Data Protection Using Recursive Inverse Function

Teng Xu, Hongxiang Gu, and Miodrag Potkonjak
Computer Science Department
University of California, Los Angeles
{xuteng, hxgu, miodrag}@cs.ucla.edu

***Abstract*—Data security and privacy have emerged to become an important issue in various types of applications. Although many cryptographic cyphers are proposed to leverage the issue, they normally suffer from the problems of either requiring large power/bandwidth consumption or employing linear system which is easy to break. To solve the problem of traditional cyphers, we have proposed a new hardware security primitive: recursive inverse function (RIF) designed on the field-programmable gate array (FPGA). The RIF takes advantage of a pair of inverse functions, building a recursive scheme for message encryption and decryption. The inverse functions are defined as a pair of functions where each function implements a mapping being inverse to the mapping of the other function. On the top of it, the recursive structure guarantees the input-output mapping to be statistically extremely hard to predict. The RIF can be easily implemented using hierarchical lookup-table (LUT) structures with low delay and power overhead. Using our proposed RIF structure, we have demonstrated how the RIF can be incorporated into a processor design to enable the data protection. Finally, we implement our scheme on a Xilinx Spartan-6 FPGA device to analyze the performance and the overhead.**

I. Introduction

The security and privacy of data have emerged to become an important issue in many applications. In the domain of hardware data storage, there have been many possible ways for the attackers to tamper or to steal the data, for example, through the communication channel, the processors, or the memories. Among all the above data attacks, the memory attacks are among the most common type, where the assumption is that the contents stored in the memory can be read or even modified by the attackers.

To protect the data from memory attacks, a most intuitive way is to use security ciphers to encrypt and to decrypt the data. However, there has been two well-known requirements in the design of security ciphers. The first is low power and low bandwidth which is due to the fast growth of resource constrained systems, such as mobile devices and wireless sensor networks. All of the above systems are highly constrained by the battery life and the available network bandwidth. The second requirement is the confusion and diffusion of the system which are two properties of the operation of a secure cipher identified by Shannon [1]. For example, any linear system can be easily attacked by using statistical models. Traditional cryptography designs often fail to meet either of the two requirements. Therefore, there is a need to propose new types of security primitives which are low power, low bandwidth, and employ high statistical security.

The objective goal of this paper is to propose a new security primitive, the recursive inverse function, to leverage the requirements proposed above. As the name suggested, RIF employs two major properties, respectively "inverse" and "recursive". The first and the most important advantage of RIF is to employ inverse structure for message encryption and decryption. Compared to traditional cryptographic cyphers, both operations under the inverse design are ultra lightweight. More importantly, in RIF, the bandwidth in message transferring equals to the size of message itself, only a very short random seed needs to be synchronized at the beginning of the transferring. The second important property is the "recursive". RIF achieves this by designing a scheme to recursively use historical information to encrypt the message in the current round. Thus the relation between the plaintext and the ciphertext becomes even more nonlinear and will depend on the previous calculation results.

The RIF is implemented using the FPGA-based hierarchical LUT networks. It consists a pair of inverse functions. The RIF has two hardware pieces, respectively $f_{original}$ and $f_{inverse}$. We guarantee that the mappings are the inverse to each other by manually allocating the LUT contents level by level. Based on the hierarchical structure, RIF recursively encrypts and decrypts messages such that unless the attacker acknowledges all the previous rounds of calculation results, he/she has no chance to retrieve the message in current round.

Based on the architecture of RIF, we propose a new scheme to protect the data stored in the memory. The core idea is that we incorporate RIF to prevent memory attacks by using it to secure the data transfer between the processor and the memory. Our assumption here is that the processor can be trusted while the memory has the risk to be tampered thus can not be trusted. For all the data to be stored in memory, we use $f_{original}$ in RIF to encrypt. Meanwhile, for the data fetched from the memory, we use $f_{inverse}$ to decrypt. The protection scheme encrypts all the data outside the processor with low delay, area, and power overhead. We have also implemented our scheme on a Xilinx Spartan-6 FPGA device to analyze the performance.

