Cryptanalysis of KeeLoq code-hopping using a Single FPGA

Idan Sheetrit and Avishai Wool

Computer and Network Security Lab
School of Electrical Engineering
Tel-Aviv University, Ramat Aviv 69978, Israel
idanshee@post.tau.ac.il, yash@eng.tau.ac.il

Abstract. The KeeLoq cipher is used in many wireless car door systems and garage openers. Recently the algorithm was studied and several attacks have been published. When a random seed is not used the attack on the system is fairly straight-forward. However when a seed is shared between the remote control and the receiver previous research suggested using highly parallel crypto hardware (like COPACOBANA) for breaking the cipher within reasonable time.
In this paper we show that highly-parallel hardware is not necessary: our attack uses a single FPGA for breaking KeeLoq when using a 48-bit random seed in 17 hours using a mid-range Virtex-4, and less than 3 hours using a high-end Virtex-6 chip. We achieve these results using a combination of algorithmic improvements, FPGA design methodology, and Xilinx-specific features.

Keywords: KeeLoq, cryptanalysis, FPGA

1 Introduction

1.1 Background

Wireless remote controls are popular in many applications including car access and garage door openers. Simple remotes commonly send a unique code on a known frequency to authorize access, which can easily lead to unauthorized access. A popular improvement is to use the KeeLoq code hopping system: KeeLoq consists of a low cost hardware implementation block cipher which has a large number of combinations available and won't respond twice to same transmitted code. This cipher is used by Microchip for keyless systems usually for the automobile industry and door openers [10]. Car manufactures like Chrysler, Daewoo, Fiat, GM, Honda, Jaguar, Toyota, Volvo, Volkswagen, etc. adopted KeeLoq for their cars security [9].

1.2 Related work

The KeeLoq cipher was extensively studied lately and many attacks have been published [1][2][4][5][6][7][8]. Research has shown that manufacturers often share

the same master key for most of their products: if this is the case then this master key can be recovered using side-channel attacks on one of the manufacturer's receivers [2]. When this master key is discovered the system stays secure only if a random seed is used for the shared-key derivation between the remote unit and the receiver (see section 2.2). Recently, Novotnỳ and Kasper have shown that even when a random seed is used one can break the security using a special 120-parallel-FPGA crypto cracker system called COPACOBANA[1] [3] within reasonable time.

1.3 Contributions

In this paper we show that KeeLoq can be broken despite the use of a random seed, using a single FPGA without the need for highly-parallel hardware. This is achieved by algorithmic improvement and FPGA design methodology, that together make our breaker more than 3 times faster than the breaker of [3] on the same hardware. Additionaly, our breaker can use any two captured messages, whereas [3] requires capturing two nearly-consecutive messages. Furthermore, our design requires roughly 50% of the gate count of [3], allowing us to place many more breaker blocks on the same FPGA. Combining all the above properties, our implementation can break KeeLoq with a 48-bit random seed in less than 17 hours using a single mid-range Virtex-4 chip, and less than 3 hours using a high-end Virtex-6 chip. Thus the single-chip Virtex-4 implementation is roughly equivalent to the performance of the 120-FPGA COPACOBANA system of [3], and the Virtex-6 implementation clearly has superior performance. This makes our breaker more affordable, accessable and mobile.

2 Overview Of KeeLoq

2.1 KeeLoq Algorithm

The KeeLoq algorithm is a block cipher with a 64-bit key and a block size of 32-bits. The algorithm is designed for efficient hardware implementation. As shown in Figure 1 its main components include a 64-bit shift register (FSR), 32-bit shift register (NLFSR) and a non linear feedback (NLF) function. The 64-bit key is kept in the FSR. The feedback in the NLFSR register depends on the xor result of 1 key bit, 2 taps on the NLFSR itself and the result of a non linear feedback (NLF) function using 5 taps on the NLFSR. Each clock cycle the 64-bit key FSR is cyclicly left-rotated, the NLFSR is shifted left and the new feedback created is shifted back into the NLFSR. To preform decryption, the ciphertext is loaded into the 32-bit NLFSR and the key into the key register. The entire block is clocked 528 times after which the NLFSR contains the plaintext. Encryption is done just in the opposite direction when clocking with relevant changes on the taps place (the taps should move one bit left and the key tap would start from 0).

