Enhanced Instruction Register Files for Embedded Software Obfuscation

Naoki Fujieda and Shuichi Ichikawa
Department of Electrical and Electronic Information Engineering
Toyohashi University of Technology
Toyohashi, AICHI, 441-8580, JAPAN
`fujieda@ee.tut.ac.jp, ichikawa@tut.jp`

Abstract

Computer software might be analyzed, plagiarized, and falsified unless obfuscation or encryption is applied. One of cost-effective obfuscation techniques is instruction set randomization, which modifies or enhances instruction coding of processors. Using an Instruction Register File (IRF) is one of the attractive candidates of such techniques, while it lacks quantitative evaluation about tamper resistance. In this paper, we first quantify the effectiveness of the additional coding that disguises instruction sequences. We then propose heuristic algorithms that search for sub-optimal assignments of the IRF. Our evaluation shows that the IRF that allocates a single instruction to multiple entries greatly improves the tamper resistance. Lastly, we present that the hardware overhead of an embedded processor is acceptable with a large IRF implemented on an FPGA.

1 Introduction

In recent years, the protection of software from analysis, plagiarism, and falsification has been an important issue. This property is often called tamper resistance. In particular, leakage of trade secrets is a serious problem. Though software can be encrypted, or at least obfuscated, for protection against reverse engineering, additional cost brings up another important issue. For embedded systems, anti-tamper methods should be realized with small overhead.

Instruction set randomization [1][2][3][4] is one of cost-effective obfuscation approaches. This technique protects processors from analysis or plagiarism by giving them different or additional instruction coding system that is hidden from attackers. Moreover, diversified instruction sets are naturally resistant to falsification because a malicious instruction sequence for one processor will not operate correctly on the others.

The use of an Instruction Register File (IRF) [5] is one of the attractive candidates of instruction set randomization. The IRF is a small memory that stores the most common expressions of instructions specified by the compiler. It is referred by an index written in fetched instructions. Though it initially aimed at reducing power consumption by packing multiple instructions into a single one [5], it also has resistance to tampering [6], for reference to the IRF is considered as an additional expression of instructions.

The problem is that there is no quantitative evaluation about tamper resistance of the IRF. The contents of the IRF may be guessed from the occurrence frequency of indices, or a routine to be hidden may not include enough number of references of the IRF. The possibility of such risks is not evaluated in [6], although the *side effects* of the IRF, the code reduction and the execution time, are measured.

In this paper, we study the effectiveness of the IRF against tampering, particularly reverse engineering. We first quantify the efficiency of instruction hiding, and then propose heuristic algorithms to store proper instructions in the IRF. Lastly, we show that a large IRF can be implemented with a small hardware cost on an FPGA.

2 Background

2.1 Instruction Set Randomization

Encryption of instruction memory is one of the most typical anti-tamper approaches for processors, which was used in the Execute Only Memory (XOM) [7] and AEGIS architecture [8]. Most of the methods adopt modern ciphers such as AES to decrypt instruction memory that was encrypted at compile time. Even data memory is often encrypted and decrypted on the fly. This approach is useful for applications where security is the most important concern. Nevertheless, it significantly increases the memory access latency and the hardware amount, and thus it is unsuitable for cost-sensitive embedded systems.

Instruction set randomization (ISR) can be considered as lightweight memory encryption. Compared to robust but costly modern ciphers, it is based on simple substitution ciphers for lower overhead. Both hardware [1][4] and software [2][3] implementations have been studied. Some methods utilize the characteristics of the target instruction set [4].

An important measure of ISR is the hardness to guess the original instructions from the *randomized* ones with frequency analysis [9]. Some kinds of instructions might be easily guessed from statistical properties of the obfuscated binary (e.g. the frequencies of opcode values). The analysis might be even easier if heuristics are applied. For example, the reserved fields of specific instructions are always set to zero. To prevent frequency analysis, it is important for ISR to leave less statistical information of the original instructions.

2.2 Instruction Register File

Instruction Register File (IRF) [5] is a table of frequently used instructions. The IRF is placed between instruction fetch and instruction decode stages and accessed by an index. If a specialized instruction is fetched, the corresponding instruction(s) to the index (or indices) will be read from the IRF and sent to the decoder.

