A Secure Co-Processor to Accelerate Real-Time AntiViruses via Inspection Breakpoints

Marcus Botacin[1] Francis B. Moreira[2]
Philippe O. A. Navaux[2] André Grégio[1]
Marco A. Z. Alves[1]

[1]Informatics Department - Federal University of Paraná (UFPR), Brazil
{mfbotacin, gregio, mazalves}@inf.ufpr.br
[2]Informatics Institute - Federal University of Rio Grande do Sul (UFRGS-BR)
{fbmoreira,navaux}@inf.ufrgs.br

Abstract

AntiViruses (AVs) are essential to face the myriad of malware threatening Internet users. AVs operate in two modes: on-demand checks and real-time verification. Software-based real-time AVs intercept system and function calls to execute AV's inspection routines, resulting in significant performance penalties as the monitoring code runs among the suspicious code. Simultaneously, dark silicon problems push the industry to add more specialized accelerators inside the processor to mitigate these integration problems. In this paper, we propose TERMINATOR, an AV-specific coprocessor to assist software AVs by outsourcing their matching procedures to the hardware, thus saving CPU cycles and mitigating performance degradation. We designed TERMINATOR to be flexible and compatible with existing AVs by using `YARA` and `ClamAV` rules. Our experiments show that our approach can save up to 70 million CPU cycles per rule when outsourcing on-demand checks for matching typical, unmodified `YARA` rules against a dataset of 30 thousand in-the-wild malware samples. Our proposal eliminates the AV's need for blocking the CPU to perform full system checks, which can now occur in parallel. We also designed a new inspection breakpoint mechanism that signals to the coprocessor the beginning of a monitored region, allowing it to scan the regions in parallel with their execution. Overall, our mechanism mitigated up to 44% of the overhead imposed to execute and monitor the SPEC benchmark applications in the most challenging scenario.

1 Introduction

AntiViruses (AVs) are essential solutions to protect Internet users from the multiple threats that target their systems daily [35]. Typical AVs have two operation modes for checking for viruses: on-demand and real-time. In the former, the AV matches selected files against detection rules when the user triggers a scanning procedure. In the latter, the AV matches the rules against function call arguments to identify malicious behavior in processes as they execute. To perform this real-time matching, AVs are required to intervene in the original function calls by preloading them with the AV code [13]. The new code causes significant performance overhead, as the AV's interposed code runs every time a monitored function call is invoked.

The performance overhead imposed by a real-time AV depends on the security requirements. The rate of additional cycles for a check, for a given operational scenario, in some cases, might reach 100% (see Section 2). This overhead might be prohibitive for some applications, such as gaming, so AV companies created gaming modes in their products to avoid performing background scans when users are playing [34, 6]. Whereas increasing the system performance, this mode also limits the AV's response time, in a clear trade-off decision. However, ideally, an AV should combine full system performance and full system protection.

Our key observation about the current AV's is that most of the imposed overhead originates from a sequential execution of the detection rules before the function calls (the AV check might run even on the same core than the monitored code, in case of interrupt-based monitors [31]). Therefore, to solve current AVs' performance problems, we propose TERMINATOR, a simplified yet flexible hardware coprocessor to allow AV's software components to outsource their matching routines.

The proposed coprocessor runs code routines specified by the AV's software component and interrupts the system-operation only when it identifies malicious behavior. The reliance on AV-specified code overcomes the major limitation of previously proposed AV accelerators (e.g., FPGA-based) of not being updatable by software [8]. Also, we expect a coprocessor to be a more flexible platform than FPGAs and GPUs, whose AV proposals never reached the market (see Section 5). Furthermore, considering the restricted power budget present in modern processors, due to dark silicon problems [25], a decision to add specialized hardware is aligned to the semiconductor industry, offering good usage for the transistors.

We aimed to make our coprocessor compatible with market technologies. Thus, it supports two popular, open-source matching mechanisms: the `ClamAV` antivirus engine and the `YARA` pattern matcher [60]. The coprocessor matches the supplied rules both on-demand as well as in real-time mode. When operating in real-time, the coprocessor matches the rules over the arguments from the function and system calls currently being executed in the main-CPU (the CPU that was initially executing the application). We developed a new `inspection breakpoint` mechanism, inspired by software breakpoints, that allows an AV

to flag a given region as monitored. When the main-CPU decodes this flag, it invokes the coprocessor to parallel scan the monitored region.

We designed our coprocessor as a simple ARM processor that communicates with the x86 core. Our decision to use two different ISAs eases our evaluation method while allowing manufacturers to choose the most suitable ISA combination for actual deployment. Whenever the coprocessor matches a malicious pattern, it raises an interrupt. That interruption delivers the pattern to a software-based AV (in userland) that defines additional responses (e.g., process termination). We implemented TERMINATOR as a proof-of-concept in an architectural simulator to evaluate our proposal in actual scenarios. We generated detection rules based on a dataset of 30 thousand malware samples collected in-the-wild and shared with us by a partner CSIRT[1], and measured the performance impact during the execution of the SPEC benchmark applications to understand the impact of monitoring over typical legitimate applications.

It is important to highlight that the idea of adding a coprocessor to a system is not new by itself, but it has been proposed since decades ago (see Section 6). However, many important questions are still open: Why didn't it become more popular? What prevents it? Are we in the same situation even after years of computing development? What would a coprocessor look like nowadays? We try to answer these questions in this paper. In this sense, the contributions of this work are the following:

- We propose, design, and implement TERMINATOR, a hardware coprocessor to assist AV matching procedures. We also propose the use of `YARA` and `ClamAV` rules in our novel coprocessor to match API parameters when it searches for malware. Our goal is to accomplish a coprocessor whose exclusive purpose is to act as an anti-malware device. Therefore, our goal is **not increasing AV's detection rates**, but to **make them more performance-efficient**. As far as we know, we are the first to propose this type of combined approach (`YARA` and `ClamAV` rules within a hardware coprocessor to perform real-time validation of API calls).

- We introduce the concept of `inspection breakpoints`, which allows TERMINATOR to perform checks only when the main processor executes an AV-monitored function.

- We explore the design-space of actual detection rules (`ClamAV` and `YARA`) regarding their number and complexity, when applied to the monitoring of real applications (SPEC benchmark).

- As a proof of concept, we implemented TERMINATOR in a cycle-accurate simulator and show the obtained performance results (improvements up to 44% in the most challenging scenario).

Our main experimental results indicate that:

[1]Computer Security Incident Response Team

- Outsourcing *on-demand* matching procedures to the coprocessor saves from one to 70 million CPU cycles per `YARA` rule when matching a real dataset of 30 thousand malware samples. Preventing the AV from blocking the CPU for long streamlines full system checks to occur at any time with no execution delay to other application's start.
- Our coprocessor verifies any unmodified `ClamAV` rule in parallel to the invoked function calls;
- The proposed coprocessor allows the matching of typical, regex-based `YARA` rules with no overhead: 10% of all tested `YARA` rules [61] can be matched in runtime without any modification; the remaining 90% can be matched using a delayed matching strategy.
- Matching procedures are extensible for checks beyond the ones implemented by typical AVs: Even the most complex `YARA` rules can be matched in runtime if a delayed matching procedure is adopted.
- The parallel matching mechanism saves 70 thousand cycles per inspection call, which results in mitigating up to 44% (5% on average) of the overhead imposed to execute and monitor the SPEC benchmark applications in the most challenging scenario.
- We can detect 99% of the 30 thousand malware samples in runtime by using a set of 9 custom rules to be applied by the coprocessor.

The organization of the remainder of this paper is as follows: In Section 2, we motivate our choice for a coprocessor; In Section 3, we present the design and implementation of a coprocessor-assisted AV; In Section 4, we evaluate the detection capabilities and the performance gains enabled by our solution; In Section 5, we discuss the impact of our findings and the limitations of our work; In Section 6, we present related work to better position our contributions; In Section 7, we draw our final conclusions;

2 Motivation

In this section, we show an overview of the current AntiViruses way of operation to motivate the need for an AV assisted by a coprocessor and, consequently, support our proposal and claim that it is a viable, if not required, approach for the next-generation AVs.

2.1 Current AVs Landscape

AVs checking process. Typical AVs have two operation modes: on-demand checks and real-time checks. In the first mode, AVs perform checks against files only when demanded by the users or when a full system scan is scheduled—performance-wise, on-demand checks only affect the system on determined periods. Usually, they block system operations to handle a massive number of

files. In the second mode, the AV is continuously monitoring system operation for any suspicious sign—this affects the whole-system usage with regards to performance.

Signature-based flagging. On-demand checks typically flag suspicious files using signatures which are patterns known to belong to malicious samples. The patterns can be matched using customized routines or third-parties solutions. A popular pattern matching tool in the security context is the `YARA` framework [60]. Code 1 exemplifies a popular `YARA` signature to detect PE files (Windows Executables) that were packed to avoid detection. This rule checks if the file is a valid PE binary (having the `MZ` header) and flags the binary file as packed if a high entropy value is found.

```
1 rule IsPacked : PECheck {
2   condition:
3   // MZ signature at offset 0 and
4   uint16(0) == 0x5A4D and
5   // PE signature at offset stored
6   // in MZ header at 0x3C
7   uint32(uint32(0x3C)) == 0x00004550
        and
8   math.entropy(0, filesize) >= 7.0
9 }
```

Code 1: YARA rule to detect packed PE files.

Real-time monitoring. This type of monitoring can be seen as a variation of on-demand checking, but that matches routines or system calls arguments against suspicious patterns instead of static files. Due to the distinct nature of arguments used in both approaches (routines vs. system calls), matching rules based on function/system calls may be considered a simplified version of those rules applied to entire binary files. AVs may apply runtime monitoring mechanisms to enforce security policies, such as to forbid processes from accessing a given protected directory. In this case, the AV checks if the protected directory is the calling argument of the `open` function.

Function interposition requirement. An AV needs to use function interposition to retrieve function arguments since it allows the AV code to run before the invocation of the original function call. Figure 1 illustrates a typical interposition procedure: the AV hooks into the symbol tables and makes them point to the AV's trampoline function. Therefore, when a binary invokes a function call, the execution flow is hijacked by the AV function. The AV can then decide the further actions. It might: (i) only log the function invocation to execute some remediation procedure later and resume the flow by jumping to the original function address; or (ii) actively block the function call by immediately returning from the call without invoking the original function.

Typical function interposition procedures used to be implemented by hooking to the System Service Dispatch Table (SSDT) [28], a kernel table that affects the whole system operation. SSDT hooking is nowadays prevented and blocked

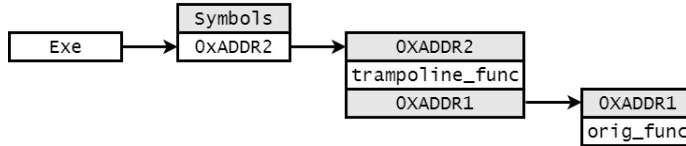

Figure 1: **Function Interposition.** A trampoline function added by AVs to interpose the original function calls.

in most OSes (from the x64-Windows Vista to the present, for instance) to prevent kernel rootkits from hooking to this table. As an alternative, AVs have been implementing interception routines by using callbacks and filters provided by the OSes [13].

Function interposition overhead. During its usage, function interposition routines impose significant performance overhead that a hardware coprocessor could mitigate. The imposition of performance penalties is unavoidable to any software-based solution, as the AV's instructions are executed in the same CPUs running the monitored code. To demonstrate the impact of this overhead in practice, we measured the execution time of the applications in the SPEC CPU 2006 benchmark [57] with and without AV solutions. We considered both an SSDT-based [28] and a filter-based AV [13]. All experiments were executed on an Intel i7-7700, 16GB computer running Windows XP and 7, as required by the tested AV solutions. All results are reported as an average of 10 executions.

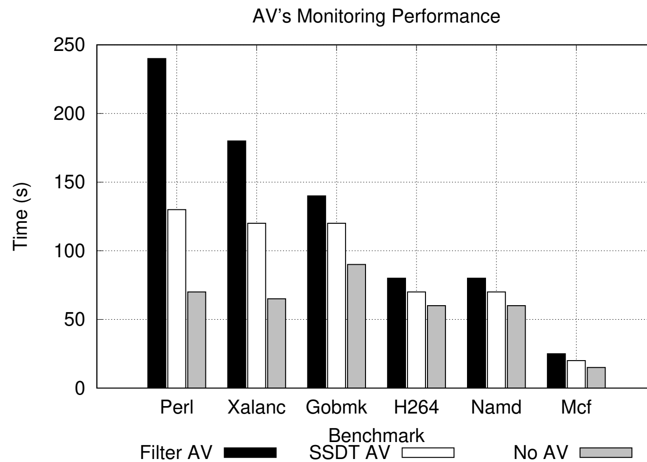

Figure 2: **AV Monitoring Performance.** Software-based monitoring approaches impose performance overhead regardless their implementation strategy.

Figure 2 shows the performance impact caused by the use of AVs for the applications that reported: (i) the two highest; (ii) two average; and (iii) the two smallest execution time values. In the worst case, the benchmark execution time more than doubled. We notice that both types of AVs impose overhead. However, the SSDT-based AV's overhead is smaller due to the viability of more granular interception routines (SSDT-based AVs can select individual APIs to hook, whereas filter-based AVs are limited to the events selected by the OS).