[1] Cost-Optimized Parallel COde Breaker based on 120 Spartan3-1000 FPGAs

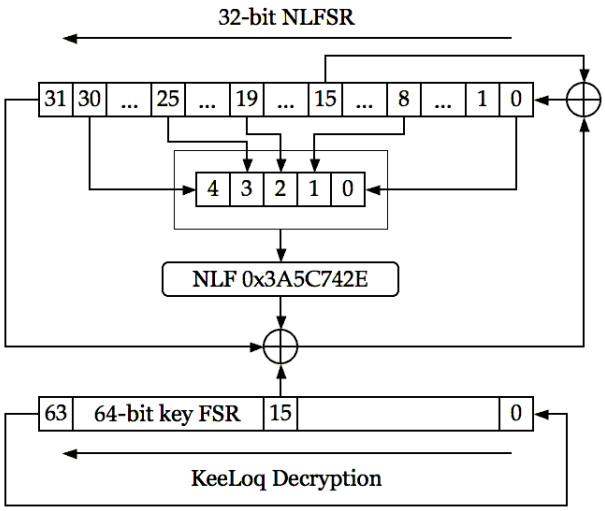


Fig. 1. Structure of the KeeLoq decryption cipher (taken from [9])

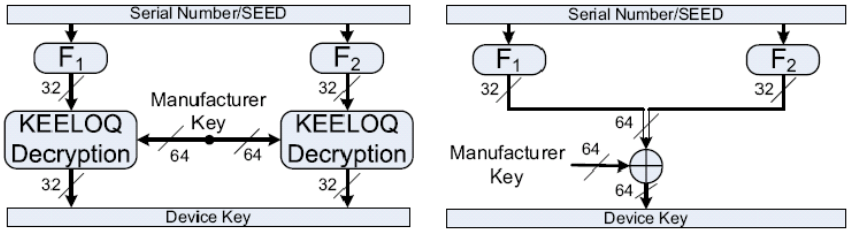
2.2 Key Derivation Schemes

Two different keys are involved in the typical KeeLoq application: a *manufacturer key* and a *device key*. As mentioned in [2] the *manufacturer key* is typically identical for all manufacturer receivers. The *device key* is derived during the learning process between the transmitter and the receiver. The "learning process" involves the *manufacturer key*, the *serial number* of the (transmitter) device and (in some cases) the *seed* which is a random number passed from the transmitter to the receiver during the "learning process" [11]. There are four possible methods to derive the *device key*[2].

Simple The simplest derivation called is Simple. In this derivation there is no use of any *seed* and only the 64-bit *manufacturer key* and 28/32-bit *serial number*[3] are needed. The *serial number* is passed through a fixed transformation (using 28 bits of the *serial number* appended with a different constant nibble for each 32 bits) and then each 32-bit half is decrypted using a simple xor function separately with the relevant 32-bit of the *manufacturer key* - see Figure 2(b).

Normal In this method the same process is used to create the input 64 bits from the *serial key* as with *Simple*. However, instead of using a xor function the Normal derivation uses KeeLoq decryption for the two 32 bits input-halves with the *manufacturer key* - see Figure 2(a).

[2] Details can be found in [11]. Here we use the naming convention from [14]
[3] The serial number could be either 28-bit or 32-bit according to [13]



(a) Normal/Secure Decrypt (b) Simple/Secure Xor

Fig. 2. Key Derivation Schemes (taken from [2])

Secure Decrypt In this method a random *seed* is used. The *seed* is used as the MSB that is prepended to the *serial number* to form a 64-bit input. There are 3 possible random *seed* lengths: 32, 48 or 60 bits; The remaining bits are taken from the *serial number*. The resulting 64 bits are split into 2 halves, each decrypted by KeeLoq using the *manufacturer key* (as in the *Normal* method).

Secure Xor In this method the same process is used to make the 64-bit input from the *seed* and *serial key* as in *Secure Decrypt*, but instead of decrypting with the *manufacturer key* the input bit string is xored with the *manufacturer key* as shown in Figure 2(b).

From the above we can see that every receiver must store the *manufacturer key*—it uses this key when "learning" to work with a new transmitter. Therefore, by a side-channel attack on any single receiver, the attacker can extract the manufacturer key—that is shared among all the manufacturer's devices [2].

In this paper we assume that the attacker knows the *manufacturer key*—e.g., from a power-analysis attack [2]. In such a scenario the *Normal* and *Simple* key derivation is completely broken, therefore we focus our attention on the seed-derived key derivation methods (Secure Decrypt and Secure Xor).

