US 20080126766A1

(19) United States

(12) Patent Application Publication

(10) Pub. No.: US 2008/0126766 A1

(43) Pub. Date: May 29, 2008

Chheda et al.

(54) SECURING MICROPROCESSORS AGAINST INFORMATION LEAKAGE AND PHYSICAL TAMPERING

(76) Inventors: **Saurabh Chheda**, Amherst, MA (US); **Kristopher Carver**, Chicopee, MA (US); **Csaba Andras Moritz**, Amherst, MA (US)

Correspondence Address:
FISH & RICHARDSON PC
P.O. BOX 1022
MINNEAPOLIS, MN 55440-1022

(21) Appl. No.: **11/981,178**

(22) Filed: **Oct. 31, 2007**

Related U.S. Application Data

(60) Provisional application No. 60/856,593, filed on Nov. 3, 2006.

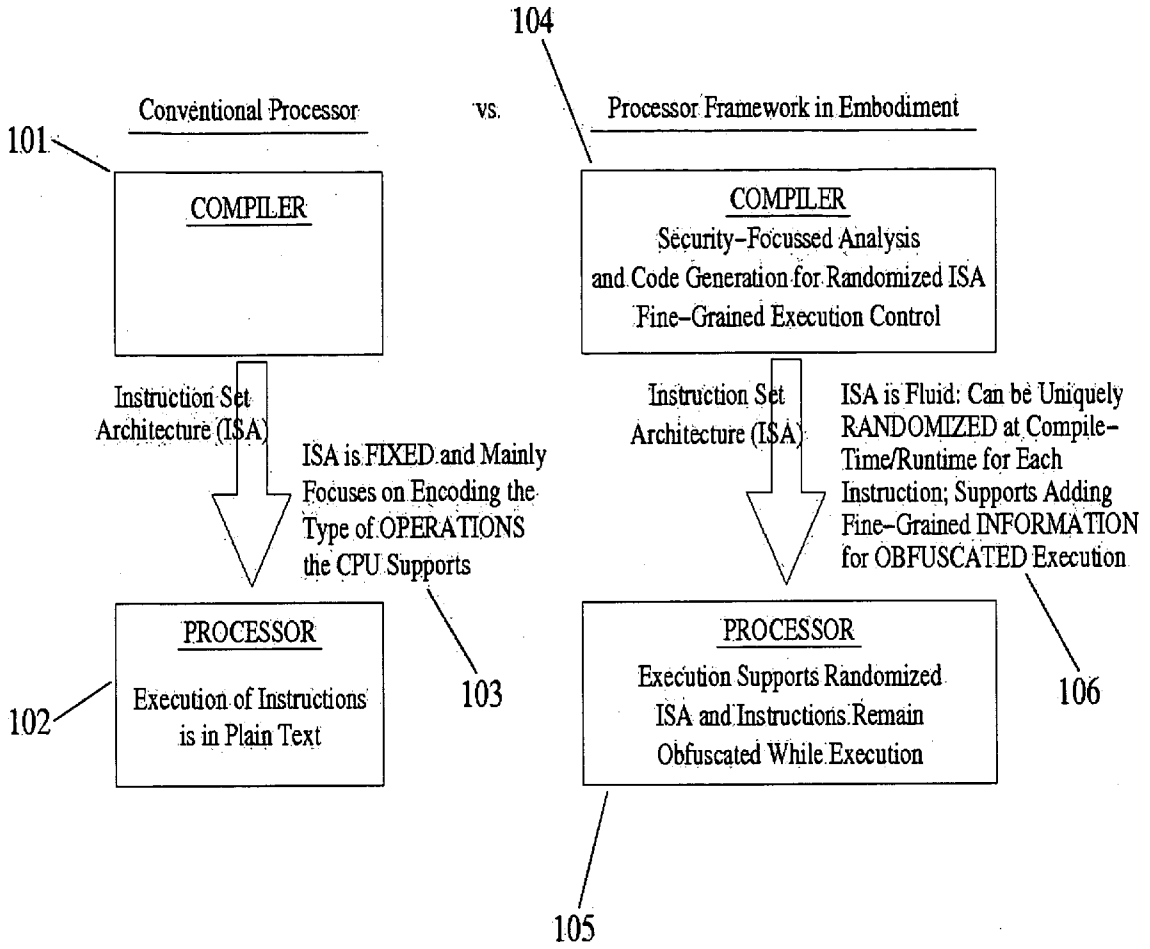
Publication Classification

(51) **Int. Cl.**
G06F 9/30 (2006.01)

