Design and Implementation of Instruction Indirection for Embedded Software Obfuscation

Naoki Fujieda[a,*], Tasuku Tanaka[a], Shuichi Ichikawa[a]

[a]*Department of Electrical and Electronic Information Engineering, Toyohashi University of Technology, 1-1 Hibarigaoka Tempakucho, Toyohashi, Aichi, 441-8580, Japan*

Abstract

Instruction Register File (IRF) was originally proposed to reduce the power consumption of a microprocessor by providing the indirect access to frequently executed instructions. The IRF is also an attractive and cost-effective unit to protect embedded software from analysis, plagiarism, and falsification. For this purpose, the correspondences between IRF entries and their original instructions must be concealed. This means the instructions in the IRF should be carefully selected both to have more instructions be executed through the IRF and to flatten the distribution of the indices of the IRF.

This paper presents two heuristic algorithms, precision-oriented and time-oriented, to find sub-optimal assignments to the IRF. According to the evaluation results, the precision-oriented algorithm obtained the same as or very close to the optimal assignment of an IRF with 48 or less entries. The time-oriented algorithm found a sub-optimal assignment of a 1024-entry IRF in 16 milliseconds, whose precision was 0.5% inferior to the precision-oriented solution at a maximum. The hardware cost of a 1024-entry IRF on an FPGA was modest: 2 RAM elements and 0.8% increase of the logic elements.

Keywords: embedded system, secure processor, software protection, instruction fetch, instruction register file

1. Introduction

To protect intellectual property, the defense of software against analysis, plagiarism, and falsification has become an important issue. For embedded systems, it should be

*Corresponding author. Tel.: +81-532-44-1276.
Email addresses: fujieda@ee.tut.ac.jp (Naoki Fujieda), ichikawa@ieee.org (Shuichi Ichikawa)

attained with a minimal overhead because of their strict limitations on resource or performance.

Reverse engineering consists of three steps: acquisition of machine code, disassembly and decompilation. Disassembly is a conversion of machine language into assembly language, while decompilation is a translation of an assembly into a human-readable code of high level language. Tamper resistance of software is obtained by obstructing at least either of them. Specifically, there are the following three major ways to protect software:

- encryption of the machine codes to make them meaningless without a hidden key [1, 2],
- obstruction of disassembly by confusing disassemblers or preventing them from accessing necessary information about the instructions [3–10], and
- obstruction of decompilation by scrambling the structure of the program [11, 12]

Instruction set randomization (ISR) [3–8] is an approach to protect software by obstructing disassembly, where each processor (or group of processors) has a different instruction set. Software is protected from analysis or plagiarism by 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 (or one group of processors) does not operate correctly on the others.

The use of an Instruction Register File (IRF) [13, 14] 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 in fetched instructions. Though it initially aimed at reducing power consumption by packing multiple instructions into a single one [13], it also has resistance to tampering [15]. It is based on the fact that a reference to the IRF is considered as another expression of the instruction, which is incomprehensible as long as the contents of the IRF are hidden. However, the contents may be guessed from the occurrence frequency of indices. Some routines may not be obfuscated enough by the lack of references to the IRF. Such risks was not evaluated in [15], although the code reduction and the execution time with the IRF were measured as its *side effects*.

This paper examines the effectiveness of the IRF against tampering, particularly reverse engineering. It is suitable for embedded systems, such as manufacturing machinery, because of its small overhead on resource and performance. Without protection, valuable know-how and trade secrets that their software contains might be easily uncovered and stolen. Moreover, once their software are analyzed, injection of malicious code by abusing buggy program or unauthorized modification of software might be also possible. When the IRF-based ISR is applied to, most of the program code is expressed by indices of the IRF, which are not understandable without the IRF contents. It can also prevent malfunctions of systems or serious accidents due to falsified software.

The main topic of this paper is proper selection of the contents of the IRF, including the quantification of the efficiency of instruction hiding, proposal of selection algorithms, and their evaluation. This paper also includes an implementation of a large IRF on an FPGA to show that its hardware cost is comparable to other ISR approaches.

The rest of this paper is organized as follows: Section 2 provides overviews of the related studies and prerequisites for the IRF on utilizing it for tamper resistance. In

Section 3, we introduce a scale of tamper resistance for the IRF contents considering frequency analysis and show its characteristics. In Section 4, we propose a branch-and-bound algorithm to find the optimal assignment and evaluate it. Section 5 describes heuristic algorithms to find a sub-optimal assignment and their evaluation with two instruction sets. Section 6 presents and evaluates an FPGA implementation of the IRF. In Section 7, we make some discussions about possible attacks to our method. Finally, we conclude the paper in Section 8.

2. Background

2.1. Protection against Reverse Engineering and 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) [1] and the AEGIS architecture [2]. 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 resources, and thus it is unsuitable for cost-sensitive embedded systems.

Protection of software is also achieved by obstructing at least either of disassembly or decompilation. As an obstruction of disassemblers, Linn and Debray [9] proposed a method to disorder disassemblers to generate erroneous assembly codes in IA-32 by inserting junk bytes that obscure the partitions of instructions. Monden *et al.* [10] proposed a finite state machine-based approach where the interpretation of an opcode varied with the value of the internal state machine. Obfuscation methods against decompilers were structured by Collberg *et al.* [11]. In particular, control flow obfuscation disturbs the reconstruction of the control flow of the original program. It is based on opaque predicates, or conditional codes whose outcomes are actually constant but not easily deduced, followed by unreachable bogus codes [11, 12].

Instruction set randomization (ISR) may be categorized into obstruction of disassembly, and also be considered as lightweight instruction memory encryption. Compared to robust but costly modern ciphers, most of them are based on simple substitution ciphers for lower overhead. Both hardware [5, 7, 8] and software [3, 4, 6] implementations have been studied. Some methods utilize the characteristics of the target instruction set [5]. Some other methods rely on stream ciphers [4, 8], whose characteristics are closer to those of memory encryption: higher safety but higher cost.

An important measure of ISR is the hardness to guess the original instructions from the *randomized* ones with frequency analysis [16]. 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 decrease statistical information of the original instructions.

2.2. Instruction Register File

Instruction Register File (IRF) [13] is a table of frequently used instructions. Figure 1 illustrates the IRF. It assumes a 32-bit RISC ISA such as MIPS and ARM. It is placed

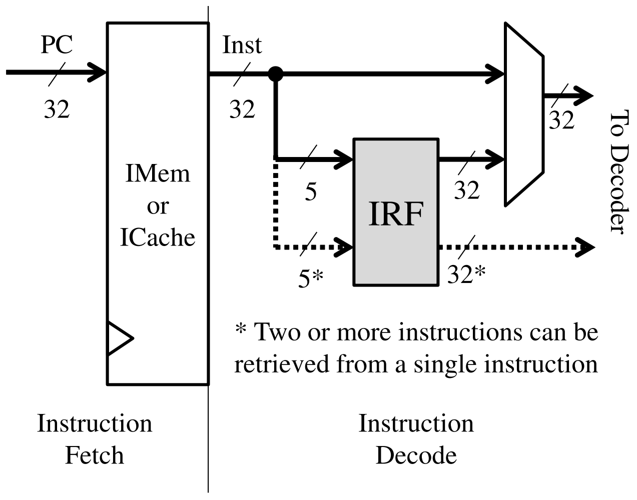


Figure 1: The Instruction Register File [13] retrieves frequently executed instructions from indices in specialized instructions.

just before instruction decoder and accessed by indices. If a specialized instruction is fetched, the corresponding instruction(s) to the index (indices) will be read from the IRF and sent to the decoder. In this paper, we refer to instructions that reside in the IRF as IRF instructions.

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 one. In the original proposal [13], both normal and specialized instructions were 32 bits long. Seven bits of specialized instructions were used for identification and the remaining 25 bits were for indices. Since the number of the IRF was set to 32, each index required $\log_2 32 = 5$ bits. Therefore, a specialized instruction contained up to five (= 25/5) continuous instructions listed in the IRF.

2.3. Using IRF for Anti-tampering

In addition to the reduction of code length, the IRF can provide randomization of a subset of the instructions [15]. It is possible to protect software from analysis by arbitrarily shuffling the mapping from indices to IRF instructions. It provides a protection from plagiarism if the mapping and the corresponding instruction sequence are diversified for each system. It makes the system robust over falsification by prohibiting IRF instructions from being executed directly from instruction memory. In comparison with other ISR methods, the IRF has an advantage of hiding information about operands.

Some ISR methods decrypt instructions when they are read from the main memory to the cache, which means that they may require a special care so that the decrypted instructions can not be read out as data. In the other methods, including the IRF-based ISR, decryption occurs right before instructions are executed. Even if instructions are read as data, obfuscated instructions will naturally be obtained. Though we assume a single-core processor in this paper, when we assume multicore environment, this strategy requires the same number of decryption units (i.e. the IRFs) as the number of processor cores, which might incur a large hardware overhead. This problem can be solved by a careful design of each unit to be small.

