A Technique Preventing Code Reuse Attacks Based on RISC Processor

Yang Li*, Zi-bin Dai and Jun-wei Li
Zhengzhou Information Science and Technology Institute, Zhengzhou, China
*Corresponding author

Keywords: Code reuse attacks, Control flow integrity, Instruction extension, RISC.

Abstract. A full-process tag inspection system was designed and experimentally verified. This system based on RISC processors can defend code reuse attacks, as well as prevent high overhead caused by the implementation of software fine-grained control flow integrity technology. By extending memory tag, adding special memory-access instructions and setting up security rules, this design achieves hardware-based fine-grained control flow integrity, which can defend against attacks of ROP, JOP and COOP. The experimental measurement have been performed to validate that this design has effective effect on defending CRA with low overhead, which is evaluated on the RISC-V platform.

Introduction

A new type of attack called code reuse attacks (CRA) has appeared as the times require. Attacker does not need to attack computers by injecting external code. Instead, they attack computers by modifying the execution order of existing internal code within the program memory.

From the development of code reuse attacks, it can be seen that code reuse attacks have been gradually shifted from such CISC processors as x86 to ARM, RISC processors. RISC-based anti-CRA technologies have also been continuously developing and improving. In order to avoid the inherent security vulnerabilities of software and the high potential overhead imposed on the system, more and more defense methods are designed based on hardware. In 2015, Ye Yanqiu proposed a method to the ARM architecture that the white list of legal jump addresses is dynamically built to complete the control flow integrity (CFI) verification while the program is loaded. The method is purely software-level with a large amount of running overhead, but its ideas can be extended to hardware[1]. In 2016, JINYONG LEE proposed an anti-CRA hardware monitor with the properties of low-overhead and high-performance based on ARM processor [2]. Its disadvantage is that the process of parsing the binary code extracted from the ARM processor is very complicated, which greatly increases the design cost. In 2017, Pengfei Qiu proposed a new control flow integrity scheme based on lightweight encryption architecture (LEA-AES) encrypting and decrypting return addresses and instructions to defend against code reuse attacks, which cause a large clock delay [3].

This research implements the control flow integrity through hardware-assisted, instruction extension, and the establishment of a full-flow tag inspection system. According to the defensive idea of permission separation, the memory unit storing the control flow transfer information is protected by extending the storage unit tag bit. According to the security model, the processor pipeline is modified and special memory-access instructions are added simultaneously, so that at various stages of the pipeline, the extended tags are used to perform checks to prevent the hijacking of the control flow. Increased delays are avoided as well as the binary code is not need to externally extract and parse in the processor because the checks related to tag bits are in parallel with the original processor pipeline.

Fine-grained Control Flow Integrity Technology Based on Full-flow Hardware Tags

This research based on Harvard architecture RISC processor follows the typical threat model for most related work. The software may contain one or more memory leaks assumed in this paper.
Once triggered, the attacker can read or write any memory location. As a hardware-based solution, the location of vulnerabilities can be listed as follows: user-mode applications, system management program and so on. At the same time, we assume that all hardware components are trusted and flawless, so attacks exploiting hardware vulnerability are beyond the scope of this research. The sample application can be used as long as the assembler is updated, and this extension can be supported by inline assembly, modifying the compiler is not a requirement of this research.

The three typical code reuse attacks are selected as the objects of research, there are: return-oriented programming (ROP)[4], jump-oriented programming (JOP)[5] and counterfeit object-oriented programming (COOP)[6]. Studies have shown that these attack methods are Turing complete. The ROP attack uses the defect that the processor doesn’t check the correctness of the next instruction after the calling function returns, it continuously changes the execution order of the programs, and reconnects program fragments (set as "gadgets") that perform different functions by overwriting multiple return addresses to generate malicious attack behavior; JOP uses serial program fragments to attack, which is similar to ROP. However, JOP does not rely on the stack to complete the control of the program flow. It uses a register to generate a "fake pc pointer", which is the same as the normal pc pointer function. However, the attacker makes this "fake pc pointer" change by self-increment or self-decrement so as to change the execution order of the normal program to achieve attack; COOP utilizes the features of C++ object-oriented programming, and uses the virtual pointer in the class, which points to the virtual function table to make a series of virtual functions (set as "vfgadgets") complete the attack in the order set by the attacker.

