Journal of Computer Virology and Hacking Techniques
https://doi.org/10.1007/s11416-020-00348-w

ORIGINAL PAPER

The self modifying code (SMC)-aware processor (SAP): a security look on architectural impact and support

Marcus Botacin[1] · Marco Zanata[1] · André Grégio[1]

Received: 7 February 2019 / Accepted: 4 February 2020

Abstract
Self modifying code (SMC) are code snippets that modify themselves at runtime. Malware use SMC to hide payloads and achieve persistence. Software-based SMC detection solutions impose performance penalties for real-time monitoring and do not benefit from runtime architectural information (cache invalidation or pipeline flush, for instance). We revisit SMC impact on hardware internals and discuss the implementation of an SMC detector at distinct architectural points. We consider three detection approaches: (i) existing hardware counters; (ii) block invalidation by the cache coherence protocol; (iii) the use of Memory Management Unit (MMU) information to control SMC execution. We compare the identified instrumentation points to highlight their strong and weak points. We also compare them to previous SMC detectors' implementations.

1 Introduction

Self modifying code (SMC) are pieces of code able to change their own structure and/or behavior at runtime [8]. Their initial usage refers to a period of storage constraints, requiring huge programming effort regarding code generation. Nowadays, SMC is often seen on payload protection cases, either benign (e.g., intellectual property protection) or malicious (e.g., binary packers for code obfuscation) [24]. SMC may also be runtime-generated on interpreted languages, such as Java and Python [17], mostly as benign cases.

SMC becomes a security concern when used on packers, as it allows malware to bypass pattern matching checks. Security solutions started to monitor code execution at distinct steps to address this issue. This monitoring requires constant checks and is usually implemented in software [24], resulting in overhead[1] (high frequency memory checks may impose performance penalties as high as 400% [32]). On the one hand, solutions performing frequent binary checks impose high-performance penalties, leading users to turn them off. On the other hand, solutions that perform less frequent checks are susceptible to timing attacks, missing code changes that happen between two checks. Therefore, improving SMC detection triggering is a relevant task to make security solutions more effective.

We may notice many architectural implications during the execution of SMC code if we look at hardware level, such as instruction and trace cache invalidation. These events suggest the processor can aid in SMC identification and on reducing software monitoring overhead. Therefore, we propose an ideal model to evaluate this possibility: SAP, an SMC-Aware Processor able to notify upper instances about any SMC execution.

Previous architectural developments to handle SMC at hardware level focused on execution speed up, aiming to solve SMC-imposed execution bottlenecks. This work, instead, focuses on identifying hardware facilities which can be leveraged to detect SMC with no significant performance penalty. More specifically, we propose implementing SMC detectors by leveraging: (i) existing hardware performance counters; (ii) instruction cache invalidation detection by the cache coherence protocol; and (iii) MMU protection bits. Therefore, we do not propose hardware changes in SAP to speed up SMC execution, but mechanisms to efficiently detect SMC. We also introduce ways to notify upper instances—Operating System (OS) and Anti-Viruses

[1] We are hereafter referring to overhead to denote the runtime monitoring overhead, as the overhead of running detection routines is unavoidable to any AV solution.

✉ Marcus Botacin
mfbotacin@inf.ufpr.br

Marco Zanata
mazalves@inf.ufpr.br

André Grégio
gregio@inf.ufpr.br

[1] Federal University of Paraná (UFPR), Curitiba, Parana, Brazil

(AVs)—about SMC execution, triggering scan mechanisms on-demand.

We classified SMC according to their impact extent over modified instructions and affected architectural entities: (i) SMC type-I affects near, cached instructions; (ii) SMC type-II affects near, pipelined instructions; and (iii) SMC type-III affects far, non-cached nor pipelined instructions. Our results show that: (i) the cache detection mechanism is able to handle SMC-type I in a general way, although it can be bypassed by specially-crafted samples using flush instructions; (ii) it is possible to build performance-counter-based detectors with existing architectural components to handle SMC type-II; and (iii) the MMU-based mechanism, although simpler, presents the best coverage and is able to detect all SMC types with near-zero logic overhead.

Target audience According to our experience, many security researchers do not understand the impact of malicious code constructions, such as SMC, at low level. At the same time, many computer architects do not have a comprehensive view of system component's impact on software security, despite having a deep technical knowledge about such components. Given this scenario, our work aims to bridge this understanding gap. Thus, more than a solution proposal, this work presents an exploratory analysis on how each software construction impacts each hardware component and vice-versa. Therefore, this paper is aimed to reach both security and computer architecture researchers in a combined effort towards enhanced security solutions.

Contributions Our contributions are the following:

– We revisit the use of SMC for malware packing and current OS and architecture support for their execution.
– We introduce a taxonomy for SMC code and detectors based on architectural detection points and detection windows.
– We evaluate the detection effectiveness and the performance impact of each aforementioned approaches.

Paper organization This paper is organized as follows: In Sect. 2, we motivate this work; in Sect. 3, we provide a background on SMC and present related work; in Sect. 4, we introduce SAP, the ideal processor model and its mechanisms to support SMC detection; in Sect. 5, we present SAP's evaluation; in Sect. 6, we discuss SMC impact as identified by SAP; finally, we draw our conclusions in Sect. 7.

2 Motivation

We here present scenarios for which improving SMC detection is essential. We first present how SMC can be used by malware to evade analysis environments. Further, we present the limits of existing software-based defenses.

SMC for bypassing malware analysis Malware samples often try to hide their malicious behavior to keep executing in infected systems. As time passes, new techniques are used to enforce stronger malicious payload protection policies, which includes using SMC. Therefore, efficient detection of SMC is an important security-related task. The methods used by the malware samples on the first bypass techniques were as simple as detecting the analysis environment by leveraging an anti-analysis "trick" and then refusing to run their malicious payloads, as illustrated in Code 1.

```
if(!anti_analysis())
{
  malicious();
}
```

Code 1 Evasive malware. Bypass consists of branching right after the execution of an anti-analysis trick.

Given the differences on the number of instructions executed on the malicious and the evasive path in this model were noticeable, which makes detection easier, attackers started to employ more similar constructions where SMC took a fundamental place. The code sample presented in Code 2 illustrates how the malicious function is called in both cases (trick succeeded or not), but its content is replaced by a malicious payload only when an anti-analysis trick succeeds.

```
if(!anti_analysis())
{
  change_page_flags(malicious);
  unsigned char *instruction =
  malicious+INST_OFFSET;
  *instruction = INST_DATA;
}
malicious();
```

Code 2 Evasive malware. Malicious function is always called, but has distinct payloads.

