2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

Enhancing Software Dependability and Security with Hardware Supported Instruction Address Space Randomization

Seung Hun Kim*, Lei Xu†, Ziyi Liu†, Zhiqiang Lin‡, Won Woo Ro*, and Weidong Shi†

*School of Electrical and Electronic Engineering, Yonsei University, Seoul, Republic of Korea.
Email: kseunghun@gmail.com, wro@yonsei.ac.kr
†Department of Computer Science, University of Houston, Houston, TX, USA.
Email: xuleimath@gmail.com, ziyiliu@cs.uh.edu, larryshi@cs.uh.edu
‡Department of Computer Science, The University of Texas at Dallas, Dallas, TX, USA.
Email: zhiqiang.lin@utdallas.edu

***Abstract*—We present a micro-architecture based lightweight framework to enhance dependability and security of software against code reuse attack. Different from the prior hardware based approaches for mitigating code reuse attacks, our solution is based on software diversity and instruction level control flow randomization. Generally, software based instruction location randomization (ILR) using binary emulator as a mediation layer has been shown to be effective for thwarting code reuse attacks like return oriented programming (ROP). However, our in-depth studies show that straightforward and naive implementation of ILR at the micro-architecture level will incur major performance deficiencies in terms of instruction fetch and cache utilization. For example, straightforward implementation of ILR increases the first level instruction cache miss rates on average by more than 9 times for a set of SPEC CPU2006 benchmarks. To address these issues, we present a novel micro-architecture design that can support native execution of control flow randomized software binary while at the same time preserve the performance of instruction fetch and efficient use of on-chip caches. The proposed design is evaluated by extending cycle based x86 architecture simulator, XIOSim with validated power simulation. Performance evaluation on SPEC CPU2006 benchmarks shows an average speedup of 1.63 times compared to the hardware implementation of ILR. Using the proposed approach, direct execution of ILR software incurs only 2.1% IPC performance slowdown with a very small hardware overhead.**

***Keywords*—Instruction location randomization, micro-architecture, code reuse attack, software security**

I. Introduction

Code reuse attacks allow the adversary to make malicious results by exploiting control flow in the existing program without any additional code injection [1], [2], [3]. Return oriented programming (ROP) attack is an representative example. Using ROP, the attacker can link small pieces of code which is known as gadgets, that already exist in the binary image of a vulnerable application. In fact, the ROP gadgets are short sequences of code, typically ending with a return or indirect control transfer instruction. Instead of injecting binary code into the memory space of an application, the attacker can use a sequence of gadget in the stack or other memory areas of the program. Each gadget ends with an indirect control transfer instruction, which transfers the control to the next gadget according to the injected gadget sequence. During the attack, the adversary can circumvent many defenses such as read-only memory [4], non-executable memory [5], kernel code integrity protections [6] since the injected part is only data (rather than code). In addition, access to ROP exploits is not difficult since they are provided in the publicly available packs [7].

Most existing defense mechanisms, such as instruction set randomization [8] or simple address space layout randomization (ASLR) cannot prevent code reuse attacks. For this reason, many solutions have been proposed to mitigate the risks of code reuse attack [9], [10]. Recently, approaches at micro-architecture level are also presented for detecting control flow violations or monitoring control flows at runtime using hardware support [11], [12], [13]. Among the previously proposed concept, instruction location randomization (ILR) is widely used due to the effectiveness of the method [14].

Pappas et al. described an in-place code randomization approach to mitigate ROP exploits by applying ILR within basic blocks [10]. The binary transformations include reordering instructions within the basic block boundaries without changing execution results. On the other hand, Hiser et al. presented complete ILR [9]. The solution completely randomizes the location of every instruction in a program. Consequently, it can thwart an attacker's ability to re-use the existing program gadgets (e.g., ROP based exploits, arc-injection attacks). In-place code randomization and complete ILR are software based approaches for mitigating gadget based exploits. ILR and other similar approaches often rely on heavy-weight runtime instrumentations or exotic binary emulation frameworks that can incur significant overhead. For minimizing such overheads, most software based approaches support either partial ILR (e.g., randomization within basic block boundary) or randomization with reduced scope to achieve acceptable performance. As more variations and less predictable control flow will increase resilience to remote attacks. In fact, Snow et al. pointed out that 32 bit address space is hard to provide enough entropy to protect systems from just-in-time code reuse attack [15]. In this case, applying a 64 bit address space can be a solution by increasing the entropy of the randomization [14]. However, partial randomization approach cannot take full advantage of increased address space while the complete randomization does. Consequently, complete ILR provides higher level

978-1-4799-8629-3/15 $31.00 © 2015 IEEE
DOI 10.1109/DSN.2015.48

IEEE computer society

security benefits by maximizing resilience with completely randomized control flows.

Besides performance and efficiency reasons, another major benefit of supporting fine grained instruction layout or control flow randomization at micro-architecture level is reduced attacking surfaces by removing the binary emulation layer. In fact, software based approach is less secure because malicious attackers can target the software emulation or interpretation layer. Similar to how out-of-order execution is hidden from the OS and compiler, our micro-architecture based approach hides the actual instruction space and minimizes the interface exposed to the hackers (attacking surface).

According to our studies, straightforward implementation of complete ILR at hardware level has major performance issues with some of the established design principles of modern micro-processor. When compared with the baseline architecture with identical cache organizations, straightforward implementation of ILR will increase the first level instruction cache miss rates since the instructions are widely spread among the memory space. That is, if ILR is to be integrated with native execution support, one has to come up with a new approach that can meet both the goals of maximizing randomness of instruction layout and efficient software execution. Last but not the least, it is a fact that software based approach suffers from lack of adoption by the end users as many companies and web services don't use the most secure software (sometimes due to cost and lack of knowledge/incentive) or apply security patches. When attack resilience is built into the micro-architecture, it can eliminate the adoption barrier by providing default attack resilience integrated with the hardware itself. To achieve all these goals and benefits, we propose micro-architecture based solution for complete ILR in this paper. Specifically, we make the following contributions:

- Introducing the performance problem of instruction fetch caused by native support of randomized instruction layout and the need for new solutions that can support software diversity with minimal impact on performance;
- Presenting the novel architecture design of one such solution that can support native execution of control flow randomized software binary and at the same time preserve the performance of instruction fetch and efficient use of on-chip caches;
- Proposing a novel control flow randomization concept that uses a lightweight mediation layer to create randomized view of instruction space without destroying instruction locality at memory hierarchy; and
- Demonstrating the effectiveness of the proposed approach using cycle based architectural model and SPEC benchmarks.

II. THREAT MODEL OF GADGET BASED EXPLOITS

Our threat model is defined as the following. An attacker is attempting to subvert a remote system via gadget and ROP[1] exploits. Software applications are distributed to the end user in binary format, and then randomized. The binary application

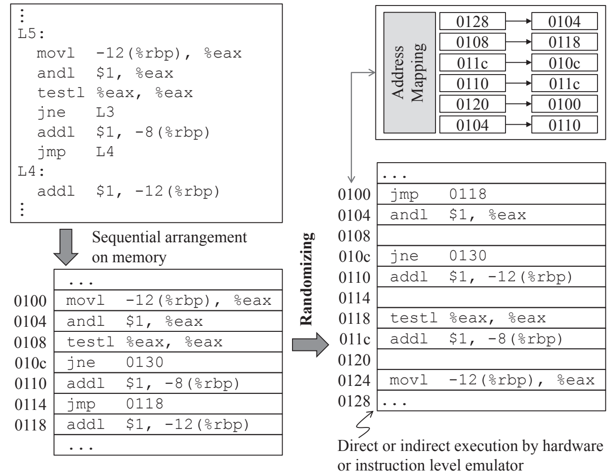

Fig. 1. Instruction space and control flow randomization of ILR. The mechanism provides control flow randomization for reducing the attack surface of gadget based exploits. Existing software based ILR uses instruction level emulator to support execution of randomized instructions.

has been tested, but not guaranteed to be vulnerability free. The program may contain weakness that can be exploited by ROP based attacks. However, the application is assumed to be free from back doors or trojans. Furthermore, we assume that there is no insider attack and the system is managed by trusted administrators. The attacker does not know/see the executable version of the binary code after randomization is applied. As such, the attacker can only launch a kind of *random attack* because the attacker can neither see (due to the lack of privilege) nor run the randomized code (because the attacker does not have physical access to the system controlled by the system administrator). Moreover, the attacker cannot observe the instruction-by-instruction state change from the operating system. Our threat model mainly focuses on attacks where applications are subverted by processing malicious data submitted by the attacker. The data may contain ROP exploits. The threat model covers a wide range of exploits, such as attacks to client-server based program, exploits to document viewers, browsers, network clients, etc. According to the previous research in the literature, randomization/diversification can effectively mitigate a wide range of security attacks because of reduced attack surface. Though this paper focuses more on the security risks associated with ROP, the proposed micro-architecture facilitated complete ILR increases a computing system's resilience against many exploits beyond ROP.