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 should be hidden. 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 simply put the most frequent instructions in it. Unfortunately, the evaluation of these risks was not presented in [15].

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 executed through the IRF may become large enough to obfuscate important routines. The frequency analysis will become much more difficult because lower-ranking instructions have almost the same occurrence probability. It may be even difficult to find which instructions reside in the IRF. Although we do not assume specific applications for this approach, it is naturally suitable for embedded systems. Since the distribution of appearance of instructions highly depends on applications, the IRF will include instructions more likely to appear when the field of applications or, if possible, the binary code itself is known before the construction of the *read-only* IRF mapping.

On utilizing a large IRF, the following concerns should be considered: 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.

In this paper, we assume that all processes use the whole of the IRF and they are treated equally on the selection of IRF instructions. Depending on the selection of instructions, a large IRF can be used as multiple small IRFs corresponding to processes, which are similar to the IRF Windows [14]. It can also be possible to give special weights on some specific applications or parts of the program. Such an application specific selection for the IRF can be useful in some cases, though it is out of scope of this paper.

We have presented the preliminary version of this study in CATA-2014 [17]. The differences of this paper from the preliminary work include:

- a detailed analysis of the entropy of the IRF indices, which corresponds to the difficulty of frequency analysis (Section 3.3),
- a branch-and-bound algorithm that obtains the optimal assignment of the IRF, along with its evaluation (Section 4),
- a new time-oriented algorithm with a reuse of calculated value that reduces the time complexity from $O(N^3)$ to $O(N \log N)$ (Section 5.2),
- the evaluation of tamper resistance with binaries compiled for an ARM instruction set (a part of Section 5.5),
- a detailed explanation and evaluation of an FPGA implementation of the IRF (Section 6), and

```
li    t0,10
li    t1,0
addu  t1,t1,t0   ; beginning of loop
addiu t0,t0,-1
bne   t0,zero,-3 ; branch to 3rd instruction if t0 == 0
nop              ; end of loop (MIPS has a delay slot)
```

Figure 2: An example of MIPS assembly to describe the difference between $C_D(I)$ and $C_S(I)$.

- discussions about comparison with other ISR-based approach and about possible attacks such as heuristic analysis of the obfuscated codes (Section 7).

3. Scale of Tamper Resistance for the IRF

3.1. Definition

In the selection of IRF instructions, the original IRF proposal [13, 15] considered only how many instructions are replaced with IRF references. However, as we have explained in Section 2, the cost of frequency analysis should also be considered, which is quantified by the flatness of the distribution of IRF indices. We introduced a scale of tamper resistance for the IRF in the preliminary work [17]. This subsection gives an explanation of the scale.

By dynamic profiling of typical applications, the number of times dynamically executed and the number of times statically appeared for each 32-bit instruction expression, I, are measured. They are defined 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 simply incremented by 1. From them, dynamic execution frequency $P_D(I)$ and static occurrence probability $P_S(I)$ are calculated as follows:

$$P_D(I) = \frac{C_D(I)}{\sum_{i \in \mathbb{I}} C_D(i)}, \;\; P_S(I) = \frac{C_S(I)}{\sum_{i \in \mathbb{I}} C_S(i)}, \tag{1}$$

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

Figure 2 shows an example of MIPS assembly code with a loop, which explains the difference between two kinds of profiles. The third through sixth instructions constitute a *for* loop executed ten times. If the program passes through this piece of code twice, C_D of the first and second instructions will increment by 2 and C_D of the third through sixth instructions will increment by 20, while C_S of the all instructions will increment by 1.

Instead of dynamic profiling, control flow analysis to the binary files can be used to obtain $C_S(I)$. It is usually undesirable that unused parts of the program affect the decision on obfuscation. Although control flow analysis might estimate more precisely which part of the program remains unused, the investigation of the profiling methods suitable for obfuscation is out of focus of this paper.

The frequency of IRF instruction executed, $\gamma(\mathrm{IRF})$, is defined as the sum of their dynamic execution frequency:

$$\gamma(\mathrm{IRF}) = \textstyle\sum_{k=0}^{N-1} P_D(\mathrm{IRF}_k), \tag{2}$$

where IRF_i is the IRF instruction assigned to the index i and N is the number of IRF entries. We formulate it with dynamic frequency P_D because most programs have small sequences of instructions executed repeatedly and they are more suitable parts for being hidden than others.

The cost of frequency analysis, $E(\mathrm{IRF})$, is represented by the Shannon entropy H of the IRF indices, normalized by its theoretical limit of $\log N$:

$$E(\mathrm{IRF}) = \frac{H(I)}{\log N} = \frac{-\sum_{k=0}^{N-1} p_S(\mathrm{IRF}_k) \log p_S(\mathrm{IRF}_k)}{\log N}, \tag{3}$$

where

$$p_S(I) = P_S(I) / \sum_{k=0}^{N-1} P_S(\mathrm{IRF}_k)^1 \tag{4}$$

When $N = 1$, we define $E(\mathrm{IRF}) = 1$ (i.e. the distribution of the IRF indices is completely flat) for convenience. We use P_S to measure the evenness of the distribution because frequency analysis is often made statically with program codes.

Based on these factors, $S(\mathrm{IRF})$ or the scale of the effectiveness of instruction selection in the sense of tamper resistance is defined as follows:

$$S(\mathrm{IRF}) = \gamma(\mathrm{IRF}) \times E(\mathrm{IRF}). \tag{5}$$

It becomes 1 if all of the executed instructions reside in the IRF *and* all of the indices appear completely evenly. On the other hand, it becomes 0 if none of the IRF instructions are executed *or* only one index of the IRF appears. The definition of $S(\mathrm{IRF})$ means that the selection should achieve a wide range of obfuscation and a high cost of frequency analysis at the same time.

In our preliminary study [17], we found that duplication of IRF entries (allowing multiple entries to be assigned to a single instruction) greatly improved the flatness of the IRF indices. We assume that indices of a duplicated instruction are equally used in the applications; that is, when an instruction I is associated with n entries, the static probability of each entry becomes $P_S(I)/n$. In the aspect of obfuscation with the IRF, this means the adoption of a homophonic substitution cipher, which is more difficult to be deciphered than a simple substitution cipher. We make a detailed discussion from this point of view in Section 7.

3.2. Use of One Kind of Instruction Profile

When only one profile of either C_D or C_S can be obtained, it might be acceptable to apply that profile to both of them. In this case, finding the optimal (or a sub-optimal) assignment of the IRF gets simpler. In particular, if each IRF instruction is assigned to only one entry, the assignment that maximizes $\gamma(\mathrm{IRF})$ always has a maximum $S(\mathrm{IRF})$. It is doubtful that such an assignment is useful in the context of tamper resistance, though it greatly reduce the computational effort.

[1] The previous paper [17] made an incorrect explanation of $E(\mathrm{IRF})$. The static probabilities of IRF instructions $P_S(I)$ must have been normalized by their sum.

3.3. Analysis of the entropy of the IRF indices

In the calculation of $S(\text{IRF})$, the most time-consuming part is $H(I)$, which contains N times of logarithmic operations. On searching for a better assignment of the IRF, it is natural to utilize the result of a neighboring assignment that was already considered. On building an assignment under the assumption that a part of it is fixed, it is useful to know the upper bound of $H(I)$. For these reasons, the rest of this section gives an analysis of $H(I)$, the entropy of the IRF indices.

Let $\text{IRF} = \{I_0, I_1, ..., I_{N-1}\}$ be an assignment of the IRF whose $s = \sum_{k=0}^{N-1} P_S(I_k)$ is already calculated. By substituting Equation 4 into 3, we obtain the following equation:

$$H(I) = -\sum_{k=0}^{N-1} \frac{P_S(I_k)}{s}\{\log P_S(I_k) - \log s\} = -\frac{h}{s} + \log s, \tag{6}$$

where $h = \sum_{k=0}^{N-1} P_S(I_k) \log P_S(I_k)$.

When an instruction $I_A \notin \text{IRF}$ is added to the IRF, s increases by $P_S(I_A)$ and h increases by $P_S(I_A) \log P_S(I_A)$. This makes the entropy of the indices of the new IRF be:

$$\begin{aligned} H(I') &= -\frac{h + P_S(I_A)\log P_S(I_A)}{s + P_S(I_A)} + \log\{s + P_S(I_A)\} && (7) \\ &= \{H(I) - \log s\} \cdot \frac{s}{s_a} + \frac{P_S(I_A)\log P_S(I_A)}{s_a} + \log s_a, && (8) \end{aligned}$$

where $s_a = s + P_S(I_A)$. Similarly, when an instruction $I_R \in \text{IRF}$ is removed from the IRF, the entropy is:

$$H(I') = \{H(I) - \log s\} \cdot \frac{s}{s_r} - \frac{P_S(I_R)\log P_S(I_R)}{s_r} + \log s_r, \tag{9}$$

where $s_r = s - P_S(I_R)$.

