Programmable Decoder and Shadow Threads: Tolerate Remote Code Injection Exploits with Diversified Redundancy

Ziyi Liu[+], Weidong Shi[+], Shouhuai Xu[†], Zhiqiang Lin[⋆]

ziyiliu@cs.uh.edu[+], wshi3@central.uh.edu[+], shxu@cs.utsa.edu[†], zhiqiang.lin@utdallas.edu[⋆]
Department of Computer Science[+], University of Houston, 4800 Calhoun Road,

Houston, TX 77004, U.S.A; Department of Computer Science[†], The University of Texas at San Antonio,
One UTSA Circle, San Antonio, TX 78249-0667, U.S.A; Department of Computer Science[⋆],
The University of Texas at Dallas, 800 W. Campbell Road, Richardson, TX 75080 U.S.A.

***Abstract*—We present a lightweight hardware framework for providing high assurance detection and prevention of code injection attacks using a lockstep diversified shadow execution. Recent studies show that hardware diversification can detect software attacks by checking the consistency of their behavior simultaneously. Unfortunately, the severe performance degradation and extra system costs caused by these methods are unacceptable in many applications. This paper presents a hardware-level, lockstep shadow thread framework to enrich the diversity of the software execution, with the facilitation from programmable hardware decoder and novel CPU support of tightly coupled shadow thread technique. Specifically, given a piece of (legacy) binary code, we first generate diversified binary versions using an offline binary rewriter and programmable hardware binary translator at runtime. Two diversified binary code images are launched as dual simultaneous threads in the hardware layer with one as the primary thread and the other one as shadow thread. Instructions from the shadow thread are not executed but just compared, and thus incur no OS side-effects. The extended CPU is able to decode instructions from both threads, and dispatch them to the next stage pipeline for a lockstep comparison. Any mismatch of the decoded instructions from the two threads caused by remotely injected binary code will be detected. Our design provides instruction set randomization (ISR) with minimal cost in performance, when compared with straightforward ISR implementation. The simulation results indicate that our framework incurs very small overheads and provides a protection against code injection attacks.**

I. Introduction

The monoculture of our computer eco-system is one of the root-causes of many of our cyber attacks today, such as code injection, code reuse, worms, and bots. Breaking the monoculture of the computer system has shown to be an effective approach to counter these large scale and rapid attacks. In the past decade, numerous techniques have been proposed to create diversified execution environments. Notable examples include address space layout randomization (ASLR) [20], [5], [7], instruction set randomization (ISR) [3], [17], data randomization [10], [6], [11], and N-Variant systems [12], [4], [19].

However, to the best of our knowledge, all these approaches are from software perspective. More or less, they often suffer from either large overhead (e.g., ISR has to at least 75% overhead according to a most fast implementation [21]), or practicability (e.g., require accessing program source code [10], [6], [11]), or have limited entropy (e.g., for 32-bits ASLR, the entropy is only 2^{16}). While N-Variant system has explored the diversity of existing ISAs, it only leverages the macro level differences of multiple computer architectures. For instance, as shown in [12], N-variant can run code compiled for several architectures (x86, ARM, MIPS) in parallel to detect code injection attacks. However, it has a synchronization problem as it employs stand-alone existing computer systems with different ISAs.

As such, in this paper, we aim to explore the hardware approaches at micro instruction level to achieve the diversification. To this end, we propose a hardware-level, lockstep shadow execution framework to enrich the diversity of the software execution, with the facilitation from hardware decoder. Specifically, given a piece of (legacy) binary code, we first generate diversified versions by remapping its machine codes through a binary rewriter. The original binary code along with the diversified binary code are launched as tightly coupled dual threads (one called primary thread, the other one called shadow thread) in the hardware layer.

Different from the conventional cycle interleaving simultaneous threads (SMT), our two coupled threads appear as one to the OS kernel. For example, our dual threads share the same OS footprint or side-effect (e.g., I/O operations from both threads are only executed once). The extended hardware is able to decode the remapped instructions, and dispatch them to the next stage pipeline for a lockstep opcode and operand comparison. If there is any difference, the hardware will flag that there is an intrusion from code injections. Our evaluation results indicate that our framework incurs minimal overheads and provides a protection against binary code injection attacks. In short, we make the following contributions:

- We introduce a low overhead machine code diversification technique that can remap machine code of the same ISA into multiple variants based on the concept of programmable decoder for high assurance of security properties.
- We propose a design of lockstep shadow thread that synchronizes and cross-validates two variants of the same application at per instruction level. Any injected attack codes will be detected due to mismatched instructions after decoded.
- We present a hardware implementation and analyze its performance.