2.2 Ideas to Improve AVs

Adding more Threads. A first idea to improve AVs is to add more processing capacities to the AV solution, which often is suggested in the form of adding more software threads. Whereas this is a solution to make on-demand scans faster, as distinct files/rules can be matched in distinct threads at distinct scanning steps, it does not solve the real-time scanning problem, since scans must be synchronized with the monitored application's execution. In fact, current AVs are already multithreaded for processing on-demand checks, but this does not eliminate the need for blocking real-time execution for scanning a coherent process context (function arguments should not be modified by the software execution while the AV scans them).

Separating Scans from the Execution. Since adding more serial processing units does not mitigate the overhead of real-time scans, the next alternative is to add parallel processing capabilities. If the monitoring code runs along with the monitored one, no performance overhead is imposed to the main software execution. There are multiple alternatives for implementing parallel scanners, ranging from extra processor cores to a specific-purpose coprocessor. These alternatives are investigated in this paper.

Using Extra CPU Cores. Extra threads might be useful for real-time scanning if they are placed in distinct CPU cores, such that the monitoring code can run in parallel with the monitored one. This strategy does not completely mitigate the performance overhead, since it still requires blocking the monitored process for a short period for context synchronization (transferring function call arguments from one core to another), but it certainly helps improving AVs as it doesn't require blocking the main CPU for the whole scanning time. In addition to the communication cost, another drawback of this alternative is that there is no guarantee that the system will have available cores during the whole system operation. In the case in which all cores are loaded with processing tasks, the AV performance is degraded to the serial mode again.

Using a coprocessor. A security coprocessor might be understood as a core that is always available for the AV's use. In addition, if placed close to the main CPU, an almost negligible communication cost is imposed. The major drawback of having a coprocessor is that it requires extra hardware and additional implementation efforts. We believe that these are drawbacks worth to be faced when security is a primary system requirement. In the following, we discuss in more detail the development of a security coprocessor for real-time AV scans.

2.3 Coprocessors Advantages

Coprocessor vs. Performance overhead. A coprocessor solves the performance's overhead problem, as it allows the monitoring code to run without spending cycles in the main-CPU by moving the tasks to a parallel processor. An overhead-free monitor could be ensured by imposing a single constraint: the monitoring routine should finish early or, at the same time, the monitored routine to avoid the main-CPU being locked on a monitoring barrier.

Coprocessor vs. Hooking issues. A coprocessor solves the SSDT hooking limitation problem, as it does not require software patching of OS structures. Hardware triggers can launch coprocessor-based routines without the need for software modifications. The hardware triggers only require having either: (i) the addresses of the function calls to be monitored; or (ii) the values associated with the system call being invoked when its invocation is trapped.

Coprocessor efficiency. A coprocessor can be more energy-efficient than typical CPUs to monitor the system since it can have specialized hardware. This efficiency is true even if both processors would execute the same number of instructions, as the coprocessor might require fewer hardware features to operate.

Updating. Our proposed coprocessor's operation can be updated by software, thus overcoming a major drawback of hardware solutions that are often limited to detect hard-coded patterns [8]. The coprocessor might share the main system memory and thus load the monitoring routines from it. It allows the monitoring routines to be updated via software, thus not breaking the current AV's operation paradigm.

Space and power-budget requirements. There is architectural and physical space for a coprocessor, as the hardware design practices have found limits for many non-specialized constructions. Physical limitations, such as the Dark Silicon problem [25], prevent chips from having their total area operating at full clock speed simultaneously. Manufacturers have overcome this problem by filling the chip with accelerators that are only periodically triggered, thus preventing the chip from melting. Our proposed coprocessor-based AV is ideal for this scenario since it is triggered only when specified function or system calls are invoked (considering a proper design in which the coprocessor is not overloaded, as discussed in Section 4).

3 Terminator: Design & Implementation

In this paper, we introduce TERMINATOR, a coprocessor that mitigates the performance overhead imposed by the monitoring and matching routines of antivirus (AV) solutions in software. Our model considers that the AV software component can send commands to the coprocessor to set its operation to the distinct operation modes. Our coprocessor can be activated on-demand via special AV requests or continuously operate in the real-time monitoring mode, intercepting and scanning function calls. This section describes the coproces-

sor operation in the real-time mode in detail since it is more challenging than outsourcing time-decoupled matching tasks.

3.1 Operation Overview

In our proposed operation model, the AV client at *userland* starts by updating its detection rules from the Internet. The downloaded rules are set to the coprocessor via an AV kernel driver. The *userland* AV starts monitoring processes creation. When a new process is created, the AV performs three steps. First, it identifies the function addresses within that process to be monitored. Second, it checks in the AV database which rules will be applied to that process. Third, it sets (via the kernel driver) inspection breakpoints in the monitored function. These inspection breakpoints will inform the coprocessor when functions that must be monitored are about to be executed. A detailed explanation of inspection triggers is presented below: (i) The coprocessor keeps idle while the main-CPU executes the binary's internal code. (ii) When the CPU decoder unit identifies a call to a monitored function, it requests an analysis to the coprocessor. (iii) The coprocessor matches the specified set of rules against the function call's arguments in parallel with the monitored function's execution (by the main-CPU). (iv) Distinct sets of rules can be applied to distinct functions and processes, thus ensuring flexibility. (v) When a malicious pattern is found, the coprocessor raises a notification delivered via the kernel driver to the *userland* AV. (vi) The *userland* AV will then decide which actions to take (e.g., process blocking).

3.2 Threat Model

In this work, we propose a coprocessor to assist AV operation. Our goal is not to replace existing AV solutions but to assist them with an efficient matching mechanism. Our coprocessor operates on x86-like processors (the most popular architecture for desktop users and malware creators), although our key insights can be applied to any architecture. Besides, we limit our system to operate on a single core-basis, i.e., each main-CPU has its coprocessor.

We focus our operation on popular operating systems (OS), especially Windows, which is the most targeted OS by malware [4]. We limited our scans to the *userland* mode since privilege escalation is out of the context of the AV engine scanner. However, other AV components can perform it. Even then, our approach could be ported to operate in the kernel. In particular, we focus on scanning the native Windows libraries when invoked by malware samples, which is popular in many attacks (e.g., Powershell-based ones [18]). Nevertheless, our strategies could be employed to general function calls. We assume that attempts to write on library memory to defeat our mechanism or patch the original library are prevented by the AV either via MMU flags [12] or integrity monitors [11]. We designed our solution to target newly-launched binaries such that their loaders are not malicious. The launched binaries might be Address Space Layout Randomization (ASLR)-aware.

Since our proposal for a next-generation AV methodology requires modifying the existing CPU architectures to add a coprocessor, we made three reasonable assumptions about computer systems' future. First, we assume that the coprocessor will operate on 64-bit processors. Thus, only the x64 calling convention [39] will be considered here. Second, we also assume that all *syscalls* will be performed via the fast *syscall* convention [16]. Third, we consider that only direct calls (even though implemented via jump tables) will be protected. Indirect control flows (e.g., a return to another function, tail calls) will be checked by the emerging control flow enforcing extensions (e.g., Intel CET [32]), such that our assumptions extend their threat models.

3.3 Terminator Modules and Operation

We describe all modules and steps for the coprocessor operation (in practice) below.

3.3.1 The AV Client

The AV component at *userland* is responsible for adding intelligence to the system, such as deciding which rules will be applied to each process and which post-detection actions shall be performed. It is implemented as a typical AV, and its only assumption is that it can count on hardware support for pattern matching. This proposal does not break the existing AV's operation paradigm. It still allows the coprocessor's detection rules to be updated via the Internet as a standard AV (the rule generation and update processes are further explained in detail).

3.3.2 The Kernel Module

This module is also a typical AV module, and it works as the interface between the *userland* AV and the coprocessor. Any communication between the AV client and the coprocessor must be forwarded via this kernel driver. It communicates with the coprocessor by writing to memory-mapped pages and Model Specific Registers (MSRs). This restriction protects the coprocessor from tampering attempts by malware samples running in *userland* [30].

3.3.3 Identifying Addresses to Monitor

In our proposed model, identifying the addresses to be monitored is delegated to the AV's *userland* component. The identification can be implemented using standard OS introspection queries, as implemented by typical security solutions [10]. This implementation choice eliminates the need for the coprocessor to have a complex x86 instruction decoder (considering that it will implement a simpler ISA), as proposed in previous work [51], thus simplifying our implementation. We expect the *userland* AV to identify the library's base addresses (ASLR-aware) and function's offsets at a process load time. When a new process is created, the AV should run this introspection procedure and request the

kernel driver to add an inspection breakpoint to this address. This breakpoint will be further identified by the main-CPU decoder, which will request a scan to the coprocessor.

3.3.4 Setting Coprocessor Rules

One of the main advantages of the proposed solution is that the rules can be set on-demand to the coprocessor and not be hardcoded in them. This flexibility is made possible by mapping memory pages shared between the coprocessor and the kernel. The AV's *kernel* component configures the coprocessor by writing the rules to these memory pages. These memory pages will be mapped to the coprocessor with executing and reading permissions-only to prevent their modification. We made such writes only available from the kernel to prevent *userland* malware from tampering with the coprocessor operation, thus following the most-privileged monitoring rule [52].

3.3.5 Triggering a Scan

An essential requirement for the efficient coprocessor operation is to have a precise inspection trigger, which allows inspection to occur only when required. Processor-wise, the simplest solution would be to modify the CPU's decoder unit to invoke the coprocessor whenever a `call` instruction is detected. The drawback of this solution is that it overloads the coprocessor with check requests for non-monitored functions and outsources to it the responsibility of identifying whether a given call is due to a monitored process or not. On the other hand, if we assign the AV and the main-CPU the responsibility of invoking the coprocessor only when required, we can simplify the coprocessor design. Thus, we propose the concept of `inspection breakpoint`, a trap flag added to instructions by the AV to signal to the coprocessor (via the main-CPU decoding) the beginning of a monitored region. This technique is inspired by the trap flags used by software breakpoints [11], which replace the first byte of targeted instructions with a trap flag that raises an exception when decoded by the CPU. This exception is handled by debuggers, which restore the original byte and perform their inspection routines. We here propose a modified trap flag that does not invoke a software debugger but the hardware coprocessor, thus, removing all the overhead caused during an exception. In our proposal, we limit the trap to be set to `call` instructions, limiting the replaced byte range to a single value to eliminate the need to restore it at the software level. The CPU decoder itself can restore the byte. Since the architecture already supports trap flags, we believe that the current CPU's instruction decoders can be easily adapted to invoke the coprocessor when a flagged instruction is identified. As an advantage of our method, we do not require the coprocessor to match the program counter addresses against a list of monitored addresses as implemented in previous coprocessors described in the literature [1].

Unfortunately, there is not a single, universal mechanism to cover all coprocessor usage scenarios. Therefore, we propose two types of inspection break-

point's flags to be applied according to the context: (i) the one applied to invoked function's instructions, and; (ii) the one applied to invoke function's instructions. The first trap type consists of a new instruction that indicates that the proceeding code is a monitored function. This technique can be applied in cases where compiler-support is available to generate this new instruction. It is a multi-byte instruction whose first byte indicates its operation (security check request) and the immediate indexes the set of rules to be applied (as defined by the AV when setting this inspection breakpoint). The second trap type modifies the software breakpoint trap flag to be applied to the calling instructions when no compiler support is available. We enforce that this type of trap can only be applied to the `call` instruction, so we can immediately proceed with the instruction decoding and do not need to restore the original byte as in a typical software breakpoint handling routine. According to the support provided by the underlying platform, we identified four usage scenarios for these techniques, as summarized in Figure 3 (specifically on items 5, 7, 9 and 10). The greater the support, the greater the security guarantees. In any case, we opted to support all cases to allow a smooth transition from existing support to our hypothesized scenario.

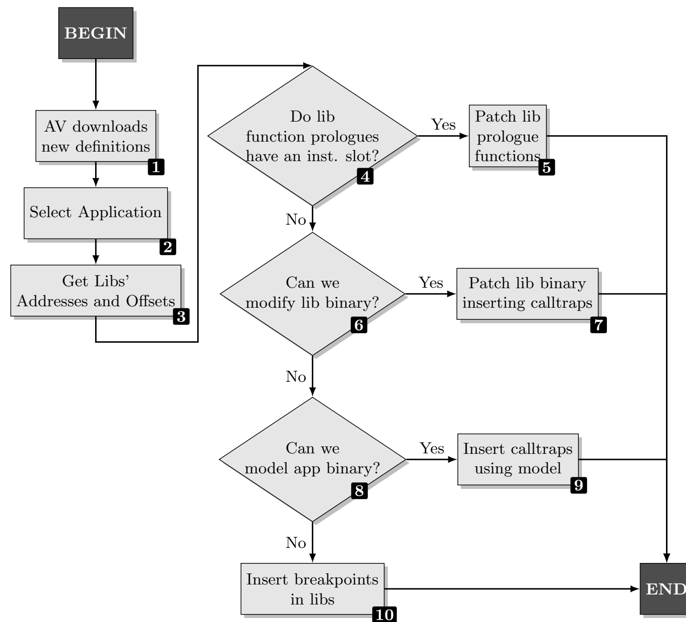

Figure 3: **Coprocessor Operation Summary.** There are four distinct ways to set up an inspection point.

