PROTECTING TODAY'S PROCESSORS

WHITE PAPER

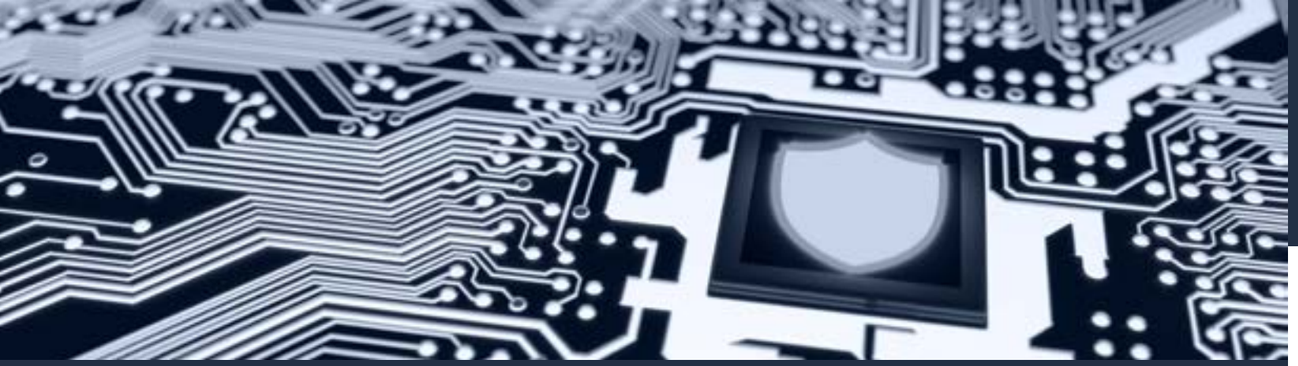
CoreGuard®: A Silicon IP Security Solution
version 2

Jothy Rosenberg
Founder & Executive Chairman

FEBRUARY 2023

Original version by Greg Sullivan, 2018

DOVER MICROSYSTEMS



This paper introduces CoreGuard®, a silicon IP security solution that can prevent the network-based cyberattacks plaguing our software-based systems. In this paper, we evaluate existing security technology, outline the approach and architecture, and make the case for a truly secure processor.

Introduction

CoreGuard is a silicon IP security solution that detects and blocks entire classes of cyberattacks. CoreGuard's hardware consists of logic added to existing SoC designs. This hardware logic enforces per-instruction, per-word, insubvertible, fine-grained micropolicies that are defined in software. The hardware logic also maintains a strict separation between application software, including the operating system, and CoreGuard micropolicies and associated data. CoreGuard's micropolicies can enforce a wide range of security requirements, including protection of memory against buffer overflows, prevention of control flow hijacking (such as Return-Oriented Programming attacks[1]), and management of data integrity and data confidentiality.

Evaluating Security Technology

Before investing in technology to mitigate potential cyberattacks, you should consider the following questions:

1. What classes of cyberattacks are you trying to prevent? Security professionals sometimes call this the *threat* model
2. Is the security technology itself subject to cyberattack? It is vital that new security technology you add does not just add a larger *attack surface* for the would-be attacker.
3. If the *threat landscape* changes—that is, if the attacks you are worried about change—is the security technology able to adapt?

[1] ROP (Return-Oriented Programming) on Wikipedia: https://en.wikipedia.org/wiki/Return-oriented_programming

QUESTION 1

What is your threat model?

To answer question #1, it is helpful to look at the Common Weakness Enumeration site maintained by MITRE[2]. As the name suggests, the CWE site documents classes of software errors (aka weaknesses) that can be exploited by attackers. An exploited weakness is labeled a vulnerability and categorized in MITRE's Common Vulnerabilities and Exposures (CVE) list of all unique attacks ever observed.

CoreGuard's threat model is quite general: we assume that all software running on a system's host processor is untrusted and potentially malicious. That includes both the operating system and application software. CoreGuard blocks entire

QUESTION 2

Is the security technology itself subject to attack and compromise?