When the number of entries for an instruction $I_D \in \text{IRF}$ is changed from m to n, s remains unchanged and h of the new assignment, h_d, will be:

$$\begin{aligned} h_d &= h - m \cdot \frac{P_S(I_D)}{m} \log \frac{P_S(I_D)}{m} + n \cdot \frac{P_S(I_D)}{n} \log \frac{P_S(I_D)}{n} \\ &= h - P_S(I_D) \cdot (\log n - \log m). && (10) \end{aligned}$$

Therefore, the entropy of the new IRF indices is:

$$H(I') = H(I) + \frac{P_S(I_D) \cdot (\log n - \log m)}{s}. \tag{11}$$

We then consider adding an instruction I_A that has a static occurrence probability of p (i.e. $P_S(I_A) = p$). The following equation is obtained from Equation 7:

$$H(I') = -\frac{h + p\log p}{s + p} + \log(s + p). \tag{12}$$

Differentiating Equation 12 with respect to p gives:

$$\begin{aligned}\frac{d}{dp}H(I') &= -\frac{\frac{1}{\ln 2}+\log p}{s+p}+\frac{h+p\log p}{(s+p)^2}+\frac{\frac{1}{\ln 2}}{s+p}\\ &= \frac{h-s\log p}{(s+p)^2}. \end{aligned} \tag{13}$$

Hence, $H(I')$ has a global maximum at $p = 2^{h/s}$. In this case, the ratio of h to s of the new assignment h_a/s_a is the same as that of the original:

$$\frac{h_a}{s_a} = \frac{h + 2^{h/s}\cdot\frac{h}{s}}{s+2^{h/s}} = \frac{h(s+2^{h/s})}{s(s+2^{h/s})} = \frac{h}{s}. \tag{14}$$

This means that keeping on adding instructions with a static probability of $2^{h/s}$ maximizes the entropy of the IRF indices. In case that $p < 2^{h/s}$, the ratio h_a/s_a satisfies the following inequalities:

$$\frac{h_a}{s_a} = \frac{h+p\log p}{s+p} > \frac{s\log p + p\log p}{s+p} = \log p, \tag{15}$$

$$\frac{h_a}{s_a} = \frac{h+p\log p}{s+p} < \frac{h+p\cdot\frac{h}{s}}{s+p} = \frac{h}{s}. \tag{16}$$

Rearranging them, we obtain the following inequality,

$$p < 2^{h_a/s_a} < 2^{h/s}, \tag{17}$$

which means the new maximum point goes between the previous maximum point and P_S of the added instruction. This fact helps us prove that maximizing γ(IRF) always has a maximum S(IRF) under the assumption mentioned in Section 3.2.

4. Optimal Selection of IRF Instructions

4.1. Branch-and-bound Algorithm

Since S(IRF) is defined, finding the optimal selection of IRF entries is now formulated as a combinatorial optimization problem to maximize S(IRF). Though this problem is specific to the proposed approach, various algorithms for combinatorial optimization problems can be applied with small modifications. It is generally difficult to compute the exact solution in a practical time. However, when the size of the problem (i.e. the number of IRF entries) is small, a branch-and-bound algorithm with a proper pruning of a search tree might help to give the exact solution.

Let N be the number of IRF entries and $\{I_0, I_1, ..., I_{M-1}\}$ be the candidates for IRF instructions. If the instructions $I_0, I_1, ..., I_{k-1}$ have the respective numbers of assigned entries $n_0, n_1, ..., n_{k-1}$, the subproblem of the selection of IRF entries is to find the rest of the assignment $n_k, n_{k+1}, ..., n_{M-1}$ that maximize S(IRF) under the condition of $U = N - \sum_{i=0}^{k-1} n_i = \sum_{i=k}^{M-1} n_i$ (where U is the number of unassigned entries).

For pruning, we now consider the upper bound of S(IRF). To maximize γ(IRF), we will select the top U instructions in P_D from $I_k, I_{k+1}, ..., I_{N-1}$. In respect of E(IRF),

```
1: if $U = 0$ or $k = M$ then
2:     $g\_new \leftarrow g$
3:     $s\_new \leftarrow s$
4:     $h\_new \leftarrow h$
5: else if $s\_max[k][1] > 2^{h/s}$ then
6:     $g\_new \leftarrow g + g\_max[k][U]$
7:     $s\_new \leftarrow s + U \times 2^{h/s}$
8:     $h\_new \leftarrow h + U \times 2^{h/s} \times h/s$
9: else
10:    $g\_new \leftarrow g + g\_max[k][U]$
11:    $s\_new \leftarrow s + s\_max[k][U]$
12:    $h\_new \leftarrow h + h\_max[k][U]$
13: end if
14: return $g\_new \times (-h\_new/s\_new + \log s\_new)/\log N$
```

Figure 3: Pseudocode for calculating the upper bound of S(IRF).

although we should add instructions with a static probability of $2^{h/s}$, it is often that all of the instructions under consideration have lower probability than $2^{h/s}$ (i.e. the highest P_S of them is lower than $2^{h/s}$). In this case, selecting the top U instructions in P_S maximizes E(IRF).

Figure 3 shows the calculation of the upper bound of the scale. The input g, s, and h are the sum of P_D, P_S, and $P_S \log P_S$ of the current IRF assignment, respectively. A pre-calculated array $g\_max[k][U]$, $s\_max[k][U]$, and $h\_max[k][U]$ stands for the sum of P_D, P_S and $P_S \log P_S$ of the top U instructions with respect to P_D, P_S and $P_S \log P_S$ in $I_k, I_{K+1}, ..., I_{N-1}$, respectively. If there are no unassigned entries or no instructions to be assigned, the present IRF is complete: the upper bound of S(IRF) is the same as the current S(IRF) (Lines 2–4). If the highest P_S in the instructions under consideration is higher than $2^{h/s}$, there is a chance of adding instructions with P_S of $2^{h/s}$ (Lines 7 and 8); otherwise, the top U instructions in P_S are selected (Lines 11 and 12). In both cases, the top U instructions in P_D are used for the upper bound of γ(IRF) (Lines 6 and 10). The upper bound of S(IRF) is obtained in Line 14 by rearranging Equations 3, 5 and 6.

4.2. Evaluation Methodology for Optimal Selection

For profiling, we use 36 traces of MiBench [18], compiled and statically linked with gcc 4.7.3, uClibc 0.9.33.2, and binutils 2.21. The target instruction set is MIPS32 Release 1, with floating point instructions. The compile options are the same as the defaults of MiBench. Some files were slightly modified to avoid compile errors probably caused by the difference of compiler versions. The instruction profiles are gathered with a modified version of SimMips version 0.7.5 [19].

For a performance reason, the branch-and-bound algorithm is implemented with C, parallelized with OpenMP, and executed on Ubuntu 12.04 LTS. The number of candidates M is set to $N \times 5/4$ (rounded down). Instructions that have the highest $P_D + P_S$ are chosen as the candidates. In addition of the list of selected instructions and the scale of tamper resistance, we evaluate the execution time of the body of the algorithm using a workstation with a Core i7 3770 and 16 GB of DRAM. It is measured three times and the median value is used as the result.

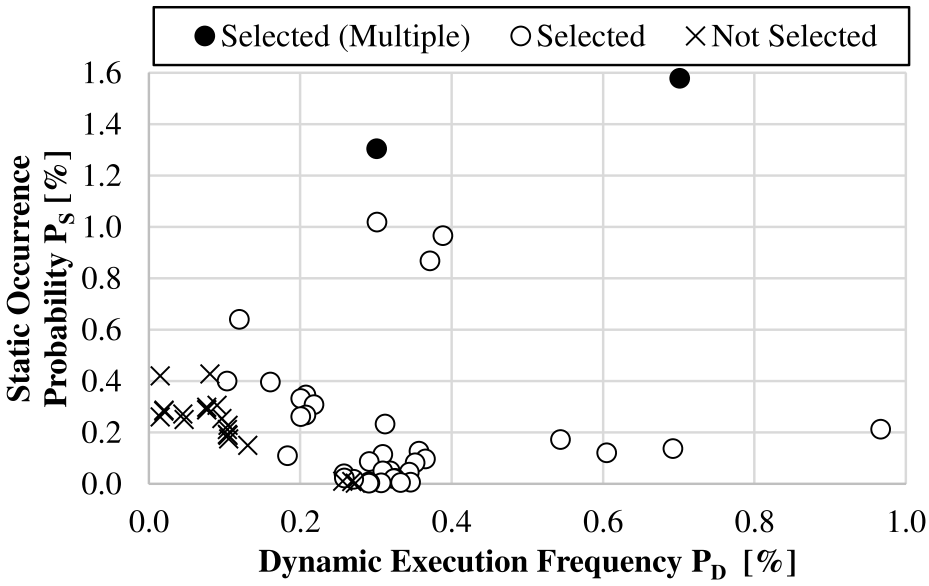


Figure 4: P_D–P_S plot of candidates for the optimal selection of a 48-entry IRF (except `nop`).

