PAPER

Evaluation of Register Number Abstraction for Enhanced Instruction Register Files

Naoki FUJIEDA[†a)], ***Member***, **Kiyohiro SATO**[†], ***Nonmember***, **Ryodai IWAMOTO**[†], ***Student Member***, ***and*** **Shuichi ICHIKAWA**[†], ***Senior Member***

SUMMARY Instruction set randomization (ISR) is a cost-effective obfuscation technique that modifies or enhances the relationship between instructions and machine languages. An Instruction Register File (IRF), a list of frequently used instructions, can be used for ISR by providing the way of indirect access to them. This study examines the IRF that integrates a positional register, which was proposed as a supplementary unit of the IRF, for the sake of tamper resistance. According to our evaluation, with a new design for the contents of the positional register, the measure of tamper resistance was increased by 8.2% at a maximum, which corresponds to a 32.2% increase in the size of the IRF. The number of logic elements increased by the addition of the positional register was 3.5% of its baseline processor.

key words: *computer architecture, embedded systems, instruction register files, secure processors*

1. Introduction

Tamper resistance is a property of software protection against analysis and falsification. Instruction sequences can be easily disassembled because an instruction set, or relationship between an instruction and a machine language, is usually open to everyone. Malicious codes might be inserted and executed because the instruction set is usually common to all individual systems. Hiding some necessary information from attackers is one of the approaches to improve tamper resistance.

Instruction set randomization (ISR) [1]–[8] is one of the cost-effective methods to obstruct disassembly and code injection. It modifies or adds the relationship between an instruction and a machine language. Hiding the relationship makes analysis difficult. Diversifying it for each system prevents falsification. ISR is also categorized into lightweight instruction encryption; it can be implemented with a smaller cost than secure processors that rely on modern ciphers, such as AEGIS [9] and XOM [10].

An instruction register file (IRF) [3], [11], a small table that supplies frequently executed instructions, can be used as a means of ISR. It is placed after an instruction fetch stage of a processor and the instructions listed in the IRF can be accessed by shorter indices. Although it was originally proposed to reduce the instruction fetch energy by compressing instruction sequences using indices, it also hides the contents of the table, or the relationship between an index and an actual instruction, outside the processor. Our previous study [3] showed that the number of IRF entries should be relatively large (e.g. 1024) to balance its hardware cost and the risk for its contents to be guessed.

In this paper, we focus on a positional register [3], [11], which keeps register numbers used by recently executed instructions. The IRF was proposed with some additional methods, including the positional register, to supply more instructions from the IRF by merging multiple instructions with similar expressions into a single group. The purpose of this paper is to achieve further improvement of tamper resistance of the large-sized IRF using the positional register. A new design of the positional register is proposed for an efficient merge of instructions. The cost of hardware implementation is evaluated in addition to the tamper resistance.

The rest of this paper is organized as follows. Section 2 provides the overview of ISR methods, mainly using the IRF, and the measure of tamper resistance. Section 3 presents the overview and a translation algorithm of the positional register. Section 4 describes a hardware implementation of the positional register, including consideration of its contents. The proposed design of the positional register is evaluated in Sect. 5 and discussed in Sect. 6. Finally, we conclude the paper in Sect. 7.

2. Background

2.1 Instruction Set Randomization

Instruction set randomization (ISR) was originally proposed as a protection from code injection attacks, while it is also considered as a kind of obfuscation technique for instruction memory to prevent disassembly. Though some methods completely rely on software such as emulation and binary instrumentation [1], [5], [6], [8], some recent methods introduced hardware support [2], [4], [7], which was more suitable for embedded systems because of their smaller performance overhead. ISR methods are categorized into two groups: static and dynamic methods. In static methods, the executable instructions are converted into randomized forms before execution. There, since the randomized instructions are usually accessible, the original instructions or the randomization key should not be guessed by the randomized instructions. This property is important not only for pre-

Manuscript received July 11, 2017.
Manuscript revised December 27, 2017.
Manuscript publicized March 14, 2018.

[†]The authors are with Department of Electrical and Electronic Information Engineering, Toyohashi University of Technology, Toyohashi-shi, 441–8580 Japan.
a) E-mail: fujieda@ee.tut.ac.jp

DOI: 10.1587/transinf.2017EDP7221

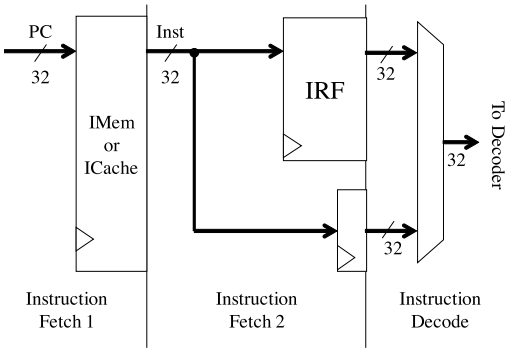


Fig. 1 Organization of Instruction Register Files.

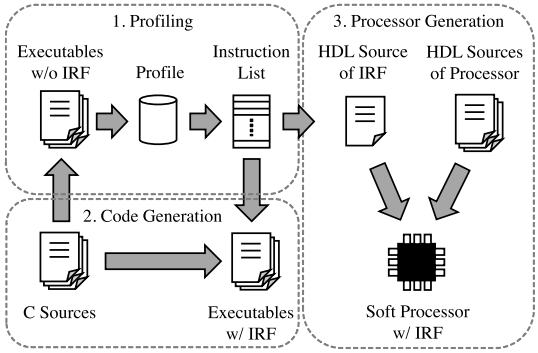


Fig. 2 The flow of code and processor generation with the IRF.

venting disassembly but also for obstructing generation of malicious codes to be injected.

For static ISR methods, the safety of the ciphers and the implementation cost are generally in a trade-off and it is challenging to satisfy both. When a method relies on a simple substitution cipher [4], [7], robustness against frequency analysis is important. If a stream cipher [2] or a finite state machine [6] is used, the internal state of the cipher should be coherent after a branch instruction is executed. It was reported that insertion of special instruction to match the internal state incurs performance loss [2], [6].

