initialized with the first block of the SD read utilizing an external tool. This allowed for development of the overall device to continue leaving complete integration directly with SD up to future work.

4) UART Driver: Read/Write to External Module: A UART master controller written by Colin Riley was used as the starting point for serial communication[46]. To set the baud rate at 19200 under the 50MHZ input clock, a counter was initialized to x"0A2C" since 50,000,000/19,200 = 2604. The initial serial controller was only designed to handle a single byte of data, but this was enough to create a full serial driver for any length messages passed both ways. The base driver is implemented within 'uart_simple.vhd' and is utilized directly within the top module. Its component interface is provided in Figure27. Within the top module, 2 separate

```
component uart_simple
   Port (
      I_clk : in  STD_LOGIC; --system clock
      I_clk_baud_count : in STD_LOGIC_VECTOR (15 downto 0); --number of cycles between baud ticks
      I_reset: in  STD_LOGIC;
      I_txData : in  STD_LOGIC_VECTOR (7 downto 0); --data to transmit
      I_txSig : in  STD_LOGIC; --transmit signal
      O_txRdy : out  STD_LOGIC; --'1' when idle, '0' when transmitting
      O_tx : out  STD_LOGIC; --physical serial output
      I_rx : in STD_LOGIC; --physical serial input
      I_rxCont: in STD_LOGIC; --receive enable
      O_rxData : out STD_LOGIC_VECTOR (7 downto 0); --data received
      O_rxSig : out STD_LOGIC; --data available signal
      O_rxFrameError : out STD_LOGIC --frame error output
   );
end component;
```

Figure 27. UART Driver Module

processes control the read and write functionality for passing entire messages between FPGA and external microcontroller. The 7 bytes of overhead are accounted for in there as well. The external module handles the external channel of communication to bridge the encrypted data between endpoints.

5) Integration: The functionality of all the individual subsections was finally combined within 'keyboard_vga.vhd' with a large number of necessary registers to facilitate the coordination of all parts. The first 2 processes are utilizes solely for controlling read and write functionality of messages to the serial bus. The next process handles reading from SD one byte at a time to store into RAM. This portion is still under development. The last 2 short processes were utilized for generating a clock divide vector as well as custom trigger clock for SPI operations. The majority of the work was spent tailoring the coordination and communication between 2 FPGAs as well as between FPGA and ARM-based CPU. Although system functionality was finally achieved, much work remains to be done in cleaning up the code and testing under a wider array of use cases.

While an ASIC can offer greater performance than FPGA implementation, fully relying on a complete hardware solution built by a 3rd party goes against the spirit of the core idea presented here. Effectively one must build their own secure phone (using hardware of their choice) for absolute security and this paper only acts as a basic guide to achieving those means.

REFERENCES

[1] E. Chabrow, "Did nsa influence taint it security standards?" [Online]. Available: http://www.bankinfosecurity.com/nist-reevaluating-cryptography-guidance-a-6068

[2] J. K. Elaine Barker, "Recommendation for random number generation using deterministic random bit generators," Jun 2015. [Online]. Available: https://en.wikipedia.org/wiki/Dual_EC_DRBG#cite_note-NIST-800-90-1

[3] J. Menn, "Exclusive: Secret contract tied nsa and security industry pioneer." [Online]. Available: http://www.reuters.com/article/us-usa-security-rsa-idUSBRE9BJ1C220131220

[4] "Revealed: how us and uk spy agencies defeat internet privacy and security." [Online]. Available: https://www.theguardian.com/world/2013/sep/05/nsa-gchq-encryption-codes-security

[5] J. Larson, "Revealed: The nsa's secret campaign to crack, undermine internet security." [Online]. Available: https://www.propublica.org/article/the-nsas-secret-campaign-to-crack-undermine-internet-encryption

[6] B. Schneier, "Computer network exploitation vs. computer network attack." [Online]. Available: https://www.schneier.com/blog/archives/2014/03/computer_networ.html

[7] D. Walter, "The best messaging apps with end-to-end encryption." [Online]. Available: http://www.greenbot.com/article/3119449/android/the-best-messaging-apps-with-end-to-end-encryption.html

[8] "$300,000 for cracking telegram encryption." [Online]. Available: https://telegram.org/blog/cryptocontest

[9] "Vault 7: Cia hacking tools revealed." [Online]. Available: https://wikileaks.org/ciav7p1/

[10] M. Gibbs, "Intel management engine's security through obscurity should scare the **** out of you," *Network World (Online); Southborough*, vol. 1, jun 2016.

[11] ——, "National security letters," *Newsletter on Intellectual Freedom (Online); Hayward*, vol. 62, no. 3, may 2013.