To answer question #2, one should consider two aspects:

First: To the extent that the security technology is implemented in software, what guarantees are given that the security software itself is not riddled with exploitable weaknesses? If a vendor says that the software has been extensively tested and subjected to review, keep in mind that (a) the same assertions have been made about innumerable software packages that have been successfully attacked, and (b) security software solutions have repeatedly been demonstrated to be vulnerable to exactly the sorts of attacks they are trying to prevent simply because they too are subject to the 15-50 bugs per 1,000 lines of source with many of these systems exceeding 10 million lines of code. Worse, because security software typically runs at high privilege, a successful attack against security software compromises the entire system.

Second: If the security software is relying on the correct operation of the host operating system, then the security software is vulnerable to every attack to which the host OS is vulnerable. There are no provably secure operating systems, with the possible exception of seL4[3].

CoreGuard does not rely on the host OS for its operation and is therefore able to isolate all of its security enforcement logic

[2] CWE FAQ description of Weakness and Vulnerability: https://cwe.mitre.org/about/faq.html#A.1
[3] seL4 product page: https://sel4.systems/

QUESTION 3

Can the security technology adapt to a changing threat landscape?

To answer question #3, consider the range of hardware design features that are intended to increase security. Most of these hardware design features are aimed at a particular sort of attack. For example, "No eXecute" (NX) bits on memory pages mitigate binary code injection attacks (though depend on the OS not being attacked), and "Trusted Execution Environment" solutions, such as Arm's TrustZone, isolate one region of code and data (deemed "Trusted") from another region of code and data (deemed "Untrusted"). Neither region is protected from exploitation of vulnerabilities such as a buffer overflow within the region, but regions are prevented from interfering with each other.

The hardware features described above, and many others, are perfectly legitimate, but each feature performs exactly one function, and there is no possibility for adaptation to changes in the threat landscape. With the CoreGuard security solution, the hardware elements guarantee Complete Mediation. This concept (from Saltzer and Schroeder's seminal 1974 paper entitled *The Protection of Information in Computer Systems*) defines that every access to every object must be checked for authority. The checks that CoreGuard enforces, however, are defined by micropolicies, and the set of installed checks can be changed over time to deal with an evolving threat landscape.

From a hardware design perspective, CoreGuard is logic that modifies existing CPU/SoC designs. If the existing CPU has built-in security features, there is no need to duplicate those protections using CoreGuard; CoreGuard can be used to target the many vulnerabilities that are not covered by existing hardware and software technology.

COREGUARD®
Protected. Trusted.

How CoreGuard Works

CoreGuard consists of three main components: Micropolicies, Metadata, and Hardware.