Although more sophisticated, this construction can also be detected due to the presence of a deviating branch–i.e. a branch which is a root cause of evasion, in the security context. State-of-the-art evasion techniques are characterized by not relying on branches for flow deviation. Willems et al. [33] presented the `delusion attack` concept, in which the same instructions behave distinctly on real machines and on emulators. As an example, consider the assembly instruction `rep movs`, which copies the number of bytes pointed by `ecx` from `esi` to `edi`, with its own address as target. As real machines perform `fetch-then-execute` atomically, no side-effect is observed. It means that the instruction will only be modified after finishing its execution. On emulators, as memory page writes are software-trapped for emulation purposes, the instruction must be re-executed after the trap routine finishes executing, thus being re-fetched after the write, resulting in a distinct behavior. Since the address from the `rep movs` instruction was modified, the emulator will actually execute the modified version of such instruction. The presented technique can be used to evade analysis systems and build attacks.

Table 1 UPX sections memory mapping. Sections are initially mapped as writable and executable at the same time

Name	Content	Permissions	Accessed
None	Header	RWE	R
UPX0	Original Code	RWE	RWE
UPX1	Unpacker	RWE	RWE
.rsrc	Resources	RWE	RW

```
1 addr ptr1 = &executable_function;
2 addr ptr2 = &writable_data_buffer;
3 target = delusion(ptr2, ptr1);
4 write_at_addr(payload, target);
5 malicious();
```

Code 3 Branch-free evasion with SMC-delusion attack.

Code 3 illustrates our proposed SMC delusion attack, which handles pointers for two memory regions: an executable function (line 1), and a writable data buffer (line 2). A delusion attack is launched having either the data buffer or the function address (line 3) as possible resulting values. The malicious payload is written in the target by the subsequent write function (line 4). If running inside an emulator, `ptr2` is considered, thus writing in the writable data buffer and not changing the original, benign function. If running on a real machine, the function offset (`ptr1`) will be considered, thus writing the payload into the function and turning it into a malicious one. In both cases, the function is always called (line 5), thus not presenting noticeable execution differences to allow evasion identification.

This attack demonstrates that anti-analysis threats could not be detected anymore due to the presence of deviating branches. Therefore, improving SMC execution detectors is essential to handle such kind of evasive threat to modern computer systems.

OS support for SMC execution Current OS rely on `eXecute Disable (XD)/No-eXecute (NX)` MMU page protections to enforce a `Write⊕Execute` [30] policy, which states executable pages cannot be writable. If a write on such pages happens, a fault is generated.

To set the NX bit, OS functions which modify memory attributes (e.g. `mmap`, `mprotect` and/or `VirtualProtect`) are trapped and instrumented to add checking capabilities. The major drawback of this approach is that the NX bit must be set when the process is launched and its memory page protections are set. Thus, either all or none SMC code is allowed to run, with no fine-grained control. As an example of this limitation, consider a benign software packed with UPX,[2] a popular open-source packer. Table 1 shows the attributed initial memory flags and the accesses performed during its execution.

[2] https://upx.github.io/.

UPX requires its pages to be initially set as both writable and executable, thus violating the `Write⊕Execute` policy. Therefore, a restrictive environment would block this benign software execution. To handle such benign cases, OS often adopt whitelisting policies. A drawback of this approach is that the SMC behavior must be known *a priori*, thus still limiting the execution of newly-created, benign software pieces.

Ideally, the OS should be able to launch processes without prior information and detect SMC at runtime, thus further deciding whether a given code should be allowed to run (benign) or not (malicious). The decision whether a given SMC execution is due to malicious code or not could be, for instance, outsourced to a third-party entity, such as an AV mechanism. Therefore, benign software whitelists would be runtime-generated instead of relying on prior code knowledge, thus scaling the protection to cover early-launched applications.

3 Background and related work

We here present an overview of how SMC is used in the security context as well as its impact on processor architectures. We also discuss existing techniques and implementations to handle SMC.

SMC and malware SMC can be implemented by distinct techniques [35], such as instruction replacement, dynamic code mutation (encryption), piece clustering and virtual machine protection. They can be used for malicious purposes (e.g., obfuscation through packing [5]) or legitimate software protection [34]. In the malware context, SMC is mostly used as packers, it means, software pieces which embed other software inside themselves to hide malicious features from static analysis. Packers like Telock build code in an overlapped way, requiring special disassemblers for correct interpretation [5].

To address obfuscated malware, the payload must be extracted (unpacked), which can be done in several ways [3], but many tools use the `run and dump` approach, i.e., they continuously inspect memory as execution proceeds, increasing the performance penalty due to the multiple required checks. Other solutions rely on emulation or binary translation [12,20], which are effective, but not intended to run in end-users machines. An effective way to detect SMC involves flagging memory pages as read-only and then trapping the page fault handler. Liu and Wang [21] implemented this approach at the software level and addressed the `SIGSEGV` signal on affected applications. OmniUnpacker [24] also monitor such writes at the OS level by calling an external scanner when syscalls are originated from OS pages. Compared to them, our goal is to provide hardware support for this kind of SMC detection approach, aiming at efficient detection and low performance overheads.

SMC inside the computer architecture Intel processors have native support for runtime code modification in their many variations (e.g., self modifying code in the local core, cross modifying code in a distinct core). Intel Core 2, for instance, makes use of an inclusion filter that acts as an instruction cache monitor [11]. From the architecture's perspective, the first stage impacted by SMC execution is the instruction fetch. Since instruction bytes are cached in the instruction cache once requested, those should be updated when the memory code region is written. As a general policy, processors tend to clear the whole instruction cache, a drawback pointed in [17,31,37]. The second impacted stage is the instruction decoder. As the newly fetched bytes could represent another instruction, cached instructions must be discarded. On Intel processors, the decode buffer is often named trace cache and has to be invalidated, as pointed in the manual: "For the Pentium 4 and Intel Xeon processors, a write or a snoop of an instruction in a code segment, where the target instruction is already decoded and resident in the trace cache, invalidates the entire trace cache.".

The processor back-end (formed by the rename, dispatch, execution, and write-back stages) will also be impacted. Since modern processors present big Reorder Buffers (ROB) to support aggressive out-of-order execution within a superscalar pipeline, instructions can be affected by a write after they have been already entered the pipeline. In general, processors tend to flush the entire pipeline, causing performance degradation. Since the organization changes according to the processor family, the flush conditions varies a lot, as pointed by the processor manual [18]: "IA-32 processors exhibit model-specific behavior when executing self-modified code, depending upon how far ahead of the current execution pointer the code has been modified. As processor microarchitectures become more complex and start to speculatively execute code ahead of the retirement point (as in P6 and more recent processor families), the rules regarding which code should execute, pre- or post-modification, become blurred.".

