CPU Security Benchmark

Jianping Zhu, Wei Song, Ziyuan Zhu, Jiameng Ying, Boya Li,
Bibo Tu, Gang Shi, Rui Hou, Dan Meng
Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
{zhujianping,songwei,zhuziyuan,yingjiameng,liboya,tubibo,shigang,hourui,mengdan}@iie.ac.cn

ABSTRACT

The current electronic-economy is booming, electronic-wallets, encrypted virtual-money, mobile payments, and other new generations of economic instruments are springing up. As the most important cornerstone, CPU is facing serious security challenges. And with the blowout of actual application requirements, the importance of CPU security testing is increasing. However, the actual security threats to computer systems are also becoming increasingly rampant (now attackers often use multiple different types of vulnerabilities to construct complex attack systems, not just a single attack chain). The traditional vulnerability detection model is not capable of comprehensive security assessment.

We first proposed a comprehensive CPU Security Benchmark solution with high coverage for existing known vulnerabilities, including Undocumented Instructions detection, Control Flow Integrity test, Memory Errors detection, and Cache Side Channels detection, Out of Order and Speculative execution vulnerabilities (Meltdown and Spectre series) tests, and more. Our benchmark provides meaningful and constructive feedbacks for evading architecture/microarchitecture design flaws, system security (OS and libraries) software patches design, and user programming vulnerabilities tips.

We hope that the work of this paper will promote the computer system security testing from the past scatter point and line mode (single specific vulnerability and attack chain testing) to coordinated and whole surface mode (multi-type vulnerabilities and attack network testing), thus creating a new research direction of the comprehensive and balanced CPU Security Benchmark. Our test suite will play an inspiring role in the comprehensive assessment of security in personal computer devices (PC/Mobile Phone) and large server clusters (Servers/Cloud), as well as the construction of more secure Block-Chain nodes (IOT), and many other practical applications.

CCS CONCEPTS

• **Security and privacy → Vulnerability scanners**; **Access control**; **Penetration testing**;

SecArch'18, October 15, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5991-7/18/10…$15.00
https://doi.org/10.1145/3267494.3267499

KEYWORDS

CPU Security, Vulnerability Detection, Comprehensive Benchmark

ACM Reference Format:
Jianping Zhu, Wei Song, Ziyuan Zhu, Jiameng Ying, Boya Li, and Bibo Tu, Gang Shi, Rui Hou, Dan Meng. 2018. CPU Security Benchmark. In *1st Workshop of Security-Oriented Designs of Computer Architectures and Processors (SecArch'18), October 15, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3267494.3267499

1 INTRODUCTION

With the booming of the electronic-economy, the electronic-wallets, encrypted virtual currency, mobile payments, and other new generations of economic instruments continue to emerge. Because of the temptation of potential profits, the density, intensity and diversity of attacks on the above application scenarios have risen sharply. Whether they are servers cluster as data centers, personal terminal devices, or IoT nodes, they are all at risk of attacks. The commonality of above three is that they are all inseparable from CPU security. The construction of convenient mobile payment system and virtual currency trading platform are both inseparable from this important cornerstone. While, the research on CPU security at the architecture or microarchitecture level is still insufficient. Is the existing CPUs (PC, Server, IoT node, etc.) secure enough to complete the tasks of building large-scale electronic-economic infrastructures? The resolution of this fundamental problem is imminent.

Combined with the previous analysis, there is a contradiction between the blowout of security application requirements and the insufficient CPU security research status. The existence of this contradiction has prompted the industry to urgently need a standardized set of CPU security tests, in order to give a comprehensive safety assessment. This benchmark must be comprehensive enough to reach a wide coverage, because with the continuous discovery of many different types of vulnerabilities, modern attacks often use diverse vulnerabilities to construct complex attack systems. If only a few part of the vulnerabilities are detected, it is not sufficient for the security of whole CPU system. The age of point-to-point detection is gone, and only a comprehensive security assessment system can ensure the safety of the CPU. More importantly, until now, there is still no high-coverage rate comprehensive CPU security test benchmark available in the industry and academia. Grain and fodder must be prepared in advance for the troops and horses. A comprehensive assessment of CPU security must precede large-scale applications.

The main design idea of our CPU Security Benchmark is: A large set of comprehensive, high-coverage and low-cost POC(Proof of Concept) vulnerability detection codes plus few targeted and tortuous vulnerability detection codes, and according to the application needs of different occasions, integrate the test results of different

code combination modes, and give targeted comprehensive assessment feedback. The overall structure is shown in Figure 1.