(52) **U.S. Cl.** **712/226**; 712/E09.032

(57) **ABSTRACT**

A processor system comprising: performing a compilation process on a computer program; encoding an instruction with a selected encoding; encoding the security mutation information in an instruction set architecture of a processor; and executing a compiled computer program in the processor using an added mutation instruction, wherein executing comprises executing a mutation instruction to enable decoding another instruction. A processor system with a random instruction encoding and randomized execution, providing effective defense against offline and runtime security attacks including software and hardware reverse engineering, invasive microprobing, fault injection, and high-order differential and electromagnetic power analysis.



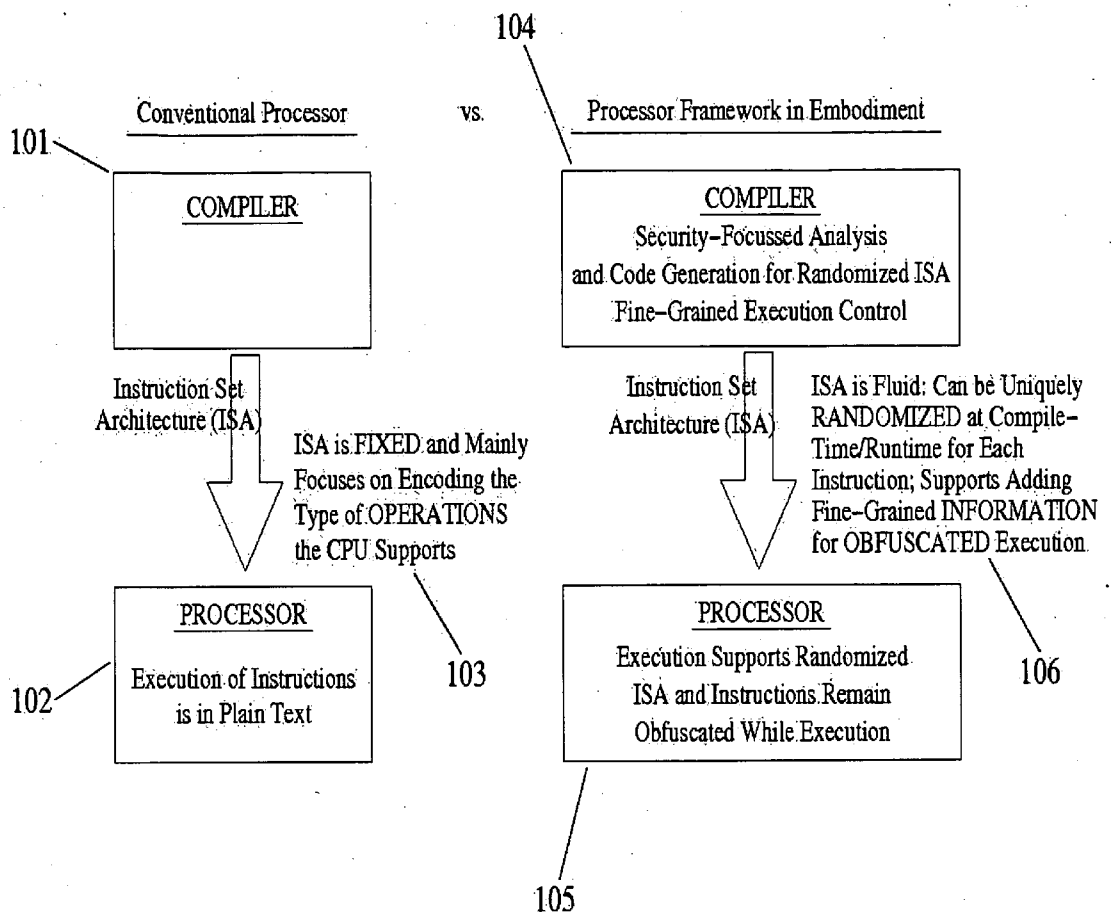

104
Conventional Processor
vs.
Processor Framework in Embodiment
101
COMPILER
COMPILER
Security-Focussed Analysis
and Code Generation for Randomized ISA
Fine-Grained Execution Control
Instruction Set
Architecture (ISA)
ISA is FIXED and Mainly
Focuses on Encoding the
Type of OPERATIONS
the CPU Supports
Instruction Set
Architecture (ISA)
ISA is Fluid: Can be Uniquely
RANDOMIZED at Compile-
Time/Runtime for Each
Instruction; Supports Adding
Fine-Grained INFORMATION
for OBFUSCATED Execution
PROCESSOR
Execution of Instructions
is in Plain Text
PROCESSOR
Execution Supports Randomized
ISA and Instructions Remain
Obfuscated While Execution
102
103
106
105