Randomness is sometimes utilized by anti-tamper methods other than ISR. A kind of falsification of running programs, called code reuse attacks [12], uses fragments of existing codes in the instruction memory. A typical policy of its countermeasure is to relocate instruction sequences to random addresses. ASLR (Address Space Layout Randomization) [13] is a coarse-grain method that has been adopted to most modern operating systems. More recently, ILR (Instruction Location Randomization) [14] and the Remix system [15] have been proposed as finer-grain methods. For CISC processors (typically x86), random insertion of `NOP` instructions may also be useful to prevent abuse of long instructions being interpreted from the middle of them [16]. Some of them are orthogonal to ISR methods and their combination with ISR might further improve tamper resistance, though it is out of the scope of this paper.

2.2 Instruction Register File

Figure 1 illustrates the organization of the Instruction Register File (IRF). The IRF is a table of instructions that are frequently executed. It was originally proposed to reduce the fetch energy of processors by code compression [11]. In a processor pipeline, it is located between the instruction fetch stage and the decode stage (shown as Instruction Fetch 2 in the figure). An IRF-resident instruction is translated into a special instruction with an index of the IRF at compile time. If the special instruction is fetched from the instruction memory, the original instruction is supplied from the IRF and passed to the decoder, while normal instructions are directly sent to the decoder.

In our previous study [3], we considered this translation of instructions using the IRF as a kind of ISR and utilized it to improve tamper resistance. This is based on the fact that, if the list of IRF-resident instructions is hidden from attackers, it is difficult to recover the original instructions from the special instructions. In other words, the special instructions can work as a kind of substitution ciphers. Note that not all instructions are obfuscated by the IRF: only a part of instructions that are listed in the IRF are translated into the corresponding special instructions. For example, if an instruction `addiu $v0, $v0, 1` is listed in the 10th entry of the IRF in one system, it will be translated into a special instruction such as `IRF #10`. Since the IRF can be shuffled for each system, that instruction may be listed in the 20th entry of the IRF in another system, where it will be translated into `IRF #20`. This diversity prevents the attackers to write malicious codes for each system. The original instruction sequence is difficult to be obtained without knowing the contents of the IRF if a large part of the instructions are translated into such forms.

The IRF alone is not enough to preventing code injection. Even though the IRF-resident instructions are translated into special instructions, they can still be executable by their original forms. This problem can be solved by forcing IRF-resident instructions to be executed only by the corresponding special instructions. We have proposed a complementary unit, called IRRF (Instruction Rejection Register File) to achieve it [17].

In the original proposal of the IRF, the number of entries in the IRF was set to 32 and thus the length of an index was 5 bits [11]. It showed a good balance between the coverage of instructions by the IRF and the number of instructions that can be packed into a single special instruction. There, the IRF was supposed to have different lists of instructions according to applications. For tamper resistance, the balance that should be considered resides between the hardware cost and the risk for the contents of the IRF to be guessed. According to our previous study [3], the number of entries in the IRF should be around 1,024 and its contents should be shared by all applications.

Figure 2 outlines the flow of the generation of executable objects and a soft processor with the IRF, which can be separated into the following three phases.

1. **Profiling.** The target applications are once built and executed normally (without the IRF). An instruction profile, or the collection of the frequencies of executed instructions, is obtained from their execution traces. The list of IRF-resident instructions are generated from the profile by selecting instructions to maximize the measure of tamper resistance explained in Sect. 2.3.
2. **Code Generation.** The target applications are built again. When the actual machine code is generated, each instruction is checked if it is listed in the IRF. If listed, it is transformed into a special instruction that refers to the IRF.
3. **Processor Generation.** Since this study assumes that the contents of the IRF is shared by all applications, the IRF can be embedded in the target processor as a ROM. The HDL source of the IRF is generated from the list. By synthesizing it with the HDL sources of the target processor, the soft processor with the IRF is obtained. The objects generated for this processor do not generally work on the other processors whose IRF are different.

The code generation phase can also be implemented as a post-processing that modified text sections of executable objects. Our current implementation adopts a post-processing method.

2.3 Measure of Tamper Resistance

On hiding the relationship between an index of the IRF and an instruction, we have proposed a measure of tamper resistance and algorithms to select IRF-resident instructions to maximize the measure. The IRF should supply as many frequently executed instructions as possible, while frequency analysis might become easier if there is a bias in the distribution of occurrence frequency of indices. Our measure was defined so that both the coverage of the IRF and the flatness of the distribution of its indices were met at the same time. The sum of the frequency of IRF-resident instructions executed $\gamma(\mathrm{IRF})$ is calculated from $P_D(I)$, the dynamic execution frequency of instruction. The flatness of the distribution, $E(\mathrm{IRF})$, is represented by its Shannon entropy, normalized by the theoretical limit, which is calculated from $P_S(I)$ the static occurrence frequency of instruction I. The measure $S(\mathrm{IRF})$ is defined as the product of them. They are formulated as follows:

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

$$E(\mathrm{IRF}) = \frac{-\sum_{i=0}^{N-1} p_S(\mathrm{IRF}_i) \log p_S(\mathrm{IRF}_i)}{\log N},$$

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

where IRF_i is the ith instruction in the IRF, N is the number of IRF entries, and p_S is calculated from $p_S(\mathrm{IRF}_i) = P_S(\mathrm{IRF}_i) / \sum_{i=0}^{N-1} P_S(\mathrm{IRF}_i)$ i.e. the relative occurrence frequency of the index i in the IRF. We showed that $E(\mathrm{IRF})$ could be significantly improved with a small decrease of $\gamma(\mathrm{IRF})$, when the most frequent instructions were given multiple entries of the IRF [3].

3. Use of Positional Register

3.1 Principle of Positional Register

A positional register is a register file that records the numbers of the registers that were referenced by recently executed instructions [11]. With the positional register, if the current instruction uses the same register as previous instructions, that register can also be specified by the positional register. An instruction that refers the positional register has its indices in the corresponding register fields, instead of register numbers. It also has additional specifier bits that determine which register fields has been replaced with indices. Having '1' in a specifier bit means that the corresponding register field has an index of the positional register and it must be translated into the actual register number before execution. Note that this translation can be applied within a basic block. If there is a branch instruction, recently executed instructions are no more uniquely determined.

When the positional register is combined with the IRF, instructions that refer to the positional register are stored into the IRF with their specifier bits. This means that the output bit width of the IRF gets longer by the length of the specifier bits. On execution, when an instruction supplied from the IRF refers to the positional register, the original instruction is restored by obtaining the actual register numbers from the positional register according to its indices and specifier bits.

The positional register is useful when there are a group of instructions of the same kind that differ in the register number and it can be replaced with the same index of the positional register. In this case, they can be described as the same expression on the IRF and thus supplied from a single entry of the IRF. As a result, it increases the number of instructions supplied from the IRF without increasing the number of entries.