III. MOTIVATION AND APPROACH OVERVIEW

Native support of fine grained ILR has many advantages than the existing software based approaches. One main benefit of integrating ILR with micro-architecture is improved performance by maintaining the efficiency of on-chip cache access.

To execute binary programs randomized by ILR such as the one shown in Figure 1[2], a special virtual machine that decodes the randomized instruction sequences at runtime is required. Figure 2 shows that when ILR is implemented on an

[1]In this paper, we decide to use ROP as a representative code reuse attack method. Details are explained in Section V.

[2]In this example figure, we assume that all instructions are 32 bit and the memory address is 16 bit. For the simplicity, we apply same assumption for the rest of the figures which show instructions and memory space.

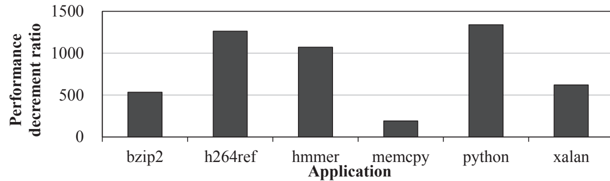

Fig. 2. Performance of implementing instruction level randomization using an instruction level machine emulator. Y-axis shows the amount of performance decrease against native execution on bare metal. The execution time increases by over hundred of times.

instruction level emulator, applications suffer from the hundreds of times slower than native executions on the bare metal CPU hardware. Although a certain optimization technique can be applied to improve performance of emulation based ILR approaches, the emulation layer without hardware support is bound to incur significant performance penalty. It should be pointed out that some software based implementation reduces the overheads by supporting ILR with limited scope. It will be not fair if one compares more efficient micro-architecture based approach with these schemes because they don't support complete randomization and de-randomization of instruction space at per instruction execution level. Our apples-to-apples comparison shows that, since randomized binary cannot be executed natively, a run-time interpreter that de-randomizes the instruction space at per instruction level will certainly incur much higher overhead and significantly reduce instruction fetch efficiency. For the studied benchmark applications, ILR can increase instruction L1 cache miss rate by more than five times on average.

A. Randomization vs. Efficient Instruction Fetch

One possible approach is to integrate the capability for direct execution of ILR randomized programs with native micro-architecture support. A simple implementation is to remove the runtime emulation layer and push its functionality into the processor. However, our in-depth studies reveal that such solution has major performance issues with the design principles of modern micro-processor. When compared with the baseline architecture with identical cache organizations, a naive implementation of ILR increases the L1 instruction cache miss rates on average by 9.4 times for 11 applications from SPEC CPU2006 benchmark suite, see Figure 3. In addition, instruction level address space randomization significantly impedes the efficiency of hardware based instruction pre-fetcher as shown by the results in Figure 3. The pre-fetch miss rates of L1 instruction cache (IL1) increase by on average 28% for the tested SPEC CPU2006 benchmarks. The reduced L1 cache efficiency is propagated to the next level cache by adding workload pressures. As shown in the figure, the unified L2 cache experience 36% of average increased loads for the tested SPEC CPU2006 applications; the amount of pressure is measured by the number of read operation from L1 cache to L2 cache.

Combining all the effects, the overall CPU performance decreases dramatically. The average IPC reduces to 61% of the baseline IPC with identical architecture settings, as shown in Figure 4. The naive implementation assumes that CPU can resolve address mapping with zero cost. Therefore, the performance penalty is entirely due to the randomization of instruction space.

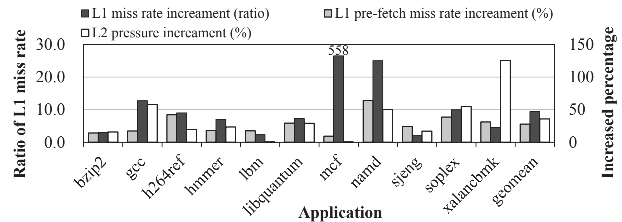

Fig. 3. The impact of naive approach on the L1 and L2 cache. In this mode, a processor directly executes a binary program with randomized layout. The program is randomized using the complete ILR approach which is described in Hiser et al.'s work [9]. CPU setting: 32KB IL1 and 512KB L2, XIOSim [16] and Zesto [17].

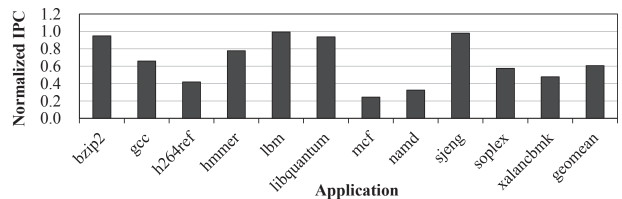

Fig. 4. Performance of straightforward implementation of ILR at micro-architecture level. Y-axis shows IPC decrease for SPEC CPU2006 applications. The value of IPC is reduced to 66% of the baseline IPC which has no ILR.

B. Approach Overview

Implementing ILR with hardware support can either remove the emulation layer and therefore, can reduce its overhead significantly. However, a negative consequence is that a naive and straightforward implementation of ILR at architectural level destroys instruction fetch locality and renders many micro-architecture components optimized for efficient instruction fetch useless. Instruction fetch plays a critical role for feeding the pipelines of a high performance micro-architecture with instructions. If ILR is to be integrated with native execution support, a new design different from the naive ILR implementation needs to be adopted.

Our solution is to satisfy the seemingly contradicting requirements of native support for ILR and instruction fetch locality (maintaining temporal and spatial localities in instruction fetch). For that purpose, we introduce an address space randomization/de-randomization interface before the instruction fetch requests are handled by the on-chip L1 instruction cache.

Analogous to the difference between physical memory space and virtual memory space, we separate randomized instruction space and de-randomized instruction space (the original virtual memory space for storing instructions). The instruction execution pipelines handle instructions in randomized instruction space and directly execute control flow randomized binary program. For efficient and high throughput instruction fetches, on-chip caches and memory hierarchy store instructions in the original layout. The instruction fetch unit and memory hierarchy create an illusion to the processor pipeline that instructions are fetched and program control transfers in randomized instruction address space. Such a design can meet both the needs of native support for executing randomized binary and efficient instruction fetch.