II. Related Work

A. Lightweight Security Primitives

There are a number of lightweight protocols for several secret key cryptographic primitives [2][3]. Most of them focus on proposing lightweight architecture for already exists cryptographic cyphers, e.g., SHA-1, AES [4]. Physical unclonable function (PUF) is a new type of lightweight security primitive first proposed by Pappu et al. [5], Majzoobi et al. demonstrated

a new methodology for low-power PUF design which enables multiple delay lines for response creation [6]. Xu et al. developed the work along the line to further reduce the energy consumption of PUF based protocols as well as to enhance the stability of PUF systems by employing completely digital architecture [7][8][9].

B. Data Protection

A wide spectrum of techniques are proposed to target data protection from different viewpoints. Tamper resistant software is defined as the one that is resilient to observation and modification. Aucksmith proposed an approach for creating tamper resistant software through employment of self-modifying and self-decrypting segment of code that is installed in unique way on in platform [10]. Another popular technology is obfuscation, which is used for the protection of software intellectual property. While most often obfuscation is conducted in compilation phases that translate source code into assembly, there are several popular technique that act during translation of assembly code into machine code [11].

III. Architecture

We introduce the hardware architecture of RIF in this section. The major issue we focus here is how to use hardware to implement an inverse mapping, essentially to meet the "inverse" property of RIF. An important prerequisite of building an inverse mapping is that each single mapping of the function must be a one-to-one mapping. Our solution to build such a mapping is to use hierarchical LUT connections. The key idea is to allocate the contents in the LUTs in such a way that any arbitrary mapping can be generated.

We start with a simple example. Consider a LUT network with k k-input LUTs, if all the LUTs take the same k inputs, then they will generate k outputs, thus forming a k-to-k mapping. Due to the fact that the output in the mapping is completely decided by the way to allocate the contents of the LUTs, thus any arbitrary k-to-k mapping can be generated as long as the LUT contents can be configured. Coming back to the design of a pair of inverse functions, the first constrain is the design of one-to-one mapping, and the second constrain is the creation of inverse mapping. Since any arbitrary mapping can be generated, thus both constrains can be easily satisfied by properly allocating LUT contents.

In the above example, we assume that all the k LUTs must take all the same k inputs. This is a prerequisite to build an arbitrary mapping. However, there still exists many flexibility in the detail LUT connection. For example, for each individual LUT, the order of the inputs does not have to be fixed. The only requirement is that each LUT must take all k inputs regardless of the order of the input pins. Instead of using $x_1...x_k$ as the inputs for all of the LUTs, we can switch the order of the inputs to be any combination of $x_1...x_k$. Meanwhile, the LUT locations to fill in the values need to be adjusted because the positions have switched.

A real system of RIF can be built on the top of the proposed structure by combining the k LUT blocks both horizontally and vertically, thus to increase the number of inputs/outputs as well as to enhance the security of the system. As for horizontal connection, we can increase the number of LUTs as well as the number of inputs. For instance, we can duplicate the structure of k LUTs with a new set of k LUTs taking a new set of k inputs. If we put them in parallel, the system will be extended to a $2k$ input-output system composed of $2k$ k-input LUTs. The generated mapping is still a one-to-one mapping since the connecting of two one-to-one mapping sequence still forms a one-to-one mapping. However, only using one level of LUTs can lead the structure to be easy to break. Therefore, in order to enhance the security of our system, we apply vertical connection. The idea is to connect the LUTs in a series to form a hierarchical structure. To be more specific, the structure can be formed by connecting the outputs of previous level LUTs as the inputs of next level LUTs which is equivalent to connect the outputs of previous mapping as the inputs of the next mapping. Due to the transitive of one-to-one mapping, the vertical connection still preserves the one-to-one property.