The key contributions of this paper are as follows:

(1) The first comprehensive high-coverage CPU Security Benchmark, which provides comprehensive CPU security assessments for a variety of commercial servers and desktops processors (including Intel, ARM, AMD, Loongson, Zhaoxin, Phytium, Hygon, Shenwei, VIA etc.).
(2) This paper extended the research field that comprehensively evaluate CPU security, and analyzed the practical utility of comprehensive CPU security testing, which pointed out the direction for subsequent researches.

2 BACKGROUND

2.1 Shortcomings of Existing Vulnerability Detection

Traditional computer system vulnerability detection mechanisms are often based on the scanning and verification of existing operating systems or administrator programs, as well as a large number of applications and code libraries. However, the amount of code in the OS Kernel or System Hypervisor or library are huge, and it is very difficult to detect the hidden gadgets.

For the open source Linux OS Kernel and Xen Hypervisor or OpenSSL library, its open source features have lowered the attack threshold to some extent. In the three years from 2015 to 2017, more than 80 new security vulnerabilities were discovered in Linux every year [4]. And a single year in 2017, more than 60 flaws were discovered in Xen [8]. For example, in the work of Matthieu et al. [17] collected more than 2,200 vulnerable files, which accounted for 863 vulnerabilities and calculated more than 35 software indicators, but only found three critical vulnerabilities in OpenSSL, and two kinds of vulnerabilities that are critical in the Linux kernel.

Microsoft's Windows OS, although not open source, has a long history of heavy historical burdens and new features that are constantly accumulating. There are also problems of bloated scale and complex code that are difficult to detect. For example, Kostyantyn et al.'s work [24] performed security vulnerabilities detection in C code, which tested over 700 test cases belonging to SARD-100 test suite of the SAMATE project and Toyota ITC Benchmark, by using three advanced runtime verification tools (E-ACSL, Google Sanitizer and RV-MATCH). The results of their experiments showed that the number of seed defects cumulatively detected by the selected tools was only 84%.

Even the complexity of a single application goes far beyond the capabilities of existing detection mechanisms. For example, Firefox [5] includes more than 36 million lines of code accumulated over 10 years. Complex code undoubtedly has more opportunities to exploit security vulnerabilities through attacks [22].

On the other hand, hackers are increasingly understanding the various levels of computer systems. Even Intel, which is tight-lipped about the underlying architecture/microarchitecture details, has an inability to avoid the continued exposure of its advanced CPU architecture flaws. Such as Rowhammer [16, 20], Spectre and Meltdown series [18, 19]. Taking spectre variant 1 as an example, it synergistically exploits the three aspects of flaws including Speculative branch bypass, Cache side channel, and the Permission check flaw of user space to kernel space. These vulnerabilities almost vertically straddle the OS, ABI(Application Binary Interface), architecture and microarchitecture computer system levels. Spectre played a beautiful combination of punches, forcing major computer architecture vendors to rethink safer architecture designs.

2.2 Limits of Existing Trusted Computing

Given the inadequacy of traditional vulnerability detection, cautious designers have to include key code in the TCB(Trusted Computing Base). The construction of TCB itself must take a different approach. Some existing practices seek to find software isolation mechanisms from lower-level physical hardware support, such as SGX [2, 9, 12] and Sanctum [13]. However, an attacker can also build a corresponding attack by exploiting the physical characteristics of the underlying hardware. For example, Wang Wenhao et al. found a series of memory side channels that broke the isolation of Intel SGX [25].

In another example, shortly after the spectre vulnerability was exposed, researchers soon discovered its derivative variant on SGX called SgxPectre [11]. This example shows that the current TCB (Trusted Computing Foundation) or TEE (Trusted Execution Environment) has a certain degree of backwardness in security concepts. The underlying hardware structure and microcode details of Intel SGX are not public. However, the old routine by concealing the underlying details to achieve security, which is easy to fall into a dead end. Ordinary users do really not have access to the underlying details. But this does not guarantee that hackers will never guess the details, not to mention that the design, manufacturing, and supply chain management of the underlying hardware and software are not necessarily trusted. The RISCV-based open source TEE being developed by Dawn Song's Keystone project [3] team (including members of the original Sanctum team) is exploring innovation in security concepts. They defined the routine that concealed the underlying details to ensure security as "Security by Obscurity". They themselves proposed "Open Security", they open source the underlying infrastructure design, so that industry and academic community can better participate in the improvement of the design. And the security of TEE under the open source, the theoretical logic is strictly demonstrated.

