ROM Circuit Edit Detecting Code

Coding and Algorithms for Memory

Yehonatan Lusky
315690677

Intro

The moment a computing platform is powered on, the code which is responsible for the boot sequence initialization takes charge. This code is usually referred to as BootROM or simply ROM- according to the type of memory this code is stored over.

ROM[1] (Read Only Memory) is a non-volatile storage which is used to store data that is not meant to be changed after the manufacturing. PROM (Programmable ROM) is a specific kind of ROM which is typically implemented as an array of some sort of fuses that can store binary data of any kind. Each fuse can be in one of two possible states, either burned or not. Accordingly, each fuse represents a single bit, a burned fuse can represent logical 1 and non-burned one a logical 0, or vice versa.

Unlike EPROM (Erasable Programmable ROM), PROM was designed to remain the same at any point (even before the end of manufacturing) and hence, has no defined mechanism to be reprogrammed. The BootROM is usually stored on such PROM from both engineering reasons and security reasons.

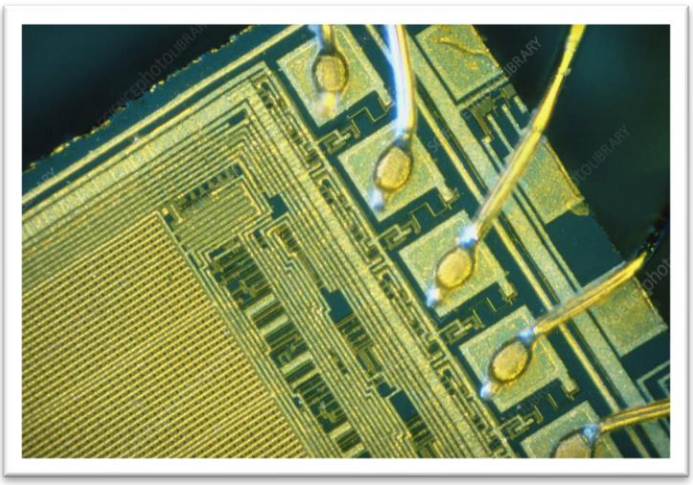


Figure 1-Intel PROM, fuses are located on the left side and control lines are on the right side

From security perspective, since the PROM allegedly can't be altered, the BootROM is the root of trust for the entire platform. Usually, it contains the most critical code in the platform initialization, it performs cryptographic keys derivation and in addition, it performs an authentication of the next part of the firmware in the chain of boot. Having the ability to alter the BootROM code could have a destructive impact in terms of the platform security.

[1] https://en.wikipedia.org/wiki/Read-only_memory

One known attack to modify the PROM is using FIB circuit edit. FIB (Focused ION Beam) is a technique which is mainly used by the semiconductor industry for failure analysis purposes. FIB allows to fix interconnects within the circuit, as such are the fuses. Performing a FIB for circuit edit purposes is a very delicate and precise operation and hence has a non-negligible chance to fail and destroy the chip.

Figure 2-FIB Machinery

In this project I will try to define the desired error detection code to deal with such attacks, I will suggest several codes that meet the defined requirements and finally, I will test these codes performance with a dedicated c framework I have created.

I aim to make a comprehensive review of the current relevant error detection codes, implement these codes to estimate their performance, and finally, try to suggest the best code in terms of performance, redundancy bits and the amount of detectable errors.

Hopefully, this project will be a basis for some practical suggestions of codes that different semiconductors will be able to use in order to protect their ROMs.

Defining the Desired Code

According to the scenario described above, we make the following observations:

1. **Bits can be altered only in one direction** – since FIB can only recover a burned fuse and not burn an unburned one, we can alter a bit only in one direction, either 1 to 0, or 0 to 1 - depends on the value corresponding to a burned fuse. This is very similar to the Z channel we learned during the semester.
2. **Increasing the number of bits to be altered makes the attack less practical** – as been stated before, we succeed in altering a single bit with a certain success rate, let's denote it by p. Assume we want to change d bits, in such scenario, the chances for the attack to succeed are p^d. Hence, increasing the bits to alter reduces the chances for the attack to succeed.
3. **Increasing ROM size is expensive** – the PROM is expensive in terms of chip area. Increasing the ROM size would result in less chip surface be left free for other important features of the integrated circuit.
4. **Data is read in a fixed chunk size** – the relevant data chunk is read every time. The size of it is determined by the bus's width and control lines. Typically, a single chunk will consist of 32, 64 or 128 bits of information.
5. **Boot time is crucial** – this is the first step in a long and complicated boot chain, therefore, reducing the BootROM execution time is important for a faster overall boot time.