Combining the proposals from the above discussion, the $f_{original}$ and the $f_{inverse}$ can be generated using two separate pieces of LUT network, realizing two mappings of opposite directions. Each LUT network can consist n levels of LUTs where each level implements a one-to-one mapping using the approach described above. To guarantee that the overall mapping is the inverse to each other, for the ith level LUTs in $f_{original}$ which implement a one-to-one mapping, the inverse mapping is implemented by the $(n+1-i)th$ level LUTs in $f_{inverse}$. Thus each level of LUTs in $f_{original}$ have a corresponding LUT network in $f_{inverse}$ at the symmetric position implementing the inverse mapping. Between levels of LUTs, the outputs of previous level LUTs are fed as the inputs to the next level LUTs. The overall structure is lightweight in terms of both time and area overhead since it only consists of a limited number of interconnected LUTs both horizontally and vertically.

IV. Recursive Encryption and Decryption

The second major property of RIF focuses on to provide a recursive scheme to encrypt and decrypt messages using the above hardware structure. Encryption and decryption are among the most popular protocols applied in various scenarios. Our motivation here is to provide a novel lightweight recursive scheme to enhance the security of the system based on the core idea that the current outputs of the system should depend on all the previous outputs, thus to dramatically increase the efforts of attacking.

Before explaining the specific flow for recursive encryption and decryption using RIF, we want to first define notations and assumptions for the scheme.

- S − initial random seed.
- Alice − the party to send and encrypt messages, owns the hardware of $f_{original}$.
- Bob − the party to receive and decrypt messages, owns the hardware of $f_{inverse}$.
- $M_1, ..., M_n$ − message flow for encryption and decryption. The flow starts from M_1 and ends at M_n.
- g − a secret function synchronized between Alice and Bob before encryption and decryption.

Algorithm 1 Recursive Encryption and Decryption

1: Alice calculates R=$f_{original}(S)$ and sends R to Bob.
2: Bob retrieves S using $S = f_{inverse}(R)$.
3: Alice wants to encrypt and send the ith message M_i.
4: Alice calculates $G_i = g(S, M_1, ...M_{i-1})$, then calculates and sends $O_i = f_{original}(G_i \oplus M_i)$.
5: Bob receives O_i. He calculates $G_i = g(S, M_1, ...M_{i-1})$, then retrieves M_i using $M_i = f_{inverse}(O_i) \oplus G_i$.
6: Step 3 to 5 are repeated until the whole message flow is transferred.

The flow of recursive encryption and decryption is shown in Algorithm 1. The algorithm has the following properties. (1) Low overhead for both encryption and decryption. Essentially, only two calculations are required in each round of message transfer process. The first is to use the inverse mapping functions, e.g., $f_{original}$ and $f_{inverse}$, they are implemented using the lightweight LUT network. The second function is the synchronization function g. It has the flexibility to be designed in different formats while preserving the lightweight property, e.g., $g(x_1, x_2, ...x_n) = x_1 \oplus x_2 \oplus ...x_n$ is an possible option which is extremely easy and light to be implemented. (2) By recursively incorporating the previous messages in the encryption and decryption process, the security of system is enhanced. It is mainly due to the fact that the outputs of encryption in the current cycle will depend on all of the previous messages, which means that if the attacker wants to retrieve the message M_i, he/she needs to first attack and retrieve all the previous messages M_1 to M_{i-1}. This recursive encryption and decryption dramatically increase the efforts the attacker needs to take to steal the message.

V. Data Protection Using RIF

We formally introduce our scheme for RIF enabled data protection. We start from the most intuitive protection diagram, then we discuss how to prevent our system from replay attacks. Lastly we present the performance and the overhead of the scheme by implementing it on the Xilinx Spartan-6 FPGA.

A. Protection Diagram

Our key idea is to use RIF to encrypt data before storing them in memory and decrypt them when fetching. The diagram of the work flow is shown in Figure 1. Every time when data needs to be stored in memory, $f_{original}$ is used to encrypt it, and when the data needs to be read back, $f_{inverse}$ is applied for decryption. In this design, we do not need to change any inner structure of the processor since we assume that the processor is trustable. However, the memory outside the processor can be tampered or even modified.