However, there is also an unfortunate fact that whether it is SGX or sanctum (keystone), up to now, still very little breakthrough on contradiction between performance and strong isolation. If they are to be truly isolated, they must sacrifice the performance of speculative interactions with the outside world. Sometimes the performance of normal interactions is sacrificed. For example, SGX does not have a secure peripheral IO interface. This inevitably loses the extended performance of the connection to the peripherals, limiting the possibility of massively parallel computing such as machine learning. Because the data in the TCB is not self-produced, the data source must come from the outside, and the processing results often require external feedback. This contradictory industry has not yet found an elegant solution, so the existing TCB deficiency and the newly added vulnerabilities on the TCB have in fact increased the necessity and workload of the security test benchmark.

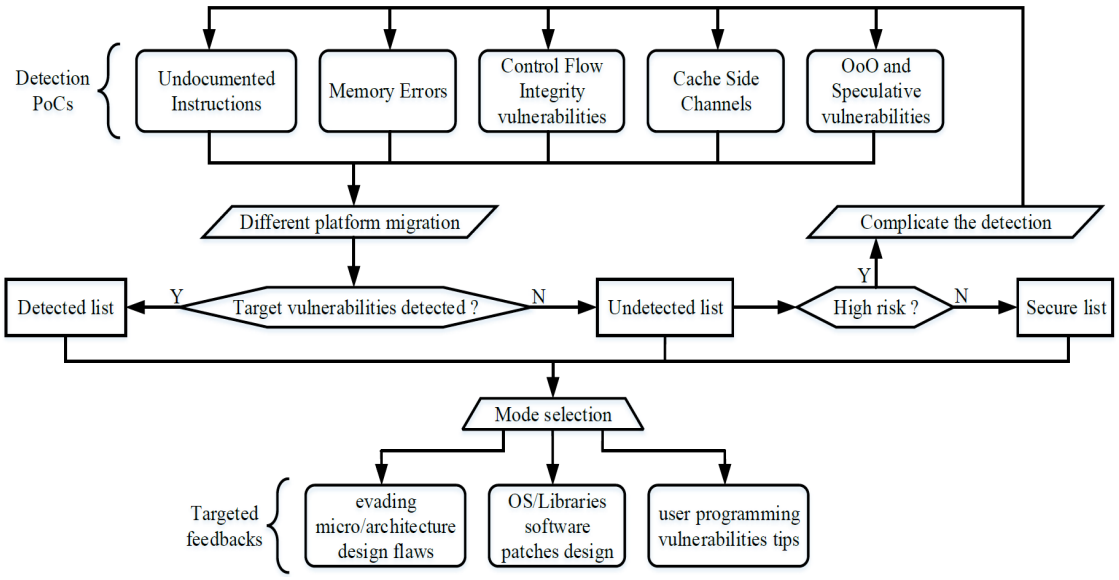

Figure 1: Overall design of CPU Security Benchmark.

2.3 Comprehensive CPU Security Testing Challenges

Designing a comprehensive high-coverage CPU security detection standard mechanism, in order to make up for the above shortcomings, faces many challenges:

Differences between different platforms. Different implementations are required on different architectures. Different architectures use different Instruction Set Architectures (ISA). Different compiling softwares, the underlying API (application interface functions) supported are also different. So even with code that is identical in principle, code that runs successfully on one platform may not work when ported to another platforms. And sometimes, the code migration works of different platforms are quite heavy.

Explosive diversification of vulnerabilities. The security coverage of the CPU is wide, and security vulnerabilities and attack methods are emerging one after another. How to ensure the high coverage of the evaluation? How to follow up with the test code as the attack progresses? How to take into account the loopholes we are not good at? An important unfortunate fact in the security field is that for a vulnerability detector, even if 999 of a thousand vulnerabilities are detected, it is not safe to miss one; for an attacker, one success in a thousand attacks is a success. The process of circumventing subjective deviations and constantly moving closer to objective laws is long and painful.

Attack chain is deeply complicated. Every hacker has his own means and methods, which vary from person to person. For the vulnerability tester, the simpler the test code, the smaller the side effects on the original system, the better. However, unfortunately, with the complexity and variety of vulnerabilities, vulnerability features are hidden in layers. Many of the original effective detection methods have been bypassed, and the detection process has become more and more tortuous. The key is that we can't let all kinds of hackers take the initiative to expose their own attack routines and combination strategies to us. Therefore, we start from the mechanism, testing on the basis of control flow integrity, side channels, memory errors, speculative execution vulnerability and so on. Then through the in-depth study of the real attack chain, we gradually improve the process and combination mode of our test. Recurring the whole attack chain is costly. This is not the purpose of our benchmark, but a last resort for some slippery flaws.

