Steganographic Resources of FPGA-based Systems for Approximate Data Processing

Kostiantyn Zashcholkin[a], Oleksandr Drozd[a], Svetlana Antoshchuk[a], Olena Ivanova[a] and Oleg Sachenko[b]

[a] *Odessa National Polytechnic University 1, Ave. Shevchenko, Odessa, 65044, Ukraine*
[b] *West Ukrainian National University, 11, Lvivska Str., Ternopil, 46009, Ukraine*

Abstract

The program code controls the operation of FPGA (Field Programmable Gate Arrays) chips. Through the use of program code it is possible to interfere with the operation of these chips. Operational monitoring (integrity, authenticity, distribution paths) of program code is one of the most effective means of preventing illegal interference with FPGA-based systems. Traditional methods of monitoring program code have the problem of storing control data. A promising approach is to embed the monitoring data into the FPGA program code in a steganographic manner. In this case, the fact of the availability of monitoring data and the fact of monitoring is not obvious to an external observer. There are methods for steganographic embedding of additional data into the FPGA program code. Methods of this kind are based on equivalent transformations of the program code. This paper proposes and experimentally substantiates the presence in the FPGA program code of new resources for steganographic data embedding. These resources arise in the programming code of FPGA systems that perform approximate data processing. In the paper, inessential LUT (Look Up Table) FPGA units are classified as such resources. These units are involved in the calculation of the least significant (discarded during rounding) bits of the result and do not participate in the calculation of the most significant (retained) bits. Also inessential bits of the program code of LUTs are referred to new resources. These bits are bits that are not accessed during the calculation of the result of the operation. The experiments performed have shown the possibility of organizing stego containers in the FPGA program code using the identified additional resources. The volume of such stego containers has been estimated. It is shown that this volume is sufficient for storing monitoring data, which are used for operational monitoring of the FPGA program code.

Keywords 1

Steganographic approach, FPGA-Based Systems, program code of FPGA, integrity monitoring, LUT-oriented architecture, stego container, floating-point format, approximate data processing, complete arithmetic operation

1. Introduction

Programmable components are now very widely used in computer systems. The ability to programmatically control the operation of programmable chips provides significant advantages over specialized chips. The main advantage is that the operation of programmable components can be modified at any time. This makes it possible:

- eliminate errors found in the program code after putting the system into operation;
- improve the functioning of the component at any stage of system lifecycle;
- adapt the functioning of the device to changes in its operating conditions.

CMIS-2021: The Fourth International Workshop on Computer Modeling and Intelligent Systems, April 27, 2021, Zaporizhzhia, Ukraine
EMAIL: const-z@te.net.ua (K. Zashcholkin); drozd@ukr.net (O. Drozd); asgonpu@gmail.com (S. Antoshchuk); en.ivanova.ua@gmail.com (O. Ivanova); olsachenko231@gmail.com (O. Sachenko)
ORCID: 0000-0003-0427-9005 (K. Zashcholkin); 0000-0003-2191-6758 (O. Drozd); 0000-0002-9346-145X (S. Antoshchuk); 0000-0002-4743-6931 (O. Ivanova); 0000-0001-9337-8341 (O. Sachenko)

All these modifications can be done by replacing the program code or parts of it. However, this ease of modifying the functioning of a programmable component also has a negative side. It consists in the possibility of malicious interference in the functioning of a computer system through the program code.

One of the most effective ways to counteract unauthorized actions against program code is operational monitoring [1]. There are considerable developments (methods, means, tools) which provide different variants of program code monitoring: integrity monitoring [2]; authenticity monitoring [3]; distribution monitoring [4]; monitoring the legitimacy of the use [5] of the program code. All of these types of monitoring are based on the well-researched mathematical apparatus of hash functions [6, 7] and message authentication codes [8, 9]. Based on the methods built on this mathematical apparatus, monitoring data are created with the help of which monitoring is performed.

However, all existing types of monitoring have a problem. This problem is the way monitoring data is stored and accessed. There are three most common approaches to storing monitoring data:

1. attaching monitoring data to program code [10];
2. storage of monitoring data together with the program code in memory or in file system [11];
3. storage of monitoring data in a remote database (monitoring data are requested from the database at the time of monitoring) [12].

All three of the above approaches to storing monitoring data make open the fact of the availability of monitoring data and the fact that monitoring is performed [13, 14]. The protection of monitoring data (including by means of encryption) is leveled by the openness of the fact of their presence. Knowledge of this fact creates the possibility of manipulating monitoring data to falsify monitoring.

There is an approach to storing monitoring data that differs from these common approaches. This approach is based on the use of the theory of digital steganography [15]. Steganography aims to protect data by secretly embedding (hiding) data of one type in data of another type. Protected data are embedded in a stego container [16], which is a specially prepared information object. When using the steganographic approach, the very fact that the data is protected becomes hidden. The main requirement for this approach is the absence of any changes (functional, visual, structural) in the stego container after the embedding of additional data into it.