B. Replay Attack

A major challenge of the scheme is how to prevent the replay attack. Assume that the attacker can access and overwrite the data stored in the memory, this will result the processor to read in wrong encrypted data. Our solution to prevent this is to add a new RIF to control the write signal of the memory. We name the two functions in this RIF as $f_{original2}$ and $f_{inverse2}$ to distinguish. The $f_{original2}$ is owned by the processor while the $f_{inverse2}$ is owned by the memory. The replay attack prevention is demonstrated in Algorithm 2. The essence is to use the RIF for authentication. In order to prevent the attacker from stealing historical authentication code through the communication bus, the processor takes advantage of the dynamic key. For instance, in Algorithm 2, it uses the current time as the authentication key. Only when the decryption results in the memory indicates the key is close enough to the current time, the write to the memory is enabled. Note that since the time cost of data transfer plus encryption and decryption is short, thus t should be very close to t_0.

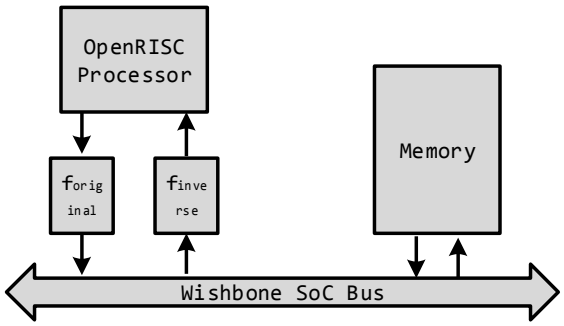

Fig. 1: The RIF-based data protection flow between OpenRISC processor and memory.

Algorithm 2 Replay Attack Prevention

1: The processor has the $f_{original2}$ and the memory has the $f_{inverse2}$.
2: The processor has an encrypted message M to store in the memory.
3: The processor fetches current system time t and calculates T=$f_{original2}(t)$.
4: The processor sends M and T to memory through the communication bus.
5: The memory calculates t=$f_{inverse2}(T)$.
6: The memory fetches current system time t_0.
7: The memory enables the write of M only when $t_0 \approx t$.

C. RIF Implementation

We implement the RIF based data protection scheme on OpenRISC reference platform system-on-chip (ORPSoC) [12]. The system is implemented on a digilent atlys development board with Xilinx spartan-6 LX45 FPGA. The OpenRISC core will serve as the processor in the protection scheme, and a DRAM will serve as the memory/storage system. Our implementation will be a demonstration of how our RIF design can be adopted in a real world application. Our RIF module resides right between ORPSoC and the external RAM. Whenever a write action is performed, the 32-bit data will go

through the $f_{original}$ and the data will be encrypted. Whenever the memory controller is accessing the data inside the RAM, the data will first go through the $f_{inverse}$ and be decrypted. In both functions in the RIF, we use LUT ladder network with 32-bits input-output mapping. Both functions contain 4 levels of LUTs.

D. Overhead

We first present the FPGA resource overhead of the RIF scheme as shown in Table I. We can see that the largest overhead is in terms of LUTs and slices, and they take around $8\% - 9\%$ overhead.

Type	Original	Protected	overhead
D-flipflop	6705	6780	1.12%
LUTs	12560	13584	8.15%
Slices	4457	4854	8.91%
DSP48A1s	4	4	0%

TABLE I: FPGA resource usage before and after protection, overhead is calculated comparing to original design.

We also analyze the timing overhead. Each time the system intends to access the memory, the data passes through the RIF. Additional module will certainly introduce latency to the overall performance of the whole ORPSoC system. To assess the performance impact introduced to the entire system, we run embedded benchmark programs on our design and compare their performance with the result from original design. Our benchmark suite includes Dhrystone [13], CoreMark [14], a subset of MiBench [15], and a subset of Zlib [16]. The normalized run time for these benchmark programs are shown in Figure 2.