While the presented invalidation procedures solve the correctness problem, the already implemented SMC detectors do not assist security at all. A straightforward enhancement would be to add an alert routine to the existing detectors—a solution discussed in this work.

Overall solutions for SMC handling rely on additional hardware; [13] presents a full ×86 implementation with a morphing code support layer; [4] presents the Selective Snoop Protocol (SSP), which minimizes the number of cache invalidation by the snoop protocol due to SMC, thus saving energy. Such approaches, however, imply on aggressive redesign changes. In addition, most changes are performance-focused (e.g., cache): [17] suggests reducing the number of stalling instructions due to SMC to reduce cache invalidation rate and overhead; Other approaches, such as selective cache line change [31] and the line-address buffer concept [37] opt to invalidate only the SMC-affected lines. None of them consider security alerting. Although porting these approaches to detect SMC for security purposes seems straightforward, we are not aware of any other work proposing such move. Therefore, we present over this article how SMC detection could be performed based on distinct instrumentation points.

SMC and code generated at runtime A precise SMC definition is a bit hard since it can refer to many scenarios. Some work opt to cover a broad scenario, such as in [8], which defines SMC as "any program that loads, generates, or mutates code at runtime". In this paper, we consider this definition to encompass all cases in which code is mutated in runtime. This might include polymorphic and metamorphic code [6] if they employ SMC constructions. Our goal, however, is not to cluster metamorphic samples, but to detect when a code mutates itself, thus possibly becoming malicious in runtime. More specifically, in our view, SMC may be defined in at least three scenarios, based on the probability of the code modification to cause hardware side-effects (e.g., cache and/or pipeline invalidation). We consider how far the modified code is from the modifier instruction and whether the instruction was previously executed or not as proxies for the probability estimation.

SMC type-I (cached instructions) When an instruction is modifying an instruction of its neighborhood (e.g., in the same function), it is very probable the original instructions had been previously cached, either by being previously executed or by block associativity/prefetching, resulting in cache invalidation, which can be detected by cache instrumentation.

SMC type-II (pipelined instructions) the last version of the modified instruction (regardless of its type) might be present in the pipeline even when evicted from cache, resulting in forwarding issues. This can be detected by pipeline monitoring hardware counters.

SMC type-III (distant instructions) if an instruction modifies a far code, such as a function placed on a distinct memory page, the previous data is not cached nor pipelined, not triggering any execution check. The only affected functional unity is the MMU.

In addition to malware, which often employ SMC for all three discussed scenarios, modern software also performs SMC for legitimate uses, such as Just-In-Time (JIT) compilation and interpreters [2,17]. Contrary to malware, they usually generate code in the third case, far ahead of their code generators, and the generated codes do not modify themselves. This way, many researchers do not call them SMC, but runtime generated code. This is a controversy since the 90's (see the LISP case [26]).

4 SAP: an SMC-aware processor

SMC-Aware Processor (SAP) is our ideal model of a processor to investigate and evaluate the impact of implementing SMC detection mechanisms in distinct CPU components. The main idea of SAP is to raise warnings and call upper instances (OS or AV) when SMC execution is identified, thus reducing the performance penalty of continuously monitoring binaries to identify code changes. SAP is able to raise interrupts when SMC execution is detected, thus causing a traditional, software-based AV to be called on-demand. Using hardware flags to trigger an AV is not new (Microsoft submits patent claims on that since 2007 [29]), but SAP is not limited to flag pages as suspicious, but it also provides a precise mechanism to trigger inspection on effectively modified pages.

SAP introduces independent modifications in three distinct components: the CPU cache coherence protocol is modified to alert when cache code invalidation is performed; flushes in the CPU pipeline are monitored through hardware performance counters; and MMU is instrumented to enforce write protections to executable pages.

4.1 Threat model and assumptions

In SAP, we design a CPU able to detect both in-place and runtime-generated SMC cases. We focus SMC occurrences in user-land due to their prevalence, although no technical restriction is imposed to handle kernel cases. SAP monitors the system in a system-wide way, thus not requiring users to specify which processes will be monitored, However, SAP allows individual process monitoring to be turned off on-demand to allow benign SMC processes to run without any impact (whitelisting). Therefore, we assume no prior-execution blocking of writable pages by the OS, thus SMC processes can be normally launched and benign SMC cases will be runtime-whitelisted. Similarly, we assume no application signing requirement enforcement, thus allowing users to freely download apps from the Internet, as most end-users do in current desktop OS versions.

Regarding packers, SAP covers only SMC cases due to instruction modification and not due to instruction encoding in virtual machines, which does not structurally impact processor execution, our focus in this work. Finally, our goal is not to entirely replace existing AVs, but to assist them with new, efficient inspection triggering mechanisms.

4.2 SMC-aware cache

When a Type-I SMC happens, the instruction cache is directly affected. Since it is not writable, it should be updated by an operation that performs partial or full invalidation and re-fetch. Therefore, a straightforward approach to detect SMC is to raise an exception during instruction cache invalidation. This mechanism does not impose a modification on the coherence protocol, but a side-effect during instruction cache invalidation, which allows the cache to directly detect code changes. This mechanism is illustrated through the SMC in Code 4, which overwrites the increment instruction (`inc eax`) with `NOP` instructions in the odd executions, starting from the second loop iteration. From this point to the end, the code does not accumulate any value anymore.

```
for(i=0;i<3;i++){
  if(i%2==1){
    void *func_addr = (char*)foo +
        FUNC_ADJUST;
    instr = (unsigned char*)func_addr
         + INST_OFFSET;
    *instr = nop; *(inst+1) = nop;
  }else{ acc++;
```

Code 4 SMC code. Accumulator variable is overwritten.

We executed this example on a pre-decoded cache modelled with Intel PIN, a dynamic binary inspection framework that allows system components to be modelled without requiring binary recompilation and/or patching [22]. The cache was modelled (as shown in Code 5) to detect when the decoded instruction for the same index differs.

```
UINT64 idx = ((UINT64)PC >>
     LINE_INDEX) && (LINES-1);
for(int j=0;j<LINES;j++){
  if(cache[idx][j].valid && cache[idx
       ][j].tag==PC){
    if(current->disassembly != cache[
        idx][j].decoded){
      // SMC Code
```

Code 5 SMC-aware cache. Checking decoded instruction for the same index.