According to these observations we demand the following requirements from our code:

1. **We want the code to detect as many asymmetric errors as possible** – dealing with more asymmetric errors allows us with handle more complicated attacks.
2. **We want our code to use as few redundancy bits as possible** – less redundancy bits directly implies a smaller ROM.
3. **We want a systematic code that can be easily implemented in hardware** – a simple implementation of the code means that it is both efficient from runtime perspective and the amount of surface it occupies over the integrated circuit.

Note: In this paper we assume that asymmetric errors change the bit value from 1 to 0. This is only a matter of convenience as the different codes will work errors in both possible directions.

Summation Code

This section describes an error detection code for asymmetric errors called summation code. This code can detect any amount of asymmetric errors within the a given word. The code will work as follows:

Given a word $x \in \{0,1\}^n$ the code will generate an ICV (Integrity Check Value), $icv \in \{0,1\}^{\lfloor \log n \rfloor + 1}$. The ICV represents the number of 0's in the binary representation of x. There can be at most n 0's at the binary representation of x, hence, $\lfloor \log n \rfloor + 1$ bits are needed to represent this number. The ICV will be appended to x inside the memory.

Given a word $y \in \{0,1\}^n$ that has to be verified that no circuit edit was performed, the algorithm will simply count the number of 0's in the binary representation of given word y and compare it with the corresponding ICV stored in memory.

Example:

Let's take the following 16-bit value:

0000-1111-0100-0010

It has ten 0's, hence the ICV will be:

1010

Which is the equivalent of 10 in binary representation.

As been stated before, FIB circuit edit can change bits only from 1 to 0. Therefore, any circuit edit applied to a word x in memory will increase that amount of 0's inside of it. On the other hand, any circuit edit performed on the ICV will create a new ICV' such that $ICV' \leq ICV$ since only 1's turn into 0's, meaning that ICV' represents a different x that should contain less 0's.

Later on, while performing a literature review of the relevant codes, this code was found out to be a known code called Berger code (Berger 1961).

Modulo Summation Code

Following the summation code presented in the previous section, a trivial enhancement in case we would like to use less than $\lfloor \log n \rfloor + 1$ check bits (The amount of bits the ICV consists of) was to use a modulo summation code. This means that the ICV would simply be the amount of 0's modulo 2^r where r represents the desired amount of redundancy bits.

In contrast to the previous summation code, in the modulo summation code isn't perfect and can't detect any amount of errors. For a given word $x \in \{0,1\}^n$ the modulo summation code will produce an ICV which is the same for another word $y \in \{0,1\}^n$ in case $0's_y \ mod \ 2^r = 0's_x \ mod \ 2^r$. As been told before, every $y \in \{0,1\}^n$, can be reached from a different word $x \in \{0,1\}^n$ using circuit edit in case $x \geq y$.

As this code is not a very strong one as we will see later. It is capable of detecting at most r errors.

Theorem: The modulo summation code can detect at most r errors.

Proof:

Let's assume we have r redundancy and a given word $x \in \{0,1\}^n$ such that its bits can be represented as $x_1, x_2 \dots x_i \dots x_n$. In case we can't alter the ICV, the amount of bits required to chance in x in order to reach a different word $y \in \{0,1\}^n$ such that $icv_y = icv_x$ is exactly 2^r.

However, we can also alter the ICV. Let's represent the ICV bits as $icv_1, icv_2 \dots icv_r$. Each bit with index i we alter in the ICV, reduces the amount of bits we have to alter in x by 2^{i-1}. Therefore, if we want an easier attack we will simply alter as many bits as possible in the ICV. The ICV contains at most r bits which could be 1 and hence can be altered. Additionally, we will have to alter at least one more bit in x.

Therefore, at the worst case, changing in total $r + 1$ bits will result in another valid word.