II. System Overview

A. Remote Code Injection Attacks

In the past decades, a variety of code injection attacks has appeared in the wild. For instance, buffer overflow attack injects attack code in the stack, leading to the overwrite of the function return address or a function pointer to point to the entry of the injected code [16]. Recently, hackers and

978-3-9815370-2-4/DATE14/©2014 EDAA

researchers have been targeting embedded systems as their attack focus (e.g., [13], [1], [2]). In [13], the author describes an approach by which SQL injection is used to gain remote access to arbitrary files from the file systems of Netgear wireless routers. In [2], the authors demonstrate exploiting HP-RFU LaserJet printer firmware vulnerability which allows arbitrary injection of malware into the printer's firmware. The key for these attacks is that the injected machine code will overflow the buffer to gain privilege access on the device firmware. These code injection based attacks can only succeed if the injected foreign code is compatible with the execution environment. For example, program may crash when injecting MIPS machine code to a process running on x86 system because execution of the illegal opcode. Our approach is based on this observation, and we create an execution environment that can load a variant shadow thread, so that the CPU can always check the correctness of the instructions in the primary thread by comparing each decoded instruction with the corresponding instruction in the lockstep shadow thread. Hence, even an attacker succeeds in injecting malicious code, the injected code can be detected by our system because they produce mismatched instructions after decoded by the primary thread and shadow thread. It is important to note that code injection attacks cannot be prevented by techniques such as non-executable stack.

B. Approach Overview

Threat Model Our threat model is defined as the following. An attacker is attempting to penetrate into an operating system via code injection attack. Software applications are distributed to the end user in binary format, and then diversified with support for shadow thread. The binary application has been tested, but not guaranteed to be vulnerability free. The program may contain weakness that can be exploited by code injection. However, the application is assumed to be free from back doors or trojans. Furthermore, we assume that there is no insider attack. The attacker does not know/see the executable version of the binary code. 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 scrambled code (because the attacker does not have access to the decoder, which is bound to the processor of the program owner/user). Our threat model mainly focuses on attacks where a system is subverted by processing malicious data submitted by the attacker. The data may contain injected code. The threat model covers a wide range of exploits such as compromising a system through a hosted client-server application, attacks to networked printers, wireless consumer appliances, networking gears (e.g., network gateway), internet access points, network clients, networked smart grid devices, embedded systems, etc.

Overview The system framework of our design is shown in Figure 1. At a high level, there are three steps. First, the binary code needs to be diversified off-line by a binary rewriter. At this stage, the machine codes of a binary will be extracted and translated line by line via a machine code diversifier. In particular, the opcode of each instruction will be replaced with another unique value based on an encoding map. A swizzle operator will reorder the positions of the result machine codes. The original code image can have multiple different diversified copies. Each one has its own encoding map. The binary rewriter will also be responsible to pack different versions of the binary codes along with their mapping information as well as its swizzle rules into a single binary file. Though diversified, all the binary images contain identical instructions in the ISA.

Next, at execution time, multiple versions of the machine code images will be loaded into separate application contexts (simultaneous co-threading with one primary executing thread and one shadow thread) by a special loader. Meantime, this loader will also load the decoder maps that are protected into the kernel space. Though each thread has its own virtual memory space and page tables, they are tightly coupled using instruction level lockstep synchronization. Each thread has its own program counter, because the two binary images need to be fetched into the instruction caches and decoded by the CPU. However, different from the conventional SMT architecture, the two threads have identical program counters.