4.3. Selected Instructions

Figure 4 presents the distribution of P_D and P_S of candidates for the optimal selection of a 48-entry IRF. The X- and Y- axes are the dynamic execution frequency P_D and the static occurrence probability P_S, respectively. A white circle represents a candidate that was selected as an IRF instruction with a single entry. A black circle stands for a candidate that was assigned to multiple IRF entries. An X mark means a candidate that was excluded from the IRF. One of the candidates, `nop` instruction, is not shown in the figure because its probability was too high.

Most of the instructions that were not listed in the IRF had low dynamic frequencies P_D, which would decrease $\gamma(\text{IRF})$ if selected. The few other instructions had quite low static probabilities P_S. They would harm the flatness of the indices distribution $E(\text{IRF})$. Instructions with very high P_S, which might also decrease the flatness, were divided into multiple entries rather than being removed from the IRF.

4.4. Calculation Time for Optimal Selection

Figure 5 plots the execution time of the branch-and-bound algorithm. The X-axis is the number of IRF entries N and the Y-axis is the execution time in a logarithmic scale. For $N \geq 24$, the execution time increased exponentially. If it continued to grow, it would take 7.6 years to give an assignment of an IRF with only 64 entries! For a large IRF, heuristic algorithms are essential to find a sub-optimal selection.

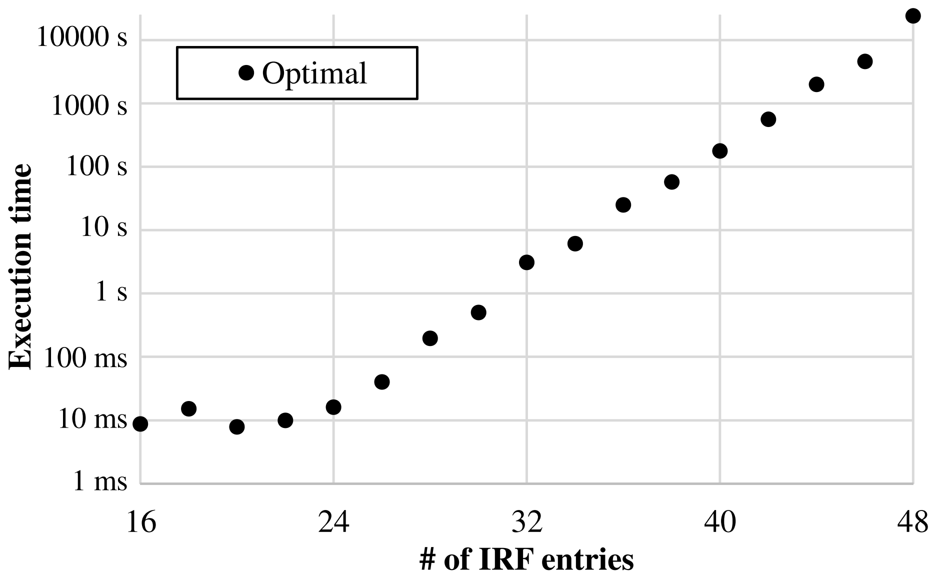


Figure 5: Execution time of the branch-and-bound algorithm to find the optimal selection.

5. Heuristic Algorithms

This section describes two heuristic algorithms, precision-oriented [17] and time-oriented, both of which are based on a local search. A genetic algorithm is often used for combinatorial optimization problems, especially when there is high possibility of falling into local optimum. Though a local search is much simpler than a genetic algorithm, it can give solutions sufficiently near to the optimal for some problems. This section also presents the evaluation of our algorithms, including how far their solutions are from the optimal (in Section 5.4) and how long they take (in Section 5.6).

5.1. Precision-oriented Algorithm

Figure 6 shows the *precision-oriented* algorithm, which has been proposed in the preliminary version [17]. 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+P_S. The lists of candidates for addition and removal are called $List\_add$ and $List\_rem$, respectively. $List\_add$ can be limited to about $3N/2$ instructions that have the highest $P_D + P_S$. The resulting combination does not change in most cases.

The main loop of the algorithm is composed of three parts:

- finding a candidate for addition (c_add) and the number of entries to be swapped (n_add) where the increase of the scale by adding n_add entries of c_add divided by n_add is maximized,
- finding n_add candidates for removal (C_rem) that have the minimum decrease of the scale by their removal, and

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

Figure 6: Pseudocode for selecting IRF entries (Precision-oriented) [17].

- replacing instructions in C_rem with *c_add*.

They corresponds to Lines 5–8, 9–14, and 15–20 in Figure 6, respectively. The swap occurs only if the expected increase of the scale by addition is more than the expected decrease by removal; otherwise, the current combination of IRF is output as a sub-optimal solution.

Note that Equations 8, 9 and 10 simplify the calculation of the scale in Lines 6 and 11. With this simplification, the time complexity of this precision-oriented algorithm changes from $O(N^3)$ to $O(N^2)$, for the complexity of a search for the candidate becomes $O(N)$ and the number of repetition is up to N.

5.2. Time-oriented Algorithm

For a larger IRF, we propose a time-oriented algorithm where the time complexity is reduced to $O(N \log N)$ at the sacrifice of the precision of the assignment. The points of the algorithm are to evaluate candidates with simpler functions and to use priority queues for quick selection of candidates. The time-oriented algorithm consists of two steps that have different evaluation functions.

Figure 7 shows the first step of the algorithm to reduce the number of candidates for IRF instructions to N. Duplication of entries is not considered in this step. In the beginning of the algorithm, it calculates the expected increase of S by a non-IRF instruction (Line 4) and the expected decrease of S by an IRF instruction (Line 5). Since these values are never updated and the accuracy of expectation is low, before swapping the candidates for addition and removal, the algorithm checks if the scale will actually be increased (Line 11). If the check fails, either of the candidates is removed from the corresponding list (Lines 15–18).

Figure 8 outlines the second step of the algorithm that selects the entries to be duplicated. In this step, each IRF instruction candidate keeps the number of entries

```
1: $IRF \leftarrow$ any $N$ instructions
2: $Queue\_add, Queue\_rem \leftarrow$ new priority queue
3: for $inst$ in all instructions do
4:     $Queue\_add$.insert($inst, S(IRF + inst) - S(IRF)$)  if $inst \notin IRF$
5:     $Queue\_rem$.insert($inst, S(IRF) - S(IRF - inst)$)  if $inst \in IRF$
6: end for
7: while $Queue\_add$.max_priority> $Queue\_rem$.min_priority do
8:     $c\_add \leftarrow Queue\_add$.max_key
9:     $c\_rem \leftarrow Queue\_rem$.min_key
10:    $IRF\_new \leftarrow IRF + c\_add - c\_rem$
11:    if $S(IRF\_new) > S(IRF)$ then
12:        $IRF \leftarrow IRF\_new$
13:        $Queue\_add$.delete($c\_add$)
14:        $Queue\_rem$.delete($c\_rem$)
15:    else if rand(2) = 0 then
16:        $Queue\_add$.delete($c\_add$)
17:    else
18:        $Queue\_rem$.delete($c\_rem$)
19:    end if
20: end while
```

Figure 7: Pseudocode for selecting IRF entries (Time-oriented, Step 1).

```
1: $IRF \leftarrow IRF$ calculated in Step 1
2: $Queue\_add, Queue\_rem \leftarrow$ new priority queue
3: for $inst$ in $IRF$ do
4:     $Queue\_add$.insert($inst, P_S(inst)$)
5:     $Queue\_rem$.insert($inst, P_D(inst)$)
6:     $Multi[inst] \leftarrow 1$
7: end for
8: while $Queue\_rem$ is not empty do
9:     $c\_add \leftarrow Queue\_add.max\_key$
10:    $c\_rem \leftarrow Queue\_add.min\_key$
11:    $IRF\_new \leftarrow IRF + c\_add - c\_rem$
12:    if $S(IRF\_new) > S(IRF)$ then
13:        $IRF \leftarrow IRF + c\_add - c\_rem$
14:        $Multi[c\_add] \leftarrow Multi[c\_add] + 1$
15:        $Queue\_add$.delete($c\_add$)
16:        $Queue\_add$.insert($c\_add, P_S(c\_add)/(Multi[c\_add] \times 2 + 1)$)
17:        $Queue\_rem$.delete($c\_add$)
18:    else if $S(IRF\_new) < S(IRF) - threshold$ then
19:        break
20:    end if
21:    $Queue\_add$.delete($c\_rem$)
22:    $Queue\_rem$.delete($c\_rem$)
23: end while
```

Figure 8: Pseudocode for selecting IRF entries (Time-oriented, Step 2).

currently assigned, which is represented as $Multi$ in Figure 8. The candidate for addition (i.e. duplication) is selected according to the static occurrence probability P_S divided by $Multi$ times 2 plus 1 (Lines 4 and 16). This division corresponds to the Sainte-Laguë method in voting systems, which favors those who receive fewer assignments [20]. It shows a good balance between the evenness of the distribution of IRF entries and the number of instructions that remain in the IRF. As the candidate for removal, the least frequently executed instruction, i.e. the instruction with the least P_D, is selected (Line 5). Just the same as the first step, a swap of the candidates will occur only if the scale will actually be increased (Lines 11–14). This step continues until every instruction is either duplicated or examined once as a candidate for removal, though we can abort it if the decline of the scale after the swap reaches a certain threshold (Lines 18–19).