When running the sample code in the aforementioned prototype, the following result is observed (Code 6).

```
13F7111DE inc eax;
13F7111DE nop;
13F7111DF nop
<<SMC Detected>>
```

Code 6 SMC-aware cache executing the SMC sample.

As expected, the first iteration caches and executes the increment instruction (2-byte-long at `0x11DE`), and the second iteration changes the instruction with two `NOP` instructions at `0x11DE` and `0x11DF`, respectively, thus triggering SMC detection. The proceeding iterations execute the `NOP` sled, but as it now corresponds to cached, decoded instructions, SAP does not trigger SMC detection.

Detection evasion with forced cache flushes A drawback of this mechanism is that it can be defeated by SMC performing a cache flush right before code modification, as the cache would have no prior data to compare. Modern processors present many instructions able to flush cache lines, such as clflush, invd, and wbinvd. They were originally created to keep synchronization and consistency among storage systems, but can also be used to leverage such kind of attack. The use of cache flush instructions is supported by compilers through the use of inline asm or intrinsics [25]. To exemplify this type of attack, consider the code shown in Code 7, which flushes the cached line associated to the instruction to be modified right before modifying it.

```
instr = (ptr*)func_addr + OFFSET;
_mm_clflush(instr);
*instr = nop;
```

Code 7 Forced cache flush before instruction change.

To support the use of such instructions, we have complemented our prototype to reflect the proper behavior, as shown in Code 8.

```
string dis = INS_Disassemble(ins);
if(dis.find("clflush"))
    clear_cache(addr);
```

Code 8 Cache simulator. Forced cache flush support.

When running in such environment, our flush-based, modified SMC example was able to stealthily modify the instruction cache content. Therefore, detecting this threat requires the development of an alternative solution: A heuristic approach could flag forced cache flushes as suspicious and to raise an interrupt to trigger a scan procedure. With that, our solution was able to detect the execution of the flush instruction (003515A7 clflush zmmword ptr [eax]). We consider the cache flush heuristic as reasonable since most legitimate applications do not perform cache flushes. To confirm that, we disassembled 2870 PEs and DLL binaries from System32 folder of a clean Windows installation, thus considering them as benign. Only 6 (0.2%) of them presented at least one clflush instruction and 160 (5.5%) presented at least one wbinvd instruction.

In addition of being a requirement for stealth SMC, cache flush is also used on a variety of side-channel cache attacks, such as privileged information leakage [16] and cryptography attacks [19,36]. Therefore, this complementary, heuristic approach, as implemented in SAP, may aid also on non-SMC threat detection.

Cache monitor implementation The implementation decision which presents the best cost-benefit is to instrument the L1 cache, as it presents high-rates of hits. However, we could easily extend this detection mechanism through the whole cache hierarchy. To detect SMC on any cache level, we consider that a simple addition of one control bit per cache line would be necessary. This extra control bit, would be responsible to identify cache lines which holds executable code (executable cache line). In this sense, during any modification of an executable cache line, the cache hierarchy should generate an exception on the requester processor. Figure 1 illustrates the proposed modification.

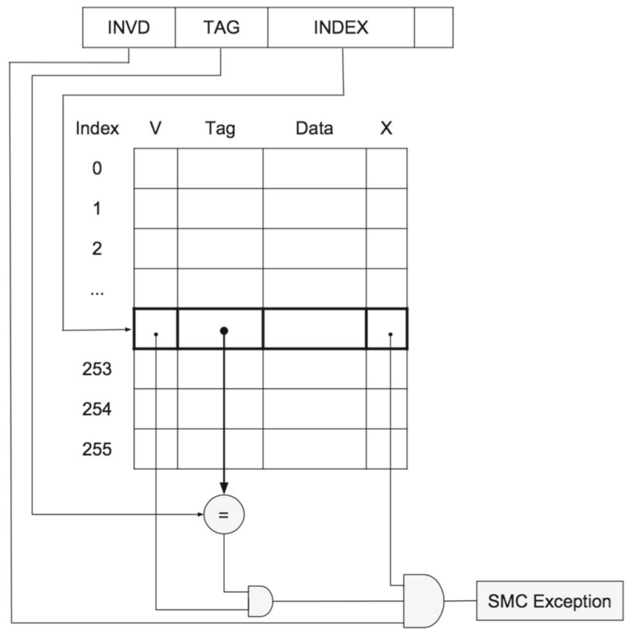

Fig. 1 SMC-aware cache. An exception is generated when an invalidation is performed on valid executable cache line

4.3 SMC-aware pipeline

The presented cache modifications enable processors to detect SMC type I. We here extend our model to also cover SMC type II. As SMC type II affects the processor pipeline, we looked for existing hardware features associated with execution metrics which could provide us with execution metadata to be leveraged for SMC detection.

Intel's processors natively support SMC execution through mechanisms that handle SMC-imposed corner-conditions. Despite such hardware support, few interfaces to them are exposed to the surface (e.g., programming interfaces). Therefore, we are required to adapt existing features for our purposes. More specifically, we adapted existing hardware counters to work as pipeline monitors for SMC execution.

Intel's processors present a mechanism called Precise Event Based Sampling (PEBS) [18], which allows a hardware counter to store its data in OS pages and to raise an interrupt when it reaches a user-defined threshold. The PEBS is able to monitor the MACHINE_CLEARS.SMC event, which "counts the number of times that the processor detects that a program is writing to a code section and has to perform a machine clear because of it.".

As the PEBS works as an event counter and we are interested on detecting any SMC execution, we set the interrupt threshold of PEBS to one. Therefore, every time an SMC event is identified, the counter is incremented, causing an overflow and raising an interrupt. Notice that this approach is deterministic and does not require polling the counter value. Also notice that the same effect can be achieved in distinct

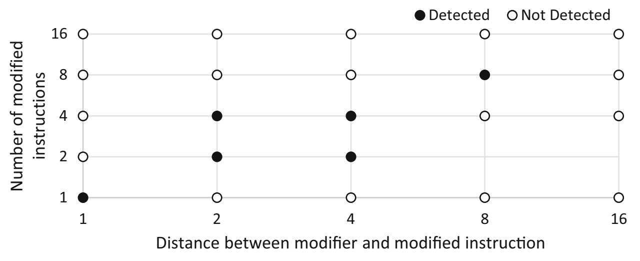

Fig. 2 Effectiveness of event counter as a SMC detector

scenarios and platforms by setting the hardware counters to their maximum values.