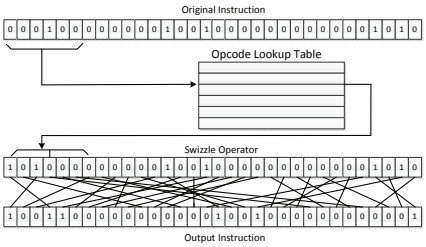

Fig. 2. Process of Binary Instruction Diversification

Then, the fetched instructions are translated according to the decoder maps (loaded into the CPU). The extended CPU compares the translated and decoded instructions between the primary and shadow thread. Instructions from the shadow thread are not executed because the two threads contain identical instructions. The shadow thread occupies minimal hardware resources (much less than what is required in the conventional SMT). We extend the CPU pipeline by supporting the proposed "lockstep simultaneous" co-threads. In particular, the modified CPU can fetch and compare instructions from different versions of binary images at the same time. Before the instruction gets executed, the instruction translator will translate the scrambled machine codes into the original instructions. After translation, the extended CPU will compare the decoded shadow thread instruction with the corresponding instruction from the primary thread. A mismatch will occur when remotely injected codes are fetched and decoded. The CPU will record such mismatch and raise an exception when the instruction from the primary thread is committed.

III. Architecture and Design

A. Machine Code Diversification

There are two steps involved for binary code diversification. First, the opcode of each instruction is replaced with another value according to a lookup table. Next, positions of each machine code bit are scrambled using a programmable swizzle operator. In RISC type of ISA, the opcode often has a fixed length. For instance, 32bits MIPS has a 6-digit Opcode. As shown in Figure 2, the original instruction is mapped to a new instruction with a new opcode according to a lookup table. The lookup table contains an array of unique 6-bit random numbers that are chosen by a system administrator or randomly generated. Using 32bit MIPS as example, this table defines the mapping for all the MIPS opcodes. An input opcode will be translated according to this table, e.g., 000100 is translated into 101000 in our example. The programmable swizzle operator will scramble the order of all the machine code bits. In the figure, the first bit is moved to the fourth bit, the second bit is moved to the eleventh bit, and so on.

B. Programable Code Translation

We implemented the proposed design at hardware layer using gate-level logic. As shown in Figure 4, machine code of a fetched instruction will first be reordered by a programmable swizzle operator, and then translated back to the original instruction using a lookup table. The lookup table is designed

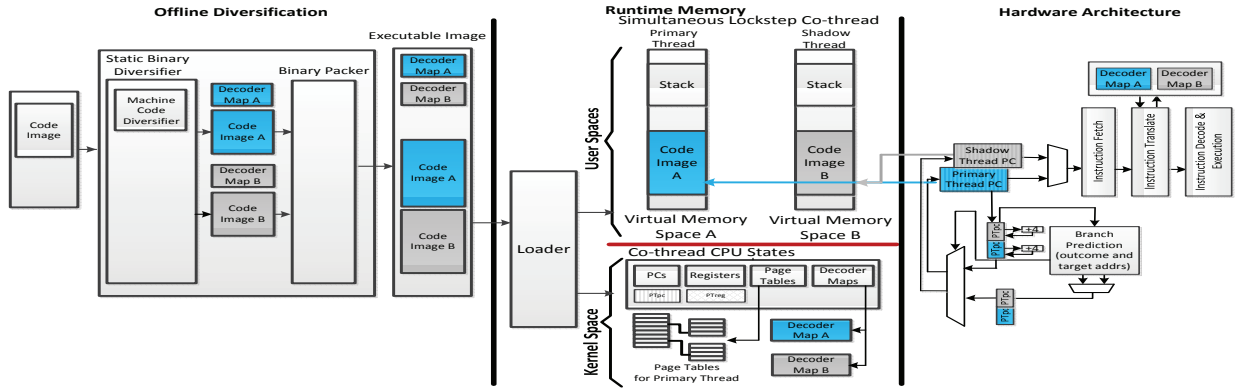

Fig. 1. Overview of Hardware Supported Binary Code Diversification and Lockstep Dual Thread Execution Model. Given a binary image, a binary rewriter can produce two diversified thread images. One behaves as a main thread, and the other one acts as a shadow thread. The two threads contain identical instruction sequences of the same ISA but encoded with different binary format. When executed by the hardware, the threaded micro-architecture will fetch and decode both threads in lockstep and cross-validate the decoded instructions.

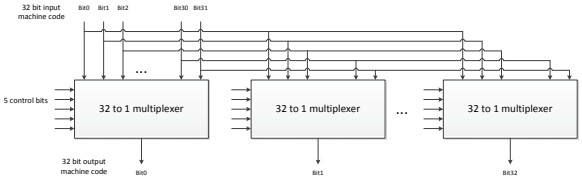

Fig. 3. Implementation of Swizzle Operator

as an array of bits similar to a directly mapped cache with opcode as index. The programmable swizzle operator is more complex because we need to support programmable N-to-N remapping of bit locations. Basically, we design a hardware logic that can arbitrarily swap machine code bits according to a set of programmable control bits.