FIG. 1

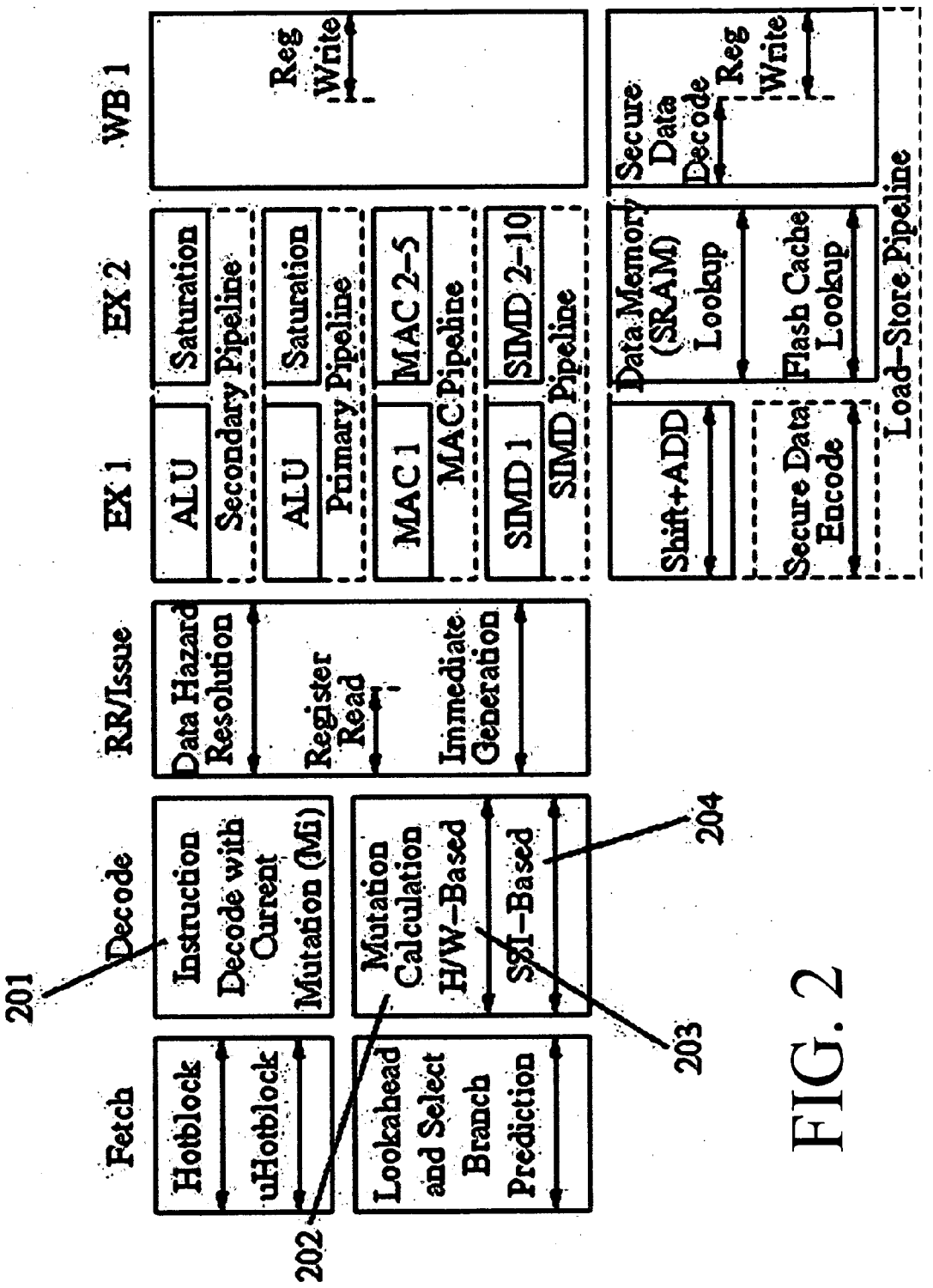

Fetch
Decode
RR/Issue
EX 1
EX 2
WB 1
Hotblock
uHotblock
Instruction Decode with Current Mutation (Mi)
Data Hazard Resolution
Register Read
ALU
Secondary Pipeline
Saturation
ALU
Primary Pipeline
Saturation
MAC 1
MAC Pipeline
MAC 2–5
SIMD 1
SIMD Pipeline
SIMD 2–10
Reg Write
Lookahead and Select Branch Prediction
Mutation Calculation H/W–Based
S&I–Based
Immediate Generation
Shift+ADD
Data Memory (SRAM) Lookup
Secure Data Encode
Flash Cache Lookup
Secure Data Decode
Reg Write
Load–Store Pipeline
201
202
203
204