We name this approach virtual control flow randomization (VCFR), which means that the processor executes instructions

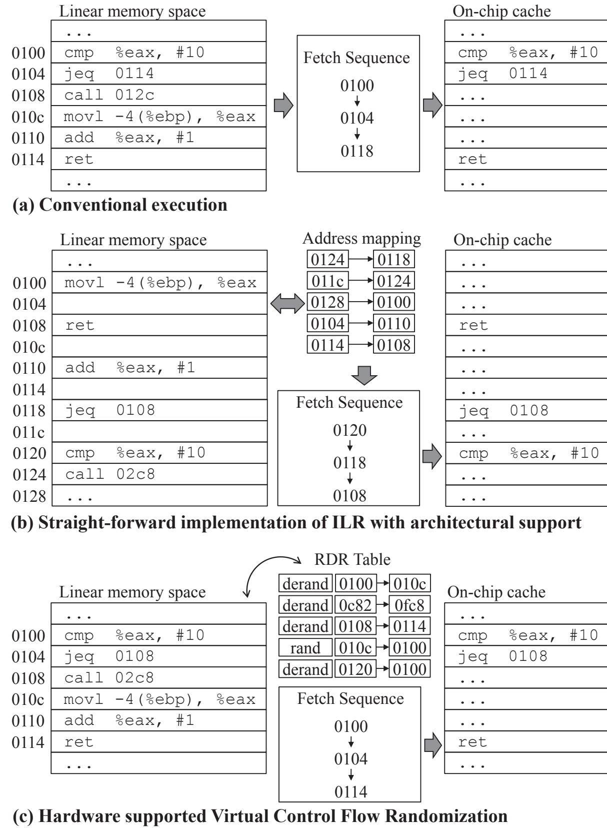

Fig. 5. Comparison of instruction fetches and memory layout in three settings: no randomization applied, ILR with straightforward architectural support, and hardware enabled virtual control flow randomization (a control flow randomization mediation layer creates randomized view of instruction address space to the processor execution pipeline and the instructions are stored in the memory hierarchy with preserved locality).

in virtually randomized instruction address space. From the perspective of randomizing instruction space and control flow, virtual control flow randomization achieves the same randomization effect as the original ILR [9], which uses a software virtual machine and rewrite rules to execute an ILR randomized binary or a straightforward hardware implementation of ILR. A main difference between our virtual control flow randomization (VCFR) and ILR is that in our approach, instruction locality is preserved. In VCFR, the control flow of a binary executable is randomized similar to ILR. This randomized view of control flow is presented to the processor execution pipeline. However, the binary instructions are still stored in the memory hierarchy (both on-chip caches and off-chip memory) in the original layout, which effectively preserves the instruction locality. A control flow randomization layer is situated between the processor pipeline and the memory hierarchy. The control flow randomization layer is light-weight as it is implemented as lookup tables that convert instruction address in randomized control flow address space to the original instruction address space or vice versa.

Figure 5 illustrates instruction space layouts in the memory hierarchy and instruction fetch sequences (assuming that instructions are fetched in program order) under three different scenarios, original unmodified binary executable, ILR transformed binary with direct hardware execution support, and our virtual control flow randomization approach. As shown in the figure, in both ILR and VCFR, control flow of the binary is randomized. However, in VCFR, instruction layout is preserved. A virtual instruction space layout (randomized) is presented to the processor execution pipeline. The processor pipeline uses randomized control flow for fetching instructions from the memory hierarchy. Some main differences between straightforward ILR implementations and VCFR are highlighted in Table I. Details of the architecture design will be presented in the next section.

TABLE I. DIFFERENCES BETWEEN STRAIGHTFORWARD ILR AND VCFR

	No Randomization	Naive Hardware Support for ILR	Our Approach (VCFR)
Execution	no control flow randomization	randomized control flow	randomized control flow
Instruction locality	preserved	destroyed	preserved
Instruction prefetch	effective	not effective	effective
Control flow diversity	no diversity	diversified	diversified

IV. Architecture and Design

A. Control Flow Randomization

Figure 6 shows the processes and steps performed by instruction level address space randomization. The randomization software, a binary rewriter, takes a third-party program as input and generates a new binary output that with randomized instruction layout. The new binary is semantically identical to the original one but using randomized control flows and instruction layouts. For ROP based exploits, the consequence of instruction space and instruction level control flow randomization is that the existing gadgets cannot be found any more. The large randomization space prevents an attacker from mounting a gadget based attacks [9].

The first step is to disassemble the binary image and perform offline static analysis. For such purpose, we use IDA Pro, a recursive descent disassembler [18]. For complete scan of disassembled code, we also use objdump. The control flow randomization software constructs CFG (control flow graph) from the disassembled program binary. Basic blocks are the nodes in the CFG. They consist of a sequence of instructions. Once the code is disassembled, basic blocks are easily detected with the leader algorithm. Relocation information can also be obtained. Entry points of basic blocks include all targets

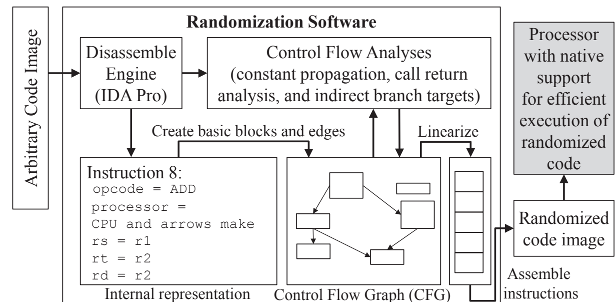

Fig. 6. ILR and architectural support for efficient execution of the randomized binary executable.

of direct control flow transfers, and all instructions directly following direct or indirect transfers.

Detecting the edges for the CFG is straightforward. One can get them directly from the disassembled codes for direct control flow transfers that encode their targets in the binary instruction itself. For indirect control transfers, we use a conservative approach at the beginning and assume that all the instructions at relocatable addresses can be the targets. This means to connect all indirect control flow transfer instructions with all possible (relocatable) targets when CFG is initially constructed. Then edges are analyzed and pruned using the established techniques described in [19], [20], [21], which apply an array of analyses. Fall-through edges are added to all basic blocks ending with an instruction that does not unconditionally transfer control.

Indirect control transfer using constant code address can be analyzed with constant propagation [19], [21]. Constant code address propagates over the CFG with instructions as producers of the code addresses (e.g., fetched from constant data segment) and indirect control transfers as the consumers. A simple but effective heuristic used in Hiser et al.'s work is to perform a byte-by-byte scan of the program's data, and disassembled code to determine any pointer-sized constant which could be an indirect branch target [9]. As shown in their work, this easy to implement approach is often sufficient. In our approach, we use both the same heuristic and a simple constant propagation analysis to recover relocation information for indirect control transfers that use constant code addresses. The analysis is performed on registers over the CFG. Our analysis does not modify any instructions that compute code addresses. The assumption is that the code addresses in the original program are relocatable. At the stage when the CFG is re-assembled into a binary image, the relocation information provides sufficient details on how code address computations need to be adapted. Analyses from us and others show that code address computations are rare in real world applications. To be conservative, we don't directly modify the code address computation.

After analyzing all the control transfer instructions, the ILR software will traverse the instructions and assign different addresses in the code space. To ensure that the control transfer instructions still branch to the correct locations, the randomization software will modify the direct control transfer instructions with the new target addresses. Jump tables and code addresses stored in the data sections are modified accordingly as well based on both reallocation information and results of indirect control transfer analysis. However, it is not feasible to completely resolve target addresses of all the indirect control transfers. Table II shows static number of indirect control transfers compared with the static number of direct control transfers in some SPEC CPU2006 benchmark applications. In this table, indirect control transfers include both control transfers from registers and computed control transfers. Also, indirect function calls include calls from registers and calls using computed function addresses. To provide a failover solution for indirect control transfer instructions, we use an approach similar to the work of Hiser et al. [9], which allows certain indirect control instructions to use the original target addresses. This means that for safe execution, some indirect branches may jump to the un-randomized address space. This can be supported by adding entries to the randomization/de-randomization table. The tables also contain address mapping entries to redirect program execution back to the randomized control flow space for continued execution after jumping to un-randomized addresses. To reduce attacking surfaces and prevent frequent jumps to the un-randomized addresses, we mark each safely randomized instruction address with a single bit tag in the randomization/de-randomization table (named randomized tag). For an un-randomized address, when its tag is set, execution control is prohibited from jumping to that location. As a result, ROP attacks can only be mounted by exploiting gadgets in the un-randomized addresses.

TABLE II. STATIC ANALYSIS OF CONTROL FLOW FOR SPEC CPU2006 BENCHMARK APPLICATIONS.

Applications	Direct control transfers	Indirect control transfers	Function calls	Indirect function calls
bzip2	27277	654	4474	654
gcc	149512	1464	51933	1605
h264ref	38650	884	6986	1409
hmmer	35438	556	7783	751
lbm	26074	620	4300	622
libquantum	27129	546	4686	636
mcf	25607	512	4214	582
namd	33497	618	5958	906
sjeng	30021	585	5280	709
soplex	49577	1271	15673	2587
xalan	126790	2915	63965	15465

To randomize `call` and `return` instructions, our control flow randomization approach analyzes the `call` instructions in a program to determine if the return address can be safely randomized. When the randomization approach decides that it is safe to randomize the return address, we have two options, one software based and one architecture based. One option is to rewrite the `call` instruction and replace it with a sequence of equivalent instructions that push a randomized return address to the stack. For instance, a `call bar` can be converted into two instructions, `push randomized_return_addr;` followed by a jump instruction, `jmp bar`. This approach expands size of the original program.