Following, we detail the early mentioned scenarios.

```
1  PcreateProcessNotifyRoutine
     (...){
2    pid = GetProcessId();
3    libs = EnumProcessModules(
         pid);
4    addr = GetModuleAddress(
5        libs[target_lib],
6        target_function);
7    VirtualProtect(addr,
         WRITABLE);
8    __intrinsics_set_trap(addr);
9    VirtualProtect(addr,
         NOT_WRITABLE|EXECUTABLE)
         ;
10 }
```

Code 2: Process Creation Callback.

```
1  target_function(...)
2  {
3      entry_checkpoint(
4          empty_slot()
5      );
6      first_task();
7      internal_calls();
8  ...
9      exit_checkpoint();
10     return;
11 }
```

Code 3: Target Function Code.

```
1  64 8b 04 25 30
2  90 90 // NOP SLED
3  67 8b 40 02
4  75 e1
```

Code 4: Original Assembly Code.

```
1  64 8b 04 25 30
2  12 34 // RULE PATCH
3  67 8b 40 02
4  75 e1
```

Code 5: Patched Assembly Code.

1. **Compiler-Supported Invocation.** The best-case scenario when designing a next-generation system is when the newly launched systems fully support the new technology. In this sense, we can hypothesize that future systems will provide compiler support for invoking our coprocessor. This hypothesis is reasonable since our threat model defines that we aim to protect native OS libraries. OS vendors have no reason to refuse to armor their libraries against subversion. Indeed, Microsoft has already been proposing a secure CPU [41] that could be adapted according to our proposal. Therefore, the most efficient solution to invoke the coprocessor is to add the instruction-based inspection trap flag at the beginning of each function that can be potentially monitored. For such, the compiled libraries should provide an empty slot filled by the AV with the set of rules to be monitored (or specify that a given function will not be monitored) by setting (or not) the monitoring flag (see Code 3). The slot can be filled by default, for instance, with a `NOP` instruction that is lazy-patched by the AV when it decides that the function should be monitored (see example from Code 4 to Code 5). This solution allows AV companies to add monitoring flags to a distinct set of functions, according to their detection policies. The compiler can assist the lazy-patching procedure by providing `intrinsics` that AV can invoke on-demand (see Code 2). Attackers aware of this invocation mode might try to circumvent the coprocessor invocation by calling a function at a different offset than its entry point. To mitigate this bypass, we rely on the fact that modern systems employ control flow integrity policies that ensure that a function invocation

is only valid if the execution flow traverses given checkpoints. We suggest the coprocessor invocation be performed in an atomic manner during the checkpoint verification. We believe that this is a viable option since Microsoft has been deploying a similar strategy in its implementation to support the Intel CET technology [55]. It is essential to highlight that we expect these instructions to be protected from patching by the malware as the threat model assumes that integrity checks are performed.

2. **Binary Rewriting.** In some cases, the original library binary might not have been compiled with support to the coprocessor invocation. However, if no restriction is imposed, this binary can be rewritten to include these instructions, which allows us to add legacy support to this binary.

3. **Offline Binary Characterization.** In some cases, however, binary rewriting is not an option. For instance, if a company enforces a security policy in which only signed libraries are allowed to run, we will not be allowed to rewrite the binaries functionally unless we possess the proper crypto keys. In cases where no compiler support is available, we cannot trap the invoked function instructions. The coprocessor will not have an exact instruction trigger to know the function entry points (i.e. no compiler-assisted function annotation). On the other hand, we can still efficiently trap the invoking function's instructions by adding an inspection trap flag in the respective `call` instructions. A challenge to deploy this technique is to know which are the `call` instructions to be trapped. A possible solution for this problem is to develop offline a given binary call model, which can be done by the AV company and distributed to the users via a regular Internet-based update. Whereas this might sound like a significant limitation at first glance, there are multiple scenarios in which it is feasible. For instance, Internet browsers are often attacked by malware and are thus continuously monitored by AVs to prevent these attacks. We believe that it is entirely viable for an AV company to develop a model for the function invocations of this type of widespread application since these will likely be used (thus monitored) every day.

4. **On-Demand Breakpoints.** In the worst-case scenario, we have neither compiler support nor a previously-created model but still want to verify the software. This case might happen, for instance, when a user is opening a recently downloaded binary from the Internet or via e-mail attachments, and this interacts with system libraries. The only alternative, in this case, is to discover the calling addresses in runtime. Therefore, the AV can rely on traditional software breakpoints to stop at the beginning of the monitored functions every time they are invoked by a distinct instruction of the monitored process. When a software breakpoint is reached, the AV can proceed as a traditional debugger and restore the original instruction byte. Our approach also adds our proposed inspection breakpoint trap flag to the calling address and specifies the set of rules to be matched by the coprocessor by directly communicating with it via the shared memory

(instead of indirectly leveraging the instruction opcodes considered in the best-case scenario). Once the inspection trap flag is set, further invocations originated from that same address will be automatically trapped by the main-CPU decoder and forwarded to the coprocessor, as in previous cases. An initial overhead is imposed to handle the software breakpoints in the first function invocations. However, even in this case, the imposed overhead is smaller than the one imposed by traditional AVs since the analysis code is outsourced to the coprocessor and not executed by the main-CPUs. In the long-term, no overhead is imposed by this approach as the AV sets inspection trap flags after each software breakpoint.

3.3.6 Scanning Rules

We designed the operation model not to break the current AV's operation assisting it with additional, efficient hardware monitoring capabilities. We expect signatures, rules, and heuristics to be developed by the AV's companies and distributed via the Internet (as currently done). We also expect signatures and rules to be developed using the best practices [45] (see [14] for a discussion on `YARA` rules optimization). In our prototype, we considered standards already tested in the market and academia, such as ClamAV [59, 47] and YARA [63, 43] rules. ClamAV signatures are hashes and regular expressions distributed in a format that the AV engine can match against the parsed fields of the input files. YARA rules are complex rules also distributed in plain text. The YARA framework is usually responsible for lifting their representation and compiling them in memory to a finite state machine. Our model considers that the AV companies will distribute already-compiled YARA rules and interpretable ClamAV rules. We consider this to be feasible, as AV companies already have their custom frameworks to manipulate `YARA` rules [5]. We did not consider the instructions and the time taken to compile these rules or to parse input fields in all experiments. We compiled them statically for both sets of rules and expect AV companies to do the same since we do not want to add to our coprocessor the complexity of handling higher-level constructions such as the library or system calls. Besides, binary debloating procedures [50] might limit the distributed code to the minimum required set of instructions.

Rules Selection. Since we expect all rules to be distributed by the AV companies as already-compiled code, the selection of the ruleset to be applied to a given function or process turns in the choice of which binary region to be loaded. Therefore, after a flagged instruction's immediate value is read by the coprocessor, it is used to index a table that loads the correct rule set to be applied.

3.3.7 CPU to Coprocessor Communication

Once a function invocation is trapped, the coprocessor needs to receive the function arguments. The x64 calling convention ensures that at the moment that a call is executed, all arguments are already stored in the main-CPU's

registers. Therefore, these values need to be copied to the coprocessor's registers. The CPU communicates to the coprocessor via a shared bus. In our proposed model, the CPU generates `MOV` instructions via the microcode unit. These instructions move data from all main-CPU registers to the coprocessor's ones via memory-mapped registers. This implementation simplifies the coprocessor design since the main-CPU solves pipeline forwarding issues by design. This modelling decision also allows us to measure the CPU-coprocessor communication cost natively in a cycle-accurate simulator running the specified x86 architecture. Therefore, the effects of latency and the cost of generating additional instructions to trigger a coprocessor scan were considered in the experiments presented in Section 4. We highlight that alternative implementations to this communication mechanism would involve the coprocessor querying data on-demand, which would involve higher abstraction components, such as the cache coherence protocol, and would not benefit from the arguments placement in the main CPU registers. This would degrade the solution performance back to the case of a traditional multi-core system, also evaluated in Section 4.

Speculative Detection. A corner case of Terminator operation is that exceptions might occur in the main CPU after a scan using the coprocessor was launched. In this case, the scan proceeds as usual. If the exception occurred due to speculative execution, Terminator will also speculatively try to detect the threat. We consider that if a malicious construction is detected in any execution flow (speculative or usual), it should be detected.

3.3.8 Scan Engine

The rules execution are performed by the coprocessor core. Our only requirement is to have a Float Point Unit (FPU) to support rules involving entropy calculation. It could be, for instance, a reduced version of the x86 ISA. In our experiments, we considered an ARM Cortex-A53 [49] (detailed in Table 1), which implements ARMv8-A 64-byte instruction set architecture (ISA). This core model has been used as the "little" more efficient counterpart in big.LITTLE architectures such as Qualcomm's Snapdragon 810. We consider this model a viable alternative since ARM coprocessors are already incorporated into current architectures for monitoring purposes (e.g., even in the same die for AMD Secure Processor [36]). Nevertheless, we note that we could still use a similar coprocessor implementing a subset of the x86-64 ISA with no extensions.

The suggested coprocessor model has a bigger data cache than required by our proposal (i.e., considering L1 and L2 caches), which suggests its area can be cut by half, as the cache memory area dominates current processors. In our simulations, we remove the L2 cache initially proposed in the ARM Cortex-A53, as it was a large shared cache between multiple cores, whereas we want a single core and keep minimal area usage. We connected the coprocessor's L1 caches (each 8 KiB) to the main-CPU's Last Level Cache.

Notifying Infections: After matching a suspicious pattern, the coprocessor needs to notify the *userland* AV. In our proposed model, the coprocessor raises an interrupt delivered to the main-CPU to be handled by the kernel driver.

Table 1: Core parameters for Sandy Bridge and ARM's Cortex-A53.

	Sandy Bridge	**Cortex-A53**
Processor's Clock Frequency	2 GHz	2 GHz
Pipeline: Execution Type / Stages	OoO Superscalar / 18	In-order Superscalar / 8
Instruction Fetch: Width / Queue Size	16 B / 18	16 B / 13
Instruction Decode: Width / Buffer Size	up to 5 uops / 56	up to 2 uops / 28
Uop Cache	1536 uops	None
Scheduler: RS Entries / Dispatch Width	54/up to 6 uops	36/up to 2 uops
FUs: ALU/Int.Mul./Int.Div.	3/1/1	2/1/1
FUs: Branch Exec./Address Gen. Unit	1/2	1/1
Vector Units: Int./FP	3/3	2/2
Register Bank Size: Int./Vector	160/144	32/32
Buffers: ROB/Load/Store	168/32/16	NA/NA/NA
Branch Predictor	Tournament	2-level PAg, 3072-entry PHT
Miss-Prediction Penalty	15-20 cycles	7 cycles
Branch Target Buffer	4-way, 4096 entries	Direct-mapped, 256 entries
Return Stack	16 entries	8 entries
Cache Line Size	64 B	64 B
L1 Data Cache / Latency	8-way 32 KB / 4 cycles	Direct-mapped 8 KB / 3 cycles
L1 Instruction Cache / Latency	8-way 32 KB / 4 cycles	2-way 8 KB / 3 cycles
Private L2 Cache / Latency	8-way 256 KB / ca. 11 cycles	None / None
Unified Last Level Cache / Latency	Inclusive 12-way 16 MB / ca. 28 cycles	None
TLB L1 / Miss penalty	4-way, 64 entries / 7-8 cycles	Dir. Mapped, 10 entries / 2-3 cycles
TLB L2 / Miss penalty	4-way, 512 entries / 20-21 cycles	4-way, 512 entries / 11-22 cycles
L1 Data Prefetcher	Adjacent Line prefetcher	Data Stride Prefetcher
L1 Instruction Prefetcher	IP Stride	Next-line Prefetcher
L2 Data Prefetcher	DCU Stream prefetcher	None

This interrupt could be delivered via a new security check, such as the ones introduced by Intel CET [32], which already presents an available interrupt handler for future developments. Alternatively, we can rely on Non-Maskable Interrupts (NMI), since it was designed for handling error cases (in this case, a security check failure) and was previously leveraged by other security solutions described in the literature [10].

3.4 Prototyping

Our main goal in this work is to present a real-world solution for the malware detection performance problem. We previously presented the theoretical design that should lead to performance penalty mitigation. We should also evaluate whether this is accomplished in practice and if other non-ideal conditions (e.g., memory latency) and other constraints (e.g., resource competition by multiple running code pieces) are present.

Thus, we also prototyped our solution in a cycle-accurate processor simulator to evaluate its operation in real-world conditions [3]. Our simulation modeled one coprocessor (ARM Cortex-A53) per processor core (Sandy Bridge architecture). Each coprocessor transparently shares the context and address spaces of the respective main-CPUs.

Our implementation is constrained because Intel's decoder stage only supports ideally 5 instructions per cycle, and we need to generate new instructions when a monitored call instruction is decoded. To overcome this challenge, we modified the decode stage of the Sandy Bridge core as follows: Upon encountering a monitored call instruction `X` (e.g., `syscall`), the decoder only proceeds

its operation if X is the single instruction being decoded in that cycle (much like legacy instructions requiring microcode).