2.3 The Code Hopping Protocol

The *code hopping protocol* is the common KeeLoq mode of operation that is used for keyless entry systems, primarily for vehicles and home garage door openers (according [12],[13]). Code hopping is a method by which the code is different on every key press. This is done by maintaining 16-bit counter that is synchronized between the transmitter and receiver. Each time the transmitter button is pressed, the counter is incremented by 1. This 16-bit counter is padded with other parameters (10 bits called "discrimination" and 6 other configuration bits) to produce a 32-bit block. This block is encrypted using KeeLoq with the *device key* that was created earlier by one of the key derivation schemes. The

resulting 32-bit ciphertext is sent together with other parameters (some of the *serial number* bits and configuration) to the receiver. Note that all the message components, except for the 32-bit ciphertext, are sent in clear.

When the receiver receives the message it: 1) checks if the *serial number* is equal to the learned *serial number*; 2) decrypts the contents and extracts the sent counter value; 3) verifies that the sent counter is in its reception window (within 16 or 32 of the last received counter value).

An important observation is that the 16 padding bits encrypted together with the counter are in fact known: the 10 discrimination bits are the 10 LSB bits of the *serial number* (which is also sent as plaintext on every transmission), and the other 6 bits consist of a 4-bit button status which is also sent as plaintext[4] and 2 bits of counter overflow which are practically constant[5].

3 Cryptanalysis Keeloq

3.1 Assumptions

From previous works and relevant documentation [2][13] we can assume that the *manufacturer key* is same for major groups of interest (same manufacturer). This means that with access to a single receiver the attacker can extract the *manufacturer key* which would be relevant to most or even all of that manufacturer's products. In the attack described in this paper we assume the *manufacturer key* is known. Our attack also assumes having 2 recordings of the same transmitter (full hop-code words), meaning 2 ciphertexts and the relevant data sent with them. Note that unlike [3] we do not require the two messages to be intercepted within a short time interval—any two messages will do.

Our goal is to design an FPGA breaker that will implement a brute-force attack on all possible random *seed* values to discover the *device key* based on these 2 recovered messages.

3.2 Using Interesting KeeLoq Properties

When implementing a brute-force attack we need to test whether our key guess is correct or not. The approach taken by [3] is to decrypt the two separate transmissions and compare their constant bits (bits known to be equal on both transmissions[6]) and, in parallel, compare the numeric distance between the two decrypted counters. The authors assume that the two messages were intercepted within a short period in time, so the counters should be close in value—thus the

[4] This is always true for the Microchip HCS301 model (see [12]), but isn't true when using a specific feature on HCS410 called "extended serial number" (see [13])

[5] If we assume that a user uses the remote transmitter 8 times a day the overflow would change after more than 20 years.

[6] In [3] only the 10 discrimination bits are being compared

filtering will yield a small set of possible device keys with high probability[7]. The main drawback of this approach, from an FPGA design point of view, is that it requires 2 copies of the KeeLoq decryptor to be constructed, i.e., twice the area.

However, using the observations of Section 2.3, we can do much better, and use just a single KeeLoq decryptor. Since we know the 16 bits of padding, we can compare the decrypted 16 MSB bits of one transmission to the known padding value and filter out the mismatching keys. Doing so would result with a set of key candidates. The second step is to reduce the set of candidates to a handful of possible keys by using the same process with the second ciphertext sample.

3.3 Breaker Algorithm

In this section we focus on 32-bit *seed*s, the case for 48 or 60-bit *seed*s is similar. As before we assume that the *manufacturer key* and *serial number* are known. For a candidate *seed* value $seed_i$ the KeeLoq operation can be described as:

$$F_{Km,Serial}\left(seed_i\right) = Kd_i \tag{1}$$

where Km is the *manufacturer key*, *Serial* is the serial number and Kd_i is the candidate *device key*. Note that a 32 bit seed size changes only half of the device key with new *seed* value as described in Section 2.2. Therefore when using a brute-force attack there is no benefit from actually deriving the key on the unchanging half of the key. Since the changing half of the key goes over all 2^{32} possibilities, again there is no need to derive it - a simple counter suffices.

Using the candidate *device key* to decrypt the ciphertext sample(C_1):

$$V_{1,i} = DEC_{Kd_i}\left(C_1\right) \tag{2}$$

$V_{1,i}$ should be compared to the constant values we know (described on 3.2). If the values are equal then Kd_i is relevant to next step and would be added to a key set named K_1. After all 2^{32} possibilities for Kd_i have been tested, the next step would be to use K_1 to decrypt the second ciphertext sample (C_2) and compare $V_{2,i}$ to the constant string value known for this sample and create a very small set of keys name K_2,

$$V_{2,i} = DEC_{K_{1,i}}\left(C_2\right) \tag{3}$$

Note that the set K_1 would be stored on the host and that Eq.3 can be calculated on the host (when K_1 isn't empty) in parallel to Eq.2 on the FPGA.