Der Jei Lin & Bella Bose Error Detecting Codes

At this point I've continued my research by making a more comprehensive literature review about different systematic, asymmetric error detecting codes. I was looking for two types of codes, codes which detect any amount of errors and codes which can detect up to a certain amount of errors but with smaller redundancy and better efficiency.

While it looks like the Berger is the indeed best code to detect any amount of errors when the redundancy bits aren't a factor, I've found several interesting papers. All those papers are mentioned in the bibliography part, and they are aimed to deal with up to a certain amount of errors. Among those papers, the most relevant paper was by Der Jei Lin and Bella Bose (Lin 1985).

In their paper Det Jei Lin and Bella Bose propose two codes.

Code 1

The first code requires at least 2 check bits and works as follows:

1. Given a word $x \in \{0,1\}^n$, count the amount of 0's in the binary representation of x. We'll denote this value using S_0.
2. Calculate $C_0 = S_0 \ mod \ 2^{r-1}$
3. Set the ICV to be: $ICV = C_0 + 2^{r-2}$

The code described above can detect at most $2^{r-2} + r - 2$ errors. The two most significant bits of the ICV can be either 01 or 10, hence due to the nature of the asymmetric errors, it is not possible to alter between these two states which allows us to handle at least 2^{r-2} errors. In addition, in a similar analysis to the modulo summation code, this code can handle at most r-2 additional errors that may occur by zeroing out all the bits in the rest of the check bits. (The full proof can be found inside the paper)

Code 2

The next code presented in their paper requires at least 4 check bits and works as follows:

1. Given a word $x \in \{0,1\}^n$, count the amount of 0's in the binary representation of x. We'll denote this value by S_1.
2. Calculate $C_1 = S_1 \ mod \ 6 * 2^{r-4}$
3. Take the 3 most significant bits of C_1 and replace them with 4 bits according to any desired mapping between them to 2-out-of-4 values. An example for such mapping is:
$$f(000) = 0011, \quad f(001) = 0101, \quad f(010) = 0110, \quad f(011) = 1001,$$
$$f(100) = 1010, \quad f(101) = 1100.$$

The described code allows us to detect at most $5 * 2^{r-4} + r - 4$ errors. The analysis is similar to the code described previously. We can't change the 4 MSB in the ICV and in addition we have to alter at least r-4 bits in the remaining part.

Summary: Error Detection Capabilities

To sum it all up, the following table describes how many errors each of the codes can detect at worst case.

Redundancy Bits	Summation Code	Summation Modulo Code	Code 1	Code 2
r	2^r*	$r+1$	$2^{r-2} + r - 2$	$5 * 2^{r-4} + r - 4$
5	32	6	11	11
6	64	7	20	22
7	128	8	37	43
8	256	9	70	84
9	512	10	135	165
…	…	…	…	…

* When there aren't enough check bits, the summation code simply becomes the summation modulo code.

Creating a Testing Framework

At this point I had 4 different codes and I wanted to evaluate their performance. To do so, I've created a dedicated framework written in C. Though a simulation in an FPGA (Field Programable Gate Array) was probably the best approach to evaluate their performance it was out of my reach, therefore, I decided to implement these codes using C code while using as many operation as possible that are executed efficiently inside the CPU core, i.e. shift left, modulo, and, or and so on.

Though this is probably not the most accurate way to evaluate these codes performance, hopefully it has given a good estimation of which code are more efficient and by how much.

I will not elaborate much more on the framework itself here as all the relevant code is located and documented inside the following Github repository.