We decode X into 5 uOps, fully occupying the decode stage limit. The first three instructions are STORE instructions which store register contents to the memory address indicated during process creation as the buffer for the coprocessor communication. Thus, this allows the processor to transmit X's parameters to the coprocessor. This project decision is beneficial in multiple scenarios. For instance, in the case of a Linux system call, the six parameters specified by the SYSV calling convention for syscalls fit into a single cache line, which will cause a single store in the write-combining store buffer in the L1 cache (as long as the address is aligned). The same effect is observed for most Windows calls. The fourth uOp sends a request to the coprocessor itself, signaling a request in its buffer. We modeled this as another STORE instruction, but we created a different port for a direct request, making this buffer behave like a cache buffer. By doing so, we can faithfully emulate a "request buffer" for the coprocessor, where we change the write penalty to limit how frequently the coprocessor can handle call analysis. We set the write penalty to 72000 cycles (see the penalty definition experiment in Section 4), which limits our emulated coprocessor to only handle an analysis request every 72000 cycles. The final uOp performs the call itself. We highlight that this call is dependent on the previous 4 uOps, as we created artificial register dependencies within the register renaming step to simulate this dependency faithfully.

We simulate an antivirus coprocessor that performs rule matching asynchronously. The core running the application does not wait for the AV analysis to complete, and it might have several pending requests for analysis in the AV coprocessor buffer request. In case the request buffer is full of pending requests, the main-CPU stalls until it can send the request (i.e., the request to the coprocessor is at the reorder buffer head, thus stalling any other commits, as x86 processors commit in order).

4 Evaluation

In this section, we evaluate Terminator regarding the multiple aspects of its operation. First, we present the results of design exploration that cover multiple scenarios. Later, we present a simulation on how Terminator acts when assisting an AV in an actual scenario.

Exploration 1: On-demand checks. We evaluated the performance gains enabled by Terminator when used by an AV to outsource on-demand matching procedures. Due to the parallelism, the number of saved cycles in the main-CPU is proportional to the coprocessor's number of instructions. We performed the experiments matching all 224 rules present in the official YARA repository [61], which encompasses the detection of malware, packing, and malicious documents. We only measured the signature matching cost since we considered pre-compiled YARA rules (compiled statically). We matched all rules against a dataset of 30 thousand real malware samples obtained from a partner CSIRT. This dataset

was already validated in previous studies [19, 7]. All tests use the same system described in Section 2.

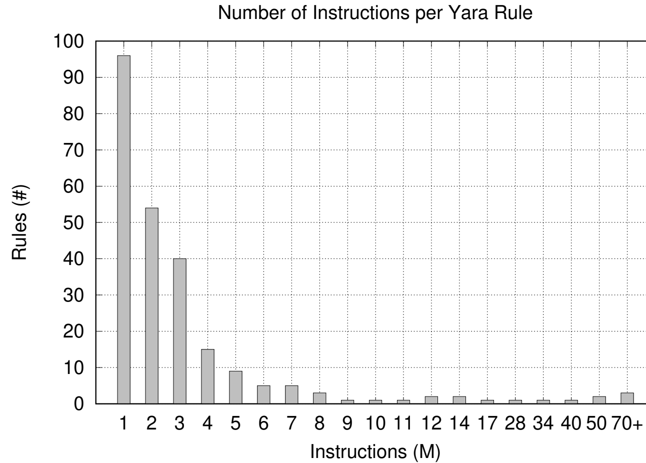

Figure 4: **Match on-demand of YARA signatures against a dataset of 30K malware samples.** The average number of instructions spent by each rule shows that a significant number of CPU cycles could be saved by outsourcing the matching to the coprocessor.

Figure 4 shows the distribution of the average number of instructions required to match each YARA rule against the whole dataset. YARA rules are significantly distinct among themselves according to their goals. Although identifying a file by its header is a simple task, finding an unaligned pattern over the whole file is more complicated. More than half of the rules (around 58%) in the dataset execute between 1 and 2 million instructions to match all files. However, around 2% of the rules are complex, requiring over 50 million instructions to match the whole dataset. Millions of CPU cycles were saved for all the rules as these instructions were not executed in the main-CPU. Notice that, in this case, the overhead mitigation does not depend on how fast the coprocessor is: Every instruction outsourced to the coprocessor saves main CPU cycles regardless of how fast their corresponding implementation will be executed in the coprocessor.

In total, the matching of this dataset in the coprocessor saved 1.03 billion cycles from the main-CPU each time a full system scan was requested, thus eliminating 100% of the system scan overhead. Currently, full system scans are a dedicated task that AVs perform upon a reboot. However, our mechanism enables full system scans to occur concurrently with other processing tasks.

Exploration 2: Background checks. In addition to full system scans, AVs also perform full file checks during ordinary runtime operations. For instance, AVs often check files as soon as they appear in the filesystem. This type of scan is often backgrounded to reduce the performance penalty. Although it

does not have the same high impact as a full system scan, its effects can still be perceived by other high-performance applications, such as games, as mentioned in Section 1. We believe that our proposed coprocessor can also help mitigating performance penalties in this scenario. To confirm that possibility, we implemented a filesystem kernel driver that invokes a userland process to match created files with `YARA` rules as soon as the files are placed in the file system, as performed by the commercial tool `YaraGuard` [46]. We developed our version of the tool to be able to instrument it with performance monitoring code. The baseline tool uses the standard `YARA` framework to perform the match. We also implemented an alternative version that outsources this step to our coprocessor. Both versions launch the `YARA` rules presented in the previous experiments for all created files.

To evaluate the impact of the AV background checks over the applications, we need to identify the frequency of checks performed in background by the AVs. We approximated this number by the frequency of filesystem accesses performed by the AVs when the OS is idle, i.e., when the user is not directly interacting with any application but OS is still performing background tasks. We considered that all accessed files would be checked. This experiment stresses a typical time opportunity (idle time) employed by AVs to launch their scan procedures. It provides us a lower bound for the scan performance impact, as any system interaction would certainly increase these values.

We compared the AV performance when running with and without coprocessor support. To allow a fair comparison of the two implementations, we should consider the same scan frequency and accessed files in all runs. Therefore, we traced OS interactions within a time frame and replayed them against the two implementations.

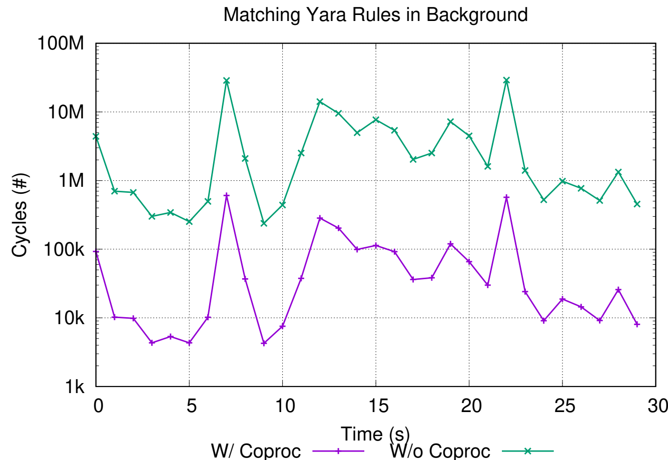

Figure 5: **Yara checks in background.** The coprocessor-assisted AV version is one order of magnitude faster.

Figure 5 shows the difference in the number of CPU cycles spent by the two approaches. We notice that both graphics have similar curves, as the number of CPU spent cycles is proportional to the number of files accessed in a given time. However, the number of CPU cycles spent by the coprocessor-based version is one order of magnitude faster (requiring fewer cycles) than the version without the coprocessor support. It happens because the verification routines are not performed in the main-CPU. Despite lower, the number of spent cycles is non-negligible, as some CPU cycles are still required for the userland application to be invoked by the kernel and then trigger the coprocessor scan. We believe that such reduction might enable AVs to keep performing scans even when the CPU is executing high-performance tasks like games.

Exploration 3: Hardware Design. From a design perspective, the main advantage of deploying a co-processor is the possibility of designing specific hardware settings for the task at hand. Ideally, we were free to evaluate any design principle. Despite that, we opted to remain tied to the most common market alternatives and decided to evaluate only the variation on the number of Arithmetic Logic Units (ALUs), which is reasonably common when designing an Instruction-Level Parallelism (ILP)-capable processor. More specifically, we are simulating a co-processor with multiple, replicated specialized pipelines, a possibility brought by the simulation nature of this work and that might or not concretize according to the processor vendor's design choices.

We evaluate the impact when varying the number of ALUs by analyzing the execution of the same `YARA` rules described above. Figure 6 shows the average number of cycles spent to match three representative categories of rules (packer identification, malware detection, and CVE identification). We limited the presentation to these categories for the sake of readability and concision.

As expected, each matching task requires a diverse number of cycles since the rules have different complexities. Therefore, each class of rules presents distinct performance gains (in the number of cycles). Despite that, all of them presented performance gains when adding more ALUs. A higher gain was observed when moving from 1 to 2 ALUs. Smaller gains start to be observed from this point, as the processing load started to be better distributed among the two ALUs. There are almost negligible gains after the addition of 4 and 5 ALUs. For the overall case, we discovered that considering 3 ALUs result in gains for any class of `YARA` rules, thus resulting in performance gains in comparison to a standard processor when performing pattern matching tasks.

Exploration 4: Real-time Checks. Although saving million CPU cycles by outsourcing on-demand checks is an exciting result, the coprocessor's usage scenarios that bring more noticeable performance gains are its application in real-time, as it was designed for. During real-time operation, the performance gain derives not from the total amount of instructions outsourced to the coprocessor at once but from the number of checks outsourced to it in the long-term. To understand this overhead mitigation's impact, we evaluated how many rules and which complexity level can be outsourced to the coprocessor to be executed in parallel with the function calls. In this first experiment, we considered the matching rules used in the `ClamAV` , as they are more focused on detecting mal-

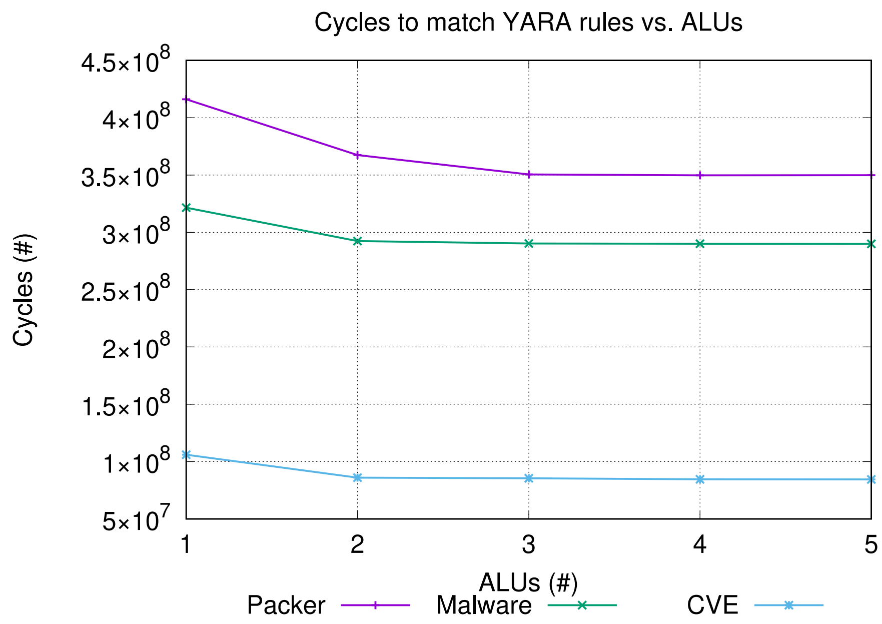

Figure 6: **Cycles to match `YARA` rules vs. ALU.** Adding more ALUs to the coprocessor saves processing cycles.

ware binaries (thus simpler) than the more generic `YARA` rules present in the repositories that also detect malicious documents and objects.

The first challenge for this type of operation is that function and system calls have different complexities. Considering that each function call performs a distinct operation and takes distinct arguments as input, they may require different amounts of instructions to be executed. For this evaluation, we considered the SPEC benchmark again, as it provides a diverse set of combinations of function calls and parameters. Figure 7 shows the average number of instructions for some popular function calls invoked during the execution of the SPEC benchmark's applications. We notice, for instance, that creating a new process is the most complex task among all function calls, as it is composed of multiple internal sub-tasks, such as allocating memory and retrieving a new process identifier (PID) [38].

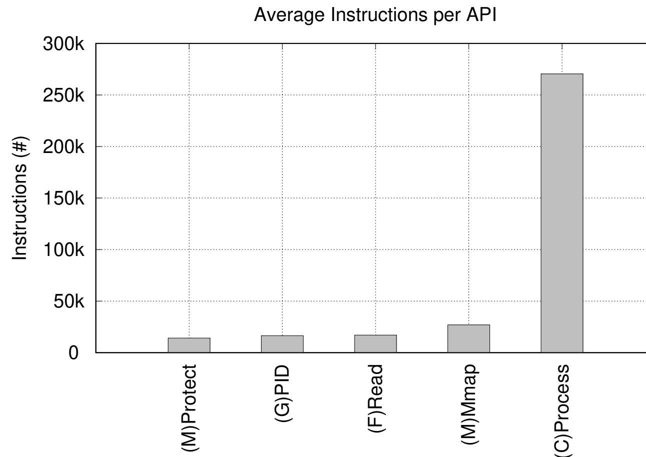

Figure 7: **Average CPU cycles per API calls.** The complexity of each API call is significantly different, thus the variation on the number of spent CPU cycles.