The second approach is to support randomized return address automatically by implementing at architecture level the mechanism to push the corresponding randomized return address to the stack. This option has the advantages of being fully transparent to the randomized binary program and at the same time maintaining the constant size for all the `call` instructions with randomized return addresses. In details, assume that a `call` instruction at address X, `call bar`, is executed by the pipeline and the un-randomized return address is X+4. When the instruction is executed, the processor core will look up the randomization/de-randomization table to find out the randomized return address corresponding to X+4. Then the processor core will push the randomized return address instead of the original return address into the stack. When this approach is applied, for each randomized return address, the randomization/de-randomization table stores an entry that maps the un-randomized return address to the randomized return address. It is the randomized return address that is pushed to the stack. For both options, when the function returns to the caller, either randomized or un-randomized return address will be popped from the stack and used as the next program counter. If the popped address is a randomized return address, it represents an instruction address in the virtual control flow address space. When the instruction is fetched, the address will be de-randomized first using the randomization/de-randomization table and then the

de-randomized address will be used for accessing the memory hierarchy. If the popped address is un-randomized address, the randomization/de-randomization table contains an entry indicating that the address is un-randomized by clearing the randomized tag. In this case, the next instruction after return will be fetched using un-randomized address.

Similar to what are reported in the prior study, not all return addresses can be safely randomized; as an example, return addresses associated with indirect call are not randomized [9]. Furthermore, for x86 based binary, a `call` instruction is often used for purpose other than invoking a subroutine. For example, to support location independent code or data, it is common to read the value of the instruction pointer (since, by definition, the relative address is relative to the instruction's location). However, there is no instruction to obtain the value of the instruction pointer on x86, a simple solution is to execute a `call` instruction with the next instruction address as the target location, which causes the target address to be saved on stack. Then the next instruction can read the pushed address from the stack by moving it to a register (e.g., `ebx`). This can be achieved by using either a `pop` instruction or a `mov` instruction. In this case, randomizing return address may cause problems for the location independent code or data. Another example is C++ exception handling [22]. In C++, the exception handing routines use the return address to find out the exception handling codes by walking through the stack. This is because at compile time, the C++ compiler cannot decide if when a function makes a call to another function, the callee will throw out an exception or not. As a result, the compiler will put the exception cleanup code into the caller. To randomize return address for C++ program with exception handling, there are two choices. One is to modify the exception handling tables to match with the control flow randomization. This is the approach used by Hiser et al. [9]. A second approach is to modify the processor architecture in such a way when exception happens, the original un-randomized return address will be returned to the exception handling routines. The details can be found in Section IV-C.

After all the instructions are relocated, the randomization software will reassemble the instructions into a new binary image. The new binary contains randomized control flow at instruction level and the associated randomization/de-randomization tables. A processor with virtual control flow randomization support can execute the program using the new binary image and the associated tables.

B. *Micro-architectural Support for Executing Randomized Binary*

There are many advantages of supporting our control flow randomization approach at micro-architectural level. These include, (i) eliminating the virtual machine that is often required for software based ILR and consequently reducing the attack surfaces; (ii) improved efficiency due to native and direct execution of the control flow randomized instructions; and (iii) better performance of memory hierarchy (e.g., cache miss rate, pre-fetch efficiency) brought by the concept of virtual control flow randomization (executing a binary program in randomized control flow instruction space but storing the program in the memory hierarchy using un-randomized instruction memory layout).

At micro-architecture level, the processor maintains a randomization/de-randomization layer that bridges the two instruction memory spaces (control flow randomized versus un-randomized). This mediation layer creates a virtual view to the processor pipeline that the instructions are fetched and executed in randomized instruction space. Depending on the contexts, an instruction address may be randomized (e.g., retrieve a randomized return address from a `call` instruction) or de-randomized (e.g., fetching from level 1 instruction cache using a randomized address). The system can maintain mapping tables to store entries for randomization and/or de-randomization. Similar to page tables, the tables for randomization and de-randomization are stored in the kernel as part of the process context and protected from illegitimate accesses. They cannot be accessed or modified by the application process in user space.

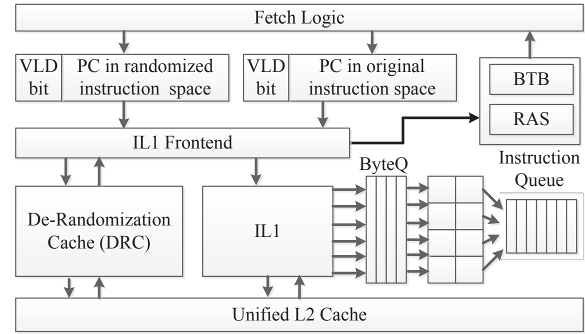

Fig. 7. Block diagram of instruction fetch in randomized instruction space. There are two program counters (PCs), one in the randomized space (RPC) and the other one in un-randomized space (UPC). At architecture level, control flow follows the PC in the randomized space. Both PCs are used for instruction fetch. When the original PC is absent, hardware will de-randomize RPC by looking up the DRC lookup buffer. DRC is a small cache that stores address pairs for both instruction address randomization and de-randomization. DRC shares L2 with IL1.

At run time, entries of the randomization/de-randomization tables can be cached on-chip using a DRC lookup buffer (de-randomization cache). The DRC lookup buffer stores frequently accessed randomization and/or de-randomization translation entries, see Figure 7. The DRC lookup buffer can be implemented as a cache (e.g., directly mapped). It acts as a mediation layer between processor execution pipeline and the memory hierarchy. There can be two buffers, one for randomization and the other one for de-randomization. For more efficient usage of silicon resources, we use one unified lookup buffer for storing entries of randomization and de-randomization. For each entry, there is a single bit tag (derand tag) indicating what kind of translation entry is stored. If the tag is set, the entry is used for de-randomizing a randomized address. Otherwise, it is used for randomization. In addition, there is a valid bit. When the valid is clear, the entry is not occupied.

Furthermore, for efficient program execution, the instruction fetch unit contains two program counters (PCs), one for the randomized instruction space (RPC) and the other one for the un-randomized space (UPC). This means that UPC always stores the de-randomized address of RPC. For un-randomized address, UPC and RPC are the same. For un-randomized entry, the DRC lookup buffer contains a tag to indicate that the address is not randomized. At architecture level, control flow follows the PC in the control flow randomized space. Both PCs are used for instruction fetch. When UPC is absent, the

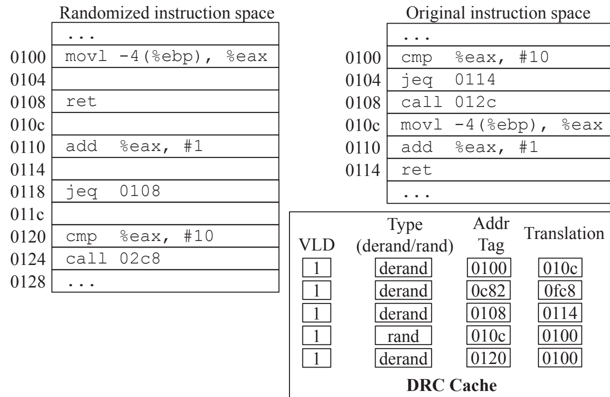

Fig. 8. DRC cache organization and its usage for translating control flow addresses between randomized and the original instruction space. Instructions are stored in on-chip caches using the original layout, which results in better cache performance than directly storing them in randomized space. For a randomized PC (RPC), DRC converts the address into the original location. The figure shows an example program in both randomized and the original space. DRC stores necessary address translation pairs for executing the program.

micro-architecture will de-randomize the address in RPC by looking up the DRC lookup buffer.