3.4 Expected Number Of Key Candidates

In this section we calculate the probability of a random key passing the condition on each step. Each filtering step of Eq.2 or Eq.3 reduces the expected number

[7] Our analysis shows the expected number of keys passing the filter is linear in T, the maximum allowed distance between the counters

of possible keys to a $1/2^{16}$ fraction. Assuming that a-priori there are M possible device keys ($M = 2^{32}$ for a 32-bit *seed*) then after 2 filtering steps, using 2 ciphertext samples, we obtain the following:

$$E\left[|K_2|\right] = M\frac{1}{2^{16}}\frac{1}{2^{16}} \tag{4}$$

so for $M = 2^{32}$ we expect to find a single possible key.

3.5 Complexity

Naive If we create a new device key on every brute-force cycle and compare two decryption results to each other (as done by [3]), the number of KeeLoq operation would be:

$$2^{32}\left(2 + 1 + 1\right) \tag{5}$$

However note that half of these operations are done in parallel.

Our Attack Our attack evaluates one KeeLoq decryption (of C_1) on 2^{32} key candidates, we then decrypt C_2 only on smaller set of suspicious keys (2^{16} on our example). Thus the number of KeeLoq operations would be:

$$2^{32} + 2^{16} + 1 \tag{6}$$

2^{32} operations for decrypting C_1, 2^{16} for decrypting C_2 and 1 to calculate the known 32 bit half-key.

In comparison to the method of [3], both methods have to process roughly the same enumeration space. However, the advantage of our method is that it needs 50% of the hardware, which means that we can place twice as many breaker modules in a chip and double our attack speed. In addition our implementation uses some special FPGA features that make it even more compact even with full loop unrolling, thus allowing a faster clock rate. Finally, our attack is effective even if the two intercepted messages do not have close counter values.

4 Implementation

4.1 Baseline

In a baseline implementation the base element of the breaker is a standard KeeLoq decryptor (as shown in Figure 1) implemented with loop-unrolling technique (Choosing a loop-unrolling technique would better use the given hardware on brute-force attack in comparison to many independent breakers). In this implementation we follow the design of [3]. It uses 4 levels of loop unrolling followed by a line of flip-flops (FF). Novotnỳ & Kasper's design uses 2 separate copies of the logic block for the search of the correct key. When we implemented their design we used no manual optimization, only the automatic optimization tools were applied.

4.2 Full Loop Unrolling

Figure 3 shows a fully loop-unrolled (LUR) KeeLoq decryption implementation. In this figure the $8 \rightarrow 1$ block implements the NLF and Xor used (as shown in Figure 1) to calculate the next bit which is used as input for next LUR level. We use 528 levels of unrolling.

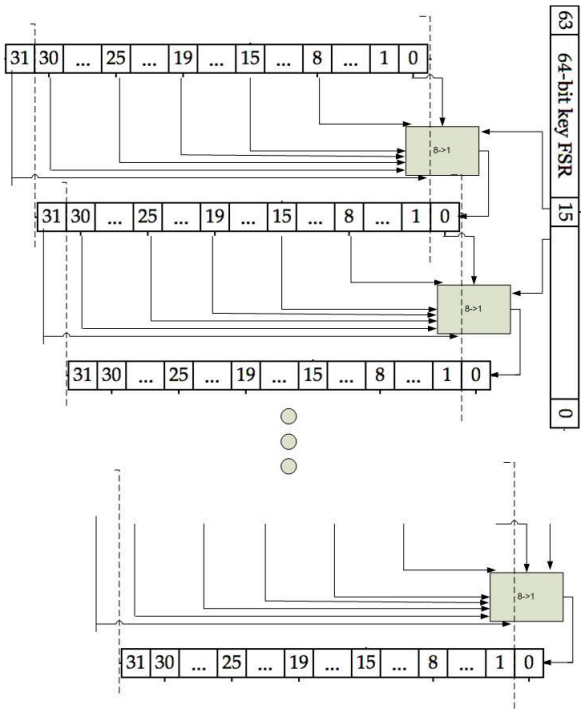


Fig. 3. Loop-Unrolling of KeeLoq decryption

A key observation is that in the design in Figure 3 there is a lot of structure in the columns. For instance, we can see that bit 15 in the first round appears as a tap to the $8 \rightarrow 1$ block, and then it is used again in round 5 when the bit is in position 19. Thus we need to implement "vertical" shift registers. A useful way to show the shift registers needed is shown in Figure 4. In this figure each row marks a LUR level (there should be 529 rows). Numbers in gray boxes indicate the taps for the $8 \rightarrow 1$ function which outputs the new bit for next level (all other bits are just copied). The first row (number 0) is the input of the LUR, which has taps on the 0,8,15,19,25,30,31 bits. The result of the $8 \rightarrow 1$ function is marked as -1 and is used in the next level. The last (528th) row would be the LUR output. From this figure it is easy to see that each bit "waits" several cycles before being used.