FIG. 2

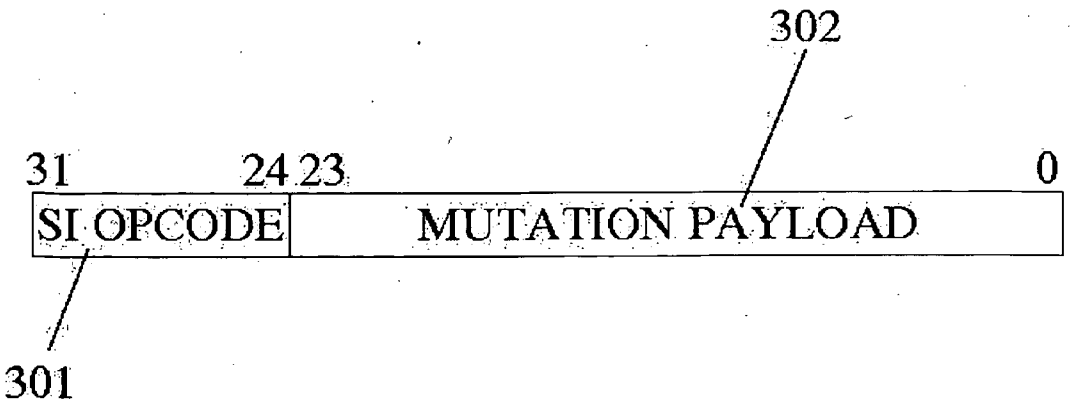


FIG. 3

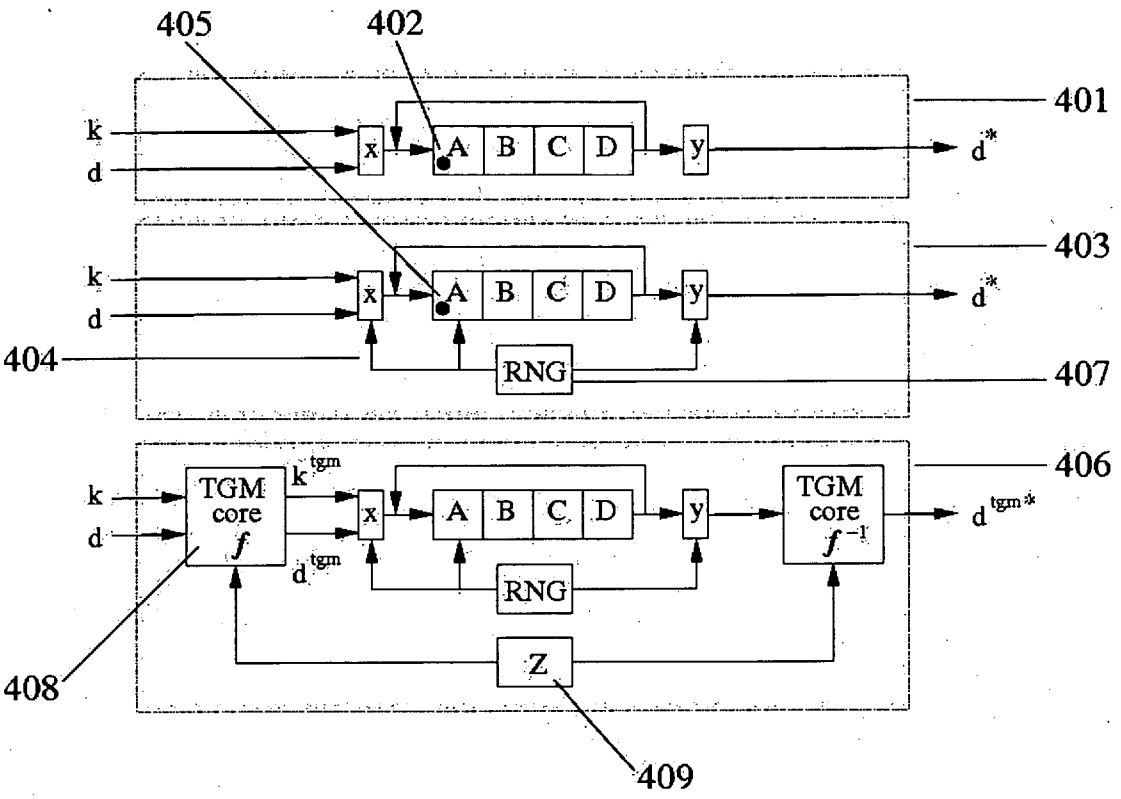

405
402
401
k
d
x
A
B
C
D
y
d*
403
404
RNG
407
406
TGM core f
k tgm
d tgm
TGM core f -1
d tgm*
408
Z
409