Table 1 shows an example of the translation of instructions with the positional register. The 2nd instruction reads and writes the register `$t0`, which is the same as the destination of the 1st instruction. This register number can be replaced with an index of the positional register, corresponding to the previous destination (shown as `dst[0]` in the table). The 4th and 6th instructions can refer to the previous destination in the same way. After that, the 2nd, 4th and 6th

Table 1 Example of positional register expressions.

Line	Original	Positional
1	`add $t0, $s0, $s1`	`add $t0, $s0, $s1`
2	`lw $t0, 0($t0)`	`lw dst[0], 0(dst[0])`
3	`add $t1, $s0, $t0`	`add $t1, src[1], dst[0]`
4	`lw $t1, 0($t1)`	`lw dst[0], 0(dst[0])`
5	`add $t2, $s0, $t1`	`add $t2, src[1], dst[0]`
6	`lw $t2, 0($t2)`	`lw dst[0], 0(dst[0])`

instructions are now described as the same expression of `lw dst[0], 0(dst[0])`. They can be restored from a single entry of the IRF, even though their original instructions differ each other.

3.2 Expected Benefit on Tamper Resistance

The approach of the positional register is also applicable for improving the measure of tamper resistance. When more instructions are supplied from the IRF, the dynamic coverage of the IRF or γ(IRF) is also increased, which potentially improves S(IRF).

Moreover, it has an advantage from a cryptographic point of view. In the IRF without the positional register, an entry of the IRF corresponds to a single instruction. When the positional register is applied, the corresponding instruction may differ according to context (or the sequence of recently used registers), like an FSM-based ISR approach [6]. Matching of the internal state of the positional register after a branch is quite simple: the history of the register numbers at that point is simply ignored from the subsequent instructions. It may make analysis of the obfuscated instruction sequence more difficult, though its effect is not considered in the calculation of S(IRF).

3.3 Selection of Candidates of Expressions

When an instruction is converted in order to reference the positional register, there is a choice of expression if a register number in that instruction corresponds to multiple indices. Or, it might be the best not to convert it to an index in the first place. Table 1 also includes an example of such a choice. In the 3rd instruction, the register `$t0` is translated as the previous destination (`dst[0]`); however, it also can be found in the previous source (`src[0]`) and the destination of the 1st instruction (`dst[1]`). It has four candidates of expression for this instruction, including the use of the original register number (`$t0`). It should be converted so that more instructions can share the same expression. If expressions are wrongly selected, contrary to the example shown in Sect. 3.1, the same original instructions might be described as different expressions.

In this paper, a greedy algorithm is proposed to select instruction groups that have frequently appeared expressions. It is because finding the optimal combination of expressions is a combinatorial optimization problem and cannot be solved in a practical time. In addition, once a group of instructions is selected, the other expressions of them should be excluded from the subsequent selection. This means that dynamic programming, which is commonly used for combinatorial optimization problems, is not suitable for this problem. The selected expressions are treated as candidates to be stored in the IRF. They are passed to the selection algorithms of IRF contents proposed in the previous study [3].

Figure 3 describes the pseudocode of the selection algorithm, which calls a subprogram to remove overly rare expressions to be selected, shown in Fig. 4. Note that

```
 1: Insts ← list of instructions
 2: Value ← empty associative array
 3: for each inst in Insts do
 4:     inst.exps ← all expression of inst
 5:     inst.rate ← P_D(Insts[i]) + P_S(Insts[i])
 6:     for each exp in inst.exps do
 7:         if Value[exp] does not exist then
 8:             Value[exp] ← 0
 9:         end if
10:         Value[exp] ← Value[exp] + inst.rate
11:     end for
12: end for
13: Candidates ← empty array
14: M ← the number of candidates to be selected
15: V_th ← lower bound of Value of candidates (if known)
16: RareExps ← array of exp such that Value[exp] < V_th
17: for i in [0..M − 1] do
18:     if i mod remove_interval = 0 then
19:         remove_rare_expressions
20:     end if
21:     find best_exp that has the maximum Value
22:     Candidates ← Candidates + best_exp
23:     for each inst in Insts do
24:         if inst.exps includes best_exp then
25:             for each exp in inst.exps do
26:                 Value[exp] ← Value[exp] − inst.rate
27:                 if Value[exp] < V_th then
28:                     RareExps ← RareExps + exp
29:                 end if
30:             end for
31:             clear inst.exps
32:         end if
33:     end for
34: end for
35: make IRF with Candidates by existing method
```

Fig. 3 Pseudocode for selecting expressions with a positional register.

each instruction, *inst*, has a supplementary data structure of *inst.exps* and *inst.rate*. The following is an overview of the algorithm:

1. enumerate all possible expressions with and without the positional register for each instruction (Line 4),
2. calculate the evaluation value for each instruction (Line 5),
3. calculate the evaluation value for each expression of instruction by summing up the values of instructions that have that expression (Lines 6–11),
4. find the expression that has the maximum evaluation value and add it to the candidates (Lines 21–22),
5. exclude instructions that have the selected expression from the subsequent selection (Lines 23–33),
6. repeat the steps 3–5 until the number of candidates reaches the predefined number M (Line 34), and
7. select the contents of the IRF from the obtained candidates using the existing algorithm [3] (Line 35).

In the same way as the selection of candidates in the existing algorithm [3], the evaluation value of instruction I is calculated as $P_D(I) + P_S(I)$. The predefined number M is usually set to $N \times 5/4$, where N is the number of IRF entries. Even though a larger M is set, the final contents of the IRF will

```
1: remove exp included by RareExps from Value
2: for each inst in Insts do
3:     remove exp included by RareExps from inst.exp
4: end for
5: remove inst from Insts such that inst.exp is empty
6: clear RareExps
```

Fig. 4 Pseudocode for removing expressions unlikely to be selected (*remove_rare_expressions* procedure).

not be changed at all in most cases.

This algorithm may be similar to the original proposal of the IRF and the positional register [11], though it was not clearly described in the paper. The difference of our algorithm is that the selected expressions are used as candidates to be stored in the IRF, rather than directly stored into the IRF.

The algorithm also adopts an optimization to reduce the time for selection. If the lower bound of the evaluation value to be selected as a candidate can be estimated from, for example, the previous execution of the algorithm, expressions with the evaluation values less than the lower bound (V_{th} in Fig. 3) can be excluded from the subsequent selection. In the algorithm, such expressions are marked by the list *RareExps* (Line 16 and Lines 27–29), and removed from the selection in a predefined interval (Lines 18–20 and Fig. 4).