As shown in Figure 3, one can use n N-to-1 muxes to design the swizzle operator. The ultimate goal of the swizzle operator is to scramble a n-bit input machine code into another n-bit output machine code. Using 32 bits MIPS ISA as example, the input comprises a 32-bit machine code. After swizzle operation, each bit of the machine code will be put into a new position. For the 32-bit machine code output, each bit location stores the value from an arbitrary input bit out of the 32 input bits. A 32-to-1 multiplexer that takes 32 bit inputs can arbitrarily choose an input bit as the output based on 5-bit control signals. For implementing our design, one can use 32 32-to-1 multiplexers to create a 32-to-32 swizzle operator. There exist alternative designs that are efficient in logic area. However, we use this design in our evaluation for simplicity. One advantage of our design for binary format diversification is its low overhead in terms of hardware cost and latency. Alternative designs such as a straight-forward hardware implementation of ISR [17] will incur more overhead in area and decoding latency because ISR relies on heavyweight cryptographic functions. Simulation results show upto 1000% performance slowdown under naive ISR hardware implementation.

C. Lockstep Shadow Threads

Our lockstep shadow threads share some similar features with the conventional cycle based simultaneous threading (SMT) but with several major differences. In a conventional SMT machine, the minimum resources needed for one independent thread execution are an execution unit, a private register file, and a separate stack-space. Similar to the simultaneous multithreading technique, the lockstep threads can issue instructions each cycle by alternating instruction fetches between the primary thread and the shadow thread. Lockstep threads enable execution of these two threads in parallel on a single core with multiple programable counters. One essential feature of the lockstep threads is that two instructions of both threads can be fetched at once. This rule ensures that the program counters of the primary and shadow thread point to the same virtual address. After translated by the TLB (translation lookup buffer), these program counters point to different physical addresses.

Different from the conventional SMT technique, in our design, the primary and the shadow thread contain identical instructions but different binary images. The two threads are synchronized in lockstep at per instruction level. This means that the corresponding instructions from both threads commit together atomically as one instruction. Since the primary and shadow thread contain identical instructions, it is not necessary to execute the decoded instructions from the shadow thread. The hardware pipeline will match the corresponding instructions from the two threads after they are decoded, and ensure that they are the same. When a mismatch is detected, an exception will occur. Taking advantages of the fact that the two threads are the same, the shadow thread doesn't need its own architect register file. It can use register values from the primary thread.

Since each thread has its own memory space, instructions that change memory states should take effects for both threads. This is achieved by applying the same updates to the physical memory of both threads at the same time and tagging on-chip cache lines with both primary thread id and shadow thread id. With this implementation, data only needs to be saved once by the primary thread. The two threads can share virtually indexed data cache entries. When a data cache line is evicted, its value will be written back to the physical memory for both threads automatically. Note that instruction cache cannot be shared because the two threads contain different binary code images.

After a machine code is translated and decoded, the hardware pipeline will first determine whether the instruction is from the primary thread or not. The instructions from the primary thread are always executed. If the decoded instruction is from the shadow thread, it will be compared with the corresponding instruction of the primary thread. Result of the comparison is stored as a single bit value in the related ROB (re-order buffer) entry of the primary thread. When instructions are committed and retired from the ROB, the matching bits will be checked. The hardware will raise exception if it detects mismatch of decoded instructions. From OS perspective, the two lockstep threads behave as one thread. OS