FIG. 4

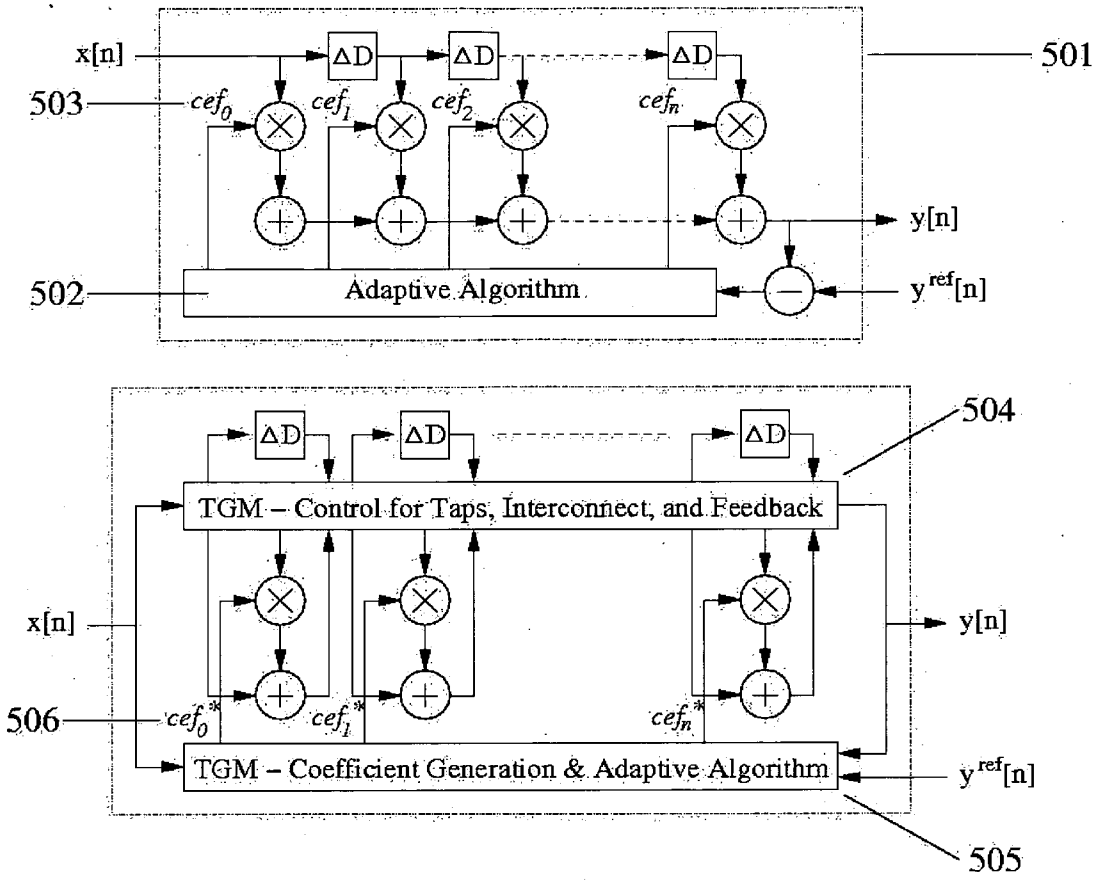

x[n]
ΔD
ΔD
ΔD
501
503
cef_0
cef_1
cef_2
cef_n
y[n]
502
Adaptive Algorithm
$y^{ref}[n]$
504
TGM – Control for Taps, Interconnect, and Feedback
x[n]
y[n]
506
cef_0^*
cef_1^*
cef_n^*
TGM – Coefficient Generation & Adaptive Algorithm
$y^{ref}[n]$
505