Instructions are stored in on-chip caches using the original layout as shown in Figure 8, which results in better cache performance than storing them in randomized instruction space. For a randomized PC (RPC), DRC converts the address into the original address. Figure 8 shows an example program in both randomized and the un-randomized space. DRC stores necessary address translation pairs for executing the program. Note that although the program is stored using the original layout, the control flow is modified. In addition, UPC cannot be directly accessed by the instructions. It is automatically updated by converting RPC or by the fetch unit.

Since DRC lookup buffer is on-chip, it has limited space. Consequently, not all entries for randomization and de-randomization can be stored in the DRC lookup buffer. One option is to include a larger level two DRC lookup buffer. However, for efficient usage of cache space, DRC can share its second level cache with the unified L2 of a processor core, which is our current design. For an address that needs to be de-randomized or randomized, if the corresponding translation entry cannot be found in the DRC lookup buffer, the processor core will search the next level memory hierarchy until the entry is fetched. Such design eliminates the necessity of trapping into the kernel when entries of the DRC lookup buffer need to be updated. However, it requires the tables for storing randomization and de-randomization address translations to be stored in paged memory. Dedicated memory pages can be used to store these tables. We designed DRC as direct mapped cache with small size to minimize power consumption. If there is a DRC miss, L2 cache will be searched. The often small size directly mapped DRC cache consumes very small amount of energy. The design doesn't require a fully-associative DRC since the miss penalty is marginal; we will show the simulation results in Section VII. In addition, the DRC is hidden from the user space program with a simple extension to TLB.

To prevent any potential tamper of these tables by instructions executed under the application's context, these pages can be made invisible to the user space instructions. This means that during execution of an application, these address translation tables can only be accessed by the micro-architecture for the purpose of handling access misses of DRC lookup buffer. To modify these entries using instructions, the system needs to switch to the kernel mode. A simple implementation of this protection is to extend each entry of the TLB (translation lookaside buffer) with a new page visibility bit. For a page, if the visibility bit is set, it means that contents stored in the page can be accessed by the user space instructions. Otherwise, it indicates that the page is invisible to the application instructions. The randomization and de-randomization translation tables are stored in such pages invisible to the user space instructions.

According to the design, DRC cache lookup is only needed when there is a randomized control flow transfer and the randomized target address requires de-randomization, which is infrequent because branch prediction is performed using the original memory space. Additionally, even there is a DRC miss, majority of them can be found in the L2 cache, which is large enough for storing the DRC table.

C. Support for Return Address Randomization

As discussed earlier, there are many scenarios that return addresses cannot be safely randomized. It is not uncommon that instructions in the callee directly access the return address stored in the stack. One example is using a `call` instruction to find out the current program counter for implementing location independent code or data. Figure 9 shows for some SPEC CPU2006 benchmark applications, the number of functions with and without `return` instructions contained in the callees. If a randomized return address is pushed to the stack, and succeeding instructions directly access the randomized return address and use it for computing address of location independent code or data, it may lead to faulty execution. A conservative approach is to apply return address randomization only when the control flow randomization software is certain that a caller follows the conventional call return pattern. The downside of this approach is that it reduces the potential randomness of the result binary image.

To maximize the chance of return address randomization, we introduce an micro-architectural enhancement as shown in Figure 10 that allows randomized return address to be saved to the stack even when it may be directly accessed for supporting C++ exception handling or location independent code/data. When a randomized return address stored in the stack is directly fetched into a register, the micro-architecture will automatically de-randomized it by looking up the DRC buffer. Such design provides compatibility for address calculation during C++ exception handling and usage of return address

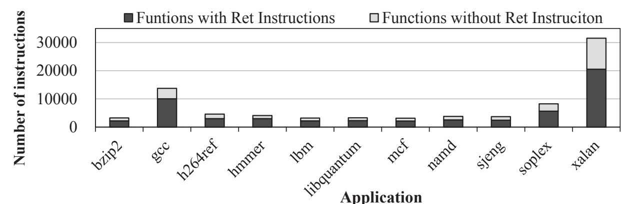

Fig. 9. Static analysis of function calls and returns for SPEC CPU2006 benchmark applications. The figure shows, for each benchmark, numbers of functions with and without `return` instructions (functions without `return` instructions may return to the caller using other x86 instructions such as `mov`).

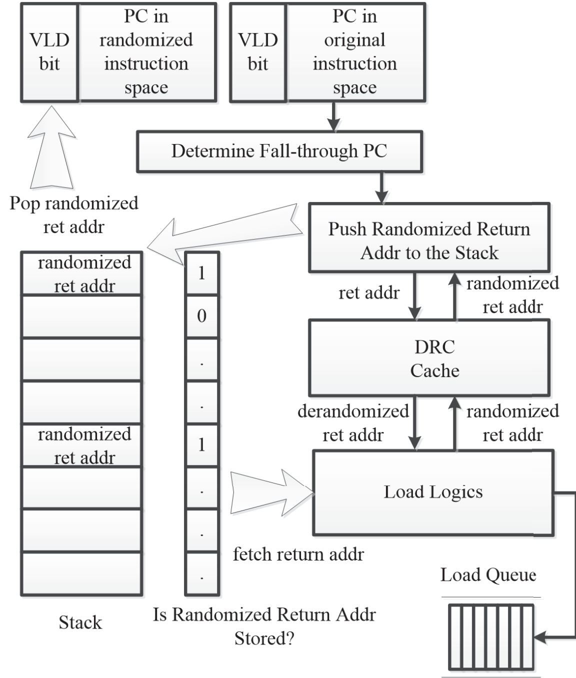

Fig. 10. Support for control transfer without using the `return` instruction. Callee in x86 may access return address store in the stack explicitly without using the call `return` instruction. The stack stores randomized return addresses.

outside the context of function call. A bitmap is used to track which stack location stores a randomized return address. For higher performance, the bitmap can be stored in paged memory. Similar to the randomization/de-randomization tables, pages containing the bitmap are set to be invisible to the user space instructions. A small cache can be used to store parts of the bitmap that are frequently accessed.

D. Remarks

Our approach mainly affects the interface between instruction fetch and the memory hierarchy for storing instructions. We carefully design our system so impacts to other microarchitecture components are minimized. For example, the introduction of two program counters (RPC and UPC) facilitates normal operations of predictors for both branch directions and branch targets. Both predictions can be based on the derandomized program counter as illustrated in Figure 7. During instruction execution and fetch, when UPC is absent, the instruction fetch unit will first de-randomize the randomized PC and then use the de-randomized PC for branch prediction. In such a way, branch prediction rates will not be affected by how instruction layout and control flow are randomized. Since our approach only randomizes instruction address space, which contains read-only data, it can be applied to multi-core or multi-processor based systems with easy. In addition, control flow randomization can be confined within the same page, which will further reduce its impact to iTLB. At system level, the main impact is to extend application context to include the de-randomization/randomization tables.

V. Security Analysis

In this section, we show the effectiveness of the proposed method to enhance software dependability. According to the definition of Jean-Claude Laprie's work, "different emphasis may be put on the different facets of dependability" such as availability, reliability, safety, confidentiality, integrity, and maintainability [23]. Therefore, we analyze how secure our framework is against ROP exploits since the security eventually includes availability, confidentiality, and integrity.

A. ROP Attacks

In terms of randomizing control flow at instruction level and thwarting ROP based exploits, our approach is equivalent with software based ILR [9]. Some main advantages of our design over the prior art include native support for direct execution of binary applications with control flow diversification, and efficient instruction fetch by preserving spacial locality when instructions are stored in on-chip and off-chip memory hierarchy. According to both our security evaluation and the prior work [9], ILR can effectively mitigate arc-injection attacks that evade ASLR. In addition to that, it was demonstrated that ILR can prevent ROP attacks that target a vulnerable Linux PDF viewer, xpdf [9]. The vulnerability allows attackers to create a shell by using crafted ROP attack. Another vulnerability in Adobe PDF viewer (9.3.0) allows arc-injection and ROP attacks [24]. It was demonstrated that randomizing control flow at instruction level can thwart attacks from crafted malicious PDF files that exploit the gadget based programming [9], [24]. In short, ILR is effective to mitigate actual ROP attacks that exploit vulnerabilities in real world applications using return based programming and gadgets.