In addition to their complexities, distinct API calls are invoked at specific rates, according to their goals. For instance, performing I/O is much more common along with the whole process execution than adjusting OS parameters, which usually only takes place at process startup. Figure 8 shows that the number of invocations is also not constant even during the benchmark execution. When the benchmark applications start, a higher number (thousands) of function calls are performed due to the need of setting up the execution environment. As the applications proceed, they spend most of their time computing over data rather than acquiring more data (few hundred calls). This initial step generates lots of interesting data to be scanned by an AV, such as libraries being loaded and handlers being opened, such that it stresses more the AV engine than the application execution in the long-term step. Similar behavior is expected

for malware samples, whose setup step includes unpacking routines, network connections establishment, processes creation, and so on.

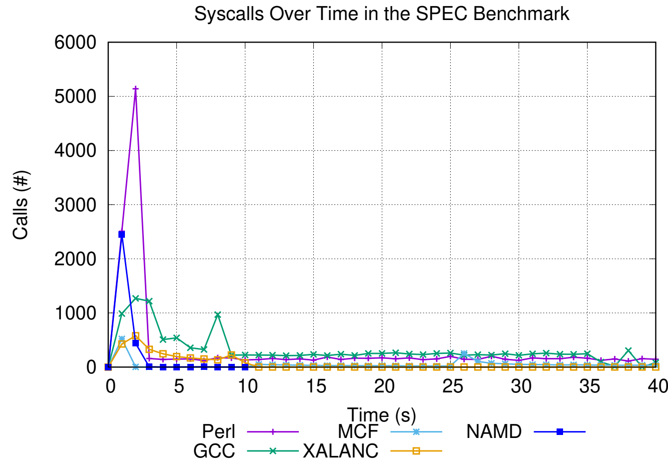

Figure 8: **Number of Calls during SPEC execution.** Most calls are made in the beginning of the execution to setup the execution environment and parameters.

Figure 9 shows the relative frequency of the most invoked APIs during the execution of the SPEC benchmark's applications. In total, we identified 421,393 distinct calls to 2,015 distinct APIs.

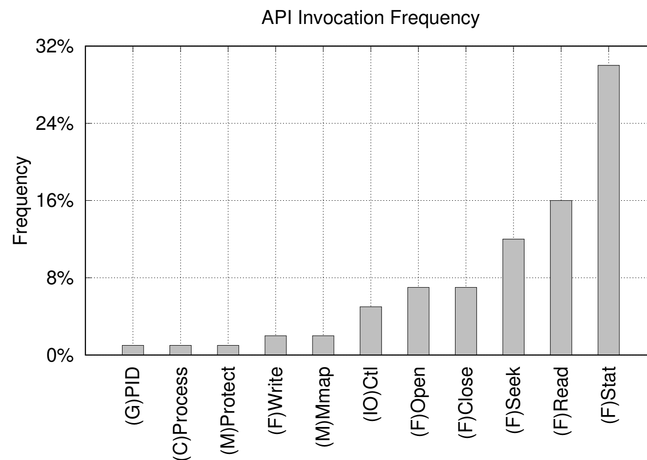

Figure 9: **API invocation distribution.** APIs are invoked by application at a distinct rate.

We leveraged these measurements to weight function calls cost and invocation frequency. We would like to avoid bounding our measurements by extreme cases, such as (i) a very slow but infrequent call (e.g., CreateProcess); or (ii) consider a very fast and frequent call but which is unlikely to be meaningful for AV monitoring (e.g., file stat). In the first case, we would assign all the remaining function calls much more time for inspection than they actually would have. In the second case, we would significantly limit the scan opportunity, when, in fact, the scan of these calls could simply be skipped. Ideally, we would like to identify an average value which allows the scan time to be as similar as possible to the actual time spent by the SPEC benchmark applications executing the monitoring functions. To do so, we weighted the identified function costs and frequencies using a `trimmed mean`, as shown in Figure 10. We established 72 thousand as the average number of instructions executed per function call.

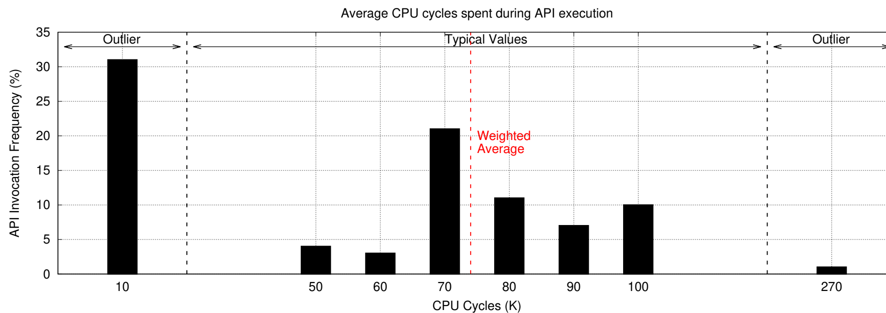

Figure 10: **Weighted Average.** Extreme values are unlikely to be monitored and thus were discarded.

Face to this function diverse scenario, we believe that considering this average is an interesting experimental strategy to understand what happens in the average case during a real-time AV scan. Nevertheless, more studies are warranted in the future to understand the AV scan of specific cases (e.g., particular functions).

This average value was used as an upper bound when executing the `ClamAV` rules in our coprocessor. We matched all arguments of all functions calls using `ClamAV` rules. We performed the match using the `ClamAV` *daemon*, so we discarded from our measurements the instructions required to load the AV. We found that each rule takes on average a corresponding number of 1,828 instructions on the main CPU to be matched in the coprocessor. Thus, an average API call of 72 thousand instructions could allow the sequential checking (in the worst-case scenario) of 39 distinct `ClamAV` rules in parallel with the main-CPU execution. This result indicates a significant reduction on the main CPU load without losing detection capabilities.

Exploration 5: Extending API monitoring to the simplest `YARA` rules. So far, we have shown that a coprocessor applying `ClamAV` rules can mitigate the overhead imposed by traditional software-based AVs, which should suffice to

establish it as a viable approach. However, the reliance on a coprocessor allows us to go beyond and propose more elaborate matching mechanisms. Typical AV's runtime checks only verify the immediate function calls arguments (e.g., if the argument of a function points to a file descriptor), but not to their indirections (e.g., the pointed file content). Increased detection capabilities would be enabled by extending the checks to these indirect arguments. We explored this possibility by evaluating the application of `YARA` rules in runtime.

We selected `YARA` rules for this experiment because its repository has more complex rules than `ClamAV` . These rules were originally proposed to match entire files, as those pointed by some function call arguments (although we are here leveraging them out of their original context to match function parameters, in a similar way to event filters [27]).

Since we do not have actual `YARA` rules designed to be applied in real-time and we did not want to bias the experiments creating our ones, we opted to evaluate the cost of applying the original `YARA` rules and supposed that their real-time versions will present the same cost and distribution. We believe that this supposition is reasonable for a first attempt towards qualifying the gains brought by a coprocessor. We evaluate the application of `YARA` rules specifically designed for real-time matching in a further-presented experiment.

A challenge for implementing this extension is that `YARA` rules are very diverse in goals and complexity, such that an AV company would have to opt for a limited set of rules to be applied. We observed that by measuring the average number of instruction taken by each `YARA` rule matching the arguments of the function calls performed by the SPEC benchmark applications. Figure 11 shows the rules distribution according to their average cost in the number of instructions. We notice that whereas the shortest rule (substring matching) takes 500 instructions per input byte, the most complicated rule (matching a complete binary) takes over 100 thousand instructions per input byte. Whereas these values are very high, we can benefit from the fact that most function calls parameters are short.

Although 90% of all rules present a cost per input byte that fits within an average API call, in practice, the extent of the function arguments to be matched will determine which rules will be selected for real-time application. Matching entire file contents is certainly impractical without adding overhead. Although AVs do not often adopt this strategy, we can still match a significant portion of binary fields. A typical PE header can be identified using less than 20 bytes [40], such that more than 10% of all `YARA` rules initially designed for offline checking could be applied in real-time without any modification and without imposing performance overhead via the coprocessor execution. If the AV vendor opt to deliver some of the remaining rules, a distinct approach is required, as following discussed.

Exploration 6: Extending API monitoring to the most complex `YARA` rules. We have shown that a parallel coprocessor can be used to completely replace traditional AV monitoring routines and extended to match some simple `YARA` rules. Let us now suppose that one's policy is to perform complex matching procedures or scan entire file contents. Whereas this is not the usual way that

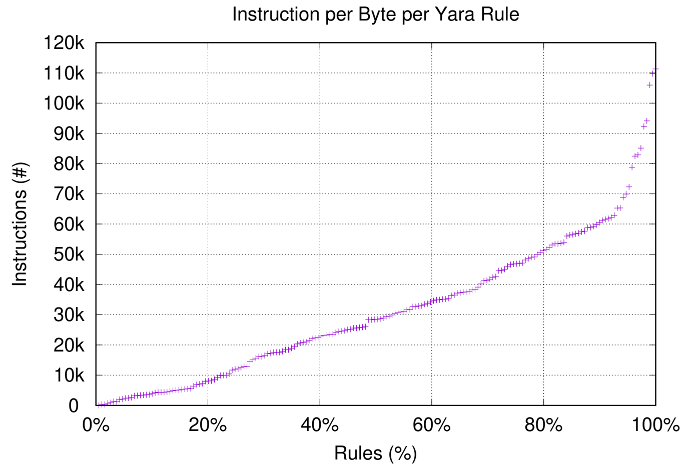

Figure 11: **Yara rules cost per input byte.** Complex rules requires more CPU cycles to be processed.

real-time AV work, we decided to explore this possibility since our proposed coprocessor enables it.

It is improbable that a complex match rule could be performed without introducing significant overhead. An alternative approach to mitigate the overhead is to amortize the matching cost by executing its instructions in the interval between two consecutive function calls.

Figure 12 shows the average number of instructions between two consecutive calls of the same function during the execution of the SPEC benchmark applications. On the one hand, I/O-bound functions, such as `read` and `write`, occur in bursts (the distance between two consecutive calls is approximate to the average number of instructions for all system calls). Thus, verification routines regarding them cannot be masked along with their execution. On the other hand, the distance between two consecutive calls to non-I/O-bound functions is significant. In the case of `CreateProcess`, million instructions are executed between two consecutive calls to it, which allows all complex YARA rules, without exceptions, to be matched in the meantime. This results reveals into a significant strategy to improve AV's performance as a whole in the cases where the checks can be delayed without security implications.

Exploration 7: Performance Analysis. The number of signatures and their complexity is not the only trade-off regarding adopting a coprocessor that should be evaluated. We should also consider the impact on performance in actual scenarios when external factors affect the code execution (e.g., memory loading latency) and the influence of resource competition by multiple code excerpts.

To evaluate that, we simulated the monitoring of the applications from the SPEC benchmark (as shown in Section 2) in the prototype described in Sec-

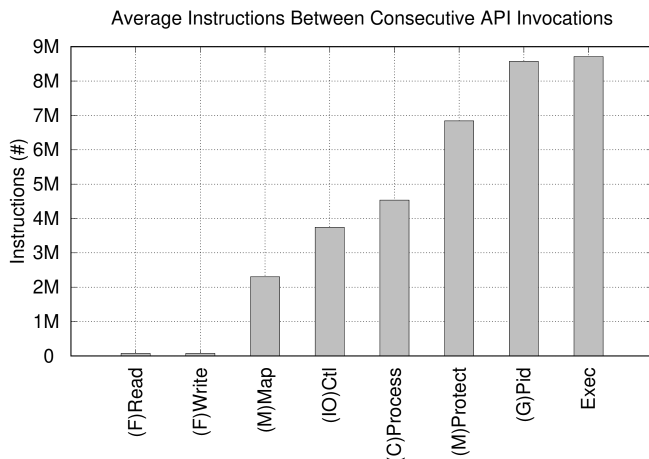

Figure 12: **Number of instructions between consecutive calls of the same function.** Many complex Yara rules can be matched within consecutive function calls. Bursting routines are exceptions to this.

tion 3.4. All traces were composed of the first 2 billion executed instructions (the excerpt with most function invocations) extracted via Intel PIN [37].

Our simulation considered three scenarios: (i) the baseline execution (BASE), with no AV monitoring; (ii) the monitoring by a software-based AV (SWAV); (iii) the monitoring by our coprocessor-supported AV (HWAV). The first case presents no overhead and no protection by definition. In the second case, we consider that the AV is executed in the main-CPU. For a fair comparison, we assume that each AV's check takes on average 72K cycles, as in our target coprocessor design goal previously presented, even though we are aware that current AVs cannot achieve this in practice. Remind that we consider custom `YARA` rules and that a typical AV has to parse multiple fields to implement these same checks. We consider that the same 72K cycles will be executed in our proposed coprocessor in the latter scenario.