3 OVERALL DESIGN OF BENCHMARK

The main testing process of our Benchmark is designed as follows: We first collected a series of common vulnerabilities and analyzed their basic characteristics one by one. Then separately built lightweight and small cost POC code model on different CPU architecture platforms, and carried out the principle verification. If after this widely covered batch of concise POC codes test, successfully detected the target vulnerability and completed the security assessment task. Then, there is no need to go through a more costly and more complex inspection process. If a certain type of high-risk vulnerability is not detected in the above period, then the attack chain collaborative utilization model of this type of vulnerability is combined to explore a more complex testing process. In order to perform targeted high-pressure tests on the entire CPU system, this system may with certain defense mechanisms. Finally, according to different application requirements, the test results of different combination modes are integrated, and the targeted comprehensive evaluation feedback is given.

A large set of comprehensive, high-coverage and low-cost PoCs plus few targeted and tortuous detection codes. First, migrate code based on different platforms. Then run all PoCs one by one. The results of the first round of operation are divided into Detected list and Undetected list according to the preset vulnerabilities. Then analyze whether the vulnerabilities in the Undetected list is high risk, and include the low-risk vulnerabilities in the Secure list, while

complicate the high-risk vulnerabilities detection code and repeats the loop detection. Finally, according to the application needs of different occasions, integrate the Detected, Undetected and Secure lists with different modes, and give targeted comprehensive assessment feedback. The overall design of our CPU Security Benchmark is shown in Figure 1.

4 COMPONENTS DESCRIPTION

4.1 Undocumented Instructions detection

The processor sometimes is not a trusted black box, in which hardware bugs, unknown machine instructions, and some unexpected behaviors in different architecture chips have been revealed by researchers. Instruction-level bugs like Pentium 0xF00F, which is CMPXCHG8B instruction code, can result in the processor ceasing to function until computer is physically rebooted [1]. Moreover, the VIA C3 processor was explored a 0x0F3F undocumented instruction code, which offers ring3 to ring0 privilege escalation [14]. We have to find these undocumented instructions and hardware bugs ahead of time. The undocumented instructions and instruction integrity should be taken seriously. While, traditional instruction verification and validation methodology focuses on the functions of processor, the undocumented instructions have been paid little attention.

There are challenges in undocumented instruction detection: the complexity of architecture instruction set of different platforms, and the magnitude instruction search space. The difficulty is that CISC ISA instruction could be up to 15 bytes long, and the search space of it is at least $1.3 * 10^{36}$ instructions. Moreover, the CISC ISA instruction format is complex, some of them are of the form of long combination of prefixes and opcodes. Fortunately, the RISC ISA instruction format is relatively simple, and the search space is $4.9 * 10^{9}$. But it is still tough work for a PC to traverse all the space of instructions in short time.

Little research work is about the detection of the undocumented instructions. Sandsifter [15] which is based on observing changes in instruction length and page fault analysis allows effective exploring the meaningful search space of the x86 ISA. But it does not cover the legal instructions in a method of unreasonable use, and the disassemble library tool it uses is not suitable for some self-defined ISA. Furthermore, the approach of random instruction generation cannot traverse all the instruction space and cannot find arbitrary complex instructions. The method of generation based on the architecture instruction formats cannot find undocumented and arbitrary complex instructions.

In this paper, an undocumented instructions detection tool is proposed. We developed an automatic instruction generator to generate instructions, execute them, and analyze the results. The tool has two advantages. First, it has the ability of full instruction code space traverse. Second, it provides efficient instruction space traversal capability. The undocumented instruction detection tool works as the following steps:

(1) Generate instructions: use automatic generation program to generate instructions;
(2) Filtrate instructions: put the instructions into memory, the processor fetches the instructions, and the decoder checks whether the instructions are executable or not;
(3) Execute instructions: run these executable instructions;
(4) Analyze executable instructions: analyze the function of the executable instructions, check the relative registers value and store illegal instructions.

The tool has been used in the undocumented instructions detection in different platforms including ARM, MIPS ISA.