The time complexity of the algorithm becomes $O(ME)$, where E is the number of expressions, because every part of the main loop, the *remove_rare_expressions* procedure (Line 19), finding *best_exp* (Line 21), and the inner *for* loop (Lines 23–33), takes $O(E)$. The *remove_rare_expressions* procedure can reduce E toward the end of calculation, while the procedure itself takes significant amount of time. This is the reason why the interval (*remove_interval* in the figure) is required. According to our evaluation (which will shown in Sect. 5.2), the interval should be around 64 to minimize the execution time.

4. Hardware Implementation

4.1 Organization

In the same way as the original IRF proposal [11], our research targets a 32-bit MIPS architecture. Figure 5 describes the three types of MIPS instructions, R, I, and T, and the corresponding register fields. Most of the R-type instructions have an opcode of zero (some floating-point instructions are exceptions [18]) and the opcode of J-type instructions is two or three. The register number can be stored in the rs, rt, and rd fields, each of which is five bits long. R-type instructions uses all of them: the rs and rt fields correspond to the source registers and the rd field corresponds to the destination register. I-type instructions uses the rs and rt fields, used as the source and the destination, respectively. J-type instructions do not have any field for a register number. Since all these fields can be replaced by the indices of the positional register, specifier bits becomes three bits long.

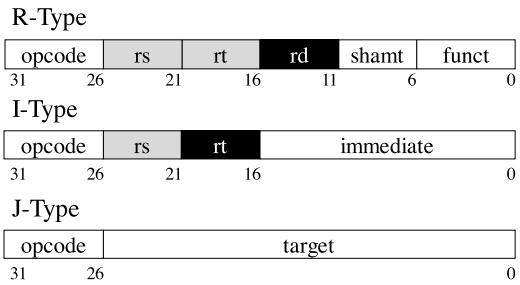


Fig. 5 Types of MIPS instructions and their register fields.

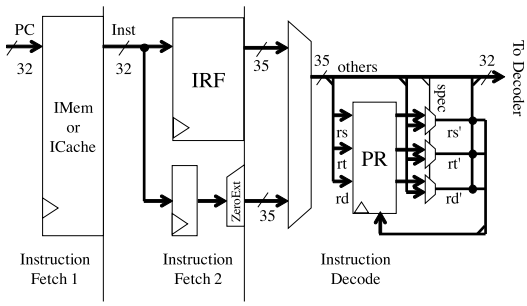


Fig. 6 IRF with a positional register (PR).

Figure 6 shows the organization of the IRF with the positional register. If the fetched instruction is a normal instruction that does not refer to the IRF, it is passed to the decoder without modification by adding zeroes to the specifier bits. When a special instruction is fetched, the IRF supplies a 35-bit instruction with specifier bits (spec). The positional register (PR) is then referenced with the value of the rs, rt, and rd fields as indices. We describe the definition of indices in Sect. 4.2. When the corresponding specifier bit is '1', the value of that field is replaced with the output of the positional register. The original instruction is now restored and passed to the decoder. After that, the obtained register numbers are recorded to the positional register in order to be referenced from subsequent instructions. As we have explained in Sect. 3.1, the translation of instructions is applied within a basic block. Though the register numbers beyond a branch might remain in the positional register, they will never be recalled.

4.2 Register Fields to be Recorded

Since the positional register records the history of the register numbers used by recently executed instructions, it can be implemented as a set of shift registers. Although the original IRF proposal [11] did not showed the detail of the positional register, it appeared that source and destination registers were distinguished and all of the referenced registers were recorded to the positional register.

This is illustrated as Fig. 7. The register fields to be recorded vary with the type of the instruction (R-type or I-type) as shown by multiplexers. As destinations, the rd or rt field is recorded for an R-type or I-type instruction, respectively. As sources, both the rs and rt fields are pushed to a

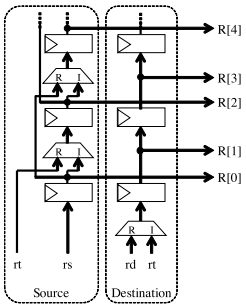


Fig. 7 The positional register that records all sources and destinations (PR-Orig).

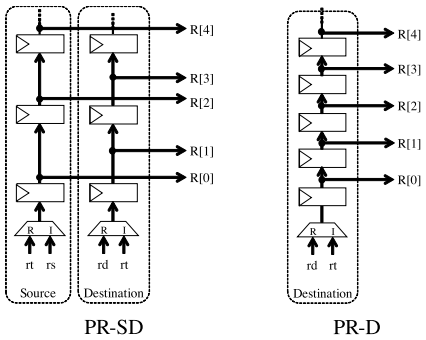


Fig. 8 The positional register that records one of sources and a destination for each instruction (PR-SD) or destinations only (PR-D).

shift register for an R-type instruction, while only the rs field is pushed for an I-type instruction. Values in the shift registers are indexed and one of them is selected as an output value of the positional register. Since there are 32 indices, the length of each shift register is 16. This type of positional register is called PR-Orig in this paper.

Figure 8 describes two simplified versions of the positional register examined in this paper. The PR-Orig type has complexity in the shift register for sources because the number of register numbers to be recorded is depending on the type of the instruction. It can be simplified by recording only one of the source registers for each instruction or no source registers at all. In the first version, called PR-SD, we record only the rt register to the source shift register, based on the result of a preliminary evaluation. Although the length of each shift register remains unchanged (16), the source shift register becomes much simpler. The second version, PR-D, does not record source registers and it only stores the history of destinations to a 32-word shift register.

We then propose another type of positional register, PR-SDPN, that aggressively extracts the similarity of instructions. For each instruction, it records a source, the destination, the next number of the destination, and the previous number of the destination. Figure 9 describes the organization of PR-SDPN. It consists of four shift registers, each of which is eight words long.