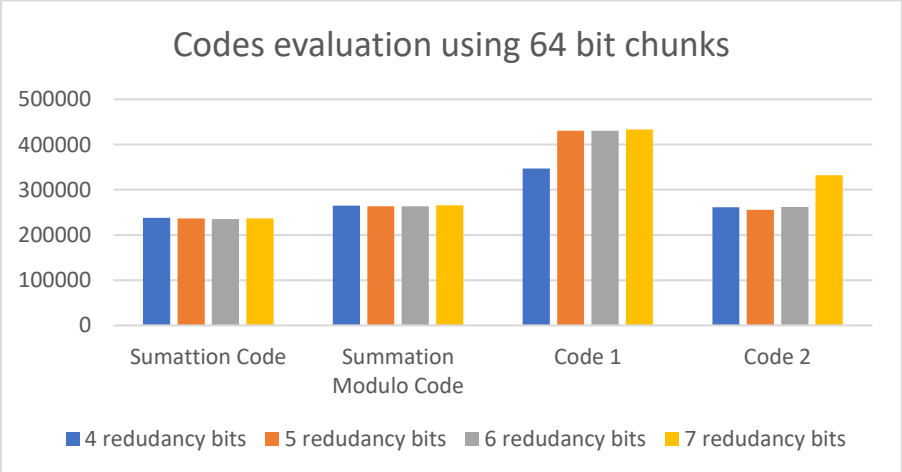
Results

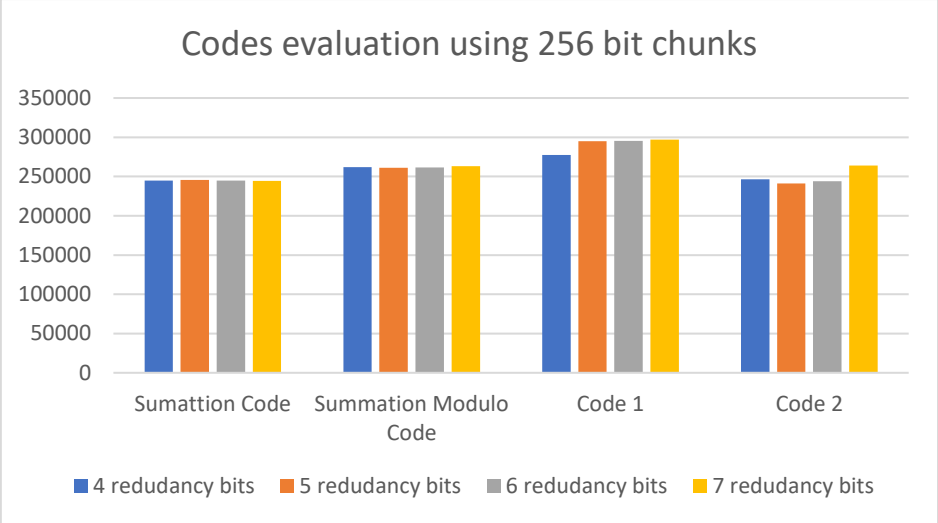
I've conducted several tests using the framework I have created. I've allocated a 16MB of dummy memory (which is a typical ROM size) and filled it with random values, I calculated the ICVs using the codes described above and made some time measurements of how long it takes for each of the codes to validate the entire memory.

I repeated each experiment 1000 times to get an average time and by doing so to eliminate the noise from the operating system. Additionally, each of the 1000 tests had a different memory since I wanted it to be as close as possible to real life scenario where we don't have control over the memory content. However, I made sure that this randomization seed is similar across different codes experiments.

In my first experiment I tried to determine whether the amount of check bits affect the performance of the different codes. I tried several chunk sizes (32 64, 128 and 256 bits) as determining the right chunk size is also important in real life systems since memory can be read multiple times and often we don't require more than few bits of the memory chunk.