When an interrupt happens, the interrupt routine is able to check the last scheduled process in the OS, thus identifying which one generated such interrupt. This information is provided to an Anti-Virus (AV) solution for additional checks. From the AV point of view, an efficient data collection procedure must be implemented, since periodic interrupt checks (polling) would degrade the solution's performance to the software-based, high-overhead scenario. To provide an asynchronous notification, we have implemented an inverted I/O routine, as described by Botacin et. al [7].

To attest the feasibility of our proposal, we developed SMC samples which modify a distinct number of instructions in distinct binary regions. Figure 2 shows the detection of SMC code as a function of the number of affected instructions and the distance between the modified and the modifying instructions. All samples were executed in a Haswell CPU, thus the reproduction of our results is guaranteed only for this micro-architecture.

Our approach succeeded on detecting SMC which caused the pipeline to flush (type II), such as the example whose self-modifying instruction modifies an instruction right before itself. We also notice, as expected, that constructions which do not cause pipeline flushes are not detected. For instance, writing an instruction that will be executed in the future and was not executed before does not affect pipeline at all. Other constructions do not cause pipeline flushes as their data dependencies are resolved through data forwarding.

4.4 SMC-aware MMU

Most modern processors implement the MMU between the processor and any cache hierarchy. Hence, before accomplishing any cache access, the MMU will first translate the virtual address to physical. This way, we can modify the MMU instead of doing cache redesign, since it handles both SMC Type-III and Type-I, and is responsible for memory data that will be cached at some moment.

A first policy for detecting SMC at MMU level is to raise an alert at each SMC code write. It can be accomplished by modifying the Translation Look-aside Buffer (TLB) to

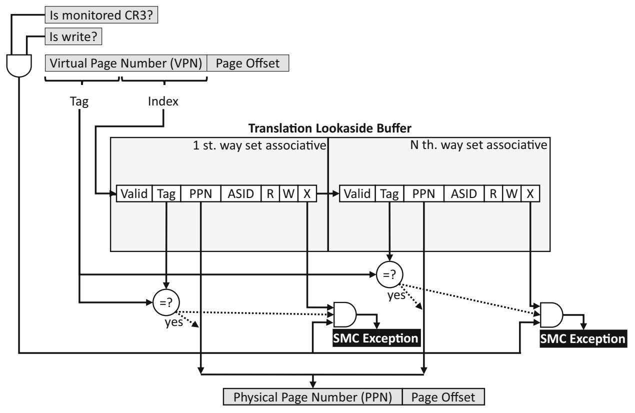
Is monitored CR3?
Is write?
Virtual Page Number (VPN)
Page Offset
Tag
Index
Translation Lookaside Buffer
1 st. way set associative
N th. way set associative
Valid
Tag
PPN
ASID
R
W
X
=?
yes
SMC Exception
Physical Page Number (PPN)
Page Offset

Fig. 3 MMU-based SMC detection mechanism

launch an SMC exception every time a write operation is translated to a page with execution permission.

Every TLB entry is formed by four fields: Virtual Page Number, Physical Page Number, Valid and Permission bits (Read/Write/eXecution). We propose to add our monitor the ability to check the Permission bits of used pages. In case of a permission violation (a write on an executable page), an alert is delivered from the mechanism to the AV solution.

Notice that this policy does not require the execution of the modified code, as it alerts the system right at the modification time. Also notice that it does not generate a segment fault exception, as page write permission is not removed.

A practical challenge is to implement this mechanism on processors with Simultaneous Multi-Threading (SMT) support, where TLB needs to know what process/thread caused the SMC. To handle such cases, we rely on the Address Space Identifier (ASID) field in each TLB line (present in Intel and AMD architectures released after 2008 [1,28]).

The proposed modification is illustrated in Fig. 3. For each matching address (tag-checked), we check whether current operation is a write (externally identified) and target address is executable. Notice the signal is activated only for monitored processes.

In current systems, such policy can only be implemented with software support, by relying on the O.S. to mark code pages as not writable and thus further handling the respective page faults. By keeping a database of pages whose permissions were removed, the page fault handler is able to disambiguate true faults from those caused by SMC writes.

Whereas following individual code writes is a precise inspection trigger, it imposes a significant overhead. An alternative policy is to check entire modified pages. It can be implemented by marking code pages as modified when they are first written and generating an alert when they are required to be executed.

This policy can be implemented in hardware by adding a modified flag to the TLB context, as shown in Fig. 4. When an executable page is writen, the modified flag is set, Further,

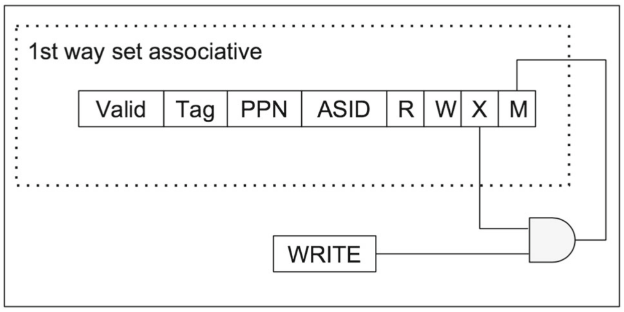

Fig. 4 MMU-based SMC detection mechanism

when the page is required to be executed, an exception is raised if the modified bit is set.

We highlight that the introduction of a modified bit makes the MMU looks like directory-based cache coherence protocols [10], which have a modified data state. SAP extends such approach such that MMU now expects code to also be modifiable.

In current systems, this policy can also only be implemented using software support. The OS can flag executable pages as not present, thus triggering page-faults at each execution attempt. By keeping state of the pages whose execution attribute was removed, the OS can disambiguate ordinary page-faults from the ones caused by SMC execution attempts.

4.5 SAP's notification handling

An important project decision when developing a security monitoring mechanism is how to handle the notifications triggered by the solution. To evaluate the best implementation choice, we implemented two distinct notification mechanisms in SAP: (i) exception handling; and (ii) page fault handling.

The advantage of handling SMC detection through exceptions is that the routines are only executed when triggered by SAP, thus not imposing any verification overhead to the system due to continuous execution. As a drawback, a new handler must be added to the system. To implement such handler, we modelled SAP exception handling on Intel PIN.