An advantage of PR-SDPN is that it can deal with registers that are not directly referenced by previous instructions. In other words, it considers the spatial locality of the

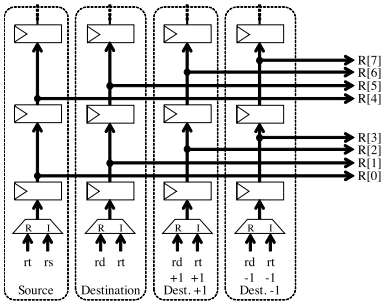


Fig. 9 The positional register that records one of sources, a destination, the destination plus one, and the destination minus one, for each instruction (PR-SDPN).

register numbers, in addition to the temporal locality. In the example of Table 1, the destination of the 3rd instruction, `$t1`, has not been used by the prior instructions. It cannot be translated by the positional register that only records the history of referenced register numbers. However, it can be interpreted as the next number to the previous destination (`$t0`) if such abstraction is acceptable. The 5th instruction can be translated in the same way, as its destination (`$t2`) is the next to the destination of the 4th instruction (`$t1`). As a result, these instructions now have the same expression of, for example, `add dst[0]+1, src[1], dst[0]`.

5. Evaluation

5.1 Methodology

For the evaluation of the selection algorithm and the tamper resistance, instruction profiles from the traces of MiBench [19] are obtained in a similar way to the previous study [3]. The programs are built for the MIPS32 ISA using gcc 4.7.3, uClibc 0.9.33.2, and binutils 2.21. In the previous study, the same instructions appeared in different locations could be grouped and their appearance frequency could be summed up. In this study, they must be treated differently because the translation by the positional register varies with their prior instructions. As a result, the profiles used in this study have 375,478 different instructions, while those in the previous study had 74,589 different (groups of) instructions. The measure of the tamper resistance is E(IRF), which is the same as the previous study and explained in Sect. 2.2. The number of IRF entries is set to 1,024. The selection algorithm, shown in Fig. 3, is implemented in Ruby (2.2.2) and executed on a Core i7 3770 processor with 8 GB of main memory. The number of candidates is set to 1,280. The construction of the IRF from the selected candidates (Line 35 of Fig. 3) is excluded from this evaluation.

For the evaluation of the hardware cost, we implemented the IRF and the positional register on Plasma [20], an open-source MIPS-like soft processor. The measure of the amount of hardware is the number of slices (logic blocks), flip-flops, LUTs and BRAMs (Block RAMs). The maximum operating frequency (Fmax) reported as a place-and-route result is also used to evaluate the performance

Table 2 The calculation time of candidate selection.

Setting	Time [s]	#Expressions (initial)	#Expressions (after removal)
PR-Orig	121.49	1,082,425	33,945
PR-D	82.99	713,773	23,944
PR-SD	118.69	1,032,405	34,274
PR-SDPN	126.96	994,471	34,269

overhead. Also, the executed instructions per cycle (IPC) is collected by running the Dhrystone benchmark. The theoretically maximum performance is calculated as the product of Fmax and IPC. The circuits are synthesized and implemented with Xilinx ISE 14.7, whose strategies are set to PlanAhead Defaults (XST 14) for synthesis and ParHighEffort (ISE 14) for implementation. It aimed for the fastest possible circuit by ignoring timing constraints (`-x` option is applied). The target FPGA is Spartan-3E XC3S500E, the default of Plasma.

The IRF with four types of the positional register shown in Sect. 4.2, PR-Orig, PR-D, PR-SD, and PR-SDPN, are evaluated. The IRF without the positional register is also evaluated for reference. A 1024-entry IRF is referred to as *IRF only* and a double-sized (2048-entry) IRF is shown as *2x IRF*. Note that it is natural for the number of the IRF to be set to a power of two because the IRF is referenced by indices. In the hardware cost evaluation, the baseline Plasma (without the IRF) is implemented, which is referred to as *Baseline*.

5.2 Results of Selection Algorithm

Table 2 shows the calculation time of selecting candidates of expressions, along with the total number of unique expressions, for each setting of the positional register. The parameters V_{th} and *remove_interval* were set to 2×10^{-4} and 64, respectively. In respect of the number of expressions, the number after performing *remove_rare_expressions* procedure for the first time (after removal) is put down with the initial number. The calculation time was strongly correlated with the number of expressions, yet it was not proportional. Most of expressions are not likely to be selected as candidates and they will be quickly removed by the *remove_rare_expressions* procedure. The numbers of expressions after the first removal were almost equivalent among the settings except PR-D.

Our prototype implementation selected 1280 candidates of expressions from about one million possible expressions in about two minutes. Though this is much longer than making the contents of the IRF from the candidate (about a second as reported in [3]), we think this is acceptable because the selection of candidates is required only once in the profiling phase explained in Sect. 2.2.

Figure 10 plots the effect of one of the parameters *remove_interval* on the calculation time of selecting candidates. The setting of positional register was PR-SDPN and V_{th} was set to 2×10^{-4}. The decrease of calculation time by removing expressions unlikely to be selected and the increase by the procedure itself was balanced when the interval was set to 64 through 128.

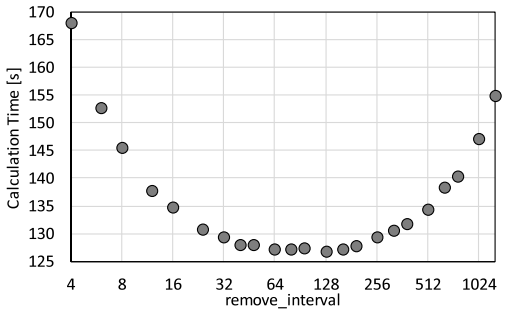


Fig. 10 The effect of remove_interval on the candidate selection of PR-SDPN.

Table 3 The results of tamper resistance evaluation.

Setting	γ(IRF)	E(IRF)	S(IRF)
IRF only	0.6708	0.9167	0.6149
PR-Orig	0.7129	0.9238	0.6586
PR-D	0.7091	0.9240	0.6543
PR-SD	0.7133	0.9241	0.6592
PR-SDPN	0.7185	0.9258	0.6652
2x IRF	0.7930	0.9337	0.7404

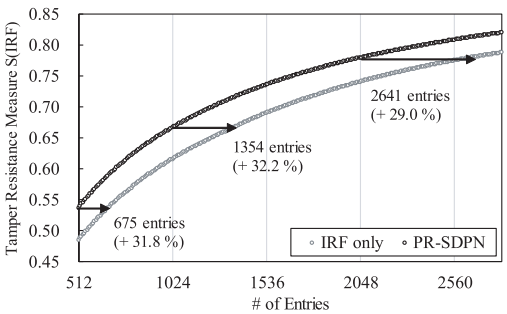


Fig. 11 Tamper resistance measures of various numbers of entries with and without a positional register.