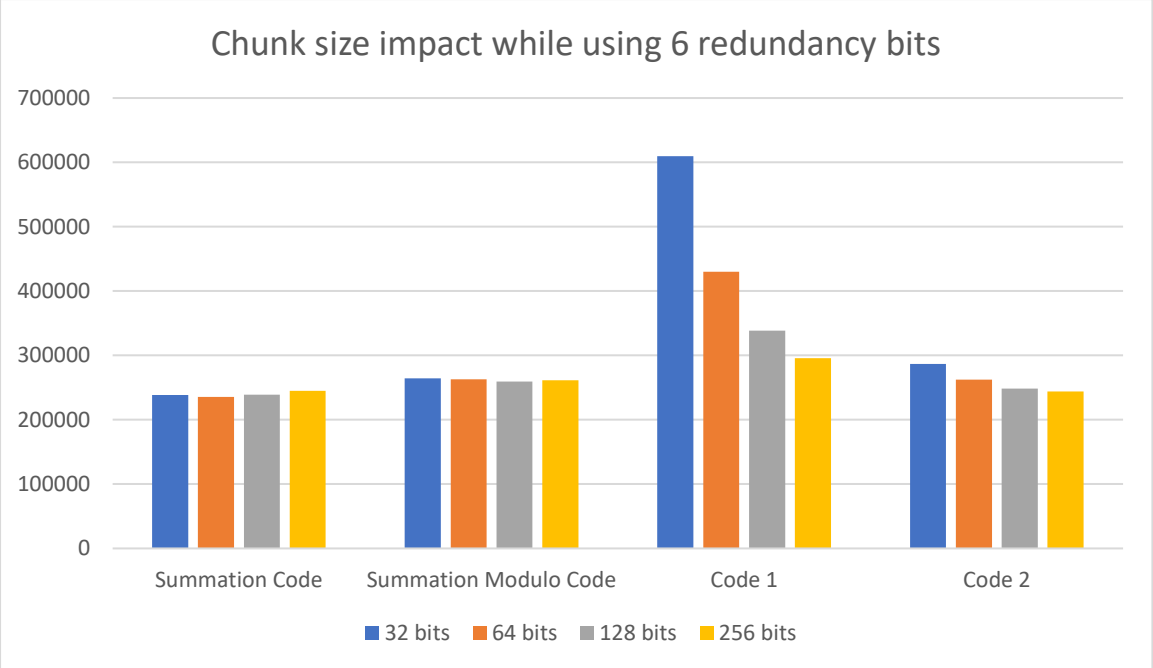
It should be mentioned that the summation code has a constant amount of check bits which is directly determined by the chunk size.





As can be seen in the graphs above and in the full results inside the appendix, usually the amount of redundancy didn't affect the performance, the only exception is code 2 when using 7 or more redundancy bits and code 1 when using 5 or more redundancy bits.

With that in mind, I decided to check how the chunks size affects the performance of the codes. I took several which are possible to work with: 32, 64, 128, 256 bits. Though in a typical ROM the chunks would usually be 64 or 128 bits.

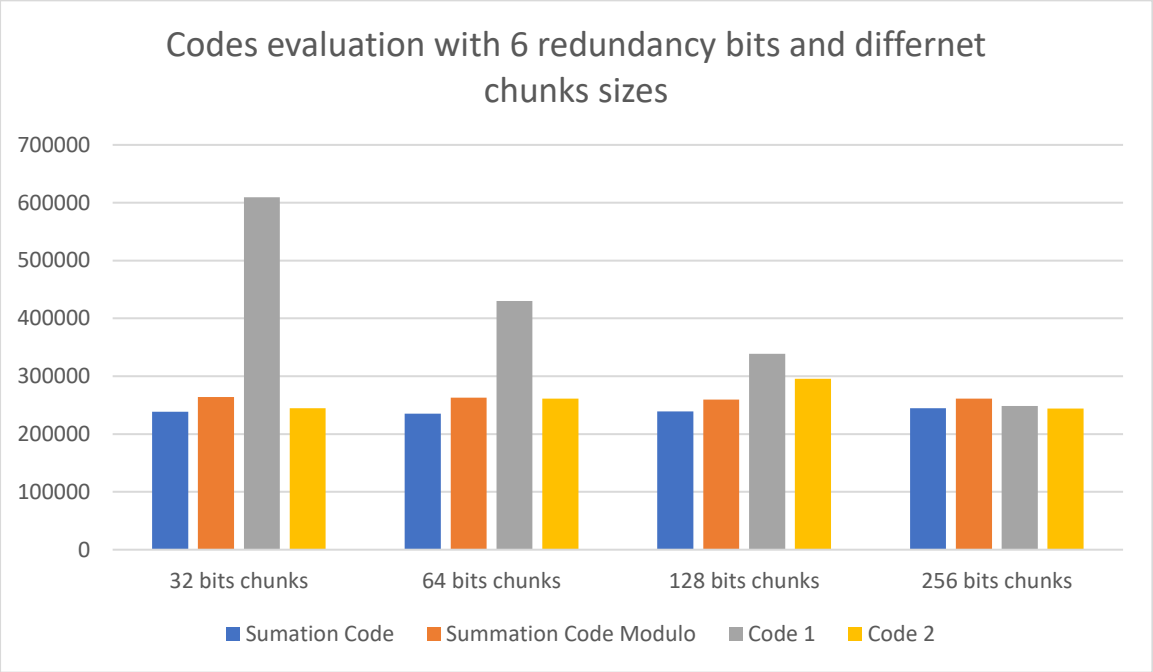


Here I present the performance while using 6 redundancy bits. Clearly the chunk size has impact especially in code 1 and code 2. I believe this is because both code 1 and code 2 complexity depend mostly on heavy computations that are performed per chunk. Thus, when increasing the chunk size and by doing so reducing the amount of chunks, the total complexity drops. On the summation code

and summation modulo code, the main activity is counting the 0's and this linearly depends on the total size of the memory and not on the size of chunks.