To entirely and fairly simulate a real AV operation, we would need to trigger checks for the same function calls that the real AVs hook. However, most commercial AVs are closed-source solutions, and their vendors do not disclose such information. Therefore, our adopted strategy simulates distinct scenarios corresponding to the hypothesized upper and lower bounds for the AV's operation and performance.

In the worst-case scenario, we consider that the AVs will trigger checks for any function calls exported by an OS library. Whereas it is unlikely that any real AV will adopt this strategy, this analysis enables a clear understanding of how useful coprocessors are in mitigating the imposed scanning performance overhead.

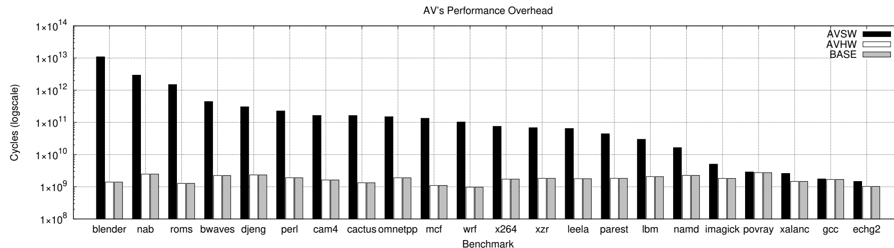

Figure 13: **Performance evaluation when tracking all function calls.** Comparison between execution without AV (BASE), execution with software AV, and execution with the proposed coprocessor model.

Figure 13 shows the experiments results. We first observe that the coprocessor scan can be triggered and executed without imposing significant overhead (AVHW close to the BASE); Its usage resulted in no performance overhead in most cases. The effects of the newly added instructions and of the communication latency are completely "diluted" in the greater amount of executed instructions to the point of becoming negligible.

We also notice that overhead penalties are unavoidable by software-based solutions, even though we try to minimize them by supposing that they are as fast as the coprocessor-based AV to inspect functions. The software-based AV overhead is bounded by the number of monitored calls performed by the benchmark. The more invocations, the greater the overhead. In this sense, the imposed overhead might be of multiple orders of magnitude greater than the application's processing times themselves for many applications. These results demonstrate that a full system monitoring at the software-level would be impractical, highlighting the need for a coprocessor to achieve this goal.

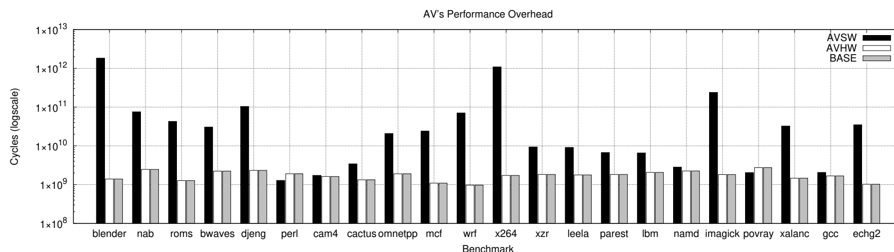

Figure 14: **Performance evaluation when tracking the C runtime function calls.** Comparison between execution without AV (BASE), execution with software AV, and execution with the proposed coprocessor model.

When analyzing the worst-case scenario, it is important to keep in mind that it includes monitoring all libraries linked to the processes. However, each library might present a distinct performance overhead if analyzed individually (which

might be the AV's choice). Figure 14 shows that most of the so-far described performance overhead is due to the monitoring of the libraries responsible for implementing the C runtime used by most applications on the SPEC benchmark.

It is also important to notice that the worst-case scenario monitors all libraries and all functions within those libraries. In a real scenario, AV companies will likely choose only some functions to be monitored. To reproduce this scenario, our challenge is to identify the minimal set of functions that an AV company must monitor. We suppose that a plausible selection is to consider only the function calls that trigger system calls. We consider syscall invocation as a proxy for the importance of the function call, such that AVs should not skip checks for these critical system operations. Thus, we believe that this set represents a lower bound for the AV's performance evaluation.

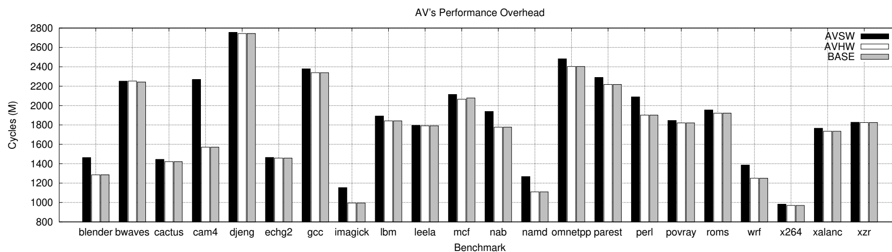

Figure 15: **Performance evaluation when tracking only system calls.** Comparison between execution without AV (BASE), execution with software AV, and execution with the proposed coprocessor model.

Figure 15 shows that, as in previous experiments, the coprocessor usage resulted in no performance overhead in most cases and only a 4.95% performance decrease in the worst case (for a single thread of the CAM4 benchmark). However, the CAM4 benchmark was impacted with a 44.4% performance penalty imposed by the software-AV (mostly mitigated with coprocessor assistance).

Even in the general case, inspecting few invocations, the coprocessor-based AV resulted in an average performance gain of 5.66% when considering all benchmarks. This result becomes more significant when considering that an AV does not inspect a single process but the multiple processes running in the system. In this sense, when considering the execution of the whole set of considered SPEC application's excerpts in the same system, the coprocessor saved 2.11 billion cycles from the main-CPU. In comparison, this number is higher than the number of cycles spent by 18 (78%) of all SPEC applications' trace excerpts. In other words, this means that any of these applications' trace excerpts (formed by 2 billion instructions) could have been individually executed "for free" when using our coprocessor-assisted system.

Terminator results in performance gain even when operating in constrained scenarios. We have tested different coprocessor's buffer sizes (from 2 to 16), and observed no performance impact for any application. This result suggests that the considered limit of 72K cycles for scanning is adequate to

avoid generating bottlenecks.

We highlight that these are the results achieved when considering only the minimal set of monitored calls. Any check added by the AV vendor would increase those overhead values to the upper bound limits presented above. Therefore, we conclude that adopting a coprocessor is a viable solution for real-time security monitoring.

Exploration 8: Implementation Alternatives. Our previous results show that parallel scans are a promising solution for mitigating the performance overhead of real-time AV scans. The remaining question is why a coprocessor should be leveraged for this task and not some alternative implementation. A frequently referred implementation alternative is the use of an extra processor core, often available in multi-core systems. To evaluate whether this is a viable alternative, we developed a software implementation of an AV. The AV is composed of two components, each one running in a distinct CPU core to stress the core communication channel: one component is responsible for hooking the monitored API functions and serializing the collected data to the other component (via a memory pipe). The other component runs in a distinct core and matches the function arguments against the `YARA` rules presented in the previous experiments.

Table 2: **Implementation Alternatives.** The communication cost diminishes the advantage of using existing processor cores.

Implementation	**Ideal**			**Sofware**		**Expected**
Cost	**Base**	**Serial**	**Parallel**	**Interception**	**Piped**	**Piped2**
Monitored	✗	✓	✓	✗	✓	✓
API	72K	72K	72K	72K	72K	72K
Interception	0K	0K	0K	34K	34K	0K
Communication	0K	0K	0K	0K	21K	21K
Match	0K	72K	0K (72K)	0K	0K (72K)	0K (72K)
Total	72K	144K	72K	106K	127K	93K
Overhead	0,00%	100,00%	0,00%	47,00%	76,00%	29,00%

Table 2 summarizes the results (i) expected from our designed co-processor and (ii) obtained from the execution of the multi-core software-based AV (average of 100 repetitions). The “Base” column shows the results expected for the base system, in which 72K instructions on average are executed per function call invocation. Since no scan is performed, this is the total execution cost. The “Serial” column shows what is expected for the serial execution of an AV scan, in which another 72K cycles on average are required to match the function arguments, thus resulting in a 100% performance overhead. The “Parallel” column shows the results expected for our designed coprocessor. Since the 72K cycles to match the scan rules are spent in parallel with the function execution, no performance overhead is imposed. The “Interception” column accounts for the cost of intercepting functions at the software level, which is not negligible, resulting in a 47% overhead. The “Piped” column shows the cost of intercepting functions at the software level and sending the collected data to another core for matching. The cost of matching is mitigated due to the parallel execution, but the cost of intercepting and transferring data imposes performance

overhead rates of 76% on average. The "Piped2" column shows the case where we suppose that we can fully mitigate the function interception cost via an `inspection breakpoint` mechanism in hardware, thus eliminating the cost of software execution. The resulting overhead (29%) is derived from the fact that the communication cost from one core to another (including data serialization at software level) is not mitigated.

In a summary, while using an extra core indeed provides some performance gains in comparison to the serial scenario, the AV-imposed performance overhead is only fully mitigated using a coprocessor close to the core running the monitored API functions. Alternatively, this result can be interpreted as the use of a coprocessor allowing a greater number of rules to be applied during the execution of a monitored API function than an extra CPU core. Therefore, we conclude that existing CPU cores would only completely mitigate the performance overhead of real-time AVs if equipped with efficient inspection and notification mechanisms, which was exactly what we designed for our coprocessor.

However, in addition to performance, one should also evaluate whether the additional cores will be available in the system for the AV usage. In our threat model, we assumed that we cannot assure that extra cores will be always available (idle) for AV processing, which, in our view, suffices for discarding this alternative as a definitive solution. Anyway, we decided to evaluate what is the actual impact of sharing a core with a process competing for resources with the AV scan. To do so, we considered a dual-core system sequentially running the SPEC benchmark applications on the first core and our software AV in the second one. We gradually scheduled the most CPU intensive processes running on our system to the second core and evaluated the impact on the execution time of the monitored application running in the first core.

Table 3: **The impact of sharing a core.** The more processes running on the same core as the scanning routine, the greater the overhead.

Processes	0	1	2	3	4	5	6
Overhead	0%	43%	47%	90%	106%	123%	133%

Table 3 shows the results for this experiment as an average of 10 repetitions. The base case is when no process is sharing the CPU with the scanner, so 0% overhead is imposed. When one CPU-intensive process from the system is moved to run on the same core, the performance (total execution time) was 43% affected. This is an expected number, since now the CPU has only around 50% of the CPU time to perform the scan of the function arguments. When more processes are moved, more degradation is observed, even though it is not linear. With 4 process or more competing for the AV scan core, the applications from the SPEC benchmark are already consuming the double of the original time to run, since the AV scan started to work as a barrier for the advance of the normal execution in the first core. Therefore, we conclude that using a distinct core for the execution of a real-time scanner is only viable if there are guarantees that

the core will be available, which is the case of our coprocessor. Alternatively, we can conclude that running the AV scan on a distinct CPU core is only a viable option if the AV checks are allowed to be non-blocking, such that detection notifications can be delayed.

Practical Case Study: Simulating an AV Operation. So far, we hypothesized multiple operational scenarios for the proposed approach. Here, we present a concrete scenario that allows us to parameterize the AV company's rules. To do so, we simulated an AV analyst's action to develop rules to detect the threats presented in the previously presented malware dataset. We developed rules based on a taxonomy of frequent malware behaviors [29].

In total, we developed 9 generic rules (for both `YARA` and `ClamAV`) that can detect and block 99% of samples present in the dataset. Table 4 presents the rules according to their complexity. The first rule (`identify writing target`) is the simplest one, as it only looks to the first two bytes pointed by a `write` call to check if it is a valid executable or not. It allows us to characterize, for instance, the writing of the `MZ` bytes as a `PE executable injection`. The last rule (`debugger check`) checks for the presence of debuggers or anti-debug techniques used by malware samples to evade detection and is the trickiest rule. Whereas Operating Systems (OS) usually have predefined APIs for this task (e.g., `IsDebuggerPresent` on Windows), we cannot invoke system functions from within the coprocessor environment. Therefore, we manually traversed OS structures to find this same information, as demonstrated viable by previous work [9].

Table 4: **Developed heuristics.** The developed set of heuristics are enough to detect and block all samples in the malicious dataset when applied over runtime-collected function arguments.

Heuristic	Goal
Check if write target starts with 'PE' or 'ELF'	Blocks File Dropping
Check allocation flags	Prevents Unpacking
Check PE fields	Blocks corrupted binaries
Check Load Library	Prevents Side-Loading
Check write memory strings	Blocks DLL Injection
Check path registry contains a substring	Blocks Writes to Autorun
Check send URL	Blocks data exfiltration via GET
Check socket target address	Blocklists malicious domains
Check EPROCESS flags	Checks for anti-debugging

The ability to traverse OS structures without requiring additional libraries shows that our solution design is viable in actual scenarios. Moreover, the reduced number of rules required to cover the entire dataset compared to the maximum number of allowed rules indicates that the approach is viable and can be scaled to cover more complex threats that require more rules to be matched.

It is essential to understand that the developed rules do not operate homogeneously. Since each rule captures a distinct malicious behavior, each rule

is triggered in a distinct malware execution stage. For instance, on the one hand, checking the MZ string is a behavior that usually appears right at the beginning of a malware operation, when the sample is being loaded or dropping its payload. On the other hand, for many malware samples, socket openings tend to appear after the malware already performed other actions in the system to allow it to exfiltrate the information it collected previously. Therefore, AV scanning procedures impose a distinct performance penalty according to the malware characteristics.