Control flow integrity technology defends against code reuse attacks by monitoring program runtime behavior. The implementation of monitoring and the setting of rules are two key points. There are two main defects that usually exist for software control flow integrity: 1) High overhead caused by monitoring the implementation of software; 2) Complex pretreatment required by obtaining control flow graphs. This paper extends the security policy on hardware level and establishes a full-process tag inspection system, which realizes the low-cost fine-grained control flow integrity and makes it impossible for an attacker to achieve the purpose of attack through software vulnerabilities.

In memory, data and instruction are highly relevant and in reality. Data is the role that determines how instruction works. As shown in Figure 1, the memory model is from Harvard architecture RISC processor, whose data and instructions are stored separately. By reclassifying the contents of memory, there are merely two sections that attackers could possibly inject data, including ordinary data segment and data address segment of memory, because these data segments and instruction address are stored and managed indiscriminately in memory as conventional data are, leaving no logical or spatial separation, it gives an opportunity for attackers to modify the address of instructions. As above, we exemplify how CRA is realized, and to solve this problem, we propose to add special tag bits to the memory unit that stores the control flow transfer information and performs memory protection to prevent the control flow from being hijacked.

![Figure 1. Memory classification and attack principles.](image-url)
Hardware Assistance and Instruction Extension Based on RISC Processor

There are 31 general purpose registers (GPR) and 1 program counter \(pc\) in the RISC instruction set architecture of the experimental platform. The width of each register is 64 bits. It is agreed to use the register \(x1\) to store the return address and use the program counter \(pc\) to save the address of the current instruction. We allow each 64-bit data word to be accompanied by a 2-bit tag. Each instruction has a 1-bit tag so that it can align the data boundaries. By modifying the tag of the instructions to set the instruction as a sensitive or an ordinary instruction, the sensitive instructions can operate on sensitive and ordinary data; and the ordinary instructions can only operate on ordinary data. We extend two new instructions "loadtag" and "storetag" to load and store general purpose register tags from memory. Figure 2 shows their instructions formats. The extended data is 66 bits, so these two new instructions are to operate the 66-bit memory unit. Similarly, the level of them can be set by modifying their tag. The mnemonics are used to distinguish instruction level, for example, "loadtag1" and "storetag1" are sensitive instruction operations, and "loadtag0" and "storetag0" are the ordinary instruction operations.

<p>| | | | | | | | | | | | | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>loadtag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>storetag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2. Instruction format for adding new instructions.

Rule Design for Defending Code Reuse Attacks Based on Security Model

This article implements the Biba integrity model[7] to protect the integrity of data, such as return addresses, target addresses, and virtual function pointers, thereby resisting ROP, JOP, and COOP attacks. According to the principles established by the Biba security model, the use of "do not read down/write up" to ensure the integrity of the data, "do not read down" means that subject can't read data below its integrity level; "do not write up" means that subject can't write data above its integrity level. The Integrity level (IL) of the corresponding data is represented by using tag: the sensitive data is IL01, and the normal data is IL00. This design statically specifies some write operations that can operate sensitive data and allows these write operations to set the memory tag to IL01. Other common write operations can only set the tag to IL00. When loading sensitive data from memory, it checks if the tag is still IL01 and judges if the protected data has been tampered with. Take the ROP attack as an example, use the new instruction "storetag1" to store the return address and set its memory tag to IL01. When loading the return address from the memory returned by the function, use another instruction "loadtag1" to check if the memory tag is still IL01. Since normal store instructions can only set the tag to IL00, if the attacker tries to overwrite the return address, the "loadtag1" instruction will find the tag mismatch and generate a memory exception to fail the check. JOP attack is for jump instructions, jump can be either unconditional or conditional, in order to further strengthen the defense of JOP, this design protects the next instruction, that is, it checks if the tag of the program counter \(pc\) of the jump destination has been tampered with to prevent the control flow from being hijacked. When the jump target is executed, the corresponding instruction is also checked according to the address which is stored in the \(pc\) register; so control flow integrity is performed by preventing the control flow from being hijacked and protecting instruction integrity. For indirect jumps, because there is an offset, it is also necessary to consider whether the offset address is modified, that is, to ensure that the register holding the offset address is not modified.