The use of digital steganography methods for storing monitoring data is as follows. The monitoring data is calculated in the traditional way. After that, they are embedded in the program code in a steganographic manner. At the same time, the embedded monitoring data must not affect the functioning of the device which is programmed by this program code. Also, the embedding should not lead to structural or visual changes in the program code. Thus, the program code acts as a stego container, and monitoring data acts as additional data embedded into this container. With this approach, the very existence of monitoring data is hidden from the outside observer. The fact that monitoring is carried out in relation to the program code is also hidden. At the time of the active monitoring stage, the monitoring data embedded in the program code is extracted. Extraction is possible if there is a stego key that describes the rules for localizing additional data in the program code space.

The approach to steganographic storage of monitoring data appeared later than other approaches to solving this problem. Because of this, traditional approaches, while having disadvantages, are used much more frequently. The steganographic approach has the prospect of taking a significant place along with the traditional approaches. Therefore, the research and development of stenographic methods and means of storing monitoring data is an important and urgent task.

2. Literature Review and Goal of the Paper

Among the programmable components of computer systems a significant place belongs to FPGA chips [17, 18]. These chips have a programming principle that differs from microprocessors and microcontrollers. Microprocessors change their behavior as a result of changing the program. However, in this case, the structure of the chip of the microprocessor remains unchanged. FPGA chips have a different programming principle. An FPGA chip consists of a large number of programmable elementary units. Modification the FPGA program code leads to the reconfiguration of the elementary units and changes in the connections between them. That is, the program code is able to change the structure of the FPGA chip [19].

In addition, the difference between microprocessors and FPGAs lies in the centralization of computations. In microprocessors, the computing process is concentrated in one or more central computing nodes. In FPGA, the computation process is distributed over a large number (hundreds of thousands, millions, tens of millions, depending on the generation of the FPGA chip) of elementary computing units. These features determine the higher performance of FPGA in comparison with microprocessors [20].

High computing performance and the possibility of reprogramming have led to the fact that FPGAs are often used as components of safety-critical systems [21, 22]. Control systems for power engineering, high-speed transport, and aerospace objects often contain FPGA components. The ability to change the program code for the components of such systems has two sides. On the one hand, thanks to this feature, the functioning of the component can be improved at any time and errors found can be eliminated. On the other hand, this possibility creates the danger of destructive influences on the functioning of the system. Under these conditions, operational monitoring of the program code (its integrity, authenticity, ways of its distribution) of FPGA chips is a very urgent task.

As already noted, the problematic place of methods for program code monitoring is the way of storing control data and the way of accessing them. One of the most effective approaches to storing monitoring data is the steganographic approach. This approach has received the greatest development for multimedia stego containers such as bitmaps, digital video and digital audio [23]. This is due to the fact that these containers are of analog origin. Digital representations of images, video and sound are obtained by registration analog phenomena. The results of this registration have been digitized and presented in the appropriate multimedia formats. As a result of such transformations, the data of elementary units (pixels, samples) of multimedia containers have an approximate representation. This means that minor distortions of this data do not lead to degradation of the container.

The low-order bits of the elementary units of the container in the spatial or transformation domain have such a small contribution to the basic function of the container that their distortion cannot be detected. Therefore, with regard to multimedia containers, steganographic embedding of additional data is efficiently implemented through non-equivalent transformations. Namely, the low-order bits of the elementary units of the spatial or frequency domains of the container are used as storage for steganographic data [24, 25].

The program code (and in particular the program code of FPGA chips) is a container with exact data. Distortions of the program code bits lead to distortions in the functioning of the device which is programmed by this code. For steganographic embedding of additional data into the FPGA program code, methods of equivalent transformation are used [26, 27]. These methods do not change the functioning of the chip and the size of program code, but allow to embedding additional data into this code.

However, when implementing approximate data processing on FPGAs, this approximation is indirectly transferred to the exact represented program code. Some of the FPGA elementary units are involved in the calculation of only least significant bits of the results [28]. The program code of such units can be changed non-equivalently for the embedding of additional data into it. Moreover, such an unequal embedding will not affect the correctness of the function of device.

Thus, in the case of approximate data processing, the FPGA program code has the potential to apply non-equivalent transformations to it. These transformations are similar to those used for multimedia containers. However, such distorting transformations do not affect the main function of the code [29].

Based on this, the *goal of this paper* is to identify and estimate the volume of resources of the FPGA program code, which allow performing non-equivalent embedding of additional data. The identification of these resources in this paper is performed for FPGA chips, which carry out approximate data processing. Resource estimation is done for the task of steganographic storage of monitoring data necessary to monitor the FPGA program code.