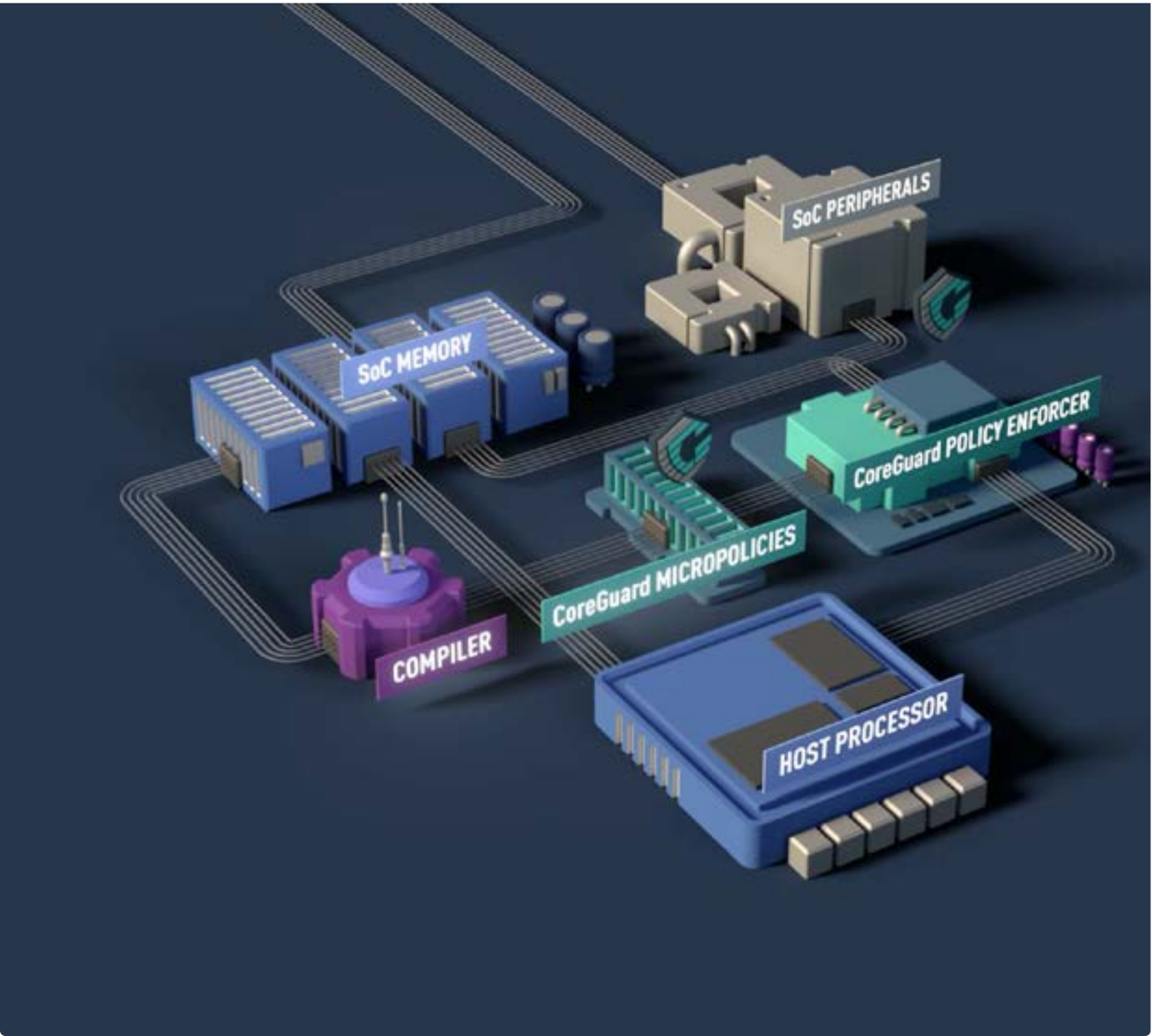
Micropolicies: CoreGuard micropolicies define the security guarantees provided by CoreGuard for a particular system. Different sets of micropolicies can be packaged for specific use cases and needs. Micropolicies are *rules* that determine, for each instruction executed by the host processor, whether that instruction is allowed by the micropolicy. Micropolicy rules are defined in a domain-specific language designed to support analysis and verification. Most implemented micropolicies are a dozen or fewer rules.

Metadata: Metadata makes processors smart about security. They inform micropolicies about important facets of the running application so they can make decisions about the validity of each instruction the host processor is attempting to execute. For any running software, there is a lot of information, metadata, in the source code that is used by the compiler to make sure the software developer has done things properly. However, that information gets thrown away by the time the final binary code is loaded into memory for the processor to execute. CoreGuard's toolchain extracts this static information from the executable file created by the compiler and uses it to inform the appropriate micropolicy. Additionally a small amount of dynamic metadata is created at runtime where, again, information critical to micropolicies did exist but is not retained such that CoreGuard could utilize it. An important instance of this is the base and length of each

allocated memory buffer, placed in metadata for each word for an allocated buffer.

Hardware: CoreGuard's policy enforcer hardware ensures that micropolicies are enforced on each and every instruction executed by the host processor, and that the code and data used by the micropolicies is inaccessible from the OS and application software. CoreGuard policy enforcer hardware logic modifies an existing SoC design.

CoreGuard can be implemented into an FPGA-based system or be integrated with a processor in one of two ways. When the SoC is using an unmodifiable core (e.g. Arm), CoreGuard can reside outside of the host processor euphamistically referred to as the *bodyguard* configuration. In this case CoreGuard obtains instruction information from the host processor via an instruction trace signal. When the SoC utilizes a modifiable core such as some RISC-V situations, CoreGuard can be integrated directly into the host processor's pipeline to maximize efficiency and reduct the amount of required logic.



The Normal Operation of a Processor

In this section, we establish terminology and concepts necessary for understanding CoreGuard by describing the normal operation of conventional processors—with nothing specific to CoreGuard. In the next section, we describe how CoreGuard integrates with a conventional processor to protect that processor from cyberattacks.