The IRF has originally been proposed to compress instruction sequences. Since the bit length of an index of the IRF is much shorter than that of an original instruction, multiple IRF instructions can be extracted from a specialized instruction. In [5], the target instruction set, MIPS, was modified so that some instructions can include an index (or indices). They can be classified as loosely or tightly packed, as depicted in Figure 1. Gray fields in Figure 1 are used to obtain instructions from the IRF. They are 5 bits long for the original IRF with 32 entries. A loosely packed instruction (the middle row in Figure 1) contains a regular MIPS instruction and an IRF instruction, while a tightly packed instruction (the bottom row) stores up to five IRF instructions. Although the original IRF proposal [5] includes a parameterization technique to improve the efficiency of tightly packed instructions, we do not consider it in this paper.

2.3 Using IRF for Anti-tampering

In addition to the reduction of code length, the IRF has an effect of ISR that is applicable to a part of instructions [6]. It is possible to protect software from analysis by arbitrarily shuffling the mapping from indices to IRF instructions. The IRF also provides a protection from plagiarism if we diversify the mapping and the corresponding

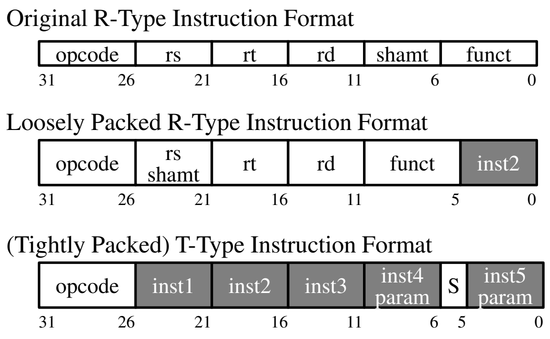

Figure 1: Additional instruction formats to reduce the code size with the IRF [5].

instruction sequence for each system. Similarly, if we prohibit IRF instructions from being executed directly from instruction memory, it becomes robust over falsification. In comparison with other ISR methods, the IRF has an advantage of hiding information about operands.

For all these merits, the IRF is not practical without modification mainly because of the following two shortcomings. First, it might be too small to obfuscate the instruction sequences that developers want to hide. Although it may be a solution to have different mappings for each process or routine, it may cause another problem of keeping the mappings themselves securely. Second, it lacks a consideration for frequency analysis. If the goal of the IRF is tamper resistance, it might be a bad idea to just put the most frequent instructions in it. Unfortunately, the evaluation of these risks is not presented in [6].

A possible approach to remedy the risks is to increase the number of the entries in the IRF and to use a mapping common to all processes. The number of instructions covered by the IRF will be large enough to obfuscate important routines. The difficulty of frequency analysis will be greatly increased because lower-ranking instructions have almost the same occurrence probability. It may be difficult even to find which instructions reside in the IRF.

On utilizing a large IRF, we have to consider the following concerns: how to evaluate the selection of IRF instructions, how to select IRF instructions, and how large the hardware or performance overhead is. We propose and evaluate a solution in the following sections.

3 Selection of IRF Instructions

3.1 Scale of Tamper Resistance for IRF

Each instruction has different occurrence frequency. The distribution of occurrence probability in object codes is plotted in Figure 2. 32-bit MIPS instruction expressions

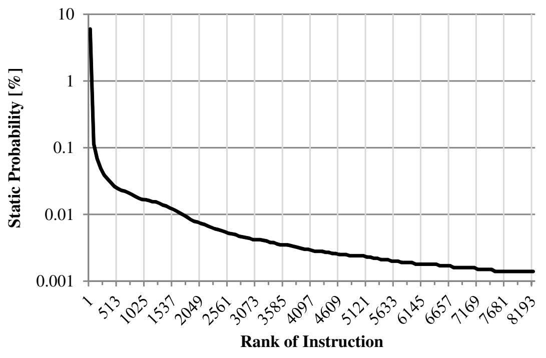

Figure 2: Occurrence Probability Distribution of MIPS Instructions.

are arranged in descending order, from left to right. The Y-axis is occurrence probability of each instruction (in logarithmic scale). The measurement condition is the same as in Section 4. The difference of the probabilities between two adjacent instructions decreases as the rank goes lower.