4.2 Memory Errors detection

Memory corruption bugs are the foundation of many kinds of mainstream attack technology at present, such as ROP, heap spray and ret2libc. As for memory errors, we proposed a layer-based, fast convergence benchmarks which consist of many elaborate test cases to analyze the defense capabilities of tested machines with Linux OS and different computer architectures. The framework of memory errors detection is shown in Figure 2.

Buffer overflow is the most typical attack in memory errors, which composes three stages: out-of-bounds access through pointer, modifying code pointer and hijacking control flow. Whether a stage takes effect is strictly dependent on the success of previous stage(s). Therefore, a three-layer framework is built to evaluate the security. Test cases in each layer evaluate whether the tested machine is susceptible to vulnerabilities in each attack stage. If an attack case fails, there is no need to test for further layers, which contributes to a fast convergence. Through these benchmarks, the stages which are vulnerable to attacks or defensed successfully will be exposed accurately.

The attackers make the pointer going out of the bound, which is the basis of buffer overflow attacks, to trigger a memory error. By excessively incrementing or decrementing an array pointer in a loop without proper bound checking, a buffer overflow/underflow will happen. By causing pointer out of bounds, but the bounds check is missing or incomplete, the pointer might be pointed to any address. If an attacker controlled pointer is used to write the memory, then any variable, including other pointers or even code, can be overwritten. Within our framework, the first layer evaluate whether pointers in tested machines are susceptible to make a pointer out of bounds.

After getting the pointer out of bounds, the next goal is to modify the code pointers, such as return addresses, function pointers and jum_buf structure, which determine the control flow. The second layer assesses whether these key code pointers in tested machines are at risk of being modified by out-of-bounds pointer.

The third stage is to load corrupted code pointer to instruction pointer register. Executing function return instruction, indirect function call and indirect jump will hijack control flow to the pointers modified in the second stage. Diverting the execution from the control-flow defined by the source code, which is a violation of the control flow integrity (CFI) policy, is estimated in the third layer.

4.3 CFI vulnerabilities detection

According to the categories of control flow transfer, we divide the the CFI benchmark into 2 parts: forward and backward [10], as shown in Figure 3.

Forward. In cpu's ISA, there are two instructions for forward control flow transfer, call and jump. The instruction call is used for calling a function, so the address followed the instruction usually is the function entry. And the instruction jump is almost used to

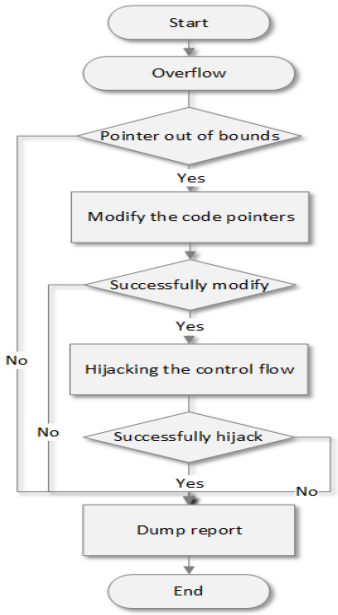

Figure 2: The structure of memory errors detection.

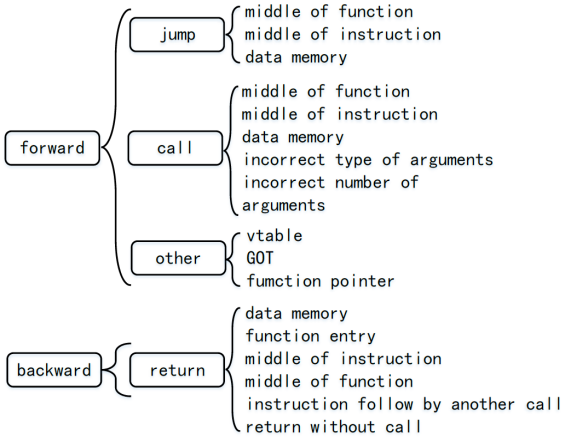

Figure 3: CFI vulnerabilities classification.

jump to a target address in local control flow, like loops and if-else statement. So some addresses are almost impossible for the forward control flow transfer instruction, for example, the middle of an instruction, which may only appear in attacks. So we make these situations to test the security of the CPU system. Function call has more limits than jump, because of its arguments. We always state the number and type of arguments in program, and compilers will help us check the correctness of arguments [23]. But the hardware doesn't check. So this is also a point for us to check in our CPU Security Benchmark. And the tests for these two instructions are as follows:

(1) jump to the middle of a function
(2) jump to the middle of an instruction
(3) jump to the instruction in data memory
(4) call to the middle of a function
(5) call to the middle of an instruction
(6) call to the instruction in data memory
(7) call to a function with incorrect number of arguments
(8) call to a function with incorrect type of arguments

The test cases list above are used to check the security of the forward control flow transfer instruction directly. And there are also many test cases to test the security of forward control flow transfer in more complex ways, which can also be used in the real world attacks. (1) virtual table pointer [21]. In object oriented language, a class object finds its virtual function address by the virtual table pointer. So we modify the virtual table pointer to test whether the system can protect the virtual table pointer and how the system to detect the changes of virtual table pointer. (2) global offset table(GOT). GOT helps the procedure linkage table(PLT) to find the address of the functions after the object files are linked with dynamic link library. We check the security of the GOT and PLT in benchmark. (3) function pointer. We check whether the system can detect incorrectness of the type and number of arguments.

Backward. In the ISA, the instruction for backward control flow transfer is just one, return. Commonly, the instruction return appears with the instruction call in pairs. And the instruction call and the instruction return are emitted by compiler as the prologue and the epilogue of a function call. So testing the consistency of the instruction call and the instruction return is necessary. Besides, the addresses of backward control flow transfer are also limited. Actually, the address for backward control flow transfer was decided when a program to call a function, which is the exactly the address of next instruction of the function call. So we change the address for the backward control flow to detect whether there is a defense for backward control flow transfer and the grain of the defense. The tests of backward control flow transfer are as follows:

(1) return to the instruction in data memory
(2) return to a function entry
(3) return to the middle of an instruction
(4) return to the middle of a function
(5) return to the instruction followed another function call
(6) different number of the instruction call and returnw

4.4 Cache side channels detection

The Cache side channel mainly uses the access time difference of the hits or misses of the caches of different hierarchies (and their auxiliary structures TLB, PTW, etc.) to leak information. This type of side channel requires the attacker to share part of the cache or its auxiliary structure with the victim. Common Cache side channels include Flush+Reload, Flush+Flush, Prime+Probe, Evict Time, AnC.

Flush+Reload: The attacker first flushes all the cache lines involved in the test with the flush instruction. After the victim accesses a cache line, the attacker traverses through the reload to find the cache line with a short time, which is the cache line that the victim just visited.

Flush+Flush: The attacker first flushes all the cache lines involved in the test with the flush instruction. After the victim accesses a cache line, the attacker traverses through the flush again to find the cache line with a long time, which is the cache line that the victim just visited.

Prime+Probe: The attacker first takes all the cache lines participating in the test to the cache of their own core, and waits for the victim on another core to access one of the cache lines (due to the cache coherency protocol, this will change the state of that cache line in attacker's location). The attacker traverses the access again to find the cache line with long time, which is the cache line that the victim just accessed.

Evict Time: The attacker first selects a target cache line and accesses the other cache lines of the same set as the target cache line, so that evicts the target cache line from the cache. Then observe the victim execution time. If the time is longer, the victim has visited the target cache line.

AnC: AnC attack uses the feature that PTE (page table entry) and common data sharing cache, to detect the location(set index) of PTEs in the cache during the PTW (page table walker) operation, ie the page offset in the page table. And the virtual address of the key data or code is obtained, finally bypassed the ASLR (address randomization).

4.5 Out-of-Order and Speculative Execution Vulnerabilities Detection

The Meltdown [19] and Spectre [18] series of vulnerabilities primarily exploit the residual effects of out-of-order or speculative execution on cache state to leak information. When the core pipeline finds out-of-order execution or speculative execution errors, architectural changes are flushed without affecting correctness; however, changes to the cache state at the microarchitecture level are maintained. Thus, the attacker can induce the processor to perform erroneous out-of-order or speculative execution, thereby leaking secret data in conjunction with the cache side channel.

4.5.1 ***Meltdown v3****: rogue data cache load CVE-2017-5754.* The attacker uses the interval between out-of-order execution and exception handling, illegally accessing kernel space data from the user space. On Intel CPUs that support out-of-order execution, you can use Intel's unique TSX (Transactional Synchronization Extensions) to suppress exceptions and repeat attacks efficiently. The code is shown in Figure 4. Of course, conditional jump instructions can also be used to suppress exceptions, such as the code in the ARM platform shown in Figure 5. On some ARM platforms, you can also access the system register that should not be accessed: TTBR0_EL1, (Meltdown V3a: Rogue System Register Read: CVE-2018-3640 [7]), the code is shown in Figure 6.