The advantage of handling SMC detection by adding verification code to the page fault handling routines is that we can rely on the already existing CPU and OS support. As a drawback, the execution of such verification routines every time a page fault is triggered will impose overhead to the system execution even when SMC detection is not triggered. This overhead is not negligible as it involves additional memory accesses and branch decisions. To evaluate this implementation, we instrumented the page fault handling routines from the Linux Kernel, as shown in Code 9.

```
1  static noinline void __do_page_fault(
       ...) {
2  // Original Code
3  if(kprobes())...
4  // Instrumentation Code
5  if(was_executable_page_written()) {
6      if(!is_allowed_process(get_pid())
           {
7          // SMC Detected
```

Code 9 SMC detection routines in the Linux kernel. The added verification instructions are executed every page fault.

The added verification routines (lines 5 and 6) are responsible, respectively, for checking if this page fault is due to SMC and, if so, if a notification should be raised. The first check will be executed for all page faults, even those caused by non-SMC execution, thus imposing overhead. Depending on the use case, this overhead may be acceptable or not. Adding overhead for inserting kernel probes (line 3), for instance, was considered as a reasonable project decision by the Linux community. Most overhead impact is imposed by the need of taking a branch for checking the fault reason (line 5), thus requiring branch prediction and speculative execution and being subject to their drawbacks. Interestingly, an effective technique that the kernel can apply to eliminate the branch need is to leverage SMC to turn the branch into `NOPs` when monitoring is disabled, as done in kernel tracing mechanisms [15]. However, implementing such strategy for page fault handling is risky as the subsystem will be responsible to modify its own code pages.

In addition to the branch which checks SMC detection, the whitelist check (line 6) also imposes a significant overhead when a benign, allowed application is executing SMC. As this check cannot be removed, the best alternative strategy for increasing performance is to move this check to hardware. By adding a monitored bit to the hardware, one can check whether a page fault or exception should be raised, thus avoiding the entire penalty of performing unnecessary context-switch. The use of an additional monitoring flag requires adding a single bit to the hardware, which can be implemented as an ordinary register. The monitoring bit is initially set and further saved and restored at context switches by the process scheduler, which requires adding the monitored flag to the OS structure. The addition of the monitored flag to the structure imposes negligible overhead to context switch but saves significant cycles while avoiding benign SMC cases from raising warnings. To evaluate this overhead, we implemented two page fault-based SAP's versions: (i) using pure software-checks within the PF handler; and (ii) using PIN-modelled hardware-assisted whitelists.

Table 2 SMC detection. Solutions comparison

App./ packer	SMC type	Cache change	Cache flush	Pipeline flush	MMU	Whitelisted
SPEC	–	✗	✗	✗	✗	✗
SMC	1	✓	✗	✗	✓	✗
SMC Flush	1	✗	✓	✗	✓	✗
UPX	3	✗	✗	✗	✓	✗
Themida	2	✗	✓	✓	✓	✗
Python	3	✗	✗	✗	✓	✓

5 Evaluation

We here present an evaluation of hardware-support for SMC execution regarding detection efficiency and performance overhead for distinct applications.

Detection Table 2 presents an evaluation of all presented detectors (columns 3–6): the proposed alert for cache invalidation, the forced cache flush heuristic, machine clear hardware counter, and the proposed MMU change, respectively. The last column specifies whether processes were whitelisted during experiments or not.

As non-SMC, benign applications, we considered all the 29 binaries from the SPEC-CPU 2006 benchmark in addition to Python-powered scripts, to exercise the whitelisting solution. As SMC, we considered the examples from Code 4 and 7 and a set of 20 distinct samples packed with UPX and Themida (packing solutions known to employ SMC in their constructions).

As expected, SPEC binaries were not detected by any mechanism, since they do not perform code modifications. Our first SMC example, in turn, was detected by the modified cache invalidation protocol. However, only this simple example was detected by such change, since the cache flushes performed by the remaining codes were effective ways to defeat code change monitoring.

In this scenario, the cache flush monitoring policy was able to detect six forced cache flushes during the execution of the Themida's loader. We confirmed this result by inspecting the instructions associated with the SMC-pointed code regions. We found the flush instructions presented in Code 10.

```
17aa388: 0f ae 7b 74 clflush 0x74(%ebx)
198b964: 0f ae 7e bc clflush -0x44(%esi)
1cbb375: 0f ae 7a ea clflush -0x16(%edx)
1d8ff8e: 0f ae 7e f9 clflush -0x7(%esi)
20368e6: 0f ae 7f 18 clflush 0x18(%edi)
2349866: 0f ae 3b    clflush (%ebx)
```

Code 10 SMC Code. Forced Cache flush in Themida's loader.

Another raised detection flag was for the pipeline clear counter. Despite being activated during Themida execution, this detector did not detect UPX execution, since the code is modified on a distinct system page. The MMU change is able to detect all cases due to its system-wide memory view. As a drawback, the Python interpreter was also flagged, requiring its process to be whitelisted.

Performance overhead Despite effective for SMC detection, a detection mechanism must not degrade performance while running benign applications, which is a drawback for many solutions. Ether [14], for instance, is a hardware-assisted VM solution that performs code unpacking, including SMC support. Its step-by-step operation imposes overhead of the same magnitude as the code being executed, not being suitable for real-time cases. Even solutions targeting end-users are affected by considerable overhead. OmniUnpacker, for instance, reports penalties of 11% and 6% on SMC and non-SMC programs, respectively. JAVA's JIT compiler Instrumentation [23] reports a 2× slowdown with a significant impact on benign programs.

To evaluate how significant is the imposed monitoring overhead, we measured the impact of each proposed mechanism while executing the same aforementioned SPEC applications and SMC binaries. Execution cycles overhead was measured by running the samples in a non-instrumented environment and further comparing it with the measured cycles in the instrument environment.

To evaluate the proposed cache modification, we executed all binaries in the PIN-modelled, SAP's prototype. As the SPEC binaries do not perform SMC execution, they did not trigger any code invalidation, thus not imposing any overhead.

To evaluate the proposed pipeline monitor, we executed all binaries on a physical machine (Haswell CPU) with enabled hardware-counters and SAP's driver loaded to handle its interrupts. As for the cache, no SPEC binary triggered a pipeline flush due to SMC, thus not imposing overhead.

To evaluate the proposed MMU modification, we have considered the instrumented Linux PF handler (with software whitelisting) and the PIN prototype (with hardware whitelisting and exception handling).