3. The proposed approach to the use of FPGA program code resources for non-equivalent steganographic embedding

Many operations that are used in approximate data processing have the same operand and result format. For example, specifications for floating-point arithmetic operations in most cases dictate the

same format and size for operands and results of operations. However, a situation often arises in which the result of an operation is larger than the size of the operands. In this case, the size of the result is brought to the size of the operands by discarding a certain number of low-order bits, followed by rounding [30, 31].

So, for example, one of the main operations – the operation of multiplying floating-point numbers, in accordance with most specifications, forms the result mantissa, the size of which is equal to the size of the operand mantissa. Multiplying the mantissa of operands that are n-bits in size naturally produces a result which contains $2n$-bits. However, the specification of the multiplication operation requires the mantissa to have the same size as the mantissa of the operands, that is, n-bits. To bring the mantissa of the result to the required format, rounding is performed, which consists in discarding the n least significant bits [32, 33].

The main elementary units of the structure of FPGA microcircuits are LUT (Look Up Table) units. In modern FPGA series, the number of such units is in the millions and tens of millions. The LUT unit has m inputs (m for different FPGA families ranges from 4 to 8) and performs the calculation of one logic function from m variables. LUTs are programmable units. Setting up a unit to implement a specific logical function is performed by a 2^m bit binary program code. Programming the FPGA microcircuit leads to tuning each of the LUT units to implement the required logic function and creating the required connections of such units with each other.

3.1. The first kind of resources for non-equivalent data embedding into the FPGA program code

The structure of the implementation on FPGA of arithmetic operations performed in approximate data processing is focused on calculating the full result with its subsequent reduction to the required format. In floating-point arithmetic, converting the result to the format of the operands is performed in two stages: first, the complete result is calculated, and then its incorrect bits are discarded during the rounding process.

Let $A = <a_n, a_{n-1}, \dots, a_1>$ and $B = <b_n, b_{n-1}, \dots, b_1>$ these are the mantissa of the operands. The result of the multiplication is the complete $2n$-bit product $C = < c_{2n}, c_{2n-1}, \dots c_{n+1}, c_n, c_{n-1}, \dots, c_1>$. Rounding is performed to bring the result to the format of the operands. As a result is formed $C_{Result} = < c_{2n}, c_{2n-2}, \dots c_{n+1}>$, which consists of the most significant n bits of the complete result. In this case, the multiplier circuit calculates both the most significant bits of the result and the least significant bits (discarded) bits of the result.

Based on this, it can be expected that the structure of the FPGA system can contain a subset of LUT units that perform the calculation of only the discarded least significant bits of the result and do not participate in the calculation of the retained most significant bits. If such LUT units are present in the FPGA system, then their program code can be corrupted, and this corruption will not lead to an incorrect result. These LUT units will hereinafter be referred to as inessential LUTs. The rest of the LUT units have a direct impact on the formation of the rounded result and therefore belong to the essential ones.

Under these conditions, the program code of inessential LUT units (in whole or its individual bits) can be used as a storage of secret data, embedded in the FPGA program code in a steganographic way.

Thus, inessential LUTs represent *the first kind of FPGA code resources* that can be used to unequally embed additional data. The experimental part of this paper aims to confirm the availability of this resource and estimate its volume.

3.2. The second kind of resources

When performing approximate arithmetic operations, not only LUT units can be inessential, but also individual bits of the program code of LUTs. We will consider as inessential bits of the program code of the LUTs the bits that are not accessed during the calculation of the most significant bits of the result. It is useful to highlight such bits only in the program code of the essential LUT units. This

is due to the fact that the program code of inessential units has been identified by us as an independent type of steganographic resource.

If the program code of the LUT units contains bits that are not accessed during the calculation of the most significant bits of the result, then these inessential bits can be corrupted. Such a distortion will not affect the correctness of the calculations when forming the rounded result of the operation.

Based on this, we define the inessential bits of the program code of the essential LUTs as the *second kind of resources* that can be used for unequal embedding of additional data. The experimental part of this paper aims to confirm the availability of this resource and estimate its volume.

3.3. The main provisions of the proposed approach

The proposed approach to non-equivalent steganographic embedding of data in FPGA code is summarized in the following principles.

Principle 1: Schema of the FPGA system for approximate data processing contains inessential LUT units. These units are involved in the calculation of the least significant bits discarded during rounding of the result and do not participate in the calculation of the retained most significant bits.

Principle 2: if the program code of irrelevant LUTs is distorted, it will not distort the final result of the arithmetic operation.

Principle 3: the program code of inessential LUT units can be used as storage for steganographic data;

Principle 4: the program code of essential LUT units may contain inessential bits, which are not accessed during the calculation of the most significant bits in result of the operation.

Principle 5: Corruption of inessential bits of the LUT program code does not affect the correctness of the rounded result of an arithmetic operation.

Principle 6: Program code of inessential bits can be used as storage for steganographic data.