4.5.2 ***Spectre v1****: bounds check bypass CVE-2017-5753.* By training the BHT(Branch History Table), the attacker always executes the branch, and suddenly accesses the out-of-range secret address in the branch. The code is shown in Figure 7. Although, after the boundary condition is resolved, the branch is error guessed and an exception is triggered. However, the secret value still remains in the cache indirectly in the form of an address.The two debugging experiences of Spectre v1 are:

(1) The open() function is needed to read the kernel data into the L1 cache before you can successfully leak the corresponding data.

(2) By nesting multiple layers of cache misses or even using TLB(Translation Lookaside Buffer) misses, extending branch resolution time, can execute more malicious instructions.

```
if((status = _xbegin()) == _XBEGIN_STARTED){
    asm __volatile__ (
      "%=:                                \n"
      "xorq %%rax, %%rax                  \n"
      "movb (%[ptr]), %%al                \n"
      "shlq $0xc, %%rax                   \n"
      "jz %=b                             \n"
      "movq (%[buf], %%rax, 1), %%rbx     \n"
      :
      :  [ptr] "r" (ptr), [buf] "r" (buf)
      :  "%rax", "%rbx");
    _xend();
}
```

Figure 4: Meltdown v3 with Intel TSX.

```
  LDR X1, [X2] ; arranged to miss in the cache
  CBZ X1, over ; This will be taken but
               ; is predicted not taken
  LDR X3, [X4] ; X4 points to some EL1 memory
  LSL X3, X3, #imm
  AND X3, X3, #0xFC0
  LDR X5, [X6,X3] ; X6 is an EL0 base address
over
```

Figure 5: Meltdown v3 on ARM.

```
  LDR X1, [X2] ; arranged to miss in the cache
  CBZ X1, over ; This will be taken
  MRS X3, TTBR0_EL1;
  LSL X3, X3, #imm
  AND X3, X3, #0xFC0
  LDR X5, [X6,X3] ; X6 is an EL0 base address
over
```

Figure 6: Meltdown v3a on ARM.

```
uint8_t array3[4096];   //size = page size, cause TLB miss
uint8_t array4[4096];

…

if (x < array4[array3[array1_size]]) {   // 3 cache misses
  temp &= array2[array1[x] * 512];       // 2 cache misses
}
```

Figure 7: Spectre v1.

4.5.3 ***Spectre v2****: branch target injection CVE-2017-5715.* The attacker trains BTB(Branch Target Buffer), induces the victim to jump to the gadget, and executes the leak code. The Spectre v2 attack process is shown in Figure 8.

Spectre v2 has two difficulties for attackers: (1) Inject malicious targets into BTB and even tamper with RSB (Return Stack Buffer) / RAS (Return Address Stack). (2) Find Gadgets in the victim code that can indirectly reveal the secret value.

For this vulnerability detection, one can simplify the process by the following means: (1) Construct gadgets in victim code. (2) The

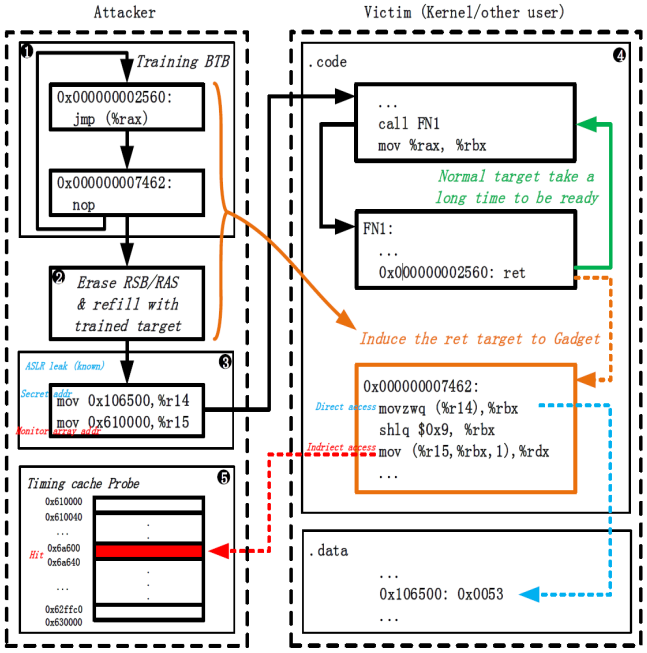

Figure 8: Spectre v2 [6, 11].

default test code has cracked ASLR (address randomization) and obtained a virtual address.