Terminology

The *Host Processor*, as we call the primary computing element, runs the application software and *Host Operating System*. The host processor is presented with a stream of native instructions, and each instruction reads, operates on, and writes registers and *memory locations*. Some locations in memory correspond to incoming data from peripherals (such as USB, network, or disk), and other locations in memory correspond to outgoing data to those same peripherals.

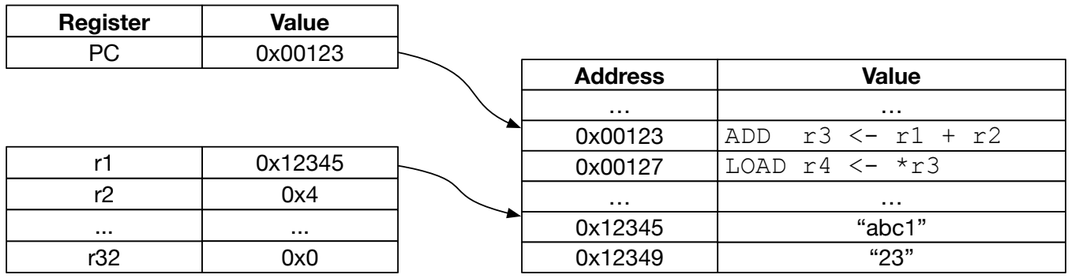
If we take a snapshot of the state of a computer (let's assume a 32-bit architecture) at a moment in time, we will see:

- The Program Counter register contains the 32-bit address of the word in memory that in turn contains the 32 bits representing the instruction currently being executed by the host CPU.
- The other registers contain *32-bit* values. On RISC-V 32-bit architectures, there are 32 general purpose registers containing intermediate values used during computation.
- The accessible *Random Access Memory* also contains 32-bit values. RAM is also known as the "heap." Addresses are locations in the heap, starting at 0 and going up to the size of the heap, counting in bytes. There are four 8-bit bytes in a native 32-bit word.

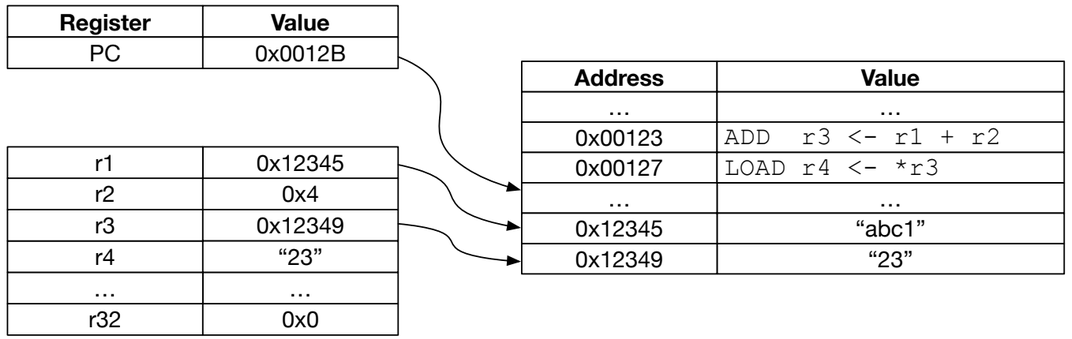
We use the term "32-bit values" above to highlight the fact that there is, in a conventional system, no way to tell whether a particular 32-bit value is the encoding of an unsigned integer, a pointer, an instruction, a floating point number, etc. A 32-bit value is just that: 32 1s and 0s. A given 32-bit value may simultaneously be a correct encoding of a native instruction, an encoding of an unsigned integer, an address of a byte in memory, a floating point number, four 8-bit ASCII characters, etc. There is no way that a conventional processor, when presented with a 32-bit value, can tell what type of value the 32 bits represent, and so the instructions presented by the (possibly exploited) application are executed blindly.

Per-Instruction Processing

The following diagram represents a snapshot of host memory state at a moment in time. We interpret the 32-bit values as if they represent instructions and sequences of characters:



The PC register points to address 0x00123, which, if interpreted as an instruction, adds the contents of registers r1 and r2 and puts the sum into register r3. The next 32-bit word, if interpreted as an instruction, uses the value in r3 as an address and copies the value found at that location to register r4. The following diagram represents the updated state after the execution of the ADD and the LOAD instruction:



We can see that the PC has advanced by two 32-bit (4 byte) addresses. Register r3 points to address 0x12349, which is the result of adding the values in registers r1 and r2. Finally, the value stored at address 0x12349, which represents the 2-character string “23,” has been copied into register r4.