For the PF-based prototype, we identified the base cost of handling a PF is, on average, 1000 cycles. Additional 100 cycles are required to perform the SMC detection check;

Table 3 Estimated overhead of Software-based SMC detectors during page fault trapping on SPEC applications

Benchmark	Penalty (%)	Benchmark	Penalty (%)	Benchmark	Penalty (%)
bzip2	3.47	mcf	3.76	wrf	3.91
namd	1.54	bwaves	3.73	perlbench	6.49
h265ref	4.61	calculix	3.57	dealli	3.12
astar	2.12	sjeng	2.74	hmmer	2.62
gobmk	3.10	cactusADM	3.70	libquantum	3.24
gcc	6.25	gromacs	4.01	sphinx3	3.76
lbm	4.27	zeusmp	3.48	povray	4.64
tonto	4.53	GemsFDTD	3.48	xalancbmk	3.85
gamess	4.05	leslie3d	3.46	specrand	3.36

another 20 cycles are required for whitelist checking in software. Table 3 shows the experimental results in terms of the relative overhead for all SPEC binaries.

We notice that performance penalties up to 6% may be imposed even to applications that do not perform any SMC-related changes, given the additional processing (whitelisting, process retrieval and flags checking) within the page fault handler. The imposed performance penalty when handling benign applications depends on how many PFs the application triggers. The more PFs, the more the added instrumentation code influences the final cycles overhead.

When leveraging the PIN prototype, no overhead was observed, as whitelist is performed in hardware and no instrumentation code is added to the PF handler. Therefore, we conclude that exception-based handling is more suitable for implementing SMC detectors than PF-based handling, as implemented by actual solutions, such as OmniUnpack.

6 Discussion

SAP allows moving SMC detection from software to hardware, thus reducing current solutions' imposed monitoring overhead. SAP allows calling an AV on-demand to check whether a given execution is malicious or not. To whitelist benign processes executions, SAP leverages a hardware-based whitelist.

Contributions SAP advances current SMC detection by not relying on a whitelist of benign process known *a priori*. By outsourcing this decision to a third-party AV, SAP allows benign SMC applications to run while still detecting malicious ones.

SAP also presents a precise, hardware-based mechanism for detecting SMC code changes, not requiring AV solutions to poll system memory, thus reducing the imposed performance penalty.

The detection mechanism operates on a per-process basis, thus allowing processes to be blocked or allowed to run individually, a significant advance over traditional whitelisting-based solutions which flags entire application classes. As an example, SAP can whitelist individual python scripts/processes whereas traditional whitelisting mechanisms would flag all python interpreter instances as benign.

SAP also reduces the imposed monitoring overhead by moving the whitelist from software to hardware. Whereas software-based solutions have to first be interrupted—for instance, by a page fault—to further whitelist a given execution, SAP does not generate interrupts for whitelisted processes, thus not interrupting their executions at any time. Also, as SAP whitelist is runtime-generated, processes can be added and removed from the monitoring list at any time, whereas current O.S whitelists are static.

Legacy support Whereas newly developed applications could be implemented in an SMC-aware way and provide OS with more fine-grained indicators about which pages should be marked as executable and writable, SAP's main advantage is to support COTS binary execution. Therefore, SAP offers legacy systems support to monitor SMC execution without requiring binary recompilation.

$W \oplus E$ policy The security policy implemented by SAP's MMU can be considered as a runtime, hardware-enforced, conditional version of the usual (W⊕E) policy. By proposing such policy implementation, we are not arguing that traditional software-based W⊕E—e.g. stack protection—should be eliminated, but extended. It is worth to notice that the traditional, static W⊕E policy is not as effective against malware as it is against external code injection attacks, because, on the contrary to external code, running malware samples may control their own flags by calling OS memory protection APIs, disabling the protection mechanisms and thus bypassing any software-based defense solution.

AV cooperation We must make clear that our goal is not to entirely replace existing AV solutions, but to assist them with new, efficient inspection triggering mechanisms. Therefore, our solution is fully compatible with existing AV solutions as well as present the same drawbacks. As an AV complement, we focused SAP on SMC detection because obfuscated-like code may prevent AV solutions of doing their best: matching malicious patterns. Other classes of attacks, however, will still require specific AV knowledge to be detected.

Interpreters A particular class of attacks which will still require AV knowledge is the script-based/interpreter-based ones, as the case of Python and Java. On the one hand, as all interpreted code will generate binary code at runtime, our solution will not be able to characterize a given code as particularly interesting for scan without external help. On the other hand, AV can instrument and monitor bytecode pieces in a less intrusive way than ordinary binaries [9,27], our focus on this work. A SAP-AV joint operation may allow for individual processes to be monitored by SAP and runtime whitelisted by the AV as they are characterized as benign. Moreover, already AV-scanned scripts could also be constantly monitored by SAP while looking for new SMC behavior without requiring AV to instrument the script bytecode.

Whitelisting Besides the case of interpreters, such as Python, ordinary binaries can also be whitelisted on-demand. However, it is important to understand the implications of such decision. Whereas binary whitelisting ensures the proper execution of a benign, SMC application, it may give opportunities for defense bypasses, as a trust relationship is established. The most noticeable evasion opportunity is related to software updates. A trojanized binary may present a legitimate behavior until be whitelisted and then turn itself into a distinct, malicious payload. Such cases must be handled by a third party mechanism, either by checking the binary is still not malicious, or by removing it from the whitelist after any update.

7 Conclusions

We revisited the SMC problem and its impact on hardware architectures, such as cache invalidation and pipeline flushes, and introduced a taxonomy for SMC code and its detection mechanisms based on their effects over distinct architectural points and detection window. We based on that to present processor changes (SAP) at cache, pipeline and MMU levels to enable it to alert upper instances (OS and AVs) about SMC execution and triggering on-demand checks, reducing traditional AV's overhead. SAP presented a perfect detection ratio (100%) with a considerable reduction on overhead penalty (0% for whitelisted and/or non-SMC code). As future work, we aim to identify non-SMC-specific counters able to assist other SMC types detection.

Reproducibility All developed prototypes were released as open source code and are available at: https://github.com/marcusbotacin/Self-Modifying-Code

Acknowledgements This project was partially financed by the Serrapilheira Institute (Grant Number Serra-1709-16621), the Brazilian National Counsel of Technological and Scientific Development (CNPq, Ph.D. Scholarship, process 164745/2017-3) and the Coordination for the Improvement of Higher Education Personnel (CAPES, Project FORTE, Forensics Sciences Program 24/2014, process 23038.007604/2014-69).