I didn't present here the results when using a different number of redundancy bits since the results are very similar as can be seen in the full results in the appendix.

Finally, I wanted to evaluate the performance of the given codes in comparison to one another. As we've witnessed increasing the chunks size may reduce significantly the computation time while reducing or increasing the check bits doesn't have the same impact. Hence, I tried to compare the different codes only when the chunk sizes are similar.



As can be seen in the graph above the summation code clearly performs the best, however when working increasing the chunks size this advantage becomes smaller and when working with 256 bits chunks this advantage even becomes negligible.

Conclusion

To conclude, in this project I've tried to find the best code that is suitable for errors detecting resulted from FIB circuit edit in ROMs. I've went through most of the research out there and it looks like asymmetric error detection problem is a solved problem for a long time as most of the papers are before 2000's.

I've dedicated quite some time to try and think of better codes on my own, however, without any luck, indeed it looks like this is a solved problem. Nevertheless, in this project I've tried to add a different input to this problem and evaluate the performance of these codes in a specific scenario.

From the results above, I believe that we can make two important observations:

1. Increasing the chunk size drastically improves the time it takes to validate the memory in Code 1 and Code 2.
2. When increasing the chunks size to 256-bit chunks, depending on the amount of check bits we are willing to dedicate for error detection, the best codes are either the regular summation code or Code 2.

Hopefully, this answers our research question and achieves this research purpose as now we are able with providing some practical suggestions to ROM development teams.

Bibliography

Al-Bassam, Sulaiman and Bella Bose. 1994. "Asymmetric/unidirectional error correcting and detecting codes." *IEEE transactions on computers 43.5* 590-597.

Berger, Jay M. 1961. "A note on error detection codes for asymmetric channels." *Information and control 4.1* 68-73.

Bose, Bella, Samir Elmougy, and Luca G. Tallini. 2007. "Systematic t-unidirectional error-detecting codes over Zm." *IEEE Transactions on Computers 56.7* 876-880.

Lin, Der Jei & Bella Bose. 1985. "Systematic unidirectional error-detecting codes." *IEEE Transactions on Computers* 1026-1032.

Ray-Chaudhuri, Dwijendra K., Navin M. Singhi, S. Sanyal, and P. S. Subramanian. 1994. "Theory and design of t-unidirectional error-correcting and d-unidirectional error-detecting code." *IEEE transactions on computers 43, no. 10* 1221-1226.

Appendix – Full Results

Redundancy bits	Code	32 bits chunks	64 bits chunks	128 bits chunks	256 bits chunks
4 Redundancy Bits	Summation Code	238496	238216	239737	244636
	Summation Modulo Code	264919	264897	260707	262048
	Code 1	449398	347098	297293	277332
	Code 2	285834	261133	248948	246372
5 Redundancy Bits	Summation Code	239395	236548	240317	245476
	Summation Modulo Code	264309	263439	259868	260958
	Code 1	609621	430776	338224	295046
	Code 2	277626	255495	244338	241169
6 Redundancy Bits	Summation Code	238492	235396	239143	244825
	Summation Modulo Code	264086	263068	259592	261495
	Code 1	609472	430221	338516	295542
	Code 2	286689	262099	248516	243899
7 Redundancy Bits	Summation Code	239120	236206	238999	244329
	Summation Modulo Code	265746	265261	261496	263309
	Code 1	614663	433097	339772	296988
	Code 2	431716	332208	285151	264177
8 Redundancy Bits	Summation Code	265790	259826	256397	259329
	Summation Modulo Code	240514	236233	233968	225293
	Code 1	613138	430040	334145	289728
	Code 2	444084	347417	297244	275205