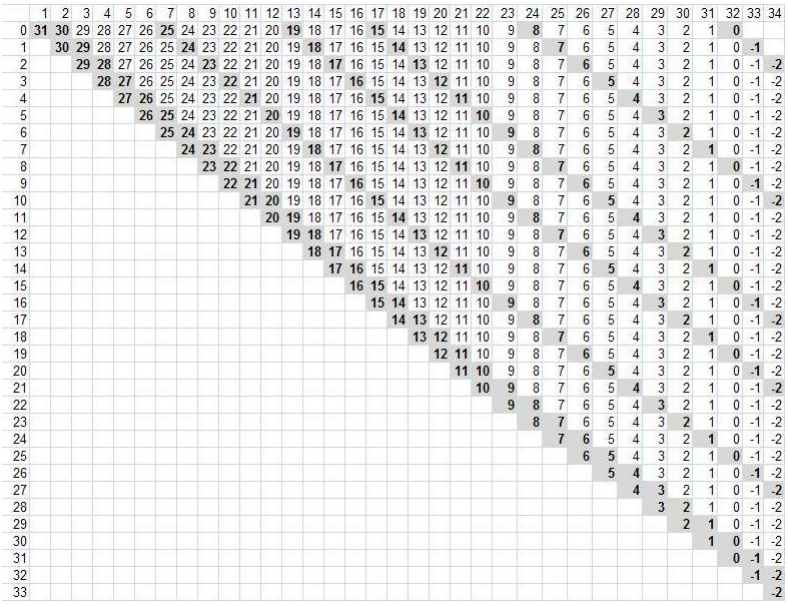


Fig. 4. Detailed Loop-Unrolling of KeeLoq - first 33 rounds out of 528

4.3 Using special FPGA features in the design

Xilinx FPGA devices have special properties that can be used in our design. In these devices the basic logic element is called LUT, and it is commonly used for creating any asynchronous $16 \rightarrow 1$ function. However, a LUT can also be used as 16-bit variable-length shift-register with a single bit clock input. This mode is called a shift-register LUT (SRL)[15][16].

When using the Xilinx Spartan-3, Virtex-2 or Virtex-4 architecture an SRL block (as shown in Figure 5) actually functions simultaneously as two shift registers, shifting the same bits: one has a variable length (determined dynamically by 4 control lines A[0-3]), and other is a fixed-length 16-bit shift-register. Thus the SRL has a single bit input (D), and two bit outputs: Q and Q15. Q which is the output of the dynamic length shift-register, and Q15 is the output of the 16-bit register. In Figure 6 we can see a basic chain of SRL blocks with different lengths (without using the Q15 output).

When using the Xilinx Virtex-5 or Virtex-6 the architecture is slightly different: a LUT can be used to implement either a single 32-bit shift-register, or a pair of dual 16-bit shift-register[17][18]—however dual shift registers implemented on the same LUT have the same dynamic length[8]. These SRL blocks are especially useful in our loop-unrolled design since they can be utilized in the construction

[8] According to our experience and conversations with Xilinx support the only way to use the dual 16-bit SRL feature is to manually choose it. The automatic optimization of Xilinx software doesn't use this feature

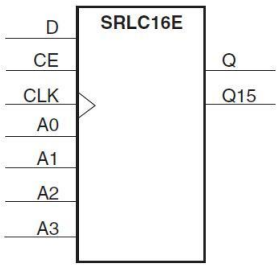


Fig. 5. SRL16 Block (taken from [15])

of the various "vertical" shift registers instead of using multiple flip-flop (FF) blocks.

4.4 The Breaker Block

The main breaker block in our optimized design is one fully loop-unrolled KeeLoq implementation as shown in Figure 3.

Considering the property mentioned in Section 4.3, note that any shift-register (which is implemented as a SRL) whose length is up to 16-bit requires no additional logic. As a result of this assumption, and the fact that the longest vertical shift register in the LUR is 8 bits, we chose to make heavy use of SRLs in our breaker.