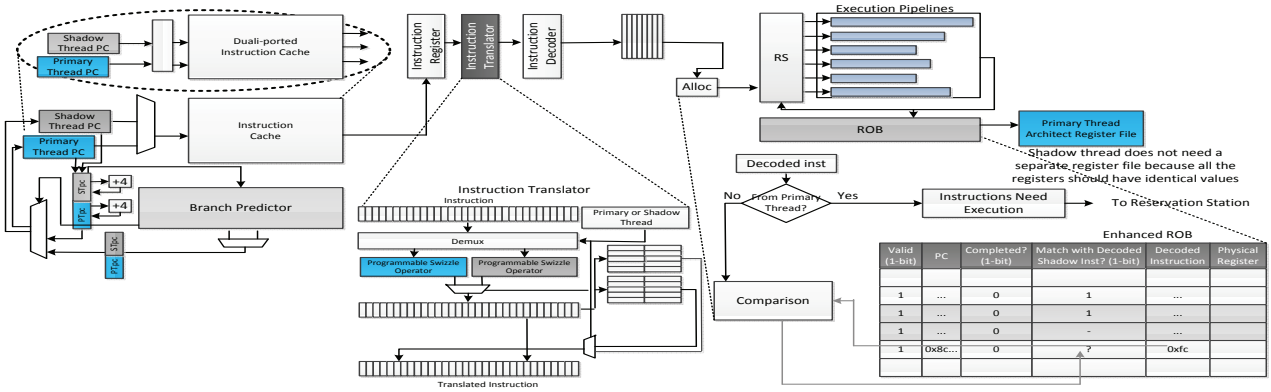

Fig. 4. Extensions to Processor Execution Pipeline. The figure shows the main extensions to an Out-of-Order threaded processor architecture for supporting the proposed lockstep shadow thread model. Each thread has its own program counter, and decoding context. When mismatch between a pair of decoded instructions from the main and shadow thread is detected, the hardware will record it in the ROB (Reorder Buffer). When the main thread instruction is to be committed and there is a mismatch, an exception will be raised and handled by the OS kernel. Most instructions from the shadow thread don't need execution because the two threads are identical.

resources are only allocated once for the thread pair. Similarly, OS level operations (e.g., sending a network packet) are only performed once. Since both threads share the same program counter and fetch instructions in lockstep. They can share the same branch prediction logic.

IV. Security Remarks

The security of our framework is fundamentally based on the fact that, (i) it is very difficult for a remote attacker to recover portions of the lookup tables; and (ii) even an attacker can recover the lookup table, he/she cannot attack both threads at the same time using the same injected codes. In this section, we analyze how secure our framework is against remote attackers.

Recall that in the threat model, an attacker attempts to penetrate into a system via code injection attack. He/she does not have the executable version of the binary code, nor be able to observe the instruction-by-instruction state change. Using the terminology from cryptographic analysis, the attacker cannot launch even the ciphertext-only attack, let alone the much more powerful known-plaintext attack, chosen-plaintext attack, and chosen-ciphertext attack. In the following, we analyze security of the system against the random attack, by evaluating the success probability of the attacker.

For 32 bit MIPS, there are 32! swizzle possibilities. Each opcode has 2^n possible values where n is the size of opcode. The relatively large search space increases the difficulties of brute fore attacks. Besides this first line of defense, the lockstep thread mode will further thwart any code injection attacks because the two threads share the same program counter and dynamic data. Injected codes work for one thread will be decoded into mismatched instructions for the other thread. Detailed studies of our solution confirms its effectiveness for detecting remote binary code injection attacks, see Section VI(B) for detailed security evaluations.

V. Evaluation

In order to demonstrate the feasibility of our system design, we have conducted several experiments and simulations. From hardware perspective, we implemented the extended hardware at register-transfer level using Verilog. In addition, we show that the area and power overhead of the extended hardware is suitable to fit in the CPU.

We show the overall performance degradation with 12 SPEC CPU 2K6 benchmarks [24]. Along with that, we list all the possible overhead by adopting our system including L1 cache miss overhead, total performance overhead, and etc. In particular, we extended GEM5 [8] simulators for cycle based full system evaluation. GEM5 is a detailed CPU architecture simulator built from a combination of M5 [9] and GEMS [18] simulators. GEM5 supports most commercial ISAs such as x86, ARM, and MIPS. It can run a full system simulation and provide a cycle based model for out-of-order processors. In addition, we implemented a full system MIPS emulator that can validate our lockstep shadow thread design at functional level.

A. Implementations

A key component of our design is the instruction translator that is able to translate scrambled machine codes into available ISA. For tuning the GEM5 simulator, the instruction translator is implemented in Verilog. Results of the Verilog implementation were used to develop the cycle based MIPS CPU model.