The Operation of a Processor with CoreGuard

Now that we have reviewed normal processing by a host processor, let’s look at how CoreGuard integrates with a conventional system to turn normal processing into secure processing.

CoreGuard micropolicies maintain metadata (that is, data about data) describing each 32-bit word in accessible RAM and the registers. Exactly what metadata is maintained depends on which micropolicies are installed.

The following table lists examples of the metadata maintained by some of the standard CoreGuard micropolicies. It is only a small sample of the rich variety of metadata that micropolicies can maintain about words in memory, and then use to enforce security micropolicies at runtime.

WITH THIS POLICY ...	THE METADATA DESCRIBES ...
Control Flow Integrity	Is the word to which the PC points an instruction? If so, is that instruction the legal target of branches or jumps? If so, which branch or jump instructions can jump here? Maintaining this metadata allows CoreGuard to detect control flow hijacking attacks, also known as “code reuse” attacks—like ROP.
Information Flow Integrity	Did the value in a register used by the current instruction come from a trusted input port? Has the value been combined with any untrusted values? When an application routine performs actions that have an effect on the world, we want to make sure that the data used to perform the action is trusted.
Memory Safety	Is this instruction a LOAD or STORE instruction? If so, is the value in the register being used for the address of a pointer? What region in memory does that pointer have access to? Does the word in memory pointed to by the address register reside in the same region that the pointer is allowed to reference? Keeping track of this metadata allows CoreGuard to detect memory safety violations, which account for a majority of current cyberattacks.
Stack Safety	Is the word in memory that is being referenced a return address? Is the instruction currently executing labeled as part of the privileged function epilogue that is permitted to access the return address? Protecting values on the stack, such as return addresses, enables CoreGuard to defeat many control flow hijacking attacks.

Recall the example machine state described in the previous section, in which the PC points to an ADD instruction followed by a LOAD instruction, and the address in register r1 points to a sequence of characters abc123.

The diagram below is a notional mapping of host registers and memory addresses to records of metadata maintained by micropolicies:

Register	Metadata
PC	just jumped from address 0x00234
r1	value is a pointer into region XYZ
r2	value came from a trusted input port
…	…
r32	

Address	Metadata
…	…
0x00123	value is an instruction. Allowed come-from addresses = {0x234, 0x345}
0x00127	value is an instruction.
…	…
0x12345	value is confidential. location within allocation region XYZ
0x12349	value is confidential. location within allocation region XYZ

A control flow integrity micropolicy would check that the metadata associated with address 0x123 allows the control transfer recorded in the metadata for the PC register.

A memory safety micropolicy would propagate the "value is a pointer into region XYZ" metadata through the ADD instruction, and would then check that the word being referenced (in this example, 0x12349) is in the same region of memory from which the pointer is allowed to read.

If any of the checks described above fail (for example, the pointer in r4 does not point within region XYZ), a micropolicy violation is raised, and CoreGuard blocks the instruction from executing.

The CoreGuard 2.0 Architecture

CoreGuard 1.0 used a rule-cache and this made CoreGuard 1.0's silicon area 87% bigger than it is now in CoreGuard 2.0. Here is briefly why. In CoreGuard 1.0, when a set of micropolicies was compiled to run on CoreGuard hardware, the Policy Compiler took all the rules defined in the Dover Policy Language (DPL) code and expanded them into a rule processing data format referred to as "concrete rules." Concrete rules define the various metadata tag combinations (both the input and output values) for allowed instructions—that is, for instructions that comply with the set of CoreGuard micropolicies and will be given the green light to execute.

CoreGuard 1.0 used a rule cache to store these concrete rules in memory so that they were readily available when CoreGuard needed them. When CoreGuard Policy Accelerator hardware processed an instruction during runtime, its Tag Processing Unit (TPU) looked to see if the instruction's input tag combination existed in the rule cache. If it found a match with a concrete rule entry, then it was a "rule cache hit" and there was no policy violation; the behavior was allowed and CoreGuard updated metadata tags as necessary using the output values. If the combination did not match any entries in the rule cache, it was a "rule cache miss," and the Policy Accelerator had to call the Policy Executor ("the PEX") to run software ("the PEX kernel") that determined whether the instruction is allowed or should trigger a policy violation. Just to be clear, the PEX was a real processor in its own right, usually a smallish RISC-V. And this was not the biggest issue for CoreGuard 1.0's area.