The following two factors should be considered in the selection of instructions: (1) the coverage of IRF instructions and (2) the cost of frequency analysis. We introduce some symbols to quantify them. We first gather an instruction profile by dynamic profiling of a typical application. Each 32-bit instruction expression, I, has the number of times dynamically executed and the number of times statically appeared. We define them as $C_D(I)$ and $C_S(I)$, respectively. They naturally differ since each instruction might be executed multiple times. When it is executed n times, C_D is incremented by n, while C_S is just incremented by 1. From them, dynamic execution frequency $P_D(I)$ and static occurrence probability $P_S(I)$ are calculated as follows:

$$P_D(I) = \frac{C_D(I)}{\sum_{i\in\mathbb{I}} C_D(i)},\ \ P_S(I) = \frac{C_S(I)}{\sum_{i\in\mathbb{I}} C_S(i)}$$

where $\mathbb{I}$ is the set of all the possible expressions of MIPS instructions.

The coverage of IRF instructions is defined as $\gamma(\text{IRF})$ with the following formula:

$$\gamma(\text{IRF}) = \textstyle\sum_{i=0}^{N-1} P_D(\text{IRF}_i)$$

where IRF_i is the IRF instruction of the index i and N is the number of entries in the IRF. The coverage becomes 1 when all of the executed instructions are included in the IRF, while it becomes 0 when no IRF instructions are executed. Most of the programs have a small sequence of instructions that are repeatedly executed. We formulate the coverage with dynamic frequency P_D because such a part is more suitable for being hidden than others.

The cost of frequency analysis is quantified with $E(\text{IRF})$ or the Shannon entropy H of the IRF index. It is formulated as follows:

$$E(\text{IRF}) = H(I) = \frac{-\sum_{i=0}^{N-1} P_S(\text{IRF}_i)\log P_S(\text{IRF}_i)}{\log N}$$

$E(\text{IRF})$ is normalized by its theoretical limit, or $\log N$. The cost becomes 1 when all of the IRF indices appear completely evenly, while it becomes 0 when only one index appears. Since frequency analysis is often made statically with program codes, we use P_S to measure the evenness of the distribution here.

To maximize the coverage, the same strategy is adopted as in the original IRF proposal [5]. It excels in the chance of instructions being obfuscated; however, the deviation of the frequency distribution is large (see the left-most part of the graph in Figure 2) and frequency analysis is relatively easy.

On the other hand, maximizing the cost leads to choosing lower-rank instructions. It makes the analysis more difficult because of smaller deviation of the frequency distribution, but most of the instructions will be left unmodified.

Based on the observation above, we define the scale of effectiveness of instruction selection, denoted by $S(\text{IRF})$, as the product of the coverage and the cost:

$$S(\text{IRF}) = \gamma(\text{IRF}) \times E(\text{IRF})$$

Even though all the instructions reside in the IRF, if the distribution is extremely skewed to a few entries, the original instruction sequence will easily be guessed by frequency analysis. Similarly, the IRF does not make any sense when the included instructions are seldom used even though the frequency distribution is completely even, In these cases, the scale should be 0 rather than (the arithmetic average of) 0.5.

3.2 Algorithm

Finding the optimal selection of IRF instructions is now formulated as a combinatorial optimization problem to maximize $S(\text{IRF})$ defined in the previous subsection. We use a heuristic algorithm based on local search to find a sub-optimal solution rather than the exact one, which is difficult to compute in a practical time.

The proposed algorithm is shown in Figure 3. Any combination of N instructions without duplication is allowed as an initial IRF instructions, though computation cost will be smaller by selecting instructions that have the highest $P_D(I)+P_S(I)$. The lists of candidates for addition and removal are called $List\_add$ and $List\_rem$, respectively. $List\_add$ can be limited to about $N/2$ instructions that have the highest $P_D(I)+P_S(I)$ next to the initial IRF instructions; the resulting combination does not change in most cases.

```
 1: $IRF \leftarrow$ any $N$ instructions
 2: $List\_add \leftarrow$ instructions except $IRF$
 3: $List\_rem \leftarrow IRF$
 4: loop
 5:     $scale\_cur \leftarrow S(IRF)$
 6:     find $c\_add$ that maximizes $S(IRF + c\_add)$ from $List\_add$
 7:     $scale\_inc \leftarrow S(IRF + c\_add)$ - $scale\_cur$
 8:     find $c\_rem$ that maximizes $S(IRF - c\_rem)$ from $List\_rem$
 9:     $scale\_dec \leftarrow scale\_cur - S(IRF + c\_rem)$
10:     if $scale\_inc > scale\_dec$ then
11:         $IRF \leftarrow IRF + c\_add - c\_rem$
12:         $List\_add \leftarrow List\_add - c\_add$
13:         $List\_rem \leftarrow List\_rem - c\_rem$
14:     else
15:         break
16:     end if
17: end loop
```

Figure 3: Pseudocode for selecting IRF entries.