5 CONCLUSIONS AND FUTURE WORKS

This paper proposed a comprehensive CPU Security Benchmark solution with high coverage for existing known vulnerabilities, including Undocumented Instructions detection, Control Flow Integrity test, Memory Errors detection, and Cache Side Channels detection, Out of Order and Speculative execution vulnerabilities tests. The Benchmark has been experimentally verified on several real platforms, and provided meaningful feedbacks for evading architecture/microarchitecture design flaws, software patches design, and user programming vulnerabilities tips.

This article is only a preliminary version of the CPU Security Benchmark. Several future works are as follows: (1) Improving the compatibility of Benchmark code for different instruction sets and reducing the porting work of users to test code. (2) Reducing the gap between our own built gadgets and real gadgets on computer system and improving the authenticity of Benchmark. (3)Further improving the coverage of Benchmark, expanding the types and quantities of more vulnerabilities and their derivatives. (4) Designing a more reasonable comprehensive evaluation process and conclusion integration mechanism of different combination modes, to make more meaningful test feedbacks. A more complete Benchmark with more features will be available in the near future.

REFERENCES

[1] 1999. Pentium Processor Specification Update, Invalid Operation with Locked CMPXCHG8B Instruction. http://www.cpu-zone.com/Pentium/Pentium%20processor%20specification.pdf

[2] 2018. Intel® Software Guard Extensions (Intel® SGX). Retrieved August 13, 2018 from https://software.intel.com/en-us/sgx/details

[3] 2018. Keystone: Open-source Secure Hardware Enclave. Retrieved August 13, 2018 from https://keystone-enclave.org/

[4] 2018. Linux kernel: CVE security vulnerabilities, versions and detailed reports. Retrieved August 13, 2018 from https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33

[5] 2018. Mozilla Firefox (2018) Project summary. Retrieved August 13, 2018 from https://www.openhub.net/p/firefox

[6] 2018. Reading privileged memory with a side-channel. Retrieved August 17, 2018 from https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

[7] 2018. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism. Retrieved August 17, 2018 from https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability

[8] 2018. XEN: CVE security vulnerabilities, versions and detailed reports. Retrieved August 13, 2018 from https://www.cvedetails.com/product/23463/XEN-XEN.html?vendor_id=6276

[9] Ittai Anati, Ittai Anati, Ittai Anati, Ittai Anati, Ittai Anati, Ittai Anati, Ittai Anati, and Ittai Anati. 2017. Intel®; Software Guard Extensions (Intel®; SGX) Architecture for Oversubscription of Secure Memory in a Virtualized Environment. In *Hardware and Architectural Support for Security and Privacy*. 7.

[10] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Stefan Brunthaler, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *Acm Computing Surveys* 50, 1 (2017), 16.

[11] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2018. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. (2018).

[12] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86.

[13] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 857–874. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[14] Christopher Domas. 2017. Breaking the x86 ISA. https://www.blackhat.com/docs/us-17/thursday/us-17-Domas-Breaking-The-x86-ISA.pdf

[15] Christopher Domas. 2018. Hardware Backdoors in x86 CPUs. https://i.blackhat.com/us-18/Thu-August-9/us-18-Domas-God-Mode-Unlocked-Hardware-Backdoors-In-x86-CPUs-wp.pdf

[16] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 300–321.

[17] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. 2017. An Empirical Analysis of Vulnerabilities in OpenSSL and the Linux Kernel. In *Software Engineering Conference*. 105–112.

[18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. In *Spectre Attacks: Exploiting Speculative Execution*.

[19] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. (2018).

[20] Rui Qiao and Mark Seaborn. 2016. A new approach for rowhammer attacks. In *IEEE International Symposium on Hardware Oriented Security and Trust*. 161–166.

[21] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Security and Privacy*. 745–762.

[22] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. 2011. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Transactions on Software Engineering* 37, 6 (2011), 772–787.

[23] Victor Van Der Veen, Cristiano Giuffrida, Enes Goktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, and Elias Athanasopoulos. 2016. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *Security and Privacy*. 934–953.

[24] Kostyantyn Vorobyov, Nikolai Kosmatov, and Julien Signoles. 2018. Detection of Security Vulnerabilities in C Code Using Runtime Verification: An Experience Report. In *International Conference on Tests and Proofs*. 139–156.

[25] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, Xiao Feng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. (2017), 2421–2434.