It is necessary to prove the availability of these resources and estimate their volume. For this, an experimental study is carried out. The experimental material is FPGA systems that perform the operation of multiplying floating-point numbers of various bits.

4. Experimental research and estimation of resources for non-equivalent data embedding into the FPGA program code

The *purpose of the experiment*: to prove the existence of the two types of resources described in this paper, suitable for non-equivalent steganographic embedding of additional data into the FPGA program code space. Also it is necessary to estimate the volume of these resources and justify its sufficiency for application in the tasks of monitoring the FPGA program code.

To carry out the experiment in the CAD system Intel Quartus Prime 20.1 Lite Edition [34], five FPGA systems were designed. Each of them is a floating-point mantissa multiplier. The size of the operands used in these systems is 4, 6, 8, 10 and 12 bits, respectively. The multiplier circuits are derived from the Intel Quartus Prime CAD library components. The implemented multipliers allow obtaining a $2n$-bit result for n-bit operands. The result of calculations before rounding and discarding inessential bits was available at the output of experimental multiplication circuits. The synthesis of FPGA systems was performed for target FPGA chips Intel Cyclone IV EP4CE15F23A7 [35].

To carry out the experiment, three software applications were developed. They will hereinafter be referred to as App1, App2 and App3, respectively.

The App1 application was implemented in the TCL language. It runs in the Intel Quartus Prime CAD environment. The purpose of App1 is to retrieve detailed information about the FPGA design structure from the Intel Quartus internal CAD database. This information contains a list of FPGA LUT units, the values of the program codes for these units and a list of links that connect the LUT units to each other. The result of the work of the App1 application was presented in a form suitable for submitting the App2 and App3 applications to the input.

App2 is developed in Delphi 10 Seattle demo version [36]. App2 receives information about the structure of the FPGA system from App1 and determines which LUT units in the system are

inessential. The functioning of the App2 application is as follows. The application sequentially iterates over the values of the operands at the input of the multiplier and stores the correct 2*n*-bit multiplication result and the correct values at the outputs of each of the LUT units in the system. Thereafter, the output values from the LUT units were sequentially corrupted. In doing so, the application analyzed which bits of the result (discarded or retained) are affected by each of the distortions. LUT units, whose inverted outputs only distorted the discarded result bits, were categorized as inessential units.

Fig. 1 shows the results of an experiment to identify inessential LUT units. These results show the percentage of inessential LUT units in experimental FPGA systems, which ranges from 39.3% to 42.5%.

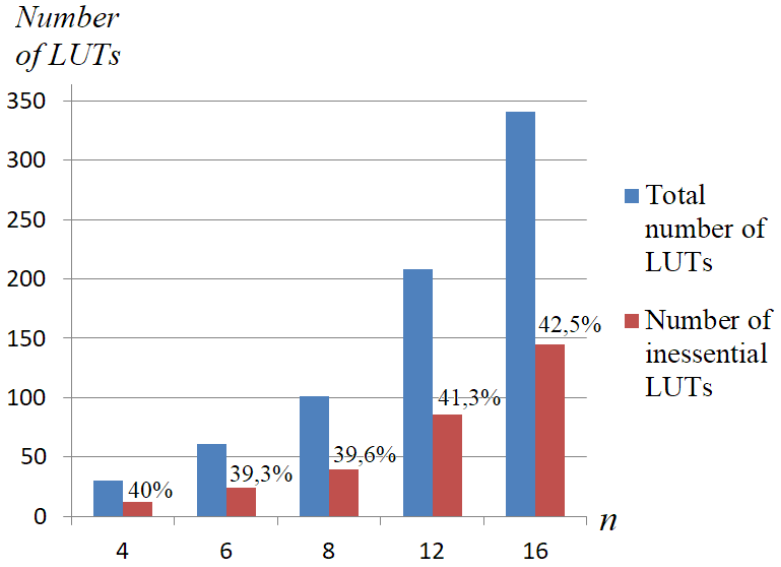

Figure 1: Number of inessential LUT units

The next part of the experiment was to determine the number of inessential bits in the program codes of the LUT units. This research was done using App3, also developed in the Delphi 10 Seattle demo version. The main window of the App3 application for 8-bit multiplier FPGA systems is shown in Fig.2.

App3 also iterates over the multiplier operands. At the same time, the application determines the bits for each of the LUT units of the FPGA-system to which the addressing is carried out. At the end of this study, each of the LUT units 1 ... 101 is evaluated by the following indicators (Fig. 2): N is the number of used inputs of the LUT unit; B1 is the number of bits that could be addressable in the LUT based on the number of its inputs used; B2 and B3 are the numbers of bits addressed in the LUT unit when multiplying the binary codes of numbers and binary codes of the normalized mantissa. Binary codes of numbers take all possible values. The binary codes of the normalized mantissa are defined at half of these values. Table 1 and Fig. 3 show the generalized count of the number of inessential bits for each of the multiplier widths.