Therefore, the real impact of outsourcing monitoring to a coprocessor is only made clear when we analyze the CPU impact of matching the developed rules against malware and goodware samples in real-time. Figure 16 shows the average CPU usage (in cycles) of the software-based AV when monitoring the first window of 1K monitored function calls of the goodware applications and the malware samples that performed at least 1K calls. The results are plotted for: (i) the set of goodware samples, which kept running all the time, and thus triggering subsequent checks (ii) the set of malware samples which could be detected right at the beginning of their execution, thus being stopped by the AV, and; (iii) the set of malware samples that could be detected only after a significant part of the trace was considered and kept triggering checks until their blocking. Since we considered averaged values of all traces, the values for an early-detected malware (10% of the dataset) and a late-detected malware (90% of the dataset) converged to 10% and 80% of the trace's length, respectively.

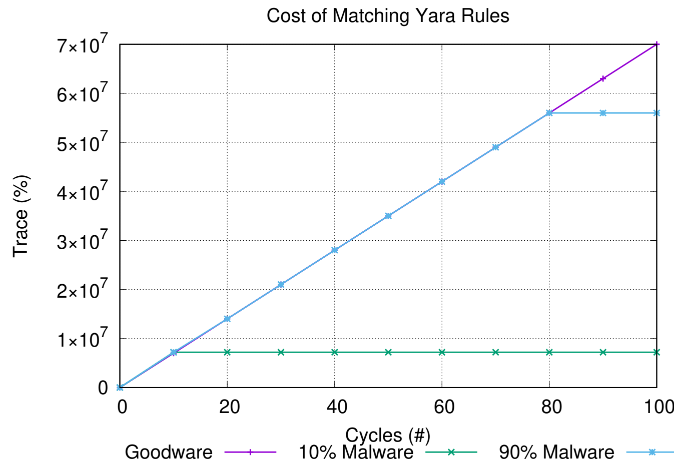

Figure 16: **Cost of matching YARA rules.** The monitoring of goodware applications keeps consuming CPU cycles.

When monitoring in real-time the samples that can be detected right at the beginning of their execution, our first observation is that the impact of a software-based AV is not so significant (less than 10M cycles, on average). However, in most cases, the samples must be monitored for a long-time before

detection, including monitoring many functions that will not act maliciously. Due to that, their monitoring, on average, costs more than 50M cycles for each 1K monitored function calls, which highlights the benefits of applying a coprocessor for this task. Highlighting that when we considered the goodware (legitimate software) samples' operation: although their function calls are monitored, a detection shall never happen. Nevertheless, the AV does not have a mechanism to prove them goodware. Thus they need to be kept under monitoring for their whole execution under the risk of an application revealing itself as malicious at some late point. This case reinforces the need to outsource scans to a specialized device to mitigate the impact over goodware during their whole operation.

5 Discussion

In this section, we discuss the impact of our findings and the limitations of our proposal.

Good detection rates relies on good policies. Whereas we provided an efficient matching mechanism for AVs efficiently applying detection rules, AV companies are still responsible for determining which rules will be applied to each API call and the number of rules to be applied according to each API complexity.

The Coprocessor should operate along with a typical AV. Our proposal does not aim to eliminate the traditional mechanisms employed by AVs. Instead, we aim to provide an efficient alternative for the most common and expensive checking conditions. Therefore, we do not expect AVs to outsource to our coprocessor all detection rules, but only the most expensive and frequently used ones. Function calls that are rarely invoked or require specific interactions with the OS should still be performed via traditional means (see Section 2).

Real-Time monitoring. Modern AVs do not rely only on checking files on disk to detect malware. They also detect malicious constructions in runtime, which significantly increases their detection capabilities. We opted to develop a coprocessor that encompasses both operation modes in this work, thus broadening the possibly adopted threat models. Our model allows AVs to apply customizable and updatable rules to function arguments. We consider this a significant advantage over previous proposals that only allowed static signature matching [8]. As showcased via the application of rules, our coprocessor can be adapted to operate using any real-time approach, which includes the emerging online, machine learning-based algorithms still costly to be implemented in software [58].

Coprocessors vs. Additional Cores. Adding extra processing units always results in trade-offs (e.g., energy, area, costs, so on). Whereas many distinct positions can be justified, we make our claim to adopt coprocessors instead of additional, general-purpose processor cores. Our main argument is that a coprocessor does not require the whole complexity of an entire core, which reduces costs. However, even if the costs are comparable, we still do not believe that

adding more cores only for this task is the best alternative compared to adding a coprocessor for synchronous real-time monitoring. The previous study has already shown that frequent thread migration's performance overhead becomes prohibitive [53] in these cases. Someone can still argue that an existing core might be used for asynchronous system checks when the system is idle. Whereas this is true as an enhancement for existing solutions, we still do not consider this as the best project decision from the development from a hardware-assisted AV from scratch, since the core communication costs will still be present [22]. Moreover, current threat models avoid relying on additional cores not to face the risk of missing an attack due to the lack of an idle core to monitor an application [17]. In fact, it has been hard to find available cores even in supercomputers since attackers started to create malware for these environments [48].

Coprocessors vs. GPUs. The massive processing capabilities of GPU indeed enable incredible opportunities for speeding up diverse security tasks. Thus, GPUs are often announced as the platform for the next-generation of security solutions. However, in practice, whereas the first research work proposing GPU-assisted security solutions date back a decade or more [2], not many of the promises concretized as real so-far. This lack of real applications can be explained by the fact that besides advantages, GPUs also have significant limitations. Although GPUs are great for AVs outsourcing the on-demand scan of a massive number of files against the same rules due to the Single Instruction Multiple Data (SIMD) paradigm. GPUs are not suitable for AVs outsourcing real-time scans, as the cost in terms of time and energy of offloading individual chunks of data (API arguments) from the CPU to the GPU would be high. Also, this usage would not benefit from the GPU's parallel processing capabilities, as distinct parameters need to be scanned for each invocation. Moreover, in many cases, such as the gaming one, moving AVs from the CPU to the GPU can be seen as only shifting the bottleneck location, as the GPU would be now overwhelmed with AV requests while rendering the game scenes. Finally, programming many distinct GPUs may be challenging, and tools like OpenCL might not be enough to assure AV's correct operation in all possible scenarios. We hypothesize that the reasons above might have limited the development of GPU-based AVs so-far. We currently are aware only of an Intel proposal to integrate AVs to their GPUs [15]. In addition to limited memory scans rather than real-time monitoring, we are not sure whether and how this deployment progressed. Therefore, we claim that a specific-purpose co-processor might be the solution to streamline efficient AV inspection in a standardized manner to all systems, users, and AV companies.

Coprocessors vs. FPGAs. A question that naturally emerges from our proposal is why a coprocessor rather than a FPGA, since it also enables offloading processing tasks to an external hardware unit. Whereas FPGAs are suitable for on-demand scans or specific, fine-grained real-time tasks (e.g., shadow stacking [17]), an FPGA would likely never reach a clock rate fast enough to allow coarse-grained, real-time APi monitoring without delaying its scan response to the main system. FPGAs can certainly be made faster when turned into Application Specific Integrated Circuits (ASICs), but the more a solution gets closer

to an ASIC, the harder to update its malware definitions, which is naturally streamlined by the coprocessor. More details about related work on FPGAs are presented in Section 6.

Kernel, Userland, and Attack Surfaces. A question derived from our developments is whether the introduction of a monitoring coprocessor could increase the attack surface and/or expose users to more risks. If an attacker could control the code that is run by the coprocessor, the attacker could inspect user data (although not modify it), which might turn into security violations. Our solution to this problem is to make the access to the monitoring code more privileged. In our solution, the monitoring code (rules) can only be written to the coprocessor via the kernel and not via arbitrary userland applications, such that an attacker would have to escalate privileges to be able to deploy malicious code in the coprocessor. However, in this case, the attacker already would have full control over the system, such that the coprocessor would not pose an additional risk by itself. Of course, this decision imposes some limitations to the AV operation: If we made the coprocessor accessible in userland, AVs could monitor APIs via ordinary DLL injection procedures instead of via more complex kernel drivers. Moreover, applications could even leverage the coprocessor for generic processing tasks triggered in the inspection points defined on their own code and/or in third party apps. However, these possibilities indeed would create a greater risk of subversion of the monitoring mechanism, such that we decided for a more restricted threat model. Alternatives to isolate contexts in the coprocessor operation are worth studying in future works.

The risks of misaligned jumps. An attempt to subvert the coprocessor invocation is to jump to the middle of a function without traversing the function prologue, thus not triggering an inspection. Notice that if one malware can invoke the original function address without traversing the trampoline function added by the current software-based AVs (see Section 2), no scan would be performed for that call. Our model expects the system to be protected with a control flow integrity mechanism to ensure that these jumps are not valid. However, as these protections and our coprocessor are decoupled solutions, one could still disable the control-flow integrity protection independently from the coprocessor. In this case, our solution is as vulnerable as a typical AV.

Multi-Core Systems. We limited our developed Proof of Concept (POC) to operate on a single core-basis (i.e., each main-CPU has its coprocessor). It brings significant advantages, such as reducing the communication latency and the competition for resources. The coprocessor application in practice over time might reveal that more straightforward rules are enough to counter a myriad of threats. Thus, coprocessors might be shared among different CPUs without affecting the performance. This shared approach would require our design to include a buffer to each coprocessor to store scan requests. It would also require the coprocessor to identify which main-CPU the requests belong to deliver the interrupt to the proper core.

The coprocessor doption depends on a CPU vendor deciding to integrate Terminator to its platform. This requires the creation of new instructions and microcode, with the corresponding adaptation of the existing control mech-

anisms (e.g., decoding). We believe these modifications are viable since CPU vendors recently proposed CPU modifications even broader than the ones required by our solution, thus suggesting that they consider these changes viable as well. For instance, Intel proposed FRED, a new set of (CALL and RET) instructions to enter and leave the kernel that cause modifications to the whole system call operation mechanism [33], including the replacement of the OS Interrupt Description Table (IDT).

AV Modeling Limitations. Commercial AVs are closed-source solutions, and the vendors do not disclose full details about their implementation choices. Therefore, our evaluations are limited to consider an AV's average behavior. Thus, most of our findings are exploratory reasoning about the possibilities. More concrete definitions, such as the maximum number of rules to be applied in runtime, can only be made by the AV vendors. We are sure that AV companies will be able to tune their solutions in a very fine-grained manner when applying their knowledge about their products' internal working.

6 Related Work

In this section, we present related work to better position our contributions.

Outsourcing security processing. The great performance requirements of some security tasks has been acknowledged by previous work. A typical strategy to mitigate the performance overhead is to outsource the processing of the scanning task to another party. For instance, Gionta et al. [26] proposed outsourcing costly AV's memory scans to the cloud. Whereas effective in mitigating the performance overhead, this approach introduces significant latency to the detection process, which is unsuitable for real-time operation. In this paper, we adopt the outsourcing idea, as exemplified by this paper, but leveraging hardware assistance to also reduce the detection latency.

Hardware coprocessors. Outsourcing complex tasks to external entities, such as hardware coprocessors, is a common practice in computer systems design [44]. This approach has been previously proposed for accelerating multiple tasks, such as image acquisition [24], floating-point operations [62], and neural networks [54], but they hardly ever address system security tasks (and more specifically malware detection).

Even though some work approach challenges that might be related to security tasks, such as template matching [23], they do not discuss the intricacies of security tasks.

Secure coprocessors. Security is usually addressed at the hardware design level for accelerating cryptography routines [42]. External monitors were also proposed to implement system monitors, such as shadow stacks [20]. These previous work, however, does not implement any rule-based detection approaches. A noticeable exception is the application of hardware modules to speed up network packet inspection routines [21]. Similar to AV matching, these procedures require significant processing capabilities that are often offloaded to hardware. These proposals still do not discuss the details of matching mechanisms, such

as rule's characteristics, as we presented for our coprocessor.

Hardware AVs. The academic literature presents some proposals for external hardware modules. However, most of them are not comparable to our proposal as they are not based on any real AV. An approach related to ours is the migration of `ClamAV` rules to hardware, as proposed in some previous work [59, 47]. The authors propose implementing specific hardware to store the rules, whereas our approach presents the significant advantage of still representing the rules at the software-level. The closest related work to ours is the FPGA-based AV [8], which proposes a particular detection method that raises an exception when a malicious execution is found. We extended this concept to complete AV matching routines. More specifically, we consider `YARA` and `ClamAV` rules as matching routines. Previous work considered the port of `YARA` rules to FPGAs [56] but presenting a specific-purpose hardware design. Our work, instead, relies on the execution of the `YARA` framework without modifications. Despite that, we found similar results about the impact of distinct patterns and input lengths in the matching performance.