5.3. Evaluation Methodology for Heuristic Algorithms

We use the same instruction profiles as we used in Section 4.3 for the MIPS ISA. To examine the sensitivity to the instruction set, we also use MiBench traces of the ARMv7-A ISA. The ARM profiles are obtained with a modified version of QEMU version 1.7.0 [21].

The heuristic algorithms are implemented with Ruby. Their input is an instruction profile, which is sorted by $P_D + P_S$ on ahead. The implemented script is executed by Ruby 2.2.2 on Cygwin 2.0.2. Since there are no standard priority queue libraries for Ruby, we uses the PriorityQueue library on RubyGems, which implements a Fibonacci heap with C [22].

We examine four algorithms shown below with various numbers of IRF entries N.

- **max_dyn** aims to maximize $\gamma(\mathrm{IRF})$ or the sum of dynamic execution frequency in the IRF which corresponds to the original proposal [13].
- **optimal** is the branch-and-bound algorithm that finds the optimal solution (Section 4.1),
- **precision** is the precision-oriented algorithm (Section 5.1), and
- **time** is the time-oriented algorithm (Section 5.2) where the threshold to abort the search is set to 10^{-4}.

5.4. Optimal Versus Sub-optimal

Figure 9 shows the scale of tamper resistance $S(\mathrm{IRF})$ in MIPS with the heuristic algorithms where the number of entries N is small enough (up to 48) to find the optimal solutions. The Y-axis is the scale relative to the optimal selection: if the selection with the heuristic algorithm is the same as the optimal one, the relative scale becomes 100 %.

The precision-oriented algorithm found assignments the same as or quite similar to (within 0.1% in the scale) the optimal. The decrease of the scale with the time-oriented algorithm varies with size but it is less than 0.3% at a maximum and about 0.15% on average. For a local search is good enough to find a sub-optimal IRF assignment, complex metaheuristics such as a genetic algorithm are not necessary for our purpose.

Table 1 explains the difference between heuristic algorithms. It shows the number of entries assigned to each of the top five MIPS instructions in $P_D + P_S$. The greatest

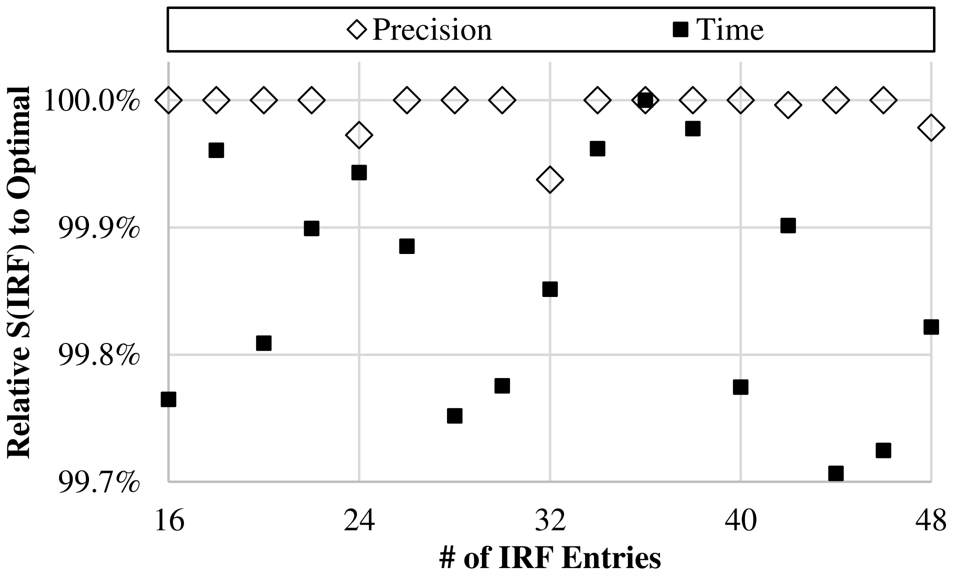


Figure 9: The scale of selection of heuristic algorithms relative to the optimal selection.

Table 1: The number of entries assigned to the most frequent instructions.

Rank	Mnemonic	precision	time
1	`nop`	57	60
2	`jr   $ra`	15	16
3	`lw   $gp, 16($sp)`	12	**0**
4	`addu $gp, $gp, $t9`	9	10
5	`lui  $gp, 2`	10	10

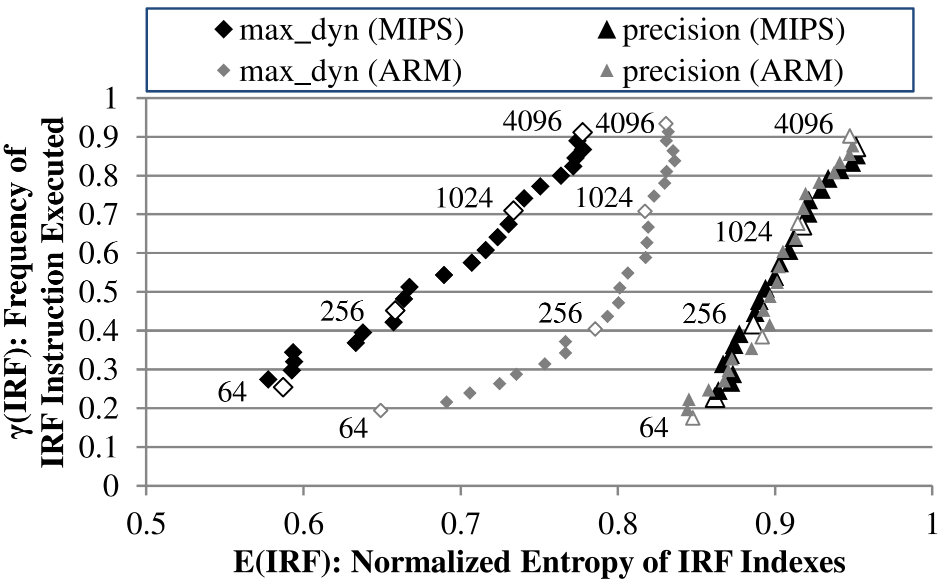


Figure 10: The difference of the breakdown of the scale with the size of the IRF.

difference is found in the third instruction, `lw`, which was given twelve entries by *precision* but removed from the IRF by *time*.

According to the evaluation results in Section 4.4, instructions with large P_S tend to be divided into multiple entries. However, simply removing some of such instructions from the IRF may also be profitable. In the time-oriented algorithm, once an instruction is removed in the first step, it is never reviewed in the second step. The slight decrease of the scale is caused by the difference of the benefit between its removal and its duplication.

5.5. ISA and IRF Size Sensitivity

Figure 10 depicts the difference of the effectiveness of our methods with the size of the IRF. The X- and Y- axes stand for $E(\text{IRF})$ and $\gamma(\text{IRF})$, respectively. Each point represents the evaluated value with a certain IRF size. The cases of 64, 256, 1024, and 4096 entries are shown as outlined points. There is an about $\sqrt[4]{2}$ times difference in the number of entries between adjacent points. The results of *time* are omitted from Figure 10 because there is little difference between *precision* and *time*.

In terms of the improvement rate of the scale with *precision*, our algorithms worked more efficiently with a smaller IRF. For example, the improvement rates with 64-entry and 4096-entry IRF in MIPS were 31.1% and 17.3%, respectively. The rate also varied with instruction sets: the increase of the scale was smaller in ARM than in MIPS. A 4096-entry IRF in ARM showed 10.2% increase with *precision*. The difference comes from the original distribution of instructions, which is especially biased among high-ranked instructions. $E(\text{IRF})$ was steeply improved in *max_dyn* with the increase of the entries, while it was gently increased in *precision*. However, it could also be said that *precision* gave a flat distribution even with a small IRF.