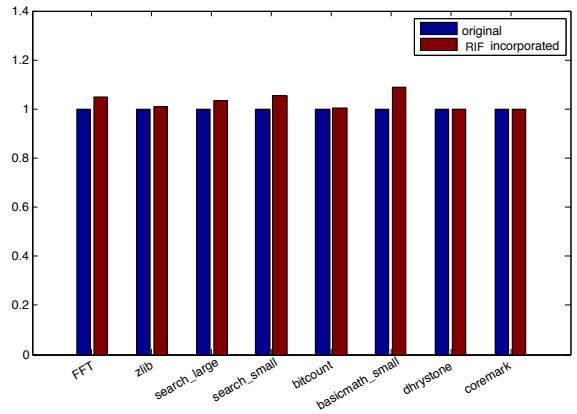

Fig. 2: Normalized runtime for both original design and RIF incorporated design.

As we can observe from the figure, small programs such as Dhrystone and Coremark show no change in performance when RIF is introduced to the design. We believe that this is caused by the relatively large instruction cache and data cache. Both caches are 8KB large. For small programs, in fact there exists no access to the memory, therefore our RIF is never used and accessed. Larger programs such as MiBench basic math shows moderate increase in runtime.

VI. Acknowledgement

This work was supported in part by the NSF under Award CNS-0958369, Award CNS-1059435, and Award CCF-0926127, and in part by the Air Force Award FA8750-12-2-0014.

VII. Conclusion

In this paper, we have proposed a recursive scheme for data protection using a novel hardware security primitive: recursive inverse function. The core idea is to implement a pair of functions that forms bijective mappings. Then we recursively incorporate historical messages in the encryption and decryption process. We have proposed the architecture of RIF using LUTs on FPGA. On the top of the architecture and the scheme, we have depicted the RIF based approach to enable data protection. Our demonstrated system implementation indicates that our protection scheme is low-overhead in terms of both area and time.

References

[1] Shannon, Claude E. "Communication theory of secrecy systems," *Bell system technical journal* 28.4, pp. 656-715, 1949.

[2] P. Yalla, and J.-P. Kaps, "Lightweight cryptography for FPGAs," *Reconfigurable Computing and FPGAs, 2009. ReConFig'09. International Conference on*, IEEE, 2009.

[3] T. Xu, J. B. Wendt, and M. Potkonjak, "Security of IoT Systems: Design Challenges and Opportunities," *ICCAD*, pp. 417-423, 2014.

[4] J.-P. Kaps, and S. Berk, "Energy comparison of AES and SHA-1 for ubiquitous computing," *Emerging directions in embedded and ubiquitous computing*, pp. 372-381, 2006.

[5] R. Pappu, B. Recht, J. Taylor, and N. Gershenfeld, "Physical one-way functions," *Science*, vol. 297, no. 5589, pp. 2026-2030, 2002.

[6] M. Majzoobi, F. Koushanfar, M. Potkonjak, "Lightweight secure PUFs," *ICCAD*, pp. 670-673, 2008.

[7] T. Xu, J. B. Wendt, and M. Potkonjak, "Digital Bimodal Function: An Ultra-Low Energy Security Primitive," *ISLPED*, pp. 292-297, 2013.

[8] T. Xu, and M. Potkonjak, "Robust and flexible FPGA-based digital PUF," *FPL*, pp. 1-6, 2014.

[9] T. Xu, J. B. Wendt and M. Potkonjak, "Secure Remote Sensing and Communication using Digital PUFs," *ANCS*, pp. 173-184, 2014.

[10] D. Aucsmith, "Tamper resistant software: an implementation," In R. Anderson, editor, *Information Hiding*, volume 1174 of Lecture Notes in Computer Science, pages 317-333. Springer Berlin Heidelberg, 1996.

[11] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," *In Proceedings of the 10th ACM Conference on Computer and Communications Security*, pp. 290-299, 2003.

[12] D. Lampret and J. Baxter, "OpenRISC 1200 IP core secification," rev. 0.12, 2011.

[13] R. Weicker, "An overview of common benchmarks," Computer, vol. 23, no. 12, pp. 65-75, Dec. 1990.

[14] Coremark, an EEMBC benchmark, 2012.

[15] M. R. Guthaus et al, "MiBench: A free, commercially representative embedded benchmark suite," *IISWC*, pp. 3-14, Dec. 2001.

[16] P. Deutsch and J.-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3," 1996.