B. Gadget Analysis

It has been shown that ILR can effectively reduce the attacking surfaces. Consequently, there is less chance for attackers to find enough number of gadgets to mount a ROP attack. To evaluate how effective our approach is to reduce attacking surfaces, we use an open source gadget tool called ROPgadget [25]. ROPgadget can scan a binary program to find specific gadgets within the executable. It has an auto-roper for build attack payload automatically with the gadgets found and facilitate the process of ROP exploitation. ROPgadget contains a database of gadget patterns. We use version 4.0.1 of ROPgadget. ROPgadget has a gadget compiler that can create attack payload using matching gadgets found in an executable binary. The assembled payloads can facilitate development of actual attacks depending on the vulnerabilities of the binary. Note that the payloads themselves are not sufficient to mount a successful attack but they form an important part of ROP exploits. The payloads can be converted into attacks when combined with vulnerabilities. If ROPgadget fails to create attack payloads using gadgets found in a binary executable, it means that even there is exploitable vulnerability, ROP attack cannot be mounted using the existing attack payload template. Typically, ROPgadget requires detection of multiple gadgets in an executable to assemble a payload. If control flow randomization significantly reduces the number of gadgets that can be found in a binary executable, the likelihood an attack payload can be assembled will become smaller because of the reduced gadget pool.

To model the environment of virtual control flow randomization faced by attackers, we modified ROPgadget to

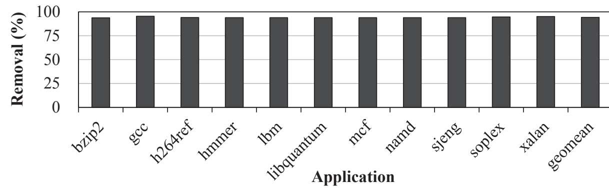

Fig. 11. Percentage of gadgets removed from SPEC CPU2006 benchmark applications after randomization. The gadets are detected using an open source gadget tool, ROPGadget [25].

take into account control flow randomization in such a way that it searches for gadgets using un-randomized instruction locations. We tested a set of SPEC CPU2006 benchmark applications using the modified ROPgadget. Without control flow randomization, for every tested SPEC benchmark application, ROPgadget is able to assemble attack payloads. After virtual control flow randomization, for all the benchmark applications, no attack payloads can be generated. This suggests that randomizing control flow can reduce the likelihood of successfully mounting a ROP attack. Virtual control flow randomization can reduce the number of gadgets. As indicated by Figure 11, on average 98% of gadgets are removed after applying our control flow randomization. In addition, many published papers in the literature already show that address space randomization is effective to resist ROP attacks.

C. Remarks

a) Entropy: As studied previously, ILR can have high entropy, which defends against attacks that try to evade the protection by reducing the entropy of a system. Since randomization is done at instruction granularity, there is a large randomization space. Although we have used ROP as a representative attack method of code reuse, return-to-libc is also powerful attack for systems. However, Shacham et al. already demonstrated that return-to-libc attack can be protected using ILR with 64 bit address space [14]. For this reason, we have used ROP as a threaten of code reuse attack in this paper.

b) Code Injection: Our primary focus of this study is to mitigate code reuse attacks with hardware support of ILR. Our solution by itself doesn't address code injection attacks [26], [8], [27]. However, our approach can be combined with solutions that are specifically designed to mitigate code injection attacks and used in conjunction with these solutions to thwart both ROP and code injection based exploits.

c) Protection of Address Translations: Similar to all randomization based approaches, a common practice to prevent leaking randomization/de-randomization tables to the attackers is to apply regular re-randomization of the binary images that will create a new sets of address translation tables and new randomized images. Even an attacker managed to obtain the old randomization/de-randomization tables, the information would be outdated for mounting new attacks. Furthermore, for online attacks, since the randomization/de-randomization tables associated with an application instance are invisible to the instructions executed in the user space, most randomized return addresses cannot be leaked to the remote attackers. Side channel exploits such as those described in earlier projects are not effective to cause randomized control flow addresses to be leaked to remote attackers [28], [29], [30]. Program counter UPC is auto-managed by the execution pipeline. It cannot be directly accessed by the instructions in the user space.

VI. Performance Evaluation

In order to demonstrate the performance of our system design, we have conducted several experiments and simulations using detailed architectural models. We studied the overall performance impacts of our design using 11 SPEC CPU2006 benchmarks [31]. Along with that, we examined the overheads to support native execution of ILR transformed programs by adopting our design such as efficiency of runtime de-randomization, execution speed of each benchmark application, and etc. In particular, we extended XIOSim [16] simulators to meet our need.

A. Implementations

We implemented a static binary rewriter which can randomize the instruction space given a third-party binary program. The output of the static binary rewriter is a binary file with randomized instruction segments and lookup tables that can be used to de-randomize the instruction space. Currently, the rewriter only works for statically linked binary with all the libraries embedded.

For performance modeling, we use XIOSim [16]. XIOSim is a highly detailed cycle based micro-architectural simulator targeted at x86 micro-processors. XIOSim is based on Zesto simulator [17]. It models Intel x86 pipeline according to best available public knowledge. Performance reported from XIOSim's models stays well within 10% of real hardware for the entire SPEC CPU2006 suite [16]. XIOSim provides detailed x86 architecture models for simulating in-order and out-of-order pipelines. The models include detailed branch predictors, branch target buffers, return address stack (RAS) predictors, cache prefetchers, memory controllers, and main memory/DRAM models. For power modeling, XIOSim integrates a modified version of McPAT [32] to create a power consumption trace. In terms of average power consumption, XIOSim's model has less than 5% deviation when compared against real measurement. We modified and extended XIOSim with the proposed architecture. The fetch stage of XIOSim is modified to use randomized instruction space, and support two program counters (RPC and UPC). The execution models of `call` and `return` instructions are modified according to our design. Instruction fetch is extended to support a de-randomization cache (DRC). The DRC cache connects to a unified second level cache shared by IL1 and DL1. A power model for DRC is also integrated with XIOSim.

B. Benchmarks

For performance evaluation, we used the single thread SPEC CPU2006 benchmark suite [31] that is a set of benchmark applications designed to test the CPU performance. We tested eleven memory intensive benchmarks of the SPEC CPU2006. In particularly, the benchmarks used are bzip2, gcc, mcf, hmmer, sjeng, libquantum, h264ref, lbm, xalan, nsmd, and soplex. The detailed descriptions of the benchmarks can be found in the webpage [31]. The simulation started when the application passed the initialization stage. The cycle based simulation executed each benchmark application for 500 million instructions or until it finished depending on which one was longer.

C. Machine Parameters

We modified the XIOSim simulator to simulate support for runtime instruction space de-randomization. The operation of our proposed scheme is verified with single issue, in-order processor. For this reason, the simulation was performed with a x86 single issue, in-order CPU model running at 1.6GHz. The overall pipeline is divided into five major components or blocks, fetch, decode, allocation (alloc), execution (exec) and commit blocks. Each component may further comprise pipeline stages, queues, and other structures. The detailed processor model includes, branch predictor (2-level gshare), BTB (branch target buffer), RAS, instruction queue, load-store queue, prefetcher, and functional units. The fetch stage includes the PC generation (i.e., branch prediction). The fetch stage of the simulator operates on entire lines from the instruction cache which are placed in the byte queue. A pre-decode pipeline performs the initial decoding of the variable-length x86 instructions to individual macro-ops, which are placed into the instruction queue (IQ) with one macro-op per entry. From here, the instructions proceed to the decoder pipelines. The instruction queue size is 18. The I-TLB and D-TLB have 64 fully associative entries. The CPU has a 32-entry load/store queue. The L1 instruction cache size is two-way 32KB, 64-byte block size, and has an access latency of 2 cycles. The L1 data cache is a 32KB, 2-way associative, write back cache with 64-byte block size, and also has an access latency of 2 cycles. The L2 cache is unified, 512KB size, 8-way associativity, 64-byte block size, and has an 12-cycle access latency.