The results of the experiments made it possible to show that inessential LUTs are present in FPGA-based floating point multipliers. The portion of inessential LUT units ranged from 39.3% to 42.5% of the total number of units in the circuit. Since the distortion of the program code of such LUT units does not lead to distortion of the results, these units constitute a resource for steganographic data embedding.

An LUT with m inputs is programmed by 2^m bit program code. Based on this, each of the inessential LUTs can provide steganographic storage of up to 2^m bits of additional data. The Intel FPGA Cyclone IV EP4CE15F23A7 chip, for which the synthesis was carried out, contains LUT units with the number of inputs $m = 4$. The program code of each LUT of this chip has a size of 16 bits. In the experimental FPGA projects, the number of inessential LUTs ranged from 12 to 145. Thus, for these cases, the volume of the stego container provided by the resource of inessential LUTs was from 192 to 2320 bits. These volumes are sufficient to store hash-sums obtained using commercially used hash functions. With an increase in the size of the multiplier operands, the volume of the stego container also becomes sufficient to store additional monitoring service information.

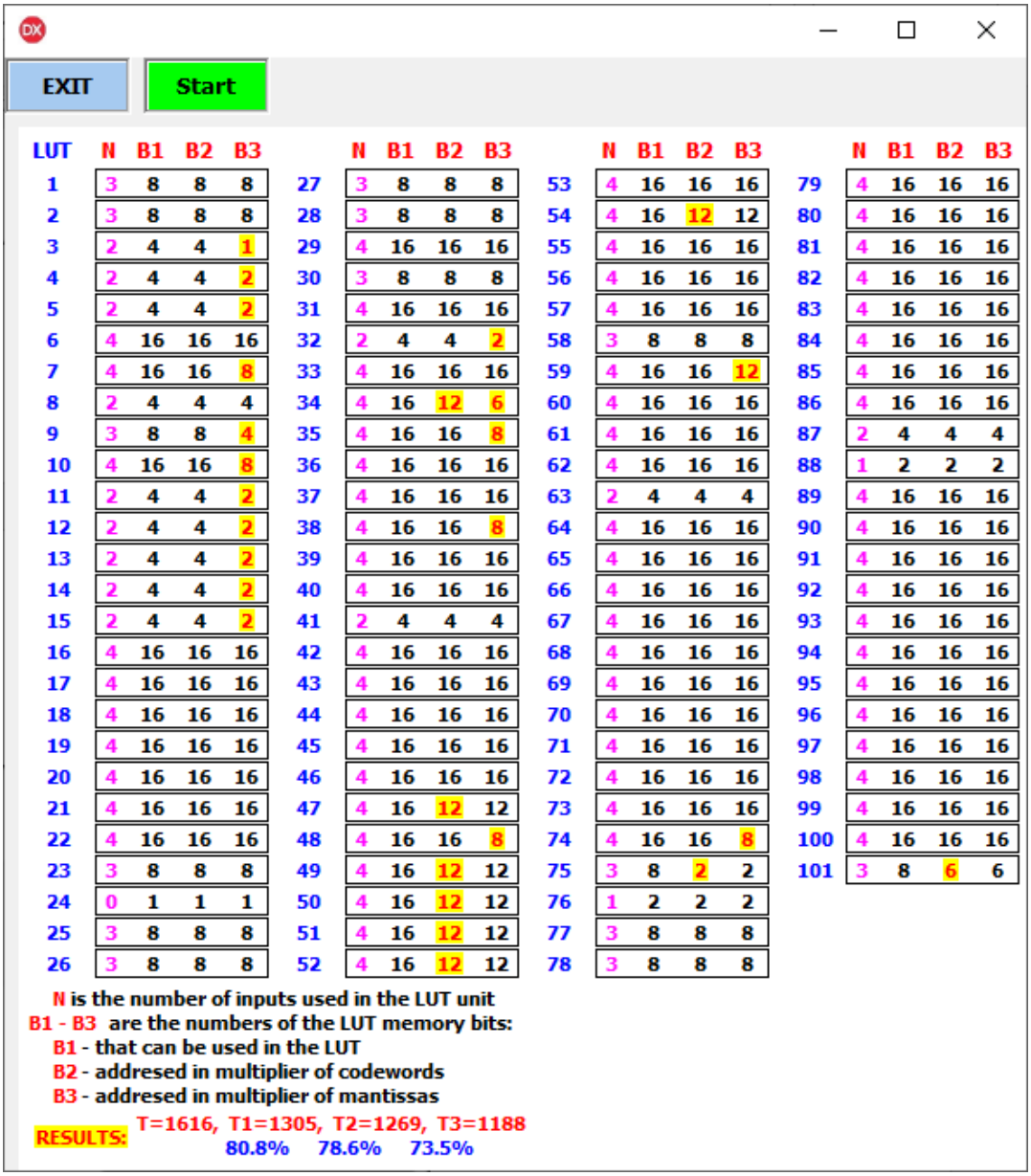

Figure 2: Window of the results for the estimation in the number of inessential bits for an 8-bit multiplier