The rule cache used a fixed chunk of memory that was often larger than it needed to be because it was difficult to "right size" the cache to accommodate a set of micropolicies that may not yet have been determined during implementation, or may grow down the road if a customer chose to add a micropolicy to their mix. Simply put, the rule cache was an area hog.

The replacement for the rule cache and PEX is called Policy Check Coding (PCC). Two key insights led to PCC:

- **INSIGHT 1**: If we can know the encoding of ALL metadata inputs and outputs when we compile a set of micropolicies, then we can precalculate what CoreGuard hardware needs to do with any instruction it encounters. In other words, if

we figure out all the answers to the test during compile time rather than runtime, we have an alternative solution to the problematic rule cache.

- **INSIGHT 2**: Since CoreGuard has all the answers ahead of time, policy checking becomes an encoding/decoding problem rather than an information retrieval (aka rule cache lookup) problem. Instead of looking something up in a rule cache, CoreGuard hardware can perform a simple calculation to determine whether an instruction is allowed. Instead of the huge rule cache, there is now a decoder that is small and fast and lives in hardware. The complex encoding process is left to the Policy Compiler, which is done offline at application compile and built time and doesn't impact the user's experience. Essentially, we've replaced the rule cache with math. Additionally, with no rule processing during runtime, there is no need for a PEX.

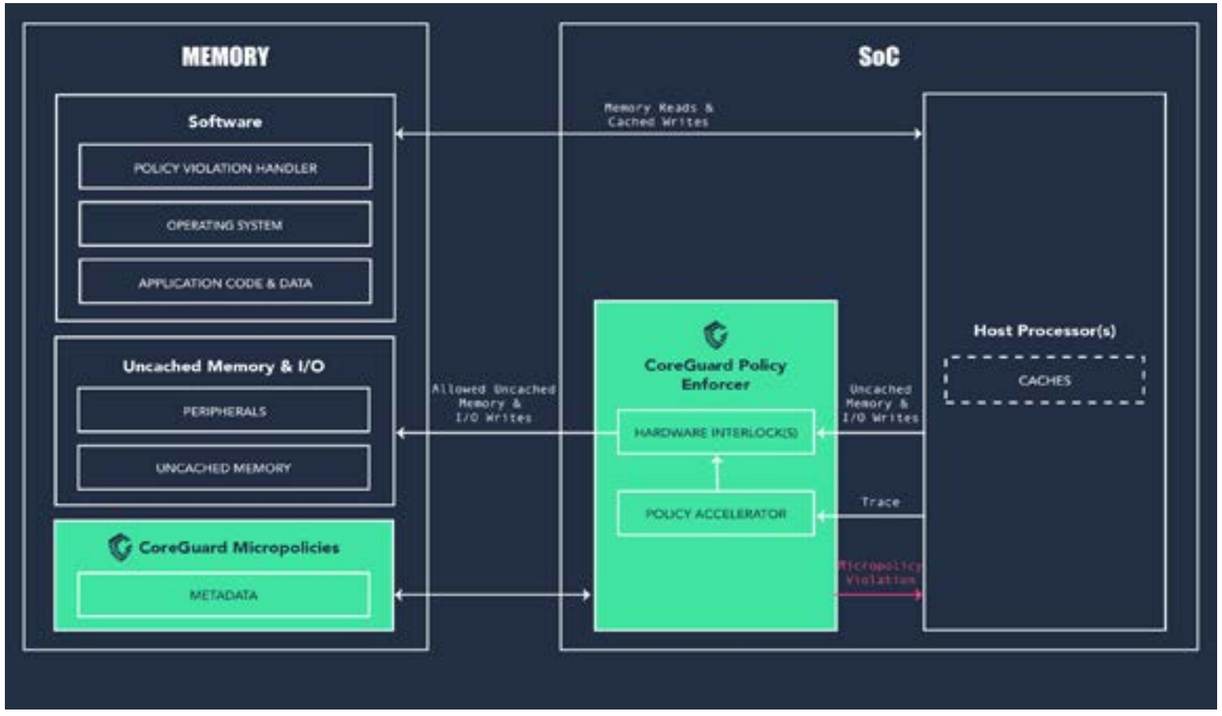
To apply these insights, PCC first tackles the challenge of representing concrete rules in a whole new way (i.e., the first insight). At a high level:

- With CoreGuard 1.0, the Policy Compiler encoded each metadata tag so that it could write out a list of what's allowed (allow rules). While each encoded tag may have been unique, this approach required lots of duplication of encodings in the generated list of concrete rules.