5.3 Results of Tamper Resistance

Table 3 summarizes the results of the tamper resistance evaluation. The proposed positional register (PR-SDPN) improved the measure of tamper resistance by 8.2%, while the increase of the measure by the existing design (PR-Orig) was 7.1%. Although most of the improvement came from the increase of γ(IRF) (7.1%) as we expected in Sect. 3.2, it also achieved some improvement in the flatness of the distribution of indices (1.0% increase of the entropy). In addition, it was shown that simplified versions of the positional register, especially PR-SD, had little effect on the efficiency of translation of instructions.

Although the addition of the positional register did not give better tamper resistance than *2x IRF*, we could measure the number of IRF entries that gave the equivalent tamper resistance to it. Figure 11 plots the relationship between the number of IRF entries with and without the positional regis-

ter (PR-SDPN and IRF only, respectively) and the measure of tamper resistance. A 512-, 1024-, or 2048-entry IRF with the proposed PR-SDPN achieved the equivalent tamper resistance to a 675-, 1354-, or 2641-entry IRF without the positional register. In other words, the addition of PR-SDPN corresponded to about 30% increase in the size of the IRF.

5.4 Results of FPGA Implementation

Table 4 summarizes the results of FPGA implementations of the IRF with and without the positional register. Except for PR-Orig, the increase of the number of slices with the positional register was up to 77, which corresponded to 3.5% of the baseline Plasma processor. In particular, the number of flip-flops (FF) did not increased at all. In Xilinx's FPGA, shift registers can be efficiently implemented in some of the LUTs [21]. Since an LUT can be configured as a 1-bit shift register of up to 16 bits long, the number of LUTs required for shift registers in the positional register becomes 30 (PR-D or PR-SD) or 60 (PR-SDPN). Considering that some other circuits, mainly multiplexers, are required for the positional register, the increase of the number of LUTs (118–163) was reasonable. PR-Orig considerably increased the amount of hardware (17.8% of the baseline) because it cannot fit to the LUT-based shift register.

Doubling the number of entries of the IRF without the positional register increased only the number of BRAMs. It was because a 1024-entry IRF was implemented by two 18-bit 1024-word BRAMs, while a 2048-entry IRF was made with four 9-bit 2048-word BRAM. All other circuits were left unmodified. Although the increase of slices and the increase of BRAMs cannot be directly compared, this increase was generally comparable to PR-SDPN.

It should be noted that increasing the number of the IRF entries requires BRAMs proportional to it, while the increase of logic units by the positional register is almost constant. The positional register effectively increases the size of the IRF by about 30% without BRAMs. Its effect becomes large if the physical number of IRF entries increases. It may lead to a design guide noting that the IRF should be enlarged to the extent as to not overuse the BRAMs before applying the positional register.

As for the maximum operating frequency, the decrease with the positional register from *IRF only* was ranged from 9.8% (with PR-D) to 11.4% (with PR-Orig). The IPC was almost unchanged, though a slight fluctuation was observed, which was mainly due to the change of the timing of the memory controller. These results imply that the addition of the positional register causes 10% decrease of system performance if the processor operates at the maximum possible frequency and the peripheral devices work at a proportional frequency to the processor's. We had already shown that the critical path of the Plasma processor included the decode stage [3], [17]. As the recovery of the original instruction with the positional register is done in the decode stage, it directly affected the critical path. We will discuss an optimized pipeline design for Plasma in Sect. 6.3.

Table 4 The results of FPGA implementations of the IRF.

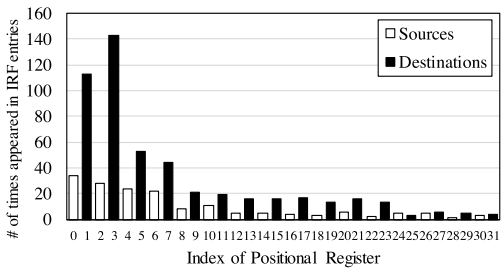
Setting	Slice	FF	LUT	BRAM	Fmax [MHz]	IPC
Baseline	2,177	834	3,921	5	36.43	0.465
IRF only [3]	2,194	822	3,972	7	31.98	0.464
PR-Orig	2,564	982	4,570	7	28.32	0.464
PR-D	2,255	822	4,094	7	28.86	0.465
PR-SD	2,255	822	4,098	7	28.71	0.464
PR-SDPN	2,271	822	4,135	7	28.40	0.464
2x IRF	2,194	822	3,972	9	31.85	0.465

6. Discussion

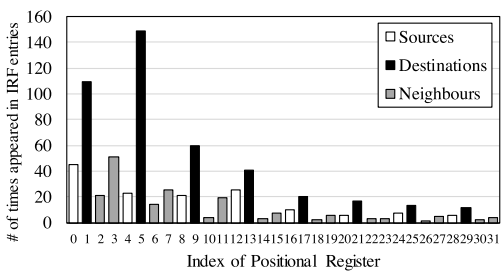
6.1 Usage of Positional Register Indices

This section further analyzes the proposed design of the positional register (PR-SDPN). Figure 12 plots the number of indices of the positional register that appear in the instruction list of the IRF. The upper graph shows the plot of PR-SD, while the lower graph corresponds to PR-SDPN. White, black and gray bars represent source register numbers, destination register numbers, and neighboring numbers of destination, respectively. The register numbers of the most recently executed instruction have the smallest indices.

According to Fig. 12 (a), the destination registers tended to be recalled by the subsequent instructions. This means that there are true data dependencies between these instructions, which is a very common situation. This is a reason that the PR-D simplification works well to some extent. Figure 12 (a) also indicated that most of the references of registers are within a four-instruction range. Translation by the positional register is applied within a basic block. Some compiler optimization, such as loop unrolling, can



(a) PR-SD



(b) PR-SDPN

Fig. 12 Distribution of positional register indices in IRF entries.

Table 5 Relative number of instruction fetch with a 32-entry IRF and a positional register.

Setting	#Fetch [%]
IRF only	62.07
PR-Orig	59.28
PR-D	59.43
PR-SD	59.31
PR-SDPN	58.64
Param-XOR [22]	57.34
2x IRF	56.47

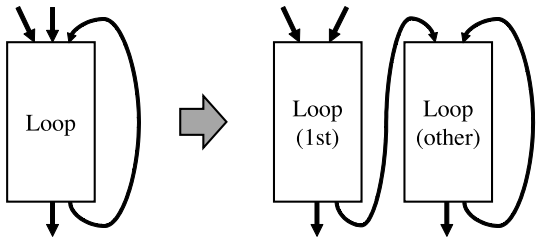


Fig. 13 Example of a loop optimization for the positional register.

make a long basic block and references to registers of distant instructions; however, it is relatively infrequent. In our evaluation, the average length of basic block was 7.83.