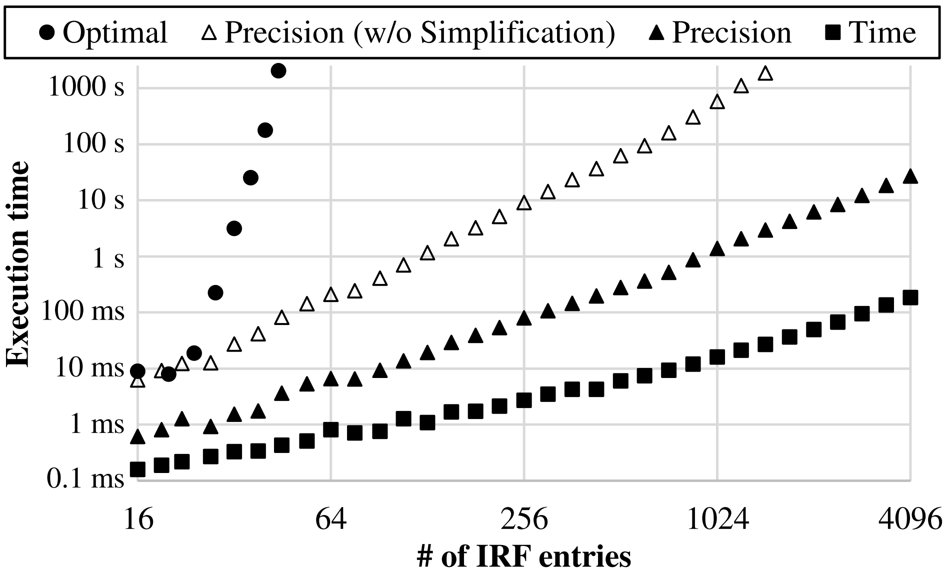


Figure 11: The difference of the execution time with the algorithm.

Comparing between the different IRF sizes, the scale of the IRF with *precision* was almost the same as that of the $2^{3/4} \approx 1.68$ times and $2^{1/4} \approx 1.19$ times larger IRF with *max_dyn* in MIPS and ARM, respectively. This result suggests that our proposal achieves the same tamper resistance as the original IRF with the smaller size of the IRF simply by the modification of the way to decide the IRF contents.

5.6. Calculation Time for Sub-optimal Selection

Figure 11 shows the difference of the execution time of the proposed algorithm in MIPS ISA. The X- and Y- axes, both of which use logarithmic scales, mean the number of entries N and the execution time. For precision-oriented algorithm, the results without the simplification of calculation shown in Section 3.3 are also shown by outlined points, which represent the preliminary study [17]. Note that a direct comparison of *optimal* with other algorithms does not make sense because it is implemented in a different way: we intend to explain its tendency.

The execution times of *precision* with and without the simplification, almost followed exponential functions. They are approximately proportional to $N^{1.95}$ and $N^{2.81}$, respectively. The time-oriented algorithm provided much better performance, which gave a sub-optimal assignment of a 4096-entry IRF in 186 milliseconds. The loss in the scale of tamper resistance was 0.2% on average and 0.5% at a maximum. The new algorithm along with the simplification for the calculation of the scale made the time for the search for IRF assignment be short enough, though an improved implementation, such as the use of C and parallelization, might further accelerate the search.

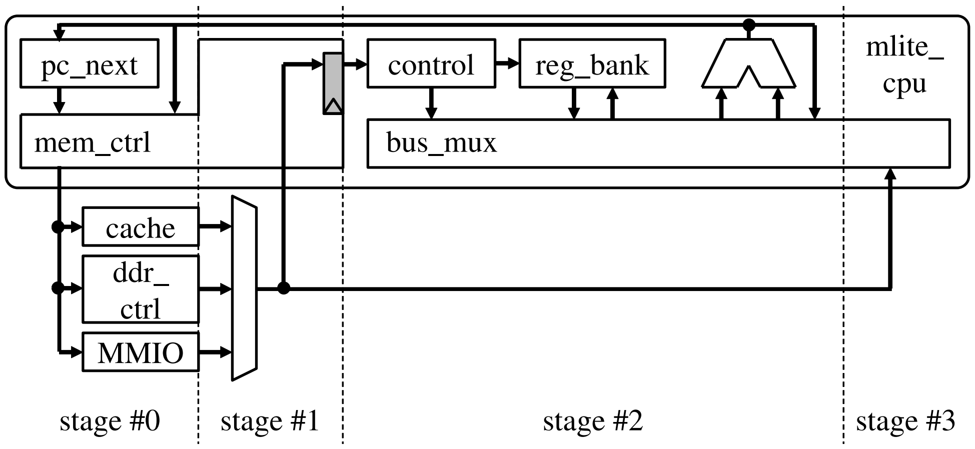


Figure 12: The *two-stage* pipeline of the Plasma processor [23], whose actual number of pipeline stage is four.

6. FPGA Implementation

6.1. Use of Block RAM

This section describes an implementation of a large IRF, where Block RAMs (BRAMs) are used for efficiency, and its evaluation on a Xilinx FPGA. To focus on the overhead of the addition of the IRF, instruction packing [13] is not applied: normal MIPS instructions are transformed to specialized ones (each of which has a predefined opcode and an index of the IRF) on a one-to-one basis. The program is modified a priori so that IRF instructions can be replaced into the corresponding specialized instructions, which are translated back by the IRF. Although we have illustrated the IRF in Figure 1 as an asynchronous RAM, BRAMs are synchronous. It means that some processors require an additional cycle or pipeline stage before instructions are passed to the decoder.

We chose Plasma [23] (the latest snapshot as of September 2013) as the target processor. Figure 12 abstracts its *two-stage* pipeline, which is actually composed of four stages. Before the instruction fetch stage (stage #1), the Plasma core, called *mlite_cpu*, determines the address to fetch in *pc_next* and *mem_ctrl* and gives it to a cache, DRAM controller (*ddr_ctrl*), and various memory-mapped I/O (MMIO) devices at the stage #0. Decode, register fetch, and execute stages are unified to the stage #2. If the instruction turns out to be load or store, the target address is calculated at the ALU and passed to *mem_ctrl*. In this case, the memory controller performs load or store instead of instruction fetch, causing a pipeline stall. The loaded data is written to the register at the stage #3. To add an IRF to Plasma, the fetched instruction latched in the gray register in Figure 12 has to be intercepted before it is passed to the decoder (*control*).

Figure 13 shows an implementation of the IRF [17]. We assume a 1024-entry IRF accessed with a 10-bit index in the Figure. IN means an instruction fetched from the instruction memory or cache. OUT stands for an instruction to be passed to the decoder. The IRF is accessed with specific 10 bits of IN, while at the same time IN is stored to a register. In the next cycle, if the stored instruction is a specialized (IRF-referring)

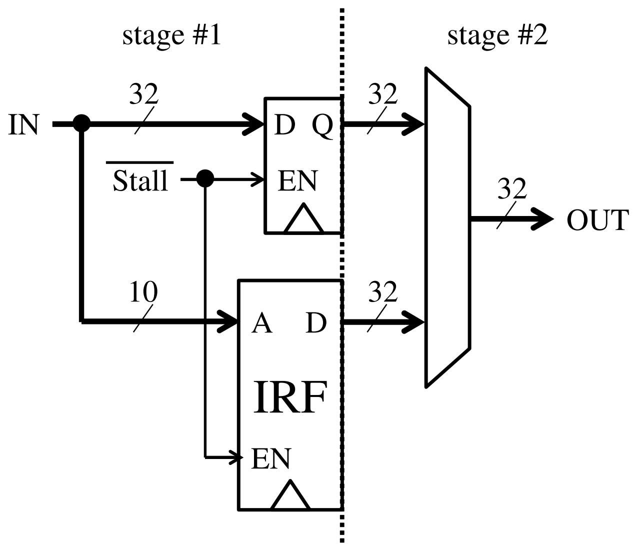


Figure 13: An implementation of the IRF, where the output of the IRF is passed to the decoder when a specialized instruction is fetched.

instruction, the IRF-side instruction is selected and passed to the decoder; otherwise, the register-side instruction is selected. In addition, when the pipeline stalls (shown as a signal Stall), the enable signals (EN) of the IRF and the registers are negated and thus the output remains unchanged.

6.2. Evaluation of Hardware Overhead

This section examines the influence of the IRF on the performance and the hardware resources of Plasma. Several approaches based on ISR and encryption are also evaluated for comparison. The encryption-based approach decrypts instructions that have been encrypted by AES with a 128-bit. Like the simple XOM architecture [1], decryption occurs on reading instructions from the main memory to the caches. Since the block length of AES is 128 bit, the DRAM controller of Plasma is modified so that it could read a 128-bit block at once. The decrypted block is buffered until another block is read: it responds to a continuous read request to the same block without overhead. To be fair, this modification is applied to all examined implementations.