The simulator integrates DRAMSim2 [33] as the memory model. DRAMSim is a cycle accurate open source JEDEC DDRx memory system simulator. It provides a DDR2/3 memory system model. It uses open page policy, and therefore attempts to schedule accesses to the same pages together to maximize row buffer hits. The DRAM model tracks individual ranks and banks, and accounts for pre-charge latencies, CAS and RAS latencies, and refresh effects. We experimented with different de-randomization cache sizes, from 64 translation entries to 512 translation entries. Each entry supports 32-bit instruction address translation.

VII. Results Analysis

For the benchmarks, the performance improvements of our approach with 128-entry DRC lookup buffer over a straightforward implementation of ILR are shown in Figure 12, the average speedup is 1.63 for all the benchmarks. For application namd, h264ref, mcf, xalancbmk, our approach achieves more than double speedup. We study the performance overhead incurred by virtual control flow randomization over the default baseline of no randomization. We further experiment with different sizes of DRC lookup buffer, 512 entries, 128 entries, and 64 entries. Figure 13 shows the results. The results indicate that increasing the size of DRC lookup buffer can improve the overall IPC. When the DRC lookup buffer has 512 entries, the average IPC for all the benchmark applications with virtual control flow randomization is almost 98.9% of the baseline condition of no randomization. With a lookup buffer size of 64 entries, on average, the applications still maintain 97.9% performance in terms of IPC, meaning 2.1% overhead.

Figure 14 shows miss rates of the DRC lookup buffer. There are two settings, 512 entries and 64 entries. The average DRC miss rate under 512-entry lookup buffer is 4.5%. When the DRC entry size is 64, the average miss rate increases to 20.6%. The results indicate that lbm and xalancbmk have the worst DRC miss rates. Note that sometimes, DRC cache miss rate is not the only factor that affects performance. When there is a DRC cache miss, the system will look up the L2 cache, which is large enough for storing de-randomization table. In short, our approach of hardware assisted control flow randomization incurs very small overhead over no randomization for the studied SPEC CPU2006 benchmark applications. The average overhead is 2.1% IPC decrease under a small 64-entry DRC lookup buffer.

In terms of power consumption, our approach incurs very small overhead. Thank to the power modeling framework already integrated with XIOSim, it is easy to modify the simulators to take into account the extra power consumed by the mediation layer that does de-randomization/randomization. Figure 15 shows the dynamic power overhead under 128-entry DRC. The average dynamic power overhead for the studied SPEC benchmarks is 0.18% of the total CPU dynamic power.

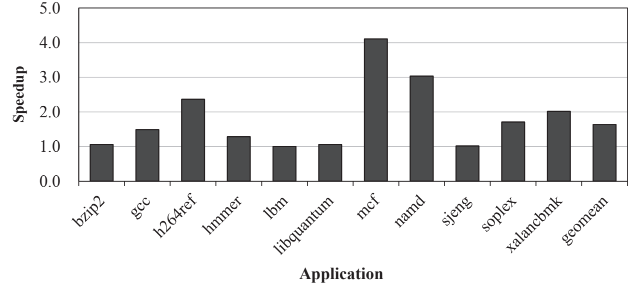

Fig. 12. Performance speedup using DRC over straightforward implementation of ILR. Y-axis shows IPC ratio. DRC setting: 128 entries. The average speedup is 1.63 for all the tested benchmarks.

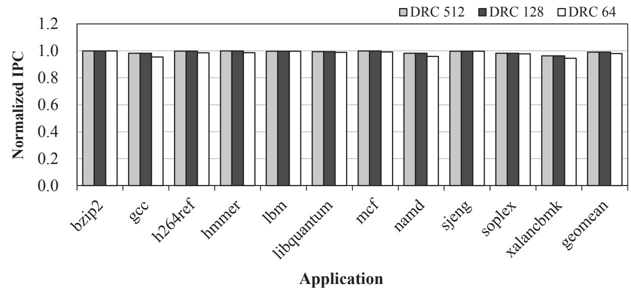

Fig. 13. Normalized IPC performance under different DRC sizes. Y-axis shows normalized IPC over the baseline IPC of no randomization. The average IPC slowdown is less than 2.1%.

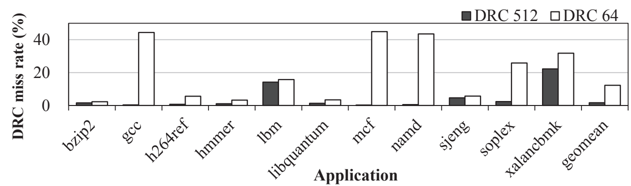

Fig. 14. DRC miss rates under two different settings, DRC with 512 entries and DRC with 64 entries. Y-axis shows DRC lookup miss rates. The average miss rate in 512-entry DRC is 4.5% and the average miss rate in 64-entry DRC is 20.6%.

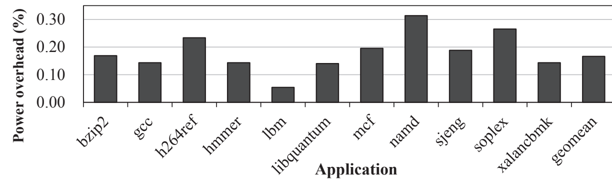

Fig. 15. Dynamic power overhead of DRC (128 entries) for SPEC CPU2006 benchmarks. Y-axis shows percentages of DRC dynamic power over CPU dynamic power. Power modeling is based on XIOSim and integrated McPAT. The average dynamic power overhead is 0.18% of the CPU dynamic power.

VIII. Related Work

A. Software Based Approaches

Address Space Layout Randomization (ASLR) is a scheme which hinders the original location of code and data objects [34], [35], [36], [37]. Some of these approaches randomize the base addresses of process images, and some of them randomize even at the basic block level (e.g., the in place randomization [10]). Some of them use virtual machine monitor to scramble the instruction addresses [38], and some of them use binary rewritters to achieve this [39]. Compared with these software based ASLR methods, our proposed mechanism guarantees more efficient randomization for the program address space with the help of micro-architectural support.

Instruction Set Randomization (ISR) is also widely used software approach to prevent code injection attack. In ISR, the underlying system instructions [40], [8] are encrypted by use of a random key set. The encrypted instructions are decrypted only before the fetch stage of the processor pipeline. Although ISR provides effective security solution for code injection attack, leakage of the encryption key or succeed in guessing procedure for the key may cause failure in the protection [41]. In addition, ISR is not designed to mitigate ROP attacks.

Randomization can be applied not only for instructions, but also for the program data. For example, all pointers that are resides in memory can be encrypted before they are used [42]; decryptions are performed only before the data is needed for the processors. Recently, XORing the data with random mask has been proposed to support probabilistic protection to cope with the memory exploiting [43], [44].

Multi-variant system which is proposed by Cox et al. is also based on software [45]. In their work, N-variant explores diversification to enhance the security. The adversary should subvert all the running variants simultaneously which is hard to achieve. In this system, different ISAs are employed and synchronized at system call level while our solution randomizes the instruction addresses.

B. Hardware Based Approach

While there is a considerate amount of research from software approaches to achieve diversification, less attention has paid on the hardware approaches. Orthrus [46] is one of the examples from hardware perspective. It protects software integrity by exploiting multi-core architecture by executing n versions using different processor cores. Kayaalp et al. describe a hardware based protection mechanism that enforces simple control flow rules at the function granularity to disallow arbitrary control flow transfers from one function into the middle of another function [47], Also, a signature-based detection of ROP is proposed, where the attack is detected by observing the behavior of programs and detecting the gadget execution patterns [12]. Though not directly targetting ROP exploits, there are solutions that attempt to use hardware performance counters to detect malware and verify control flow integrity [13], [48].

Our work distinguishes from all the prior software based and hardware based solutions. Our system tries to strengthen control flow diversity with native hardware support for improved performance and instruction fetch efficiency by preserving the instruction locality, while at the same time maximizing control flow randomness.