From Fig. 12 (b), the previous number of the destination (indices 3, 7, 11, ...) were more likely to be used than the next number (2, 6, 10, ...). When closely looking at disassembled program codes, the decrement of the destination register number is often observed in reading from and writing to the stack. A mismatch between a register number and a relative memory address had prevented such instructions from being grouped. As a result, the sum of the numbers of indices appeared in the IRF was increased from 668 (PR-SD) to 734 (PR-SDPN), which leaded more efficient grouping of the instructions.

6.2 Applying to Fetch Energy Reduction

As explained in Sects. 2.2 and 3.1, the IRF and the positional register was originally proposed to reduce the fetch energy of processors. If the proposed design of the positional register is also efficient for a smaller IRF, it may be useful for the reduction of energy consumption.

We conducted a preliminary evaluation to estimate the potential of our design for fetch energy reduction. The evaluation methodology is almost the same as that shown in Sect. 5.1. Major differences are as follows: the number of IRF entries is set to 32 (or 64 in *2x IRF*); the selection algorithm (Fig. 3) is modified to maximize γ(IRF); and the packing of instruction [11] is applied. Since the packing reserves 25 bits for indices in a special instruction, the maximum number of successive instructions to be packed is five in a 32-entry IRF (5x 5-bit indices) and four in a 64-entry IRF (4x 6-bit indices).

Table 5 summarizes the number of instruction fetch with the IRF, relative to that without the IRF. The result of a parameterization-based approach [22] is also shown as Param-XOR for reference. The proposed positional register reduced the number of fetch by 1.1% (0.6 percentage points) from the original positional register. It proved that the proposed design was useful even for a small IRF. Although it did not reach the parameterization-based approach, these approaches can be combined in principle and their combination can further reduce the fetch energy.

The number of instruction fetch can be further reduced by optimization techniques. For example, the limitation of the positional register about basic blocks can be relaxed by a loop optimization. Suppose a simple loop that contains a single basic block, as illustrated in Fig. 13. Since the loop is entered from multiple locations including the end of the loop itself, the translation with the positional register cannot be applied beyond the basic block. If the loop is duplicated and used separately by the first round and the subsequent (other) rounds, the subsequent rounds are always entered from the same instructions and thus the translation inside the loop is possible. However, such an optimization should be used under a careful consideration because it will increase the length of the code and it may affect the efficiency of an instruction cache. Also, it should be noted that this optimization does not make sense for tamper resistance unless both of the duplicated codes are hidden by the IRF.

Similarly to tamper resistance, a double-sized (64-entry) IRF minimized the number of instruction fetch. However, increasing the size of the IRF is not always the optimal solution in this case. As discussed in the previous studies [11], [22], the use of a large IRF has a trade-off between the coverage of instructions and the efficiency of packing (the maximum number of instructions packed) or the energy consumption of the IRF itself. A detailed exploration about the use of the IRF and its supplements for the reduction of the energy consumption is left for future work.

6.3 Optimization of Pipeline Design

The design of the original Plasma supposes that all registers and RAMs are synchronized with the positive edges of the clocks (except the DDR SDRAM controller). Our implementation of the IRF and the positional register, which was described in Sect. 4 and evaluated in Sect. 5, follows this design scheme. However, if we do not follow this supposition, i.e. the use of the negative edge of the clock for the processor is permitted, a more balanced pipeline design is possible, which allows an optimized implementation of the positional register.

Figure 14 describes the use of a negative edge-triggered IRF in the pipeline. The reconstruction of the instruction with the positional register, along with the selection of instruction between from the IRF and the instruction memory or cache, is moved to the latter half of the 2nd instruction fetch stage. A pipeline register is now required to store the reconstructed instruction, rather than the fetched instruction.

For the PR-SDPN positional register, we implemented its variation shown in Fig. 15. In this implementation, in-

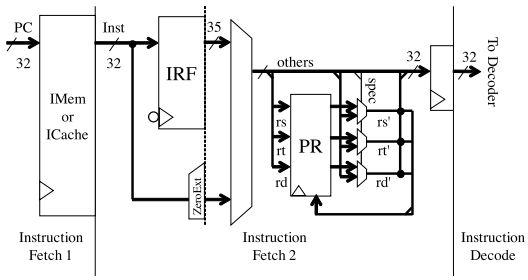


Fig. 14 Negative edge-triggered IRF with a positional register.

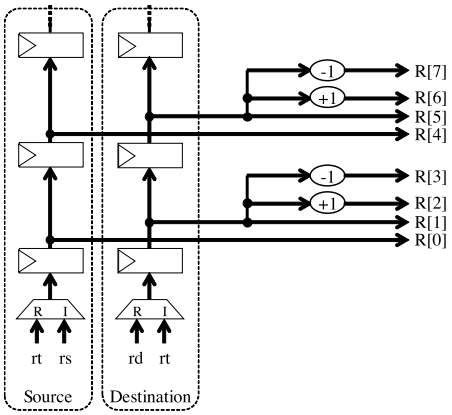


Fig. 15 A variation of the PR-SDPN positional register, which records one of sources, and a destination, and calculates the destination plus and minus one on read.

Table 6 The results of FPGA implementations of the negative edge-triggered IRF.

Setting	Slice	FF	LUT	BRAM	Fmax [MHz]
Baseline	2,177	834	3,921	5	36.43
IRF only	2,192	842	3,919	7	36.76
PR-Orig	2,408	998	4,207	7	36.10
PR-D	2,218	837	3,982	7	37.52
PR-SD	2,222	837	3,987	7	36.09
PR-SDPN	2,233	837	4,017	7	35.44
2x IRF	2,193	842	3,972	9	36.40

stead of recording the register numbers of the destination plus and minus one in shift registers, they are calculated from the recorded destination on read. It reduces the number of shift registers, while it increases the computational cost of retrieving register numbers.

We implemented and evaluated the implementations using the negative edge-triggered IRF as the same way as Sect. 5. Table 6 summarizes their results. The maximum operating frequency of every implementation was quite similar to the baseline. It is because the decode stage of the pipeline, which has the critical path, becomes the same as that of the baseline. Also, the numbers of slices and LUTs were reduced from the positive edge-triggered IRF implementations. As the timing constraints of the circuits around the IRF and the positional register got relaxed, they were optimized to reduce area rather than path delay.