- With CoreGuard 2.0, the Policy Compiler encodes each metadata tag with the micropolicy rules (both allow and fail rules) essentially "baked right in." This encoding gives CoreGuard hardware all the information it needs during runtime to easily and quickly distinguish valid combinations of metadata inputs from a much larger universe of all possible inputs.

The important point is that the Policy Compiler does this heavy lifting so that CoreGuard hardware won't have to. Which brings us to how PCC applies the second insight.

Instead of a rule cache and PEX, CoreGuard 2.0 Policy Accelerator hardware includes a small micropolicy decoder component inside its TPU. During runtime, this component takes the metadata associated with the current instruction and performs a simple calculation on the decoded values of those inputs to determine if the instruction is allowed. There is no looking for a match in a big rule cache. There is no need for additional rule processing. It is a simple calculation via a hash function, where the inputs of every allowed instruction equal one of very few answers. In coding theory, a correct answer is referred to as a "syndrome."



CoreGuard Per-instruction Processing

When the application is running, the CoreGuard Policy Accelerator hardware receives information about the current instruction (via the instruction trace) and looks at the metadata input tags associated with that instruction. (Back in the CoreGuard 1.0 days, this is when the Policy Accelerator would look for a matching combination in the rule cache.)

With CoreGuard 2.0, the Policy Accelerator's TPU instead decodes the numerical encoding for each metadata input and calculates an answer called the "PCC Allow Hash." If the answer is equal to a syndrome, the instruction can proceed. Any answer other than one of the syndromes will trigger a policy violation (the red return arrow in the diagram above notifying the host processor).

When an instruction is allowed, CoreGuard hardware then uses the same set of numerical encodings to do a different type of calculation that determines how to update metadata tag values called the "PCC Results Hash."

So in summary, CoreGuard 2.0, with the new PCC architecture, can validate any instruction (and then update metadata as required) with no need for time-consuming lookups or rule processing during runtime.