IX. Conclusion

We have developed a micro-architectural solution to enhance dependability of software defending against code reuse attack. Different from prior hardware based approaches for mitigating the attacks, our approach is based on software diversity and instruction location randomization. To address the inefficiencies of memory system and instruction fetch caused by instruction layout randomization, we propose a novel micro-architecture design that can support native execution of ILR software while at the same time preserve instruction fetch performance. Using state-of-the-art architecture simulation framework, XIOSim and a set of SPEC CPU2006 applications, we show that our solution can achieve average speedup of 1.63 times over a straightforward hardware implementation of ILR. Using our approach, direct execution of randomized binary incurs only 2.1% IPC performance overhead. Currently, the proposed idea is limited as single issue, in-order processor which is widely used in the area of low-power consuming embedded systems. However, in the near future, we will explore and extend the idea to the out-of-order superscalar processor for the contemporary high performance computing systems.

Acknowledgment

This work is supported in part by the National Science Foundation (Grant No. CNS-1205708, DGE-1433817) and in part by Memory Division, Samsung Electronics Co., Ltd.

References

[1] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” in *Proc. 14th ACM Conf. Computer and Communications Security*, 2007, pp. 552–561.

[2] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, “When good instructions go bad: Generalizing return-oriented programming to RISC,” in *Proc. 15th ACM Conf. Computer and Communications Security*, 2008, pp. 27–38.

[3] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, “Return-oriented programming without returns,” in *Proc. 17th ACM Conf. Computer and Communications Security*, 2010, pp. 559–572.

[4] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham, “Can DREs provide long-lasting security? the case of return-oriented programming and the avc advantage,” in *Proc. Conf. on Electronic Voting Eechnology/Workshop on Trustworthy Elections*, 2009, pp. 6–21.

[5] T. Dullien, T. Kornau, and R.-P. Weinmann, “A framework for automated architecture-independent gadget search,” in *Proc. 4th USENIX Conf. on Offensive Technologies*, 2010, pp. 1–10.

[6] R. Hund, T. Holz, and F. C. Freiling, “Return-oriented rootkits: bypassing kernel code integrity protection mechanisms,” in *Proc. 18th Conf. on USENIX Security Symp.*, 2009, pp. 383–398.

[7] The metasploit project. [Online]. Available: http://www.metasploit.com/

[8] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proc. 10th ACM Conf. Computer and Communications Security*, 2003, pp. 272–280.

[9] J. D. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in *Proc. IEEE Symp. Security and Privacy*, 2012, pp. 571–585.

[10] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proc. IEEE Symp. on Security and Privacy*, 2012, pp. 601–615.

[11] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: low-overhead protection from code reuse attacks," *SIGARCH Comput. Archit. News*, vol. 40, no. 3, pp. 94–105, 2012.

[12] M. Kayaalp, S. T, J. Nomani, N. Abu-Ghazaleh, and D. Ponomarev, "Scrap: Architecture for signature-based protection from code reuse attacks," in *Proc. IEEE Int'l Symp. on High Performance Computer Architecture*, 2013, pp. 258–269.

[13] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," in *Proc. 40th Annual Int'l Symp. on Computer Architecture*, 2013, pp. 559–570.

[14] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proc. 11th ACM Conf. on Computer and Communications Security*, 2004, pp. 298–307.

[15] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proc. of the 2013 IEEE Symp. on Security and Privacy*, 2013, pp. 574–588.

[16] S. Kanev, G.-Y. Wei, and D. Brooks, "Xiosim: power-performance modeling of mobile x86 cores," in *Proc. 2012 ACM/IEEE Int'l Symp. on Low Power Electronics and Design*, 2012, pp. 267–272.

[17] G. Loh, S. Subramaniam, and Y. Xie, "Zesto: A cycle-level simulator for highly detailed microarchitecture exploration," in *Proc. IEEE Int'l Symp. on Performance Analysis of Systems and Software*, 2009, pp. 53–64.

[18] Hex-rays. [Online]. Available: http://www.hex-rays.com/products/ida/index.shtml

[19] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen, "On the static analysis of indirect control transfers in binaries," 2000, pp. 1013–1019.

[20] B. De Sutter, B. De Bus, and K. De Bosschere, "Link-time binary rewriting techniques for program compaction," *ACM Transactions on Programming Languages and Systems*, vol. 27, no. 5, pp. 882–945, 2005.

[21] M. Madou. Birma: Binary rewriter for the mips architecture/matias madou. [Online]. Available: http://lib.ugent.be/fulltxt/RUG01/000/777/296/RUG01-000777296_2010_0001_AC.pdf

[22] C. de Dinechin, "C++ exception handling for ia64." in *Proc. First Workshop on Industrial Experiences with Systems Software*, 2000, pp. 67–76.

[23] J.-C. Laprie, "Dependability - its attributes, impairments and means," in *Predictably Dependable Computing Systems*, ser. ESPRIT Basic Research Series, B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, Eds. Springer Berlin Heidelberg, 1995, pp. 3–18.

[24] Libtiff tifffetchshortpair remote buffer overflow vulnerability. [Online]. Available: http://www.securityfocus.com/bid/19283

[25] J. Salwan. (2012) ROPgadget. [Online]. Available: http://shell-storm.org/project/ROPgadget

[26] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 7, no. 49, 1996.

[27] J. Pincus and B. Baker, "Beyond stack smashing: Recent advances in exploiting buffer overruns," *IEEE Security & Privacy*, vol. 2, no. 4, pp. 20–27, 2004.

[28] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, "A quantitative, experimental approach to measuring processor side-channel security," *IEEE Micro*, vol. 33, no. 3, pp. 68–77, 2013.

[29] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, "Side-channel vulnerability factor: A metric for measuring information leakage," in *Proc. 39th Annual Int'l Symp. on Computer Architecture*, 2012, pp. 106–117.

[30] T. Zhang, F. Liu, S. Chen, and R. B. Lee, "Side channel vulnerability metrics: The promise and the pitfalls," in *Proc. 2nd Int'l Workshop on Hardware and Architectural Support for Security and Privacy*, 2013, pp. 2:1–2:8.

[31] Standard Performance Evaluation Corporation. Spec cpu2006. [Online]. Available: https://www.spec.org

[32] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. 42nd Annual IEEE/ACM Int'l Symp. on Microarchitecture*, 2009, pp. 469–480.

[33] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, pp. 16–19, 2011.

[34] PaX Team, "PaX address space layout randomization (ASLR)," 2003.

[35] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *Proc. 12th USENIX Security Symposium*, 2003, pp. 105–120.

[36] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *Proc. 14th USENIX Security Symposium*, 2005, pp. 255–270.

[37] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address Space Layout Permutation (ASLP): Towards fine-grained randomization of commodity software," in *Proc. 22nd Annual Computer Security Applications Conf.*, 2006, pp. 339–348.

[38] L. Davi, A.-R. Sadeghi, and M. Winandy, "Ropdefender: a detection tool to defend against return-oriented programming attacks," in *Proc. 6th ACM Symp. on Information, Computer and Communications Security*, 2011, pp. 40–51.

[39] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proc. ACM Conf. Computer and Communications Security*, 2012, pp. 157–168.

[40] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proc. 10th ACM Conf. Computer and Communications Security*, 2003, pp. 281–289.

[41] A. N. Sovarel, D. Evans, and N. Paul, "Where's the FEEB? the effectiveness of instruction set randomization," in *Proc. 14th Conf. on USENIX Security Symp.*, 2005, pp. 10–25.

[42] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard™: Protecting pointers from buffer overflow vulnerabilities," in *Proc. 12th USENIX Security Symp*, 2003, pp. 91–104.

[43] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro, "Data randomization," Tech. Rep. MSR-TR-2008-120, 2008.

[44] S. Bhatkar and R. Sekar, "Data space randomization," in *Proc. Int. Conf. Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008, pp. 1–22.

[45] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: a secretless framework for security through diversity," in *Proc. 15th Conf. on USENIX Security Symposium*, 2006, pp. 105–120.

[46] R. Huang, D. Y. Deng, and G. E. Suh, "Orthrus: Efficient software integrity protection on multi-cores," *SIGARCH Comput. Archit. News*, vol. 38, no. 1, pp. 371–384, 2010.

[47] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: low-overhead protection from code reuse attacks," in *Proc. 39th Int'l Symp. on Computer Architecture*, 2012, pp. 94–105.

[48] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting violation of control flow integrity using performance counters," in *Proc. 42nd Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks*, 2012, pp. 1–12.