Table 1
Experiment Results

Size of operands and result		4	6	8	10	12
Size of the result before rounding		8	12	16	20	24
Total memory volume of the LUT units		480	976	1616	2384	3328
LUT units memory volume used		396 (82.5%)	747 (76.6%)	1305 (80.8%)	1991 (83.5%)	2811 (84.5%)
Essential bits	Binary code multiplication	391 (81.5%)	719 (73.7%)	1269 (78.6%)	1983 (83.2%)	2752 (82.7%)
	Normalized mantissa multiplication	287 (59.8%)	658 (67.4%)	1168 (73.5%)	1882 (79.0%)	2638 (79.3%)
Inessential bits	Binary code multiplication	89 (18.5%)	257 (26.3%)	347 (21.4%)	401 (16.8%)	576 (17.3%)
	Normalized mantissa multiplication	193 (40.2%)	318 (32.6%)	448 (26.5%)	502 (21.0%)	690 (20.7%)

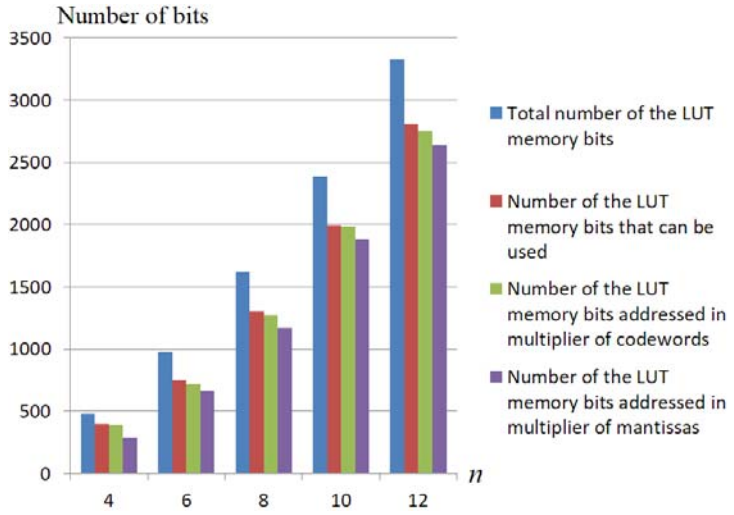

Figure 3: Number of addressable container bits

Also the results of the experiments showed the presence in the studied FPGA-systems of the second resource for non-equivalent steganographic data embedding. This resource is the inessential bits of the program code of the LUTs. These bits are not addressed during the calculation of the result. Therefore, the distortion of such bits of the program code does not lead to distortion of the calculation result. Based on this, these bits form a resource of the second kind for non-equivalent steganographic data embedding into the FPGA program code.

For experimental FPGA systems, the number of inessential bits ranged from 89 to 576. The portion of inessential bits in this case ranges from 16.8% to 26.3% of the total number of bits in program codes of LUTs. When multiplying the normalized mantissa, the number of inessential bits of the program code ranged from 193 to 690. In this case, the portion of inessential bits from the total number of bits of the program codes is from 21% to 40.2%. The resulting volumes are sufficient for storing checksums. With an increase in the size of the operands, these volumes of the stego container are sufficient to store the additional monitoring service data as well.

Thus, the experiment confirmed the presence of two kinds of FPGA program code resources that can be used for non-equivalent steganographic embedding of additional data. The experiment also made it possible to estimate the volume of stego containers, which are based on the use of these resources. It is shown that these volumes are sufficient for storing monitoring data.

5. Conclusions and Potential Directions of the Further Research

An approach to steganographic data embedding into the program code of FPGA chips is proposed. It differs from existing approaches in that it uses non-equivalent transformations of the FPGA program code. It is shown that software code of FPGA systems that perform approximate data processing, there are new resources. These resources can be used for non-equivalent steganographic embedding. The first of these resources are inessential LUTs. These units are involved in the calculation of the least significant (discarded during rounding) bits of the result and do not participate in the calculation of the most significant (retained) bits. Distortion of the program code of inessential LUT units does not lead to distortion of the calculation result. The second resource is the inessential bits of the LUTs code. These are the bits that are not addressed when calculating the most significant bits of the operation result. Distorting these bits does not distort the result. It is shown that these two kinds of resources make it possible to non-equivalently embed additional data into the FPGA program code. This does not change the operation of the device and the size of the program code.