When we experimented with the SRLs we discovered that the Xilinx automatic optimization implements the "vertical shift register" as shown in Figure 6, for a total of 5 SRLs and 2 flip-flops. However, with some careful design we can do better. Our best construction is shown in Figure 7 using only 3 LUTs as SRLs (taps are marked in gray). The design of Figure 7 is relevant for Xilinx FPGAs older than Virtex-5[9].

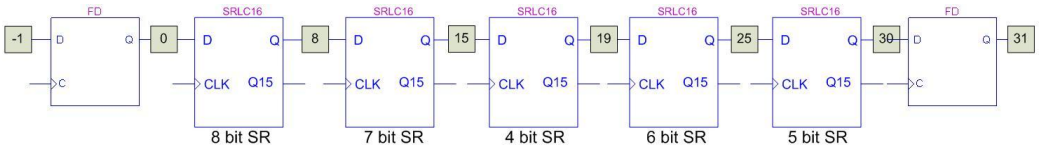


Fig. 6. LUR Column: simple implementation (2 FFs, 5 LUTs)

[9] Our best construction for the Xilinx Spartan-3 is slightly different but the same concept applies

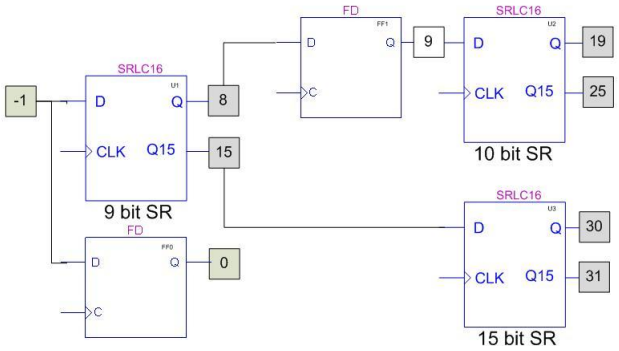


Fig. 7. LUR Column: manual optimization (2 FFs, 3 LUTs)

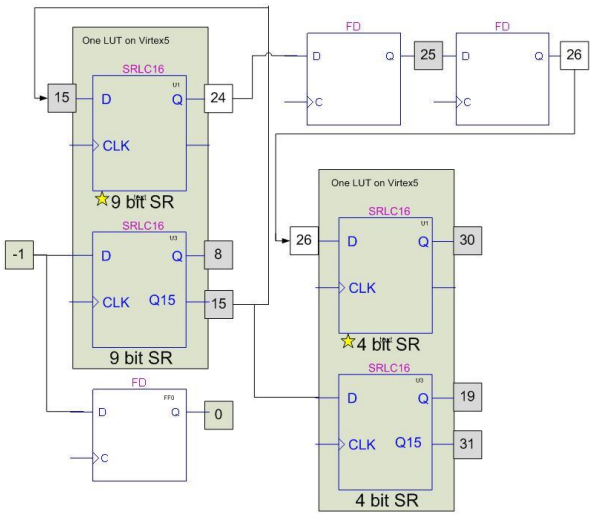


Fig. 8. LUR Column: manual optimization on a Virtex-5/6 (3 FFs, 2 LUTs)

We could use the design of Figure 7 on a Virtex-5 or Virtex-6 architecture too. However, we can also use the dual-SRL feature to reduce the LUT count even further. This is advantageous since it balances the number of LUTs and FFs. Xilinx FPGA's have roughly the same number of LUTs and FFs available, and we discovered that our design area is primarily constrained by the number of LUTs it uses. Our best design for the Virtex-5 architecture is shown in Figure 8, for a total of 3 FFs and 2 LUTs.

5 Performance Analysis

5.1 The Evaluation Environment

To evaluate our breaker designs, we used the Xilinx ISE 12.2 environment running on multicore Windows7 system for synthesizing our code and for implementing the various designs (for all the designs we used the highest optimization allowed by this environment). Our VHDL code was tested with the Aldec Active HDL 8.2 simulator for every design and configuration we used[10]. We compiled the designs assuming various target architectures, and evaluated the properties of the results.

We evaluated 3 designs: As a baseline we followed the design of Novotnỳ & Kasper, as described in Section 4.1 using 2 parallel KeeLoq decryptors to compare counters. The second design, which we call "simple" uses a single LUR machine per block with vertical SRs implemented with FFs[11]. The "simple" design uses the known half-plaintext for the brute-force attack as described earlier, so it uses half of the hardware in comparison to the baseline). The third design, which we call "optimized", includes all the manual space optimization as described on previous sections.