The amount of hardware was evaluated with the number of slices (basic logic blocks in Xilinx FPGAs), flip-flops and LUTs (as components of slices), and BRAMs. The Dhrystone benchmark was used for the performance measurement: the performance index was Dhrystone MIPS (million instructions per second): the maximum operating frequency (Fmax) of the circuit in MHz multiplied by the IPC (instructions per cycle). The circuits were synthesized and implemented with Xilinx ISE 14.7. The synthesis and implementation options were the PlanAhead Defaults (XST 14) and the ParHighEffort (ISE 14) presets with ignoring timing constraints (`-x`), respectively. The system operation was verified on the Xilinx Spartan-3E Starter Board.

We evaluated the following six implementations of Plasma:

Table 2: The results of FPGA implementations.

Approach	**Slice**	FF	LUT	BRAM	**DMIPS**	Fmax	IPC
Baseline	2,177	834	3,921	5	16.9	36.4	0.464
XOR	2,179	839	3,923	5	16.6	35.7	0.464
Shuffle-Bit	2,194	839	3,953	5	16.7	35.9	0.464
Shuffle-Op	2,241	838	4,045	5	16.6	35.7	0.464
IRF	2,194	822	3,972	7	14.8	32.0	0.464
AES	3,874	1,119	7,346	5	14.6	35.5	0.411

- **Baseline** was a modified Plasma with a 128-bit DRAM controller, configured to have *two-stage* pipeline, 4kiB instruction/data cache, and 4kiB internal memory for bootloader;
- **XOR** inserted a 32-bit bitwise XOR function with a value [3, 7] before the fetched instruction is latched;
- **Shuffle-Bit** inserted a bit-transposition logic [3, 7] before the latch;
- **Shuffle-Op** added ROMs to shuffle *opcode*, *function* and *rt* fields of MIPS instructions [5] before the latch;
- **IRF** meant the proposed approach depicted in Figure 13; and
- **AES** added an AES decryption circuit [24], which took 10 cycles to decrypt a 128-bit block, to the DRAM controller.

As we described in Section 2.3, we assume the mapping of the IRF is common to all processes, which means the IRF is implemented with ROMs whose contents are written on the circuit programming. In the existing ISR approaches, *XOR*, *Shuffle-Bit*, and *Shuffle-Op*, the value to be XORed or the mapping of the shuffling block is fixed for each circuit. Since this leads large variability among the resulting circuits, we generated ten circuits for each approach and the average value was used as the result.

The implementation results are summarized in Table 2. The increase of slices by addition of the IRF was 17 or 0.8% of the baseline Plasma, which was comparable with other ISR approaches. In the AES implementation, however, the additional circuit consumed 78% of slices of the processor itself. The IRF implementations required additional two 18 kib BRAMs, which was appropriate because the IRF was a 1024-word, 32-bit RAM. Since the power consumption of an FPGA is well correlated with the number of logic units in use, we expect that the additional power consumption will also be similar to other ISR approaches.

In respect of performance, the Dhrystone MIPS (DMIPS) of the IRF reduced by 12% unlike the other ISR approaches. The degradation was also observed in the encryption approach, which came from a loss of executed instructions per cycle (IPC) caused by a larger penalty on an instruction cache miss.

According to the reports by CAD tools, the critical path of the Plasma processor resides in the stage #2 – decoding instruction, reading register, checking if a branch is taken, calculating the next program counter, and accessing the cache. Hence, the addition of accesses to the IRF and the multiplexer to the stage #2 directly affected the length of the critical path, causing the drop of the maximum frequency (Fmax). In the other ISR approaches, the additional logic can be put at either stage: we put it at the stage #1 in this evaluation to avoid the critical path. Although this difference might affect the flexibility of pipeline, we think the problem of performance degradation is solvable as a matter of pipeline design.

7. Discussion

This section discusses the effect of one-to-multiple assignments to the IRF against frequency analysis and further possible attacks to our scheme with some countermeasures.

If a single IRF entry is assigned to each instruction, the derived instruction sequence is regarded as a simple substitution cipher. The frequency analysis works effectively to decipher this kind of sequence, since the occurrence frequency is preserved. Two or more IRF entries may be assigned to a single instruction, as in our algorithms, to equate the occurrence probability of each IRF entries. The derived instruction sequence is categorized as a homophonic substitution cipher. If the frequency profile is well scrambled, a homophonic cipher becomes more difficult target to decipher.

Although the frequency analysis on digram or multigram is still applicable [16], it is practically difficult to decipher a homophonic cipher with simple approaches, particularly when the number of alphabet is large. Therefore, heuristics and other techniques are empirically applied with dictionary-based approach. For an example, Oranchek [25] proposed a dictionary-based attack combined with genetic algorithm. Ravi and Knight [26] presented a Bayesian approach that combines letter n-gram models and word dictionaries. Thus, to decipher an IRF-based instruction sequence, the heuristics or domain-specific approaches on the target instruction set architecture would be required.

Other lightweight ISR-based approaches [3, 5, 7], on the other hand, are quite vulnerable to the frequency analysis. For the XOR function, the key might be easily guessed by extracting some most frequently appeared instructions. Deciphering bit-transposed instructions might also be easy because the appearance of 0 and 1 varies widely with bit position. Therefore, they have little effect on the resistance of reverse engineering: they are more suitable for preventing the injection of malicious instructions, along with runtime encryption like ASIST [7].

The point of our approach is to stop obfuscating the all instructions, which leads to a cost-effective obfuscation of instruction sequences. However, it also causes a risk of heuristic guess of instructions – obfuscated instructions might be guessed from the plain instructions nearby. Figure 14 shows pieces of MIPS assembly code, which were actually compiled from the C sources of MiBench, along with comments to IRF instructions. In the obfuscated instruction sequences, the commented instructions are translated into the IRF references.

The examples (a) and (b) include IRF instructions that might be predictable. In the example (a), three `move` pseudoinstructions are lined up and the center of them will be replaced to the IRF reference. However, if someone looks at this piece of obfuscated code, it might be easy for him or her to guess that it is `move t7,t1`. In the example (b), an `mflo` in Line 3 will be translated. Since *mflo* instructions are used to take the result of the previous multiplication or division from a special-purpose register, they tend to

```
1  li    s0,79
2  move  t6,t0
3  move  t7,t1      ;[IRF A]
4  move  t8,t2
5  lw    t2,0(v0)
6  xor   a3,a3,t5   ;[IRF B]
7  xor   a1,a1,t3
```

(a)

```
1  addu  a0,s1,a0
2  mult  a0,v0
3  mflo  a1         ;[IRF C]
4  addu  a1,a1,s2
5  addu  a0,s0,a1
```

(b)

```
 1  lbu   s6,-2(a0)
 2  addu  s7,t8,s7   ;[IRF D]
 3  subu  a3,v0,a3   ;[IRF E]
 4  lbu   t8,0(a1)
 5  lbu   a1,-1(a0)
 6  subu  s6,v0,s6   ;[IRF F]
 7  addu  s7,t0,s7
 8  lbu   a3,0(a3)   ;[IRF G]
 9  lbu   t0,0(a0)
10  subu  a1,v0,a1   ;[IRF H]
11  addu  s7,t8,s7   ;[IRF D]
12  lbu   t8,0(s6)
13  lbu   s6,1(a0)
14  lbu   t9,0(a1)
15  subu  t0,v0,t0   ;[IRF I]
16  addu  s7,a3,s7
17  lbu   a3,2(a0)
18  addu  a1,t5,v1
19  addu  s7,t8,s7   ;[IRF D]
20  subu  s6,v0,s6   ;[IRF F]
```

(c)

Figure 14: Examples of compiled codes and how some of them are translated into the references of the IRF.

be placed immediately after multiplication and division. Thus it might also be easy to guess them, unless the adjacent multiplications and divisions are obfuscated as well.

One of the possible countermeasures against the heuristics is to shuffle the order of instructions as far as their flow and dependency are preserved. This technique has been examined to diversify instruction sequences [27]. It weakens the relationship between the adjacent instructions. In shuffling the instruction order, placing IRF instructions consecutively may provide better obfuscation. Considering longer instruction sequences like the example (c), consecutive references to the IRF obstacles the heuristic guess of instructions. Insertion of dummy instructions may also be helpful. For example, in the example (a), any instruction that writes to the register `a1` can be inserted right before `mflo` without breaking the semantics. If one or more IRF instructions are inserted as dummy instructions, the attack will be even more difficult. Nevertheless, some unobfuscated instructions still remain regardless of their order and they might be clues to guess the obfuscated instructions. For example, `addu` instructions in Lines 7 and 16 of (c) remain unchanged, where the register `s7` is involved just like the IRF instruction D in Lines 2, 11, and 19.