References

1. AMD: AMD Secure Virtual Machine Architecture Reference Manual (2008). https://www.mimuw.edu.pl/~vincent/lecture6/sources/amd-pacifica-specification.pdf
2. ARM: Smc (2013). https://community.arm.com/processors/b/blog/posts/caches-and-self-modifying-code
3. Babar, K., Khalid, F.: Generic unpacking techniques. In: International Conference on Computer, Control and Communication, pp. 1–6 (2009)
4. Ballapuram, C.S., Sharif, A., Lee, H.H.S.: Exploiting access semantics and program behavior to reduce snoop power in chip multiprocessors. SIGARCH Comput. Archit. News **36**(1), 60–69 (2008)
5. Bonfante, G., Fernandez, J., Marion, J.Y., Rouxel, B., Sabatier, F., Thierry, A.: Codisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions. In: ACM SIGSAC Conference on Computer and Communications Security, pp. 745–756 (2015)
6. Borello, J.M., Mé, L.: Code obfuscation techniques for metamorphic viruses. JICVHT **3**, 211–220 (2008)
7. Botacin, M., de Geus, P., Grégio, A.: Enhancing branch monitoring for security purposes: from control flow integrity to malware analysis and debugging. ACM Trans. Priv. Secur. **21**(1), 1–30 (2018)
8. Cai, H., Shao, Z., Vaynberg, A.: Certified self-modifying code. SIGPLAN Not. **42**(6), 66–77 (2007)
9. Caserta, P., Zendra, O.: A tracing technique using dynamic bytecode instrumentation of java applications and libraries at basic block level. In: Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, ICOOOLPS'11, pp. 6:1–6:4. ACM, New York, NY, USA (2011). https://doi.org/10.1145/2069172.2069178
10. Censier, F.: A new solution to coherence problems in multicache systems. IEEE Trans. Comput. **C–27**(12), 1112–1118 (1978). https://doi.org/10.1109/TC.1978.1675013
11. Coke, J., Baliga, H., Cooray, N., Gamsaragan, E., Smith, P., Yoon, K., Abel, J., Valles., A.: Improvements in the Intel's Core2 penryn processor family architecture and microarchitecture (2008)
12. Debray, S., Patel, J.: Reverse engineering self-modifying code: unpacker extraction. In: Working Conference on Reverse Engineering, pp. 131–140 (2010)
13. Dehnert, J., Grant, B., Banning, J., Johnson, R., Kistler, T., Klaiber, A., Mattson, J.: The transmeta code morphing trade; software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In: International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, pp. 15–24. IEEE Computer Society (2003)
14. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: ACM Conference on Computer and Communications Security, pp. 51–62 (2008)
15. Gebai, M., Dagenais, M.R.: Survey and analysis of kernel and userspace tracers on linux: design, implementation, and overhead. ACM Comput. Surv. **51**(2), 26:1–26:33 (2018). https://doi.org/10.1145/3158644
16. Gruss, D., Maurice, C., Wagner, K., Mangard, S.: Flush+flush: a fast and stealthy cache attack. In: Interenational Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 279–299 (2016)
17. Gutierrez, A., Pusdesris, J., Dreslinski, R.G., Mudge, T.: Lazy cache invalidation for self-modifying codes. In: International Con-

ference on Compilers, Architectures and Synthesis for Embedded Systems, pp. 151–160 (2012)

18. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual. Intel (2013). https://www.intel.com.br/content/www/br/pt/architecture-and-technology/64-ia-32-architectures-software-developersystem-programming-manual-325384.html
19. Inci, M.S., Eisenbarth, T., Sunar, B.: Wait a Minute! A Fast, Cross-VM Attack on AES, pp. 299–319. Springer, Springer (2014)
20. Korczynski, D.: Repeconstruct: reconstructing binaries with self-modifying code and import address table destruction. In: International Conference on Malicious and Unwanted Software, pp. 1–8 (2016)
21. Liu, A., Wang, W.: Ascms: an accurate self-modifying code cache management strategy in binary translation. In: International Conference on Information Science and Cloud Computing Companion, pp. 405–410 (2013)
22. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'05, pp. 190–200. ACM, New York, NY, USA (2005). https://doi.org/10.1145/1065010.1065034
23. Maebe, J., De Bosschere, K.: Self-modifying Code (2003). arXiv:cs/0309029
24. Martignoni, L., Christodorescu, M., Jha, S.: Omniunpack: fast, generic, and safe unpacking of malware. In: Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), pp. 431–441 (2007). https://doi.org/10.1109/ACSAC.2007.15
25. Microsoft: x64 intrinsics. https://msdn.microsoft.com/en-us/library/hh977022.aspx
26. Mody, R.P.: Functional programming is not self-modifying code. SIGPLAN Not. **27**(11), 13–14 (1992)
27. Moret, P., Binder, W., Tanter, E.: Polymorphic bytecode instrumentation. In: Proceedings of the Tenth International Conference on Aspect-oriented Software Development, AOSD'11, pp. 129–140. ACM, New York, NY, USA (2011). https://doi.org/10.1145/1960275.1960292
28. Neiger, G.,, Santoni, A., Leung, F., Rodgers, D., Uhlig, R.: Intel virtualization technology: hardware support for efficient processor virtualization (2006). https://www.ece.cmu.edu/~ece845/docs/vt-overview-itj06.pdf. Accessed Jan 2019
29. Ray, K., Kramer, M., England, P., Field, S.: On-access scan of memory for malware, US Patent 7,836,504 (2010)
30. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: systems, languages, and applications. ACM Trans. Inf. Syst. Secur. **15**(1), 1–34 (2012)
31. Shar, L.E., Lawton, K.P.: Trace cache for efficient self-modifying code processing, US Patent 7,606,975 (2009)
32. Uluski, D., Moffie, M., Kaeli, D.: Characterizing antivirus workload execution. SIGARCH Comput. Archit. News **33**(1), 90–98 (2005). https://doi.org/10.1145/1055626.1055639
33. Willems, C., Hund, R., Fobian, A., Felsch, D., Holz, T., Vasudevan, A.: Down to the bare metal: using processor features for binary analysis. In: ACSAC, pp. 189–198 (2012)
34. Wu, M., Zhang, Y., Mi, X.: Binary protection using dynamic fine-grained code hiding and obfuscation. In: International Conference on Information and Network Security, pp. 1–8 (2016)
35. Xianya, M., Yi, Z., Baosheng, W., Yong, T.: A survey of software protection methods based on self-modifying code. In: International Conference on Computational Intelligence and Communication Networks, pp. 589–593 (2015)
36. Yarom, Y., Falkner, K.: Flush+reload: a high resolution, low noise, l3 cache side-channel attack. In: USENIX Security, pp. 719–732 (2014)
37. Zaidi, N.: System and method for tracking in-flight instructions in a pipeline, US Patent 6,237,088 (2001)