FIG. 5

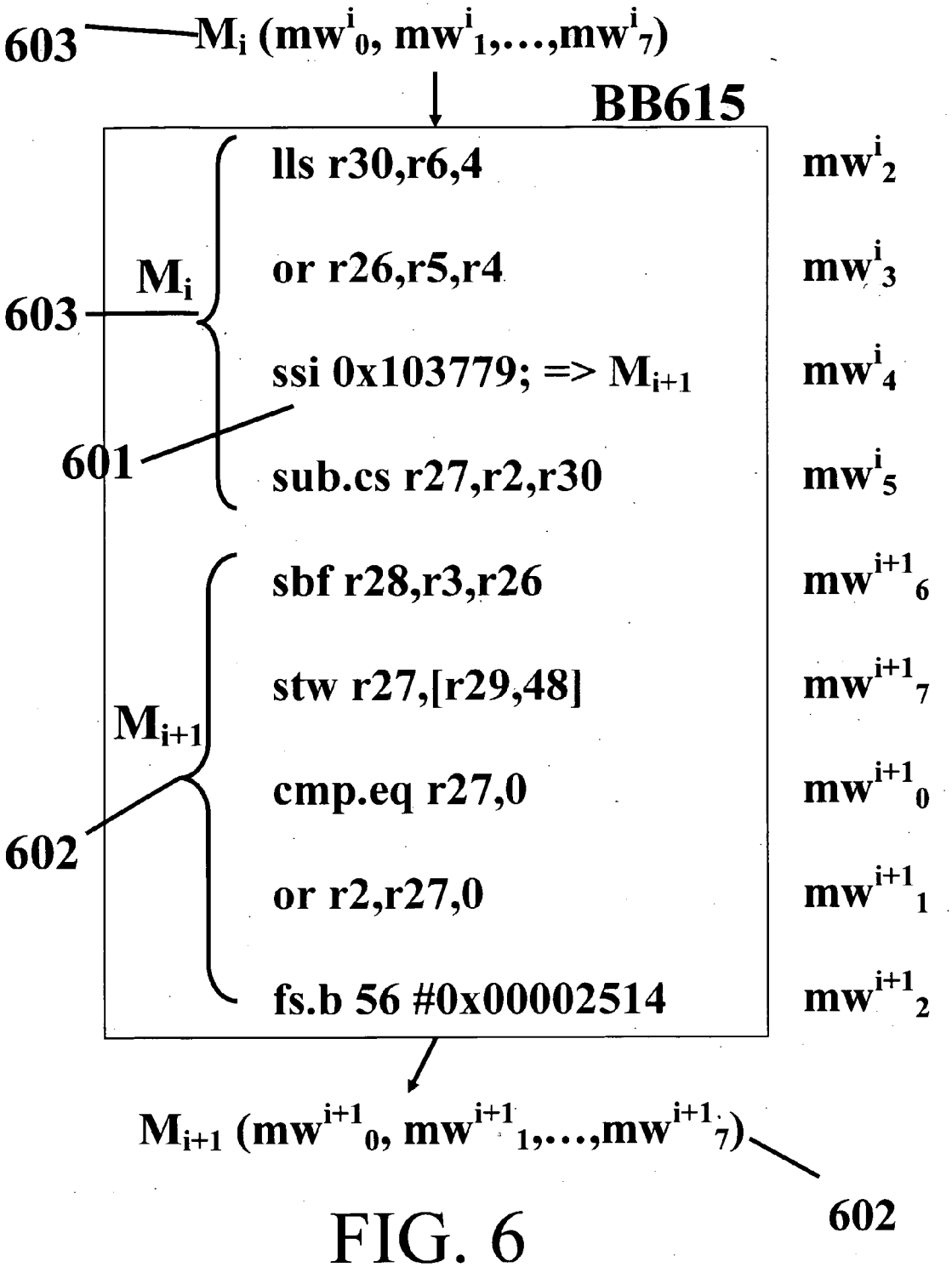

603
M_i (mw^i_0, mw^i_1,...,mw^i_7)
BB615
lls r30,r6,4
mw^i_2
M_i
603
or r26,r5,r4
mw^i_3
ssi 0x103779; => M_{i+1}
mw^i_4
601
sub.cs r27,r2,r30
mw^i_5
sbf r28,r3,r26
mw^{i+1}_6
stw r27,[r29,48]
mw^{i+1}_7
M_{i+1}
cmp.eq r27,0
mw^{i+1}_0
602
or r2,r27,0
mw^{i+1}_1
fs.b 56 #0x00002514
mw^{i+1}_2
M_{i+1} (mw^{i+1}_0, mw^{i+1}_1,...,mw^{i+1}_7)
602

FIG. 6

SECURING MICROPROCESSORS AGAINST INFORMATION LEAKAGE AND PHYSICAL TAMPERING

RELATED U.S. APPLICATION DATA

[0001] This application claims the benefits of U.S. Provisional Application No. 60/856,593, filed on Nov. 3, 2006, and Confirmation No 1421, entitled: SAFE ZONES: SECURING A PROCESSOR AGAINST INFORMATION LEAKAGE AND PHYSICAL TAMPERING, the contents of which are hereby incorporated by reference into this application as if set forth herein in full.

TECHNICAL FIELD

[0002] This invention relates generally to providing effective defense against information leakage and tampering in a microprocessor or a system where such a secured microprocessor would be incorporated. More particularly, it relates to a processor framework and methods supporting an execution based on chained sequences of small obfuscated codes called safe zones and associated randomized execution. It relates to mechanisms to make encoding of instructions in each safe zone random and unique for each chip, or compilation, and to ensure that breaking into a safe zone's encoding does not compromise another safe zone's security or does not allow leaking information from the processor outside that safe zone. The invention provides effective mechanisms across compiler, instruction set architecture, and micro-architecture layers to defend against offline and runtime security attacks including software and hardware reverse engineering, invasive microprobing, fault injection, and high-order differential and electromagnetic power analysis. The invention provides the security benefits without significantly impacting performance, power consumption, or energy efficiency during execution.