The effect of our low-space designs is that we can fit more loop-unrolled KeeLoq blocks on a single chip, which in turn means that we can test more keys in parallel and reduce the overall enumeration time.

5.2 Implementation Results

The 3 implementations above were implemented based on Virtex4-100-12 chip. This chip has 98,304 LUTs and FFs (49,152 slices), and its price is approximately $2,000. Using this chip we could implement 10 baseline breaker modules with 6.6 ns clock. We implemented 19 "simple" breaker modules on this chip with less than 5 ns clock cycle. Finally with our manually optimized the hardware implementation we managed to implement 23 "optimized" breaker modules with 5 ns clock cycle. These 23 modules and the added control logic used 26,862 FFs and 96,569 LUTs which occupied 48,797 (99%) chip slices. A single "optimized" full breaker module uses 1,333 FFs and 3,808 LUTs totalling 2,234 slices[12]. Figure 9 shows the total break time for a full brute force attack, for the 3 possible seed lengths (32, 48, and 60-bit). All the results assume the same Virtex4-100-12 chip. We can see that the optimized design reduces the attack time by a factor of 3.05 over the baseline. This improvement is due to improving the breaking algorithm, the faster clock rate, and the higher parallelism that the smaller space requirements allow.

[10] The simulation results were compared with a C code implementation, which we validated against [14]

[11] The automatic Xilinx optimization replaces chaining FFs by an SRL, as shown earlier in Figure 6

[12] Note that some FFs are shared between the breaker modules and that the control logic increases when a module is added

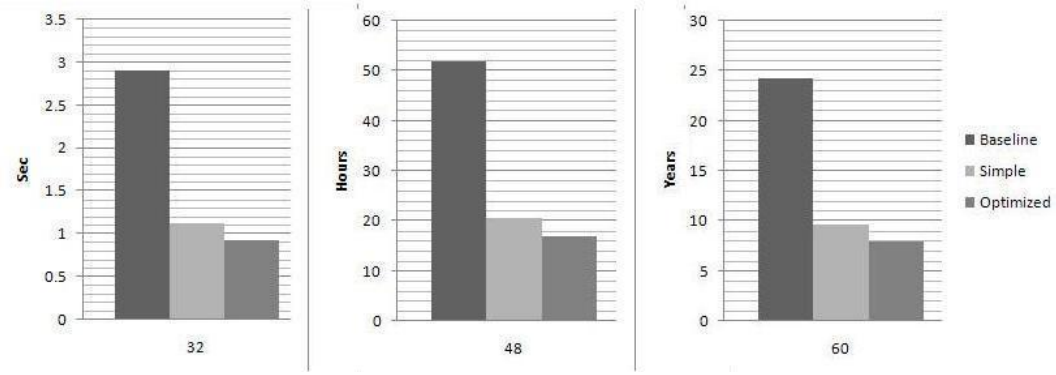


Fig. 9. Virtex-4 results comparison assuming 32, 48, and 60-bit random seed length

5.3 Comparison to previous work

Figure 10 compares the performance of our optimized design to the COPACOBANA implementation reported in [3].

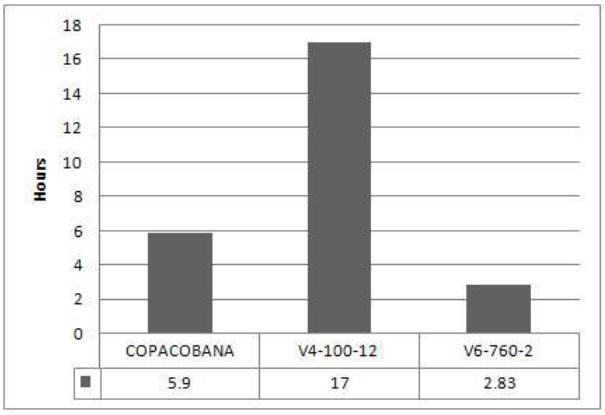


Fig. 10. Previous work comparison assuming 48-bit random seed length

Recall that COPACOBANA consists of 120 Spartan-3 FPGAs, and the authors implemented the "baseline" design on each of those chips. In comparison we show the performance of our optimized design on a *single* Virtex-4 or Virtex-6 chip.