B. Benchmarks

For performance evaluation, we used the SPEC CPU2006 benchmark suite [24] that is a set of benchmark applications designed to test the CPU performance. We tested twelve memory intensive benchmarks of the SPEC CPU2006. These include, 8 integer benchmarks and 4 floating point benchmarks. In particularly, there are bzip2, gcc, mcf, gobmk, hmmer, sjeng, libquantum, h264ref, omnetpp, calculix, lbm, and gemsFDTD. The detailed descriptions of the benchmarks can be found in [24]. The simulation started when the application passed the initialization stage. The cycle based simulation executed each benchmark application for one billion instructions or until it finished depending on which one was longer.

C. Machine Parameters

We modified the GEM5 simulator to simulate instruction translation and lockstep multi-thread support. The configuration setting is based on results of the Verilog implementation. In particular, the multicore CPU has multiple configurations with different instruction cache sizes. The simulation is performed with an out-of-order CPU model running at 2GHz and MIPS ISA. The CPU model has seven pipeline stages: fetch, decode, rename, issue, execute, writeback, and commit. We merge the instruction translation into decode stage. Each processor core has pipeline resources: branch predictor, reorder buffer, instruction queue, load-store queue, and functional units. Specifically, each processor core has two program

TABLE I
SIZE AND LATENCY COMPARISON BETWEEN DIFFERENT APPROACHES

	AES-192	AES-256	Our Binary Diversification Approach
Number of Slice Registers	5280	6848	1506
Number of Slice LUTs	4264	6503	795
Number of bonded IOBs	449	513	263
Number of Block RAM/FIFO	100	121	47
Maximum Frequency (MHz)	324.6	324.6	600
Latency (ns)	3	3	1.6

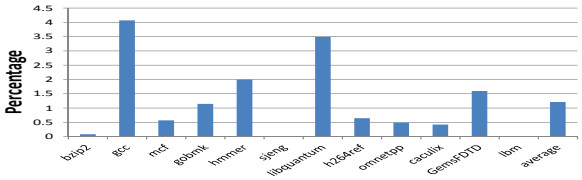

Fig. 5. Total Performance Overhead (measured as percentage of performance change)

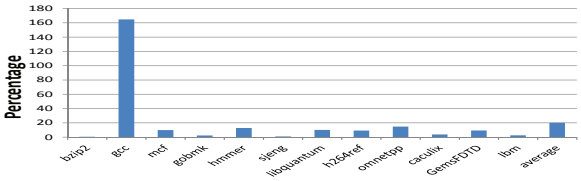

Fig. 6. Instruction Cache Performance

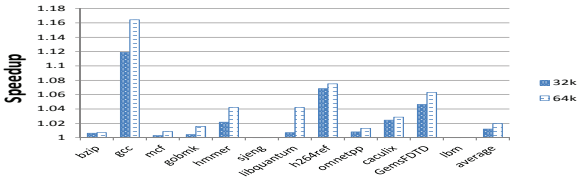

Fig. 7. Performance Improvement under Different Instruction Cache Sizes

counters and is able to support two threads simultaneously in lockstep. The I-TLB and D-TLB have 64 fully associative entries. The L1-instruction and L1-data caches are 64KB write-back caches with 64-byte block size, and an access latency of 2 cycles. The L2 cache is unified, virtually indexed, non-blocking, 2MB size, 16-way associativity, 128-byte block size, and has an 10-cycle access latency.

VI. ANALYSIS

A. Performance Analysis

A fully synthesizable implementation of the CPU extensions to support programmable binary diversification at 45nm, occupies additional 1.07 mm^2 area, and dissipates extra 32.2mW of peak power. Most of the silicon area and power are consumed by the swizzle operator and lookup table. The on-chip resource overhead is very low when compared with the size and power consumption of a modern commercial micro-processor. For example, Atom 450 fabricated in 45nm has a transistor count of 123M and die size of 66mm^2.

Based on our Verilog implementation, we compared our programmable binary diversification approach with ISA randomization using cryptographic functions such as AES. We used optimized and pipelined AES Verilog implementation as reference. All the designs were tested and verified using FPGA. Comparison of the FPGA overhead and performance is shown in Table I. As suggested by the results, our approach requires less resources when compared to ISR implemented using AES cores and at the same time achieves lower instruction decoding latency. Our approach is much simpler than the conventional ISR in both design and implementation. Though the implementation of our approach contains lookup tables and swizzle operators, ISR implemented using a cryptographic core is far more complex.