This design also implements the Bell-LaPadula[8] confidentiality model. According to the principles established by the Bell-Lapadula security model, "do not read up/write down" is used to ensure the confidentiality of information. "do not read up" means that subject cannot read object above its confidentiality level, "do not write down" means that subject cannot modify object below its confidentiality level. By using tag to indicate the sensitivity level (SL) of the corresponding
information, for example, to protect sensitive information, we can set their tags to SL11 (secret level) and force all untrusted read operations (instruction tag is 0) can only read the information with the tag SL10 (normal level). This can effectively prevent attackers from searching for components and virtual function fragments that make up the code reuse attacks from the operating system or program. This increases the program's agnosticism.

Because the information flow of the above two models is reversed and the confidentiality and the integrity indicators of most of the subject and object are consistent, for example, high confidentiality information usually has high integrity and security features, simply combining the two model will make the system unusable. In order to solve this problem, this paper proposes a unified model of confidentiality and integrity that is dynamically adjusted through tag identification. As shown in Figure 3, integrity (IL) and confidentiality (SL) are used as two dimensions for tag identification. According to the historical process of data tag identification and instruction access rights, the system dynamically adjusts the current model rules and realizes the organic combination of BLP and Biba model.

![Figure 3. Rule design based on a combination of BLP and Biba models.](image)

**Architecture Design Based on Extension of RISC Processor**

In order to support the implementation of tag function to constitute a full-flow tag inspection system, we need to implement tag extension for instruction cache and data cache of L1 cache, L2 cache as a whole, all general purpose registers (GPR) and control status register (CSR) in Harvard-structured RISC processor. At the same time, the tag table is stored in physical memory, and a hardware tag cache module is added, so that fine-grained control flow integrity technology can be implemented to support full-flow hardware tags. Through hardware extension, the L1 instruction cache provides instructions and tags for the instruction decoder (ID) stage, the L1 data cache can store tags and data together.

![Figure 4. Extended design architecture based on RISC processor.](image)
Figure 4 shows the RISC processor-based design architecture. The gray part is the extended hardware tag cache module, whose specific structure is showed in the dashed box. The metadata array stores the TAG of the physical memory address. If the TAG of the metadata array hits, the tag cache module hits and the corresponding data and tags are taken out from the data array; if the TAG of the metadata array misses, the corresponding data and tags are directly taken out from the physical memory, and the data with tags will be stored in the data array while the metadata array will store the corresponding TAG for the next use. In order to reduce the high delay of accessing physical memory and multiple transactions can be processed in parallel, we introduce a transmitter to serve the transaction, and the transmitter can maintain the current state of each transaction.

The basic processor selected in this paper consists of five pipelines, which are fetching, decoding, memory and writing back. As shown in Figure 5, the design modifies the processor pipeline, multiple check modules consisting of a tag processing unit and a tag checking unit are added to each stage of the processor and L1 data cache pipeline. In order to allow tag propagation between register data and register tags, some multiplexers are added to the MEM stage. Each check module added to the processor pipeline performs an inspection function that can be enabled at runtime, and each function is controlled by a mask. The set of all masks is placed in a 64-bit register called tagctrl, which controls the execution of different check functions in different stages of the pipeline. It can perform instruction integrity check, jump to the target address (return address, virtual function pointer) check, pc pointer address check and memory load store check in sequence. When the tag check fails, an exception is generated, the CSR is set to 0x10 and pc points to the exception instruction, thus completing the anti-CRA.