7. Conclusion

In this paper, we examined the combination of the IRF and the positional register to improve tamper resistance and proposed a new design of the positional register to achieve more efficient grouping of instructions. It recorded the next and the previous numbers of the destination register number for each instruction, in addition to a source and the destination. According to our evaluation, 8.2% increase of the measure of tamper resistance was observed, which was equivalent to increase the number of IRF entries from 1,024 to 1,354. The increase of logic elements of our FPGA implementation of the positional register was modest: 3.5% of the baseline processor.

Our future work includes an application of the IRF and the positional register to other processors, especially with other instruction sets such as x86. It will improve the availability of the IRF. A detailed exploration about the use of the proposed design for the reduction of the energy consumption is also left for future work.

Acknowledgments

A part of this work was supported by the Telecommunications Advancement Foundation and JSPS Grants-in-Aid for Scientific Research (KAKENHI) Grant Number 16K00072.

References

[1] E.G. Barrantes, D.H. Ackley, S. Forrest, and D. Stefanović, "Randomized instruction set emulation," ACM Trans. Inf. Syst. Secur., vol.8, no.1, pp.3–40, 2005. DOI:10.1145/1053283.1053286.

[2] J.-L. Danger, S. Guilley, and F. Praden, "Hardware-enforced Protection against Software Reverse-Engineering based on an Instruction Set Encoding," Proc. 3rd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, San Diego, CA, pp.5:1–5:11, 2014. DOI:10.1145/2556464.2556469.

[3] N. Fujieda, T. Tanaka, and S. Ichikawa, "Design and Implementation of Instruction Indirection for Embedded Software Obfuscation," Microproc. Microsy., vol.45, no.A, pp.115–128, 2016. DOI:10.1016/j.micpro.2016.04.005.

[4] S. Ichikawa, T. Sawada, and H. Hata, "Diversification of processors based on redundancy in instruction set," IEICE Trans. Fundam., vol.E91-A, no.1, pp.211–220, 2008. DOI:10.1093/ietfec/e91-a.1.211.

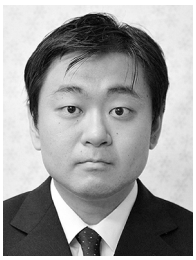
[5] G.S. Kc, A.D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," Proc. 10th ACM conference on Computer and communications security, Washington, DC, pp.272–280, 2003. DOI:10.1145/948109.948146.

[6] A. Monden, A. Monsifrot, and C. Thomborson, "Tamper-Resistant Software System Based on a Finite State Machine," IEICE Trans. Fundam., vol.E88-A, no.1, pp.112–122, 2005. DOI:10.1093/ietfec/e88-a.1.112.

[7] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, and S. Ioannidis, "ASIST: Architectural Support for Instruction Set Randomization," Proc. 20th ACM Conference on Computer and Communications Security, Berlin, Germany, pp.981–992, 2013. DOI:10.1145/2508859.2516670.

[8] G. Portokalidis and A.D. Keromytis, "Fast and practical instruction-set randomization for commodity systems," Proc. 26th Annual Computer Security Applications Conference, Austin, TX, pp.41–48,

2010. DOI:10.1145/1920261.1920268.
[9] G.E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: architecture for tamper-evident and tamper-resistant processing," Proc. 17th annual international conference on Supercomputing, San Francisco, CA, pp.160–171, 2003. DOI:10.1145/782814.782838.
[10] D.L.C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," Proc. 9th international conference on Architectural support for programming languages and operating systems, Cambridge, MA, pp.168–177, 2000. DOI:10.1145/378993.379237.
[11] S. Hines, J. Green, G. Tyson, and D. Whalley, "Improving program efficiency by packing instructions into registers," Proc. 32nd annual international symposium on Computer Architecture, Madison, WI, pp.260–271, 2005. DOI:10.1109/ISCA.2005.32.
[12] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls," Proc. 14th ACM Conference on Computer and Communications Security, Alexandria, VA, pp.552–561, 2007.
[13] "The PaX Team,". https://pax.grsecurity.net/, accessed June 20, 2017.
[14] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J.W. Davidson, "ILR: Where'd My Gadgets Go?," Proc. 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, pp.571–582, 2012. DOI:10.1109/SP.2012.39.
[15] Y. Chen, Z. Wang, D. Whalley, and L. Lu, "Remix: On-demand Live Randomization," Proc. 6th ACM Conference on Data and Application Security and Privacy, New Orleans, LA, pp.50–61, 2016. DOI:10.1145/2857705.2857726.
[16] T. Jackson, A. Homescu, S. Crane, P. Larsen, S. Brunthaler, and M. Franz, "Diversifying the Software Stack Using Randomized NOP Insertion," Moving Target Defense II, Advances in Information Security, vol.100, pp.151–173, 2013. DOI:10.1007/978-1-4614-5416-8_8.
[17] N. Fujieda, K. Sato, and S. Ichikawa, "A Complement to Enhanced Instruction Register File against Embedded Software Falsification," Proc. 5th Program Protection and Reverse Engineering Workshop, Los Angeles, CA, pp.3:1–3:7, 2015. DOI:10.1145/2843859.2843864.
[18] D. Sweetman, See MIPS Run Linux Second Edition, Morgan Kaufmann, Burlington, MA, 2006.
[19] 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," Proc. 2001 IEEE International Workshop on Workload Characterization, Austin, TX, pp.3–14, 2001. DOI:10.1109/WWC.2001.990739.
[20] S. Rhoads, "Plasma – most MIPS I(TM) opcodes,". http://opencores.org/project, plasma, accessed June 19, 2017.
[21] K. Chapman, Saving Costs with the SRL16E. White Paper WP271 (v1.0), Xilinx Inc., 2008.
[22] N. Fujieda and S. Ichikawa, "An XOR-based Parameterization for Instruction Register Files," IEEJ Trans. Electr. Electron. Eng., vol.10, no.5, pp.592–602, 2015. DOI:10.1002/tee.22123.



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.

Kiyohiro Sato received his M.E. degree in 2017 from the Department of Electrical and Electronic Information Engineering of Toyohashi University of Technology. Since April 2017, he is affiliated with dSPACE Japan K.K.

Ryodai Iwamoto received his B.E. degree in 2017 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, IEEJ, and IPSJ.