Another possible countermeasure is to merge similar instructions by parameterization or indirection. Since the originally proposed IRF was very small, two approaches were proposed in [13] to gather two or more similar instructions into a single IRF entry. One approach is to parameterize a part (or some parts) of instructions. While the original approach parameterized immediate values [13], register specifiers can also be parameterized as we have studied in [28]. However, for the tamper-aware use of the IRF, the distribution of parameters itself should also be robust against frequency analysis. The other approach is to add another way to express the register specifiers. This approach introduces a positional register specifier that stores the distance to the instruction where the corresponding register was used last. It might help the frequency distribution be flatter because some instructions will belong to two groups of IRF instructions: one expresses the register specifiers in the traditional manner and the other specifies the register positionally. Moreover, some other instructions might get resident in the IRF by reordering them to fit their positional specifiers into IRF entries.

We could provide a better protection while keeping small hardware overhead if we examined these countermeasures in detail, though we leave it as future work.

8. Conclusion

This work presented the utilization of the IRF, which provided the indirect access to instructions in processors, to protect embedded software. The key points of our proposal are summarized as follows: (1) the flatness of the index distribution of the IRF should be considered against frequency analysis; (2) the most frequent instructions should be given multiple IRF entries; (3) it is possible to find a sub-optimal assignment of the IRF quickly with a minimal loss of precision with proper simplification of the algorithm; and (4) it can be implemented with a minimal hardware cost. According to our evaluation results, the precision-oriented algorithm gave the optimal assignment in most cases when the IRF size is small. The time-oriented algorithm reduced the calculation time to about 1/100 with 0.2% less precision than precision-oriented. We successfully implemented the large IRF with a modest cost: 0.8% of additional logic elements of the target processor and 12% of the performance overhead.

Acknowledgements

This work was partially supported by a Grant-in-Aid for Scientific Research from the Japan Society for the Promotion of Science (JSPS).

References

[1] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, M. Horowitz, Architectural support for copy and tamper resistant software, in: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems, 2000, pp. 168–177. doi:10.1145/378993.379237.

[2] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, S. Devadas, AEGIS: architecture for tamper-evident and tamper-resistant processing, in: Proceedings of the 17th annual international conference on Supercomputing, 2003, pp. 160–171. doi:10.1145/782814.782838.

[3] G. S. Kc, A. D. Keromytis, V. Prevelakis, Countering code-injection attacks with instruction-set randomization, in: Proceedings of the 10th ACM conference on Computer and communications security, 2003, pp. 272–280. doi:10.1145/948109.948146.

[4] E. G. Barrantes, D. H. Ackley, S. Forrest, D. Stefanović, Randomized instruction set emulation, ACM Trans. Inf. Syst. Secur. 8 (1) (2005) 3–40. doi:10.1145/1053283.1053286.

[5] S. Ichikawa, T. Sawada, H. Hata, Diversification of processors based on redundancy in instruction set, IEICE Trans. Fundam. E91-A (1) (2008) 211–220. doi:10.1093/ietfec/e91-a.1.211.

[6] G. Portokalidis, A. D. Keromytis, Fast and practical instruction-set randomization for commodity systems, in: Proceedings of the 26th Annual Computer Security Applications Conference, 2010, pp. 41–48. doi:10.1145/1920261.1920268.

[7] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, S. Ioannidis, ASIST: Architectural Support for Instruction Set Randomization, in: Proceedings of 20th ACM Conference on Computer and Communications Security, 2013, pp. 981–992. `doi:10.1145/2508859.2516670`.

[8] J.-L. Danger, S. Guilley, F. Praden, Hardware-enforced Protection against Software Reverse-Engineering based on an Instruction Set Encoding, in: Proceedings of the 3rd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, no. 5, 2014. `doi:10.1145/2556464.2556469`.

[9] C. Linn, S. Debray, Obfuscation of Executable Code to Improve Resistance to Static Disassembly, in: Proceedings of the 10th ACM Conference on Computer and Communications Security, 2003, pp. 290–299. `doi:10.1145/948109.948149`.

[10] A. Monden, A. Monsifrot, C. Thomborson, Tamper-Resistant Software System Based on a Finite State Machine, IEICE Trans. Fundam. E88-A (1) (2005) 112–122. `doi:10.1093/ietfec/e88-a.1.112`.

[11] C. Collberg, C. Thomborson, D. Low, A Taxonomy of Obfuscating Transformations, Tech. Rep. 148, Department of Computer Science, University of Auckland (1997).

[12] H. Chen, L. Yuan, X. Wu, B. Zang, B. Huang, P. chung Yew, Control Flow Obfuscation with Information Flow Tracking, in: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009, pp. 391–400. `doi:10.1145/1669112.1669162`.

[13] S. Hines, J. Green, G. Tyson, D. Whalley, Improving program efficiency by packing instructions into registers, in: Proceedings of the 32nd annual international symposium on Computer Architecture, 2005, pp. 260–271. `doi:10.1109/ISCA.2005.32`.

[14] S. Hines, G. Tyson, D. Whalley, Reducing instruction fetch cost by packing instructions into registerwindows, in: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture, 2005, pp. 19–29. `doi:10.1109/MICRO.2005.27`.

[15] D. Chang, S. Hines, P. West, G. Tyson, D. Whalley, Program differentiation, in: Proceedings of the 2010 Workshop on Interaction between Compilers and Computer Architecture, no. 9, 2010. `doi:10.1145/1739025.1739038`.

[16] F. L. Bauer, Decrypted Secrets: Methods and Maxims of Cryptology, 4th Edition, Springer, 2006.

[17] N. Fujieda, S. Ichikawa, Enhanced Instruction Register Files for Embedded Software Obfuscation, in: Proceedings of the 29th International Conference on Computers and Their Applications, 2014, pp. 153–158.

[18] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown, MiBench: A free, commercially representative embedded benchmark suite, in: Proceedings of 2001 IEEE International Workshop on Workload Characterization, 2001, pp. 3–14. `doi:10.1109/WWC.2001.990739`.

[19] N. Fujieda, T. Miyoshi, K. Kise, SimMips A MIPS System Simulator, in: Proceedings of 2009 Workshop on Computer Architecture Education, 2009, pp. 32–39.

[20] M. Gallagher, Proportionality, Disproportionality and Electoral Systems, Elect. Stud. 10 (1) (1991) 33–51. `doi:10.1016/0261-3794(91)90004-C`.

[21] F. Bellard, QEMU, a Fast and Portable Dynamic Translator, in: Proceedings of 2005 USENIX Annual Technical Conference, 2005, pp. 41–46.

[22] Brian Schroeder. PriorityQueue — RubyGems.org [online, cited May 29, 2015].

[23] S. Rhoads. Plasma – most MIPS I(TM) opcodes [online, cited February 28, 2014].

[24] Aoki Laboratory. Cryptographic Hardware Project, Graduate School of Information Sciences, Tohoku University [online, cited February 28, 2014].

[25] D. Oranchak, Evolutionary Algorithm for Decryption of Monoalphabetic Homophonic Substitution Ciphers Encoded As Constraint Satisfaction Problems, in: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation, 2008, pp. 1717–1718. `doi:10.1145/1389095.1389425`.

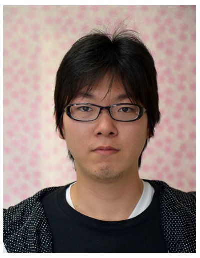
[26] S. Ravi, K. Knight, Bayesian Inference for Zodiac and Other Homophonic Ciphers, in: Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1, 2011, pp. 239–247.

[27] K. Hattanda, S. Ichikawa, Redundancy in instruction sequences of computer programs, IEICE Trans. Fundam. E89-A (1) (2006) 219–221. `doi:10.1093/ietfec/e89-a.1.219`.

[28] N. Fujieda, S. Ichikawa, An XOR-based approach to merging entries for instruction register files, in: Proceedings of the 1st International Symposium on Computing and Networking, 2013, pp. 332–337. `doi:10.1109/CANDAR.2013.60`.

Vitae

Naoki Fujieda received his D.E. degree in 2013 from the Department of Computer Science of Tokyo Institute of Technology. Since 2013, he is an assistant professor of the Department of Electrical and Electronic Information Engineering of Toyohashi University of Technology. His research interests include processor architecture, applied FPGA systems, embedded systems, and secure processors. He is a member of IPSJ, IEICE, and IEEE.



Tasuku Tanaka received his B.E. degree in 2014 from the Department of Electrical and Electronic Information Engineering of Toyohashi University of Technology. Presently, he is studying for his master's degree at that institution.

Shuichi Ichikawa received his D.S. degree in Information Science from the University of Tokyo in 1991. He has been affiliated with Mitsubishi Electric Corporation (1991-1994), Nagoya University (1994-1996), Toyohashi University of Technology (1997-2011), and Numazu College of Technology (2011-2012). Since 2012, he is a professor of the Department of Electrical and Electronic Information Engineering of Toyohashi University of Technology. His research interests include parallel processing, high-performance computing, custom computing machinery, and information security. He is a member of IEEE, ACM, IEICE, and IPSJ.