After the initialization, the algorithm repeatedly finds a candidate for addition c_add from $List\_add$ and a candidate for removal c_rem from $List\_rem$. If the increase of the scale by addition of c_add is more than the decrease by removal of c_rem, they are swapped and then removed from the lists; otherwise, the current combination of IRF is output as a near-optimal solution. The time complexity of the algorithm is $O(N^3)$, for a search for the candidate takes $O(N^2)$ and the number of repetition is up to N.

3.3 Duplicated IRF Entries

Although the problem of deviation is partially solved by the above algorithm, it still remains with higher-rank, frequently used instructions because their removal causes serious decrease in the coverage. Here we propose a modified algorithm that assigns multiple entries to the frequently used instructions. By allowing duplication, some lower-rank instructions are removed from the IRF; however, such instructions have little impact on the coverage. In this paper, we assume that we can equally use indices of entries to which a duplicated instruction is assigned; that is, when an instruction I is associated with k entries, the static probability of each entry is calculated as $P_S(I)/k$.

The modified algorithm is shown in Figure 4. Now the initial IRF instructions are also included in the candidates for addition. When we assume n as the number of entries to be added, the calculation of the expected increase of the scale by addition of an instruction is changed to the average increase of the scale by adding n entries per n. We find a pair of an instruction c_add and the number of additional entries n_add that maximizes the expected increase. If multiple entries are to be added, or n_add is two or more, n_add items that have the minimum decrease of the scale are chosen from $List\_rem$ as candidates for removal. The removal of an instruction from $List\_rem$ occurs not only when it is removed from the IRF but also when it is duplicated in the IRF. The time complexity of the modified

```
 1: $IRF \leftarrow$ any $N$ instructions
 2: $List\_rem \leftarrow IRF$
 3: loop
 4:     $scale\_cur \leftarrow S(IRF)$
 5:     find $c\_add$ and $n\_add$ that maximize
        $S(IRF + c\_add * n\_add)/n\_add$ from all instructions
 6:     $scale\_inc \leftarrow S(IRF + c\_add * n\_add)$ - $scale\_cur$
 7:     $C\_rem \leftarrow$ an empty array
 8:     $scale\_dec \leftarrow 0$
 9:     for $i$ in [1..$n\_add$] do
10:         find $c$ that maximize $S(IRF - c)$
            from $List\_rem - c\_add - C\_rem$
11:         $C\_rem \leftarrow C\_rem + c$
12:         $scale\_dec \leftarrow scale\_dec + \{scale\_cur - S(IRF - c)\}$
13:     end for
14:     if $scale\_inc > scale\_dec$ then
15:         $IRF \leftarrow IRF + c\_add * n\_add - C\_rem$
16:         $List\_rem \leftarrow List\_rem - c\_add - C\_rem$
17:     else
18:         break
19:     end if
20: end loop
```

Figure 4: Pseudocode for selecting IRF entries with duplication.

algorithm is still $O(N^3)$, while actual computation time becomes much longer than the original because of the search for n_add and the increase of the number of repetition. We may reduce the computation cost of finding n_add from the fact that the proper n is 1 for most instructions.

4 Evaluation

4.1 Methodology

We use 36 traces of MiBench [10], taken with a modified version of SimMips version 0.7.5 [11]. These benchmarks are compiled with gcc 4.7.3, uClibc 0.9.33.2, and binutils 2.21. The instruction set is MIPS32 Release 1, assuming hardware floating point unit. The compile options are same as the defaults of MiBench except asserting a flag for static compilation (`-static`). Some files are slightly modified to avoid compile errors probably due to the difference of compiler versions.

We evaluate the scale of instruction selection for a tamper-aware IRF with three algorithms as follows:

- **max_dyn** aims to maximize the coverage, or the sum of dynamic execution frequency in the IRF, which is corresponding to the original proposal [5].
- **no_dup** is the proposed algorithm without duplication, which we have described in Section 3.2.
- **dup** is the modified algorithm allowing duplication, which has been proposed in Section 3.3.

We vary the number of IRF entries, N, exponentially from 32 to 2048.