7 Conclusions

In this paper, we investigated the performance overhead imposed by software-based AntiViruses (AVs) that implement their checks via function interposition. We understand that adding overhead is inevitable to any software-based approach as AV's instructions run in the same processors as the monitored code. We proposed a coprocessor to assist the AV by outsourcing its matching procedures to hardware and saving CPU cycles. Our experiments have shown that our approach can save up to 70 million CPU cycles when outsourcing on-demand checks for matching `YARA` rules. We also designed a new `inspection breakpoint` mechanism that allows AVs to flag code regions as monitored and the coprocessor to be invoked when the main-CPU core decodes the flags. We showed that we could match on average 39 real `ClamAV` rules in parallel with typical function call invocations by outsourcing the match to our proposed coprocessor resulting in performance gains of up to 44% in SPEC benchmark applications. We consider that our approach is a viable solution for mitigating current AV solutions' performance overhead and expect that this work might foster future AV developments.

Reproducibility. The developed prototype is available at `https://github.com/marcusbotacin/Real.Time.AV` and `https://github.com/mazalves/OrCS`.

References

[1] G. Aggarwal, N. Thaper, K. Aggarwal, M. Balakrishnan, and S. Kumar. A novel reconfigurable co-processor architecture. In *Proceedings Tenth International Conference on VLSI Design*, pages 370–375, India, Jan 1997. IEEE.

[2] Malak Alshawabkeh, Byunghyun Jang, and David Kaeli. Accelerating the local outlier factor algorithm on a gpu for intrusion detection system. In *Proc. 3rd Work. on GP-GPUs*, page 1, US, 2010. ACM.

[3] M. A. Z. Alves, C. Villavieja, M. Diener, F. B. Moreira, and P. O. A. Navaux. Sinuca: A validated micro-architecture simulator. In *IEEE HPCC, ISCSS, ESS*, page 1, US, Aug 2015. IEEE.

[4] Ionut Arghire. Windows 7 most hit by wannacry ransomware. `http://www.securityweek.com/windows-7-most-hit-wannacry-ransomware`, 2017.

[5] Avast. Yaramod. `https://engineering.avast.io/yaramod-inspect-analyze-and-modify-your-yara-rules-with-ease/`, 2019.

[6] Avira. Avira antivirus: Game mode explained. `https://www.avira.com/en/blog/avira-antivirus-game-mode`, 2020.

[7] Tamy Beppler, Marcus Botacin, Fabrício J. O. Ceschin, Luiz E. S. Oliveira, and André Grégio. L(a)ying in (test)bed. In Zhiqiang Lin, Charalampos Papamanthou, and Michalis Polychronakis, editors, *Information Security*, pages 381–401, Cham, 2019. Springer International Publishing.

[8] M. Botacin, L. Galante, F. Ceschin, P. C. Santos, L. Carro, P. de Geus, A. Grégio, and M. A. Z. Alves. The av says: Your hardware definitions were updated! In *2019 14th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 27–34, UK, July 2019. IEEE.

[9] Marcus Botacin, Vitor Falcão, André Grégio, and Paulo de Geus. Analysis, anti-analysis,anti-anti-analysis: An overview of the evasive malware scenario, 2017.

[10] Marcus Botacin, Paulo Lício De Geus, and André Grégio. Enhancing branch monitoring for security purposes: From control flow integrity to malware analysis and debugging. *ACM Trans. Priv. Secur.*, 21(1), January 2018.

[11] Marcus Botacin, Paulo Lício De Geus, and André grégio. Who watches the watchmen: A security-focused review on current state-of-the-art techniques, tools, and methods for systems and binary analysis on modern platforms. *ACM Comput. Surv.*, 51(4), July 2018.

[12] Marcus Botacin, Marco Zanata, and André Grégio. The self modifying code (smc)-aware processor (sap): a security look on architectural impact and support. *Journal of Computer Virology and Hacking Techniques*, 1(1), 2020.

[13] Marcus Felipe Botacin, Paulo Lício de Geus, and André Ricardo Abed Grégio. The other guys: automated analysis of marginalized malware. *Journal of Computer Virology and Hacking Techniques*, 14(1):87–98, Feb 2018.

[14] Michael Brengel and Christian Rossow. Yarix: Scalable yara-based malware intelligence. https://publications.cispa.saarland/3360/, 2021.

[15] Peter Bright. Intel, microsoft to use gpu to scan memory for malware. https://arstechnica.com/gadgets/2018/04/intel-microsoft-to-use-gpu-to-scan-memory-for-malware/, 2018.

[16] c9x. Fast syscall. https://c9x.me/x86/html/file_module_x86_id_313.html, 2016.

[17] Sadullah Canakci, Leila Delshadtehrani, Boyou Zhou, Ajay Joshi, and Manuel Egele. Efficient context-sensitive cfi enforcement through a hardware monitor. In Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 259–279, Cham, 2020. Springer International Publishing.

[18] CarbonBlack. Who needs malware? powershell and wmi are already there! https://www.carbonblack.com/2016/04/06/who-needs-malware-powershell-and-wmi-are-already-there/, 2016.

[19] F. Ceschin, F. Pinage, M. Castilho, D. Menotti, L. S. Oliveira, and A. Gregio. The need for speed: An analysis of brazilian malware classifers. *IEEE Security & Privacy*, 16(6):31–41, Nov.-Dec. 2018.

[20] Ronny Chevalier, Maugan Villatel, David Plaquin, and Guillaume Hiet. Co-processor-based behavior monitoring: Application to the detection of attacks against the system management mode. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, page 399–411, New York, NY, USA, 2017. Association for Computing Machinery.

[21] Y. H. Cho and W. H. Mangione-Smith. A pattern matching co-processor for network security. In *Proceedings. 42nd Design Automation Conference, 2005.*, pages 234–239, US, June 2005. ACM.

[22] Theofanis Constantinou, Yiannakis Sazeides, Pierre Michaud, Damien Fetis, and Andre Seznec. Performance implications of single thread migration on a chip multi-core. *SIGARCH Comput. Archit. News*, 33(4):80–91, November 2005.

[23] A. de Vasconcelos Cardoso, N. Nedjah, L. de Macedo Mourelle, and Y. M. Tavares. Co-design system for template matching using dedicated co-processor and modified elephant herding optimization. In *2018 IEEE 9th Latin American Symposium on Circuits Systems (LASCAS)*, pages 1–4, MX, Feb 2018. IEEE.

[24] J. Dubois and M. Mattavelli. Embedded co-processor architecture for cmos based image acquisition. In *Proceedings 2003 International Conference on Image Processing (Cat. No.03CH37429)*, volume 2, pages II–591, Spain, Sep. 2003. IEEE.

[25] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, US, June 2011. ACM/IEEE.

[26] Jason Gionta, Ahmed Azab, William Enck, Peng Ning, and Xiaolan Zhang. Seer: Practical memory virus scanning as a service. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, page 186–195, New York, NY, USA, 2014. Association for Computing Machinery.

[27] GoDaddy. Procfilter. `https://godaddy.github.io/procfilter/`, 2019.

[28] André R. A. Grégio, Dario S. Fernandes Filho, Vitor M. Afonso, Rafael D. C. Santos, Mario Jino, and Paulo L. de Geus. Behavioral analysis of malicious code through network traffic and system call monitoring. In Misty Blowers, Teresa H. O'Donnell, and Olga Lisvet Mendoza-Schrock, editors, *Evolutionary and Bio-Inspired Computation: Theory and Applications V*, volume 8059, pages 180 – 189, US, 2011. International Society for Optics and Photonics, SPIE.

[29] André Ricardo Abed Grégio, Vitor Monte Afonso, Dario Simões Fernandes Filho, Paulo Lício de Geus, and Mario Jino. Toward a taxonomy of malware behaviors. *The Computer Journal*, 1(1), 2015.

[30] F. Hsu, M. Wu, C. Tso, C. Hsu, and C. Chen. Antivirus software shield against antivirus terminators. *IEEE Transactions on Information Forensics and Security*, 7(5):1439–1447, 2012.

[31] Intel. Manual. `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf`, 2016.

[32] Intel. Control-flow enforcement technology. `http://kib.kiev.ua/x86docs/Intel/CET/334525-003.pdf`, 2019.

[33] Intel. Flexible return and event delivery (fred). `https://software.intel.com/content/dam/develop/external/us/en/documents-tps/346446-flexible-return-and-event-delivery.pdf`, 2021.

[34] Kaspersky. Gaming mode on. `https://www.kaspersky.co.in/gaming-mode-on/`, 2020.

[35] F. L. Levesque, A. Somayaji, D. Batchelder, and J. M. Fernandez. Measuring the health of antivirus ecosystems. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 101–109, US, 2015. IEEE.

[36] LibreBoot. Frequently asked questions. `https://libreboot.org/faq.html`, 2015.

[37] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM PLDI*, page 1, New York, NY, USA, 2005. ACM.

[38] Microsoft. Flow of createprocess. `https://flylib.com/books/en/4.491.1.52/1/`, 2009.

[39] Microsoft. x64 calling convention. `https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=vs-2019`, 2018.

[40] Microsoft. Pe format. `https://docs.microsoft.com/en-us/windows/win32/debug/pe-format`, 2019.

[41] Microsoft. Meet the microsoft pluton processor – the security chip designed for the future of windows pcs. `https://www.microsoft.com/security/blog/2020/11/17/meet-the-microsoft-pluton-processor-the-security-chip-designed-for-the-future-of-windows-pcs/`, 2020.

[42] M. Nabeel, M. Ashraf, E. Chielle, N. G. Tsoutsos, and M. Maniatakos. Cophee: Co-processor for partially homomorphic encrypted execution. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 131–140, US, May 2019. IEEE.

[43] N. Naik, P. Jenkins, N. Savage, L. Yang, K. Naik, and J. Song. Augmented yara rules fused with fuzzy hashing in ransomware triaging. In *2019 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 625–632, China, Dec 2019. IEEE.

[44] David Nellans, Kshitij Sudan, Erik Brunvand, and Rajeev Balasubramonian. Improving server performance on multi-cores via selective off-loading of os functionality. In *Proceedings of the 2010 International Conference on Computer Architecture*, ISCA'10, page 275–292, Berlin, Heidelberg, 2010. Springer-Verlag.

[45] Neo23x0. Yara performance guidelines. `https://gist.github.com/Neo23x0/e3d4e316d7441d9143c7`, 2020.

[46] NoVirusThanks. Yaraguard. `https://www.novirusthanks.org/products/yaguard/`, 2018.

[47] N. L. Or, X. Wang, and D. Pao. Memory-based hardware architectures to detect clamav virus signatures with restricted regular expression features. *IEEE Trans. on Computers*, ""(""):"", 2016.

[48] Charlie Osborne. This linux malware is hijacking supercomputers across the globe. `https://www.zdnet.com/article/this-linux-malware-is-hijacking-supercomputers-across-the-globe/`, 2021.

[49] David A Patterson and John L Hennessy. *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan kaufmann, US, 2016.

[50] Anh Quach, Aravind Prakash, and Lok Yan. Debloating software through piece-wise compilation and loading. In *USENIX Security*, SEC'18, page "", "", 2018. USENIX.

[51] A. Raveendran, V. Patil, V. Desalphine, P. M. Sobha, and A. David Selvakumar. Risc-v out-of-order data conversion co-processor. In *Int. Symp. on VLSI Design and Test*, "", 2015. IEEE.

[52] Christian Rossow, Christian J. Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *Sec. and Priv.*, SP '12, page "", "", 2012. IEEE.

[53] Abhik Sarkar, Frank Mueller, Harini Ramaprasad, and Sibin Mohan. Push-assisted migration of real-time tasks in multi-core processors. *SIGPLAN Not.*, ""(""):"", 2009.

[54] E. Setiawan and T. Adiono. Implementation of systolic co-processor for deep neural network inference based on soc. In *ISOCC*, "", 2018. IEEE.

[55] Yarden Shafir and Alex Ionescu. R.i.p rop: Cet internals in windows 20h1. `https://windows-internals.com/cet-on-windows/`, 2020.

[56] Shreyas G. Singapura, Yi-Hua E. Yang, Anand Panangadan, Tamas Nemeth, Peter Ng, and Viktor K. Prasanna. Fpga-based acceleration of pattern matching in yara. In *Int. Symp. on Applied Reconfigurable Computing*, page "", "", 2016. Springer.

[57] SPEC. Cpu 2006. `https://www.spec.org/cpu2006/`, 2006.

[58] R. Sun, M. Botacin, N. Sapountzis, X. Yuan, M. Bishop, D. E. Porter, X. Li, A. Gregio, and D. Oliveira. A praise for defensive programming: Leveraging uncertainty for effective malware mitigation. *IEEE TDSC*, ""(""):"", 2020.

[59] T. Tran Ngoc, T. T. Hieu, H. Ishii, and S. Tomiyama. Memory-efficient signature matching for clamav on fpga. In *ICCE*, "", 2014. IEEE.

[60] Yara. Yara - the pattern matching swiss knife for malware researchers. `https://virustotal.github.io/yara/`, 2018.

[61] Yara. Yara - the pattern matching swiss knife for malware researchers. `https://github.com/Yara-Rules/rules`, 2018.

[62] X. Zhang and X. Shen. A power-efficient floating-point co-processor design. In *Int. Conf. on CS and Soft. Eng.*, "", 2008. IEEE.

[63] A. Zhdanov. Generation of static yara-signatures using genetic algorithm. In *EuroS PW*, "", 2019. IEEE.