For the benchmarks, performance overhead of the introduced binary translator is shown in Figure 5, the average system degradation is around 1.2%. GCC benchmark has the highest overhead due to the increase of instruction cache miss rate. In the proposed design, the instruction translator is merged into the pipeline stages. Thus the overhead of instruction translation is small. However, the binary size is twice as the normal binary, instruction cache miss rate will increase because of the fixed instruction cache size. As shown in Figure 6, the increased L1 instruction cache miss rate is more than 20% on average. The figure also demonstrates that the performance degradation is highly related to the instruction cache miss rate. The L1 cache miss rate of GCC benchmark raises 160% when executed as a thread pair. Our investigation reveals that of all the studied benchmark applications, GCC has the most number of conditional branches. When both threads are turned on, it leads to higher pressure on the L1 instruction cache. In addiction, Figure 7 indicates that larger instruction cache can achieve better performance. Doubling the L1 instruction cache size is enough to tolerate the instruction fetch pressure caused by running two lockstep threads.

B. Security Evaluation

Our proposed approach provides a lightweight solution for diversifying the binary to prevent code injection attacks on embedded systems. We studied and analyzed several binary vulnerabilities such as shellcode attacks, return-to-libc attacks, format string vulnerability based on the proposed prototype under an emulated MIPS environment using MIPS-gnu cross-compiling tools. The injected exploit codes were written manually and tested. The MIPS shellcode attacks were developed according to [22]. Apparently the injected code, such as shellcode, depends on the ISA support. Using our MIPS functional emulator with instruction level cross-validation between the main and shadow thread, the shellcode which contains malicious MIPS machine code, when injected to the test binary, is detected at the decoding state because it produces mismatched instructions between the main thread and the shadow thread. As a result, the attack becomes invalid and fails. In addition, we tested our solution against a real attack to MIPS based SOHO devices [13]. The vulnerability and attacks are documented in [13]. For experimentation, we isolated the vulnerable program and developed customized exploit codes according to [13]. According to the evaluation, the injected code can be detected using our diversified shadow execution solution.

VII. RELATED WORK

Creating a diversified environment is a promising approach to thwart cyber attacks as advocated by Forrest et al. [14].

TABLE II
EVALUATED CODE INJECTION EXPLOITS AGAINST OUR APPROACH. ALL ATTACKS CAN BE DETECTED.

Exploit	Attack-Vector	Detected?
shellcode [22]	shellcode code injection	✓
return to libc	code injection	✓
format string	code injection	✓
SQL injection [13]	code injection	✓

In the past decade, numerous diversification strategies have been instantiated, such as ASLR [20], [5], [7], instruction set randomization (ISR) [3], [17], data randomization [11], [10], [6], and N-Variant system [12]. In this section, we briefly compare our new lockstep diversified shadow execution with each of these techniques.

Address Space Layout Randomization (ASLR): Being a practical technique, ASLR has been widely adopted by many modern OSes such as Windows and Linux. The goal of ASLR is to obscure the location of code and data objects that are resident in memory, including the addresses of the program stack, heap, and shared library code [20], [5], [7]. Compared with all these software ASLR approaches, our work complements with them by adding another layer of randomizations from hardware.

Instruction Set Randomization (ISR): By randomizing the underlying system instructions [3], [17], ISR is an approach to prevent code injection attacks. In this approach, instructions are encrypted with a set of random keys and then decrypted before being fetched and executed by the CPU. ISR is effective for preventing code injections. However, it has significant performance slowdown and overhead, which in fact motivated us to develop this more efficient and lightweight hardware approach. Furthermore, ISR fails when the encryption key is leaked or an attacker succeeds in guessing the key [23]. In our solution, an attacker cannot fool both threads using the same injected codes even the attacker knows both maps.

Data Randomization: Besides the randomization to program instructions, program data can also be encrypted and decrypted. PointGuard [11] encrypts all pointers while they reside in memory and decrypts them only before they are loaded into CPU registers. Recent work has presented a new data randomization technique that provides probabilistic protection against memory exploits by XORing data with random masks [10], [6].

Orthrus: Orthrus protects software integrity by exploiting multi-core architecture and executing n versions using different processor cores [15]. Different from Orthrus, our solution is based on lockstep simultaneous co-threading. Furthermore, other unique properties of our solution include, variants of machine codes based on programmable binary translations, and non-executing simultaneous shadow threads.