Figure 10 shows that the 120-chip COPACOBANA is only 2.87 times faster than a single Virtex-4 (medium size) chip. Furthermore, our design implemented on a Virtex6-760-2 chip (that has more than 7 times of logic and better timing) is at least 2.09 times faster than COPACOBANA. Moreover when designing for

COPACOBANA hardware (Spartan-3) with our optimized breaker we managed to get approximately 3 times better than previous works.

6 Conclusions

Our results demonstrate that when building a break system we can use a single "mobile" breaker based on standard FPGA technology. One does not need to purchase or rent any special crypto hardware, and can easily build a homemade breaker using a standard evaluation board. The time to crack a 32-bit seed is under 1 sec, and a 48-bit seed only requires 17 hours. Thus, it is very reasonable that such a breaker could be built using only one Virtex-4 chip. Such a small system could even be installed on a hacker's car. By just recording two legitimate remote transmissions the attacker can break into the car by the next day (even assuming a 48-bit seed).

Note that in our design there are still many free FFs. This means that, theoretically, there is room for more modules based only on FFs. We have not attempted to use these FFs—in fact it is unclear whether this space can be effectively used (routing would be probably the problem)—but it may offer an opportunity for even further improvement.

References

1. Nicolas T. Courtois, Gregory V. Bard, Andrey Bogdanov.: Periodic Ciphers with Small Blocks and Cryptanalysis of Keeloq. Tatra Mt. Math. Publ. 41 (2008)
2. T. Eisenbarth, T. Kasper, A. Moradi, C. Paar, M. Salmasizadeh, and M. T. M. Shalmani: On the Power of Power Analysis in the Real World: A Complete Break of the KeeLoq Code Hopping Scheme, in Advances in Cryptology - CRYPTO 2008, pp. 203-220 (2008)
3. M. Novotnỳ and T. Kasper: Cryptanalysis of KeeLoq with COPACOBANA. SHARCS'09 Special-purpose Hardware for Attacking Cryptographic Systems, 159 (2009)
4. Indesteege, S. and Keller, N. and Dunkelman, O. and Biham, E. and Preneel, B.: A practical attack on KeeLoq. Proceedings of the theory and applications of cryptographic techniques 27th annual international conference on Advances in cryptology, 1–18, Springer-Verlag (2008)
5. Courtois, N. and Bard, G. and Wagner, D.: Algebraic and slide attacks on KeeLoq, Fast Software Encryption, 97–115, Springer (2008)
6. Bogdanov, A.: Linear slide attacks on the KeeLoq block cipher, Information Security and Cryptology, 66–80, Springer (2008).
7. Eisenbarth, T. and Kasper, T. and Moradi, A. and Paar, C. and Salmasizadeh, M. and Shalmani, M.T.M.: Physical cryptanalysis of keeloq code hopping applications, Cryptology ePrint Archive: Report 2008/058. Dostupné na: `http://www.crypto.rub.de/keeloq` (2008)
8. Kasper, M. and Kasper, T. and Moradi, A. and Paar, C.: Breaking KeeLoq in a Flash, AFRICACRYPT 2009. 5580, 402–419 (2009)
9. Keeloq wikipedia article. 30 September 2010. `http://en.wikipedia.org/wiki/KeeLoq`

10. Microchip. An Introduction to KeeLoq Code Hopping. Available from http://ww1.microchip.com/downloads/en/AppNotes/91002a.pdf (1996)
11. Microchip. Secure Learning RKE Systems Using KEELOQ Encoders. Available from ww1.microchip.com/downloads/en/AppNotes/91000a.pdf (1996)
12. Microchip. HCS301 Keeloq Code Hopping Encoder and Transponder. Available from ww1.microchip.com/downloads/en/devicedoc/21143b.pdf (2001)
13. Microchip. HCS410 Keeloq Code Hopping Encoder and Transponder. Available from http://ww1.microchip.com/downloads/en/DeviceDoc/40158e.pdf (2001)
14. Microchip KeeLoq tool, V.02.00.04, Available from http://www.microchip.com
15. Xilinx. Using Look-Up Tables as Shift Registers (SRL16) in Spartan-3 Generation FPGAs. Available from www.xilinx.com/support/documentation/application_notes/xapp465.pdf (2005)
16. Xilinx. Virtex-4 FPGA - User Guide. Available from www.xilinx.com/support/documentation/user_guides/ug070.pdf (2008)
17. Xilinx. Virtex-5 FPGA - User Guide. Available from www.xilinx.com/support/documentation/user_guides/ug190.pdf (2010)
18. Xilinx. Virtex-6 FPGA Configurable Logic Block - User Guide. Available from www.xilinx.com/support/documentation/user_guides/ug364.pdf (2009)