The experiment performed made it possible to confirm the presence of these two kinds of resources. Also, the experiment made it possible to estimate its volume of stego containers resulting from the use of these resources. The volume of the stego container provided by the resource of inessential LUTs was from 192 to 2320 bits. The portion of inessential bits in this case ranges from 16.8% to 26.3% of the total number of bits in program codes of LUTs. When multiplying the

normalized mantissa, the number of inessential bits of the program code ranged from 193 to 690. In this case, the portion of inessential bits from the total number of bits of the program codes is from 21% to 40.2%. These volumes are sufficient to store hash-sums obtained using commercially used hash functions.

The proposed approach can be used both independently and in conjunction with an equivalent approach to embedding additional data into the FPGA program code. Given the prevalence of approximate data processing among the functions of modern computer systems, embedding based on the features of operations with such data is very promising.

The direction for further research is to determine how to jointly use the equivalent and non-equivalent approaches to steganographic data embedding. Of interest is the study of a hybrid stego system in which both approaches play the same or different roles. For example, the roles of embedding, code obfuscation, or false embedding aimed at countering stego analysis.

6. References

[1] W. Stallings, Cryptography and Network Security: Principles and Practice, 7th ed., Pearson Education Limited, United Kingdom, Harlow, 2017.

[2] A. Awad, M. Fairhurst, Information Security. Foundations, technologies and applications, The Institution of Engineering and Technology, UK, London, 2018.

[3] J. Katz, Introduction to Modern Cryptography, 2nd Edition, CRC Press, Boca Raton, 2018.

[4] M. Miyanaga, H. Irie and S. Sakai, "Accelerating Integrity Verification on Secure Processors by Promissory Hash," 2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC), Christchurch, 2017, pp. 20-29, doi: 10.1109/PRDC.2017.13.

[5] M. Bishop, Computer Security, 2nd edn., Addison-Wesley, USA, Boston, 2018.

[6] Y. Yang, F. Chen, X. Zhang, J. Yu, P. Zhang, Research on the Hash Function Structures and its Application, in Proceedings of International Conference Wireless Personal Communications, 2016.

[7] J. Vacca, Computer and information security handbook, 3rd ed., Morgan Kaufmann, Waltham, Mass, 2017.

[8] X. Wu, Z. Yang, C. Ling, X. Xia, Artificial-Noise-Aided Message Authentication Codes With Information-Theoretic Security, IEEE Transactions on Information Forensics and Security, vol. 11, no. 6 (2016) 1278-1290. doi: 10.1109/TIFS.2016.2524514.

[9] S. Hirose, J. Shikata, Aggregate Message Authentication Code Capable of Non-Adaptive Group-Testing, IEEE Access, vol. 8 (2020) 216116-216126. doi: 10.1109/ACCESS.2020.3041638.

[10] T. Hovorushchenko, O. Pomorova. Evaluation of Mutual Influences of Software Quality Characteristics Based ISO 25010:2011, in: Proceedings of 2016 IEEE International Scientific and Technical Conference “Computer Science and Information Technologies, CSIT-2016, Lviv, Ukraine, 2016, pp. 80-83.

[11] M. Zaman, T. Shen and M. Min, Hash Vine: A New Hash Structure for Scalable Generation of Hierarchical Hash Codes, 2019 IEEE International Systems Conference, Orlando, FL, USA, 2019, pp. 1-6. doi: 10.1109/SYSCON.2019.8836921.

[12] T. Hovorushchenko, O. Pavlova. Evaluating the Software Requirements Specifications Using Ontology-Based Intelligent Agent, in: Proceedings of 2018 IEEE International Scientific and Technical Conference “Computer Science and Information Technologies CSIT-2018, Lviv, Ukraine, 2018, pp.215-218.

[13] S. Wang, C. Li, H. Shen, Distributed Discrete Hashing by Equivalent Continuous Formulation, IEEE Transactions on Signal and Information Processing over Networks, vol. 6 (2020) 196-210. doi: 10.1109/TSIPN.2020.2975356.

[14] S. Lysenko, O. Savenko, K. Bobrovnikova, A. Kryshchuk. Self-adaptive system for the corporate area network resilience in the presence of botnet cyberattacks, Communications in Computer and Information Science, 860 (2018) 385-401.

[15] F. Shih, Digital Watermarking and Steganography: Fundamentals and Techniques. 2nd ed., CRC Press, USA, Boca Raton. 2017.