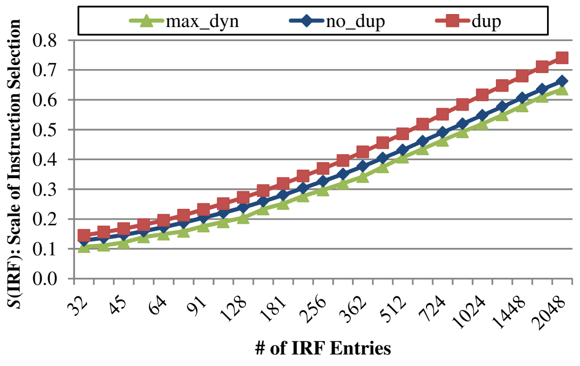

Figure 5: The number of IRF entries vs. scale of instruction selection.

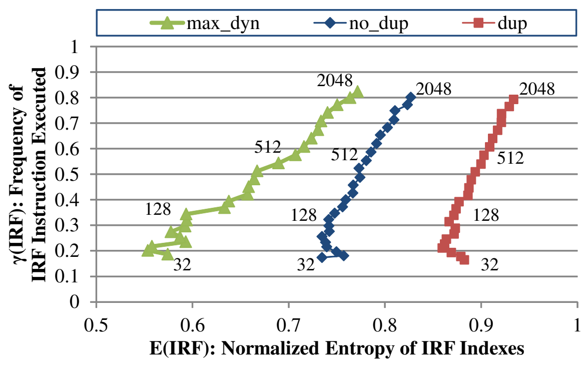

Figure 6: The difference with our methods in the breakdown of the scale.

4.2 Tamper Resistance

Figure 5 depicts the relationship between the number of IRF entries and the scale of instruction selection, which correspond to X- and Y-axes, respectively. Due to the consideration of the entropy of the frequency distribution, no_dup achieved 4% to 22% increase of the scale over max_dyn. The increase rate was higher when the number of entries was smaller. We think it is because the room for selection is large in such cases. On the other hand, dup showed 11% to 14% higher scale, with smaller influence of the number of entries, than no_dup. It worked effectively regardless of the size of the IRF.

To discuss the effect of our proposal further, we show the breakdown of the scale, or the coverage of IRF instructions and the cost of frequency analysis, in Figure 6. The X-axis is the relative entropy of the static distribution of IRF indices (or the cost of analysis). The Y-axis is the sum of dynamic execution frequency (or the coverage). Numbers near the points of the graph represent the number of entries (only the cases of 32, 128, 512, and 2048 entries are shown). The cost of frequency analysis was greatly improved by both of the algorithms. The improvement of no_dup over max_dyn is 7% to 36%; the further improvement of dup over no_dup is 12% to 20%. However, the decrease of the coverage of no_dup and dup over max_dyn was very small: it was less than 10% and 12% in no_dup and dup, respectively. Judging from the figures, our algorithms successfully enhance the protection for frequency analysis with small overhead of the coverage of the IRF.

4.3 Hardware Overhead

We implemented a processor with a large IRF on a Xilinx FPGA. We need a reasonably large number of entries to obfuscate an important routine that developers want to be hidden. From Figure 6, a 1024-entry IRF supplies about two-thirds of instructions in the execution. Fortunately, we can use block RAMs (BRAMs) in an FPGA to effectively implement an IRF of this size.

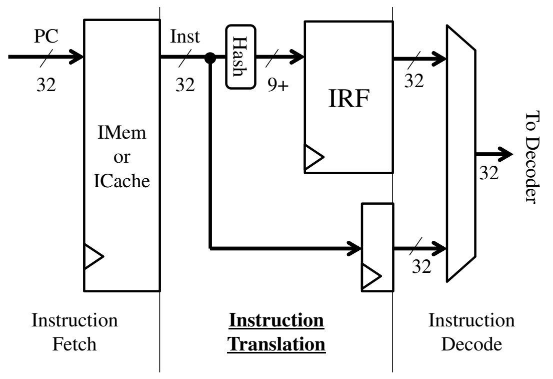

Figure 7: An additional “Instruction Translation” stage between instruction fetch and decode.

Figure 7 shows the diagram of partial pipeline of a processor with a BRAM-implemented IRF. We do not consider instruction packing: normal MIPS instructions are transformed to specialized instructions (which refer to the IRF) on a one-to-one basis. An additional pipeline stage for translation is needed because BRAMs are clock synchronous. Either the outcome of the IRF or the fetched instruction is passed to the instruction decoder. This additional “Instruction Translation” stage may slightly increase the branch miss penalty.