A Secure System Needs a Secure Processor

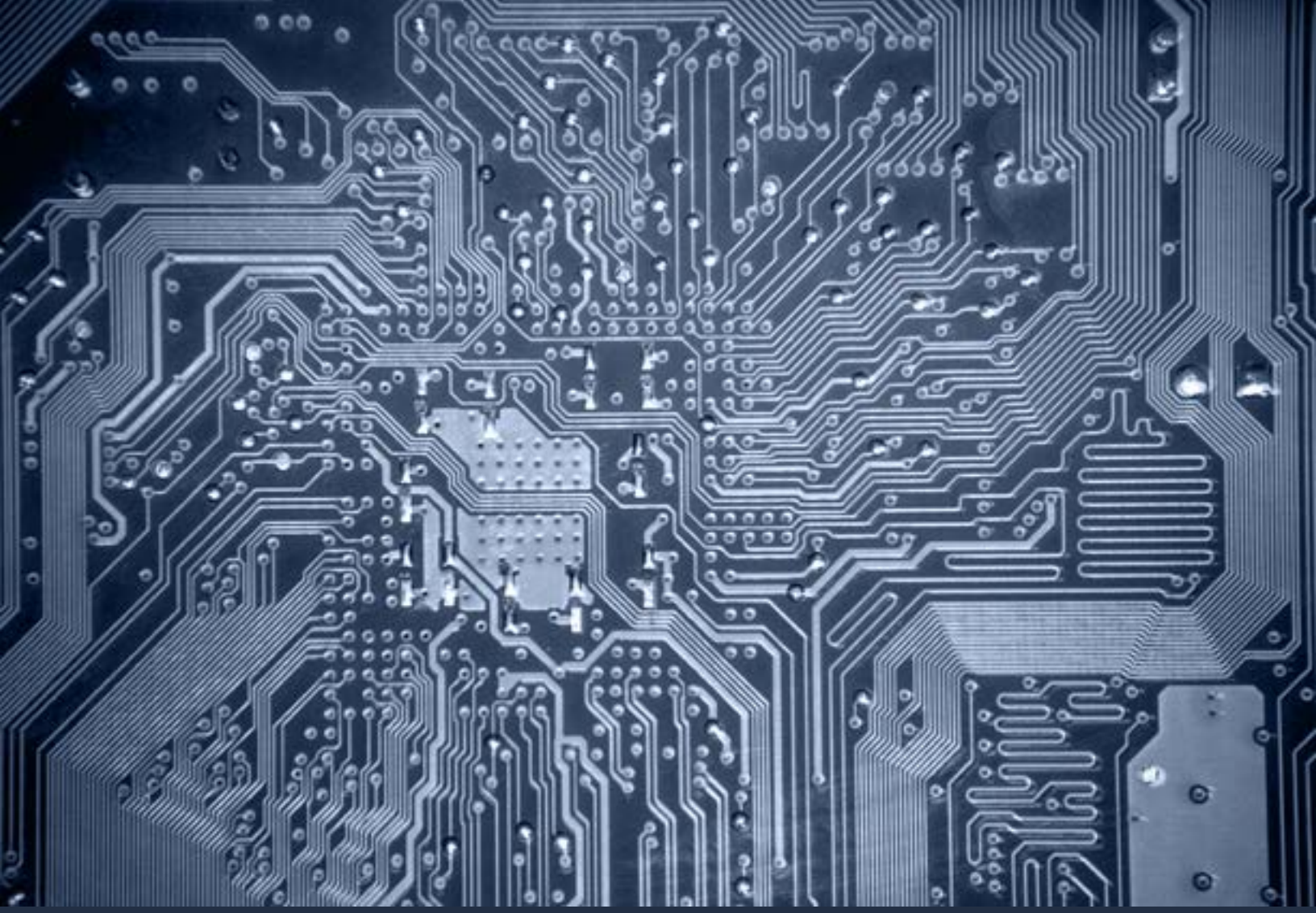
In conclusion, let's think again about a system's attack surface—the sum of the different points where an attacker can try to break in. The cybersecurity industry's focus has been on building defensive software around our networks and applications, but because software contains the vulnerabilities that are exploited by attackers, these software-only approaches have actually been making the attack surface of our embedded systems larger rather than smaller. Coupled with this grim reality is the fact that processors are not equipped to do anything about it. In our walkthrough of the operation of a conventional processor, you saw that processors blindly execute whatever instructions they are presented with, even if those instructions were exploited; they don't know the difference between good instructions and bad, and they can't enforce what they don't know.

These insights—vulnerable software and undiscerning processors—were the impetus of CoreGuard. With its architecture, CoreGuard integrates with conventional processors, on the same chip, to check every instruction for compliance with a set of micropolicies. CoreGuard is designed so that micropolicies are inaccessible from the system's OS and application software; only CoreGuard hardware can access and run CoreGuard micropolicies. Unlike software-only security products that look for known vulnerabilities, micropolicies can detect and block entire classes of cyberattack giving CoreGuard the unique position of stopping an industry-bease 95% of all cyberattacks (CVEs) that MITRE tracks.

This means that CoreGuard can even stop zero-day attacks that exploit software vulnerabilities unknown to the vendor.

As the number of IoT devices and other network-connected embedded systems grows, so does the threat of cyberattacks. We need to disrupt this cycle. CoreGuard does that with silicon security that makes today's processors—and the systems they are embedded in— immune to the cyberattacks of tomorrow.

CoreGuard can even stop zero-day attacks that exploit software vulnerabilities unknown to the vendor.



Learn More: info@dovermicrosystems.com

DOVER MICROSYSTEMS

About Dover Microsystems

Dover's lineage began in 2010 as the largest performer on the DARPA CRASH program. In 2015, Dover began incubation inside Draper Laboratory before spinning out in 2017.

Based in Greater Boston, Dover is the first company to bring real security, safety, and privacy enforcement to silicon. Dover's patented CoreGuard solution integrates with RISC processors to protect against cyberattacks, flawed software, and safety violations.

www.dovermicrosystems.com