Multi-variant System N-variant [12] is an application level framework which employs a set of automatically diversified variants to execute the same task in a loosely coupled manner. Any divergence among the outputs will raise an alarm and can hence detect the attack. Multi-variant system makes code injection significantly hard for attackers to simultaneously subvert all the running variants. Different from N-variant that employs different ISAs and synchronizes at system call level, our solution exploits machine code variants of the same ISA and synchronizes at per instruction level. Due to the coarse level of synchronization, N-variant can be compromised by an attacker if the attacker can subvert all the variants before a synchronization point is reached. As a result, our solution does not have this vulnerability. Furthermore, N-variant requires complete executions of all the variants, while in our solution, execution of shadow thread is minimized.

VIII. Conclusion

We have developed a programmable decoder based binary diversification scheme and lockstep shadow execution to fight against code injection attacks. By monitoring the diversified machine codes at instruction-level, the system can detect remotely injected binary code attacks including unknown code injections. The speed degradation of the proposed design is around 1.2% across twelve SPEC CPU2006 benchmarks. In addition, the power overhead of the extend hardware is less 1% of the modern CPU.

References

[1] Ang Cui, J. V. Print me if you dare, firmware modification attacks and the rise of printer malware, 2012.

[2] Ang Cui, M. C., and Stolfo, S. J. When firmware modifications attack: A case study of embedded exploitation. In *NDSS* (2013), The Internet Society.

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

[4] Berger, E. D., and Zorn, B. G. Diehard: probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI'06)*.

[5] Bhatkar, S., DuVarney, D. C., and Sekar, R. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proc. 12th USENIX Security Symposium* (2003), p. 8.

[6] Bhatkar, S., and Sekar, R. Data space randomization. In *Proc. Int. Conf. Detection of Intrusions and Malware, and Vulnerability Assessment* (2008), pp. 1–22.

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

[8] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. The gem5 simulator. *SIGARCH Comput. Archit. News 39* (Aug. 2011), 1–7.

[9] Binkert, N. L., Dreslinski, R. G., Hsu, L. R., Lim, K. T., Saidi, A. G., and Reinhardt, S. K. The m5 simulator: Modeling networked systems. *IEEE Micro 26*, 4 (July 2006), 52–60.

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

[11] Cowan, C., Beattie, S., Johansen, J., and Wagle, P. PointGuard™: Protecting pointers from buffer overflow vulnerabilities. In *Proc. 12th USENIX Security Symposium* (2003), pp. 91–104.

[12] Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., and Hiser, J. N-variant systems: a secretless framework for security through diversity. In *Proceedings of the 15th Conference on USENIX Security Symposium* (2006).

[13] Cutlip, Z. Sql injection to mips overflows: Rooting soho routers, 2012.

[14] Forrest, S., Somayaji, A., and Ackley, D. H. Building diverse computer systems. In *Proc. 6th Workshop on Hot Topics in Operating Systems* (1997), p. 67.

[15] Huang, R., Deng, D. Y., and Suh, G. E. Orthrus: efficient software integrity protection on multi-cores. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems* (2010), ASPLOS '10, pp. 371–384.

[16] Kc, G. S., Keromytis, A. D., and Prevelakis, V. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*.

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

[18] Martin, M. M. K., Sorin, D. J., Beckmann, B. M., Marty, M. R., Xu, M., Alameldeen, A. R., Moore, K. E., Hill, M. D., and Wood, D. A. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News 33*, 4 (Nov. 2005), 92–99.

[19] Novark, G., Berger, E. D., and Zorn, B. G. Exterminator: automatically correcting memory errors with high probability. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation* (2007).

[20] PaX Team. PaX address space layout randomization (ASLR), 2003. pax.grsecurity.net/docs/aslr.txt.

[21] Portokalidis, G., and Keromytis, A. D. Fast and practical instruction-set randomization for commodity systems. In *Proceedings of the 26th Annual Computer Security Applications Conference*.

[22] Scut. Writing MIPS/Irix shellcode. http://www.thc.org/root/docs/exploit_writing/mipsshellcode.pdf, 2001.

[23] Sovarel, A. N., Evans, D., and Paul, N. Where's the feeb? the effectiveness of instruction set randomization. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*.

[24] Standard Performance Evaluation Corporation. SPEC CPU2006.