[12] "Arm trustzone." [Online]. Available: https://www.arm.com/products/security-on-arm/trustzone

[13] F. Martin, "Expert says nsa have backdoors built into intel and amd processors." [Online]. Available: http://www.eteknix.com/expert-says-nsa-have-backdoors-built-into-intel-and-amd-processors/

[14] O. Kugler, "One time pad encryption the unbreakable encryption method," *Mils Electronic*.

[15] M. Borowski and M. Leśniewicz, "Modern usage of one-time pad," in *2012 Military Communications and Information Systems Conference (MCC)*, Oct 2012, pp. 1–5.

[16] "Sd specifications part e1 sdio simplified specification," 2007. [Online]. Available: https://www.sdcard.org/developers/overview/sdio/sdio_spec/Simplified_SDIO_Card_Spec.pdf

[17] B. Brown, "Adding a secondary sd card on raspberry pi." [Online]. Available: http://ralimtek.com/Raspberry_Pi_Secondary_SD_Card/

[18] 2017. [Online]. Available: https://www.raspberrypi.org/documentation/hardware/raspberrypi/spi/README.md

[19] T. Vaughab, "Read from sdhc card using bus pirate," 2014. [Online]. Available: https://hackaday.io/project/3686-read-from-sdhc-card-using-bus-pirate

[20] 2016. [Online]. Available: https://cdn.sparkfun.com/datasheets/BreakoutBoards/Shifting_microSD_v10.pdf

[21] 2014. [Online]. Available: http://www.ti.com/lit/ds/symlink/txb0104.pdf

[22] A. Johnstone, "spidev xfer2 v xfer," 2013. [Online]. Available: https://www.raspberrypi.org/forums/viewtopic.php?f=32&t=39384

[23] "Sd specifications part 1 physical layer simplified specification," 2006. [Online]. Available: http://users.ece.utexas.edu/~valvano/EE345M/SD_Physical_Layer_Spec.pdf

[24] P. Stoffregen, "Ps2keyboard library," 2016. [Online]. Available: https://www.pjrc.com/teensy/td_libs_PS2Keyboard.html

[25] Adafruit, "Adafruit-st7735-library," 2016. [Online]. Available: https://github.com/adafruit/Adafruit-ST7735-Library

[26] "Particle photon datasheet," 2017. [Online]. Available: https://docs.particle.io/datasheets/photon-datasheet/

[27] "Particle core datasheet," 2017. [Online]. Available: https://docs.particle.io/datasheets/core-datasheet/

[28] "Particle.publish()," 2017. [Online]. Available: https://docs.particle.io/reference/firmware/photon/#particle-publish-

[29] "Ascii table and description," 2010. [Online]. Available: http://www.asciitable.com

[30] "Base 64," 2006. [Online]. Available: https://tools.ietf.org/html/rfc4648#section-4

[31] adamvr, "arduino-base64," 2015. [Online]. Available: https://github.com/adamvr/arduino-base64

[32] M. G. Kuhn, "Eavesdropping attacks on computer displays." [Online]. Available: http://www.cl.cam.ac.uk/~mgk25/iss2006-tempest.pdf

[33] "rtl-entropy." [Online]. Available: https://github.com/pwarren/rtl-entropy

[34] "Fips pub 140-2 security requirements for cryptographic modules." [Online]. Available: https://web.archive.org/web/20070825103724/http://csrc.nist.gov/cryptval/140-2.htm

[35] "rngtest." [Online]. Available: http://linuxcommand.org/man_pages/rngtest1.html

[36] "Product delivery order requirements package checklist." [Online]. Available: http://web.archive.org/web/20141229080627/netcents.af.mil/shared/media/document/AFD-140107-011.pdf

[37] "Tempest equipment selection process." [Online]. Available: http://www.ia.nato.int/niapc/tempest/certification-scheme

[38] A. Fabio, "Theremin's bug: How the soviet union spied on the us embassy for 7 years." [Online]. Available: http://hackaday.com/2015/12/08/theremins-bug/

[39] "The thing great seal bug." [Online]. Available: http://www.cryptomuseum.com/covert/bugs/thing/index.htm

[40] E. Williams, "Tempest: A tin foil hat for your electronics and their secrets." [Online]. Available: http://hackaday.com/2015/10/19/tempest-a-tin-foil-hat-for-your-electronics-and-their-secrets/

[41] "Nsa ant product catalog." [Online]. Available: https://nsa.gov1.info/dni/nsa-ant-catalog/

[42] S. Larson, "Ps/2 keyboard interface (vhdl)." [Online]. Available: https://eewiki.net/pages/viewpage.action?pageId=28278929

[43] K. Lindsey, "vga_generator." [Online]. Available: http://papilio.cc/index.php?n=Playground.VGAGenerator

[44] "Basys 2 reference manual." [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/basys-2/reference-manual

[45] "Xesscorp, github repository." [Online]. Available: https://github.com/xesscorp/VHDL_Lib/blob/master/SDCard.vhd

[46] C. Riley, "Github repository." [Online]. Available: https://github.com/Domipheus/UART