We chose Plasma [12] as the target processor. Plasma has a 4-stage pipeline, two stages of which are consumed in instruction fetch. We found that “Instruction Translation” stage could be merged into the latter of the fetch stages. Therefore we were able to apply an IRF to Plasma without large modification of the pipeline.

We confirmed that the modified Plasma worked correctly on the Xilinx Spartan-3E Starter Board. We then evaluated the addition of hardware amount and the loss of the maximum operating frequency. The circuit was implemented with Xilinx ISE 14.6, optimized for speed. When we added a 1024-entry IRF, the additional number of slices was 73 (3.7% of Plasma) and that of BRAMs was 2. The loss of the maximum frequency was about 11%, which is due to longer critical path of instruction decode. Optimization of implementation is left as a future work.

5 Discussion

This section discusses the two following topics related to our proposal: (1) the use of pairs or triplets of instructions and (2) handling of *nop*.

In this paper, we defined the cost of frequency analysis under the assumption that it was solely based on the frequency distribution of a single instruction. Actually, there is a certain correlation between the adjacent instructions. Thus our proposal may still be vulnerable to frequency analysis using the distribution of pairs or triplets of instructions. Some software-based approaches exist to remedy such deviations. For example, instructions can be swapped as long as they are in the same basic block and the dependency is not broken [13].

In MIPS processors, *nop* (no operation) is often the most frequently executed instruction. It is true that *nop* is never related to algorithms; however, it is required in some cases due to delayed branch, dependency of load instructions, and so on. This characteristic of MIPS might not appear in other instruction sets (e.g. ARM). Investigation of the relationship between the instruction set and the frequency distribution is left for future studies.

6 Conclusion

This work presented a study on the tamper-aware use of the IRF. The key points of our proposal are twofold: (1) we should consider the entropy of the distribution of IRF index occurrence, and (2) some of the most frequent instructions should be assigned to multiple IRF entries. According to our evaluation, the protection for frequency analysis was much enhanced with small overhead of the coverage of the IRF.

The items for future studies are summarized as follows. First of all, the algorithm should be improved for higher precision or shorter computation time. Further evaluation and optimization of hardware implementation are desired for practical applications. It is also important to verify the effectiveness of the algorithm with other instruction sets.

References

[1] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proc. of CCS '03*, pp. 272–280, 2003.

[2] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović, "Randomized instruction set emulation," *ACM Trans. on Information and System Security*, vol. 8, no. 1, pp. 3–40, 2005.

[3] G. Portokalidis and A. D. Keromytis, "Fast and practical instruction-set randomization for commodity systems," in *Proc. of ACSAC '10*, pp. 41–48, 2010.

[4] S. Ichikawa, T. Sawada, and H. Hata, "Diversification of processors based on redundancy in instruction set," *IEICE Trans. on Fundamentals*, vol. E91-A, no. 1, pp. 211–220, 2008.

[5] S. Hines, J. Green, G. Tyson, and D. Whalley, "Improving program efficiency by packing instructions into registers," in *Proc. of ISCA-32*, pp. 260–271, 2005.

[6] D. Chang, S. Hines, P. West, G. Tyson, and D. Whalley, "Program differentiation," in *Proc. of INTERACT-14*, no. 9, 2010.

[7] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proc. of ASPLOS-IX*, pp. 168–177, 2000.

[8] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: architecture for tamper-evident and tamper-resistant processing," in *Proc. of ICS '03*, pp. 160–171, 2003.

[9] F. L. Bauer, *Decrypted Secrets: Methods and Maxims of Cryptology*, 4th ed. Springer, 2006.

[10] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. of WWC-4*, pp. 3–14, 2001.

[11] N. Fujieda, S. Watanabe, and K. Kise, "A MIPS system simulator SimMips for education and research of computer science," *IPSJ J.*, vol. 50, no. 11, pp. 2665–2676, 2009.

[12] S. Rhoads, "Plasma – most MIPS I(TM) opcodes." [Online]. Available: http://opencores.org/project,plasma

[13] K. Hattanda and S. Ichikawa, "Redundancy in instruction sequences of computer programs," *IEICE Trans. on Fundamentals*, vol. E89-A, no. 1, pp. 219–221, 2006.