[16] K. Zashcholkin, O. Drozd, R. Shaporin, O. Ivanova, Y. Sulima, Increasing the effective volume of digital watermark used in monitoring the program code integrity of FPGA-based systems, in Proceedings of 2019 IEEE East-West Design and Test Symposium, EWDTS 2019.
[17] Z. Zhang, L. Njilla, C. A. Kamhoua, Q. Yu, Thwarting Security Threats From Malicious FPGA Tools With Novel FPGA-Oriented Moving Target Defense, IEEE Transactions on Very Large Scale Integration Systems, vol. 27, no. 3 (2019) 665-678. doi: 10.1109/TVLSI.2018.2879878.
[18] H. Amano, Principles and Structures of FPGAs, Springer, 2018.
[19] J. Andina, FPGAs: Fundamentals, Advanced Features, and Applications in Industrial Electronics, CRC Press, USA, Boca Raton, 2017.
[20] A. Melnyk, A. Salo, Automatic generation of ASICs, in: Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems, Edinburgh, UK, 2007, pp. 311-317, doi: 10.1109/AHS.2007.36.
[21] O. Drozd, K. Zashcholkin, R. Shaporin, J. Drozd, Y. Sulima, Development of ICT Models in Area of Safety Education, in: Proceedings of the 18th IEEE East-West Design & Test Symposium, Varna, Bulgaria, 2020, pp. 115–119. doi:10.1109/EWDTS50664.2020.9224861.
[22] O. Drozd, K. Zashcholkin, O. Martynyuk, O. Ivanova, J. Drozd, Development of Checkability in FPGA Components of Safety-Related Systems, CEUR Workshop Proceedings, vol. 2762 (2020) 30-42.
[23] O. Y. Abdulhammed, Improving Encryption Digital Watermark by Using Blue Monkey Algorithm. International Journal of Computing, vol. 20, no. 1 (2021) 129-136. doi: 10.47839/ijc.20.1.2101
[24] Y. Ke, J. Liu, M. Zhang, T. Su, X. Yang, Steganography Security: Principle and Practice, IEEE Access, vol. 6 (2018) 73009-73022. doi: 10.1109/ACCESS.2018.2881680.
[25] T. Korkishko, A. Melnyk, Cryptographic processor architectures for DES algorithm, in: Proceedings of the 5th IEEE Africon Conference (Cat. No.99CH36342), Cape Town, South Africa, 1999, vol. 1, pp. 175-180. doi: 10.1109/AFRCON.1999.820788.
[26] K. Zashcholkin, O. Drozd, O. Ivanova, P. Bykovyy, Formation of the interval stego key for the digital watermark used in integrity monitoring of FPGA-based systems. CEUR Workshop Proceedings (CEUR-WS) 2623 (2020), 267–276.
[27] A. Drozd, J. Drozd, S. Antoshchuk et. al., Objects and Methods of On-Line Testing: Main Requirements and Perspectives of Development, in: Proceedings of the IEEE East-West Design & Test Symposium, Yerevan, Armenia, 2016, pp. 72–76. doi:10.1109/EWDTS.2016.7807750.
[28] J. Drozd, A. Drozd, M. Al-dhabi, A resource approach to on-line testing of computing circuits, in: Proceedings of the IEEE EWDT Symposium, Batumi, Georgia, 2015, pp. 276–281. doi:10.1109/EWDTS.2015.7493122.
[29] T. Chen, K. Hou, W. Beh and A. Wu, Low-Complexity Compressed-Sensing-Based Watermark Cryptosystem and Circuits Implementation for Wireless Sensor Networks, IEEE Transactions on Very Large Scale Integration Systems, vol. 27, no. 11 (2019) 2485-2497. doi: 10.1109/TVLSI.2019.2933722.
[30] IEEE Standard for Floating-Point Arithmetic, in IEEE Std 754-2019 (Revision of IEEE 754-2008), pp.1-84, 22 July 2019. doi: 10.1109/IEEESTD.2019.8766229.
[31] A. Anderson, S. Muralidharan, D. Gregg, Efficient Multibyte Floating Point Data Formats Using Vectorization, IEEE Transactions on Computers, vol. 66, no. 12 (2017) 2081-2096. doi: 10.1109/TC.2017.2716355.
[32] O. Drozd, I. Perebeinos, O. Martynyuk et. al., Hidden fault analysis of FPGA projects for critical applications, in: Proceedings of the IEEE International Conference TCSET, paper 142, Lviv–Slavsko, Ukraine, 2020. doi:10.1109/TCSET49122.2020.235591.
[33] O. Pomorova, O. Savenko, S. Lysenko, A. Kryshchuk, K. Bobrovnikova, Anti-evasion Technique for the Botnets Detection Based on the Passive DNS Monitoring and Active DNS Probing, Communications in Computer and Information Science, 608 (2016) 83-95.
[34] Intel Quartus Prime Standard Edition User Guide, 2020. URL: https://www.intel.com/content/dam /alterawww/global/en_US/pdfs/literature/ug/ug-qps-getting-started.pdf.
[35] Cyclone IV Device Handbook. URL: https://www.intel.com/content/dam/www/programmable/us /en/pdfs/literature/hb/cyclone-iv/cyclone4-handbook.pdf.
[36] Delphi 10 Seattle: Embarcadero. URL: https://www.embarcadero.com/docs/datasheet.pdf.