Experimental Verification Analysis Based on RISC-V Platform

By extending the RISC-V[9] instruction set architecture, the corresponding memory unit is extended with tag, a 64-bit tagctrl register is added to the register file, and a 64 KB hardware tag cache module is added between the L2 cache and DDR. Four check modules are added to the pipeline—an instruction check module, a jump check module, a pc check module, and a load store check module. We implement an anti-CRA processor architecture prototype and perform defense function verification and resource overhead analysis on the Zedboard development board. The sample programs are compiled in Ubuntu 16.04 GCC 5.2.0 and binutils 2.25 environment.
Defense Function Verification

RIPE benchmark[10]: Runtime Intrusion Prevention Evaluator (RIPE) is an open source intrusion prevention benchmark. The benchmark is a synthesized C program that tries to attack itself in a number of ways. This paper ported 54 attacks in the RIPE of x86. The ported attacks can cause any memory location to be buffer overflowed and make any target code pointer except the frame pointer be attacked, which meet the requirements of the defense function test. Through experimental verification, this design can prevent 54 all attacks. Figure 6 shows the experimental results of the test for ported ROP attacks. The command parameters for generating ROP attacks are:

```
./fesvr-zynqpkripe_attack_generator -t direct -i rop -c ret -l stack -f memcpy
```

comments:

fesvr-zynq: A front-end server running on the Zedboard ARM core to communicate with the RISC-V processor;
pk: RISC-V agent kernel;
ripe_attack_generator: RIPE benchmark attack generator.

As shown in Figure 6, the experimental result is "user tag mismatch segfault", which proves that the attack was found and caused an exception.

![Figure 6. Experimental result of ported ROP attack.](attachment:image.png)

VTable Hijacking: This paper develops a simple attack that overwrites the vtable pointer with a fake one, so the next invocation of the virtual function will invoke the attacker controlled function. Without this design protection, the simple vtable hijacking attack can call malicious virtual functions. Through our virtual pointer protection mechanism, we can prevent the attacker's virtual pointer from loading and achieve COOP protection.

Resource Overhead Analysis

Based on the synthesis results, the paper uses Slice LUTs, Slice Registers, and Muxes to quantify the resource which is required to implement a full-flow tag system on the Zedboard. As shown in Table 1, the resource overhead of this technique causes 9.01% of Slice LUTs, 11.80% of Slice Registers and 12.06% of Muxes. Compared with the external hardware monitor designed by JINYONG LEE, our design greatly reduces the resource overhead.

<table>
<thead>
<tr>
<th>Site Type</th>
<th>Hardware Added</th>
<th>Total Baseline</th>
<th>Hardware Added Over Total Baseline</th>
<th>External Hardware Monitor Over its Total Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice LUTs</td>
<td>5664</td>
<td>62839</td>
<td>9.01%</td>
<td>22.71%</td>
</tr>
<tr>
<td>Slice Registers</td>
<td>3230</td>
<td>27382</td>
<td>11.80%</td>
<td>15.00%</td>
</tr>
<tr>
<td>Muxes</td>
<td>320</td>
<td>2653</td>
<td>12.06%</td>
<td>13.79%</td>
</tr>
</tbody>
</table>

Table 1. Synthesis results comparison.
Conclusion

This paper mainly studies a kind of anti-CRA defense technology based on RISC processor—full flow tag inspection system. Through memory tag extensions, a low overhead fine-grained control flow integrity is achieved on the hardware by adding special memory-access instructions and formulating security rules. The verification and evaluation is completed on the RISC-V platform. The experimental results show that the design can effectively resist code reuse attacks by adding a low resource overhead. It is expected that our proposed design may find a potential applications in reality, such as security processor, trusted computer, secure system on chip (SoC) and so on.

References