[0003] Furthermore, systems that incorporate a microprocessor with above technology can rely on the trust and security provided inside the processor to defend against different kinds of information leakage and tampering attacks including both invasive and non-invasive methods. Additionally, systems that in addition incorporate microprocessors with lesser security that would run applications, could be still effectively defended with the addition of a security microprocessor designed with the proposed invention.

BACKGROUND

[0004] Processing devices are vulnerable to security attacks including software attacks, invasive attacks by removing layers of packaging and different types of non-invasive attacks like fault injection and power analysis, etc. Attacks are also often categorized as in-wire when an attack does not require physical presence of an attacker. An example of such an attack is through the internet or other connection to another system. Non in-wire attackers would need typically to have access to the system.

[0005] This section mainly focuses on attacks that require considerable resources or Class III such as funded organizations with unlimited resources. Other lesser sophisticated attacks are similarly defended. A list of some of the available defense mechanisms is also described after the attack scenarios.

[0006] Attack categories: There are several sophisticated attack strategies reported. First, there are non-invasive side-channel attacks based on differential power analysis, electromagnetic analysis, and fault injection. Attacks based on power and electromagnetic analysis utilize the fact that encryption devices leak key data electromagnetically, whether by variation in power consumption or electromagnetic radiation. Differential power analysis (DPA) is very effective against cryptographic designs and password verification techniques. Electromagnetic analysis allows more focused observation of specific parts of a chip. Fault injection attacks typically require precise knowledge of the time instances when faults are injected and aim, e.g., at modifying memory bits to allow extraction of side-channel information. There are several reported successful side-channel attacks, e.g., recovery of password in Freescale MC908AZ60A, AES ASIC implementations, and smart cards.

[0007] Another attack category is based on invasive methods. Chips can be decapsulated front-side and/or rear-side manually using nitric acid and Acetone, or automatically using concentrated HNO3 and H2SO4. The more advanced approaches for reverse engineering have the capability to gather information about deep-submicron designs using Optical Imaging (OI), or Scanning Electron Microscopy (SEM). SEM yields higher-precision reverse engineering, often with sufficient detail for building gate-level models enabling VHDL simulation. SEM-based Voltage Contrast Microscopy is used to read memory cells.

[0008] Some attacks are based on recovering data from erased locations (e.g., caused by tamper-detection related zeroization logic) in SRAM and non-volatile memory due to data remanence—see successful attack on PIC16F84A. Other attacks are semi-invasive, e.g., UV or X-rays based, and can be completed without requiring removal of passivation layers.

[0009] Microprobing attacks would rely on removing the polymer layer from a chip surface, local removing of passivation layers, cutting through metal layers and using Focus Ion Beam (FIB) probes. FIB allows 10-nm precision to create probing points and/or restore security fuses. There are several companies specializing in chip reverse engineering, e.g., Chipworks and Semiconductor Insights at the time of submission of this patent.

[0010] Because microprocessors are vulnerable they cannot provide defense against sophisticated attackers. When added to systems such as an embedded device, mobile phone, or personal computer, the whole system's security is affected by the lack of a trusted component. In such systems an attacker has several ways to attack including by modifying and tampering with the software, attacking in memory, attacking the operating system, or physically attacking the processor itself. Existing solutions are not adequate whenever high security is necessary. This includes application such as premium content security, access to enterprise resources, devices used in power plats, defense systems, government systems etc.

[0011] Defenses: State-of-the-art approaches offer limited defense against Class III attacks. Partial defense is provided by techniques including tamper detection with top metal layer sensors, operating voltage as well as temperature sensors, highly doped silicon substrate to defend against a rear-side attack, sophisticated security fuses including those in memory arrays, zeroization logic of security-sensitive state in case of tamper-detection, encryption of memory content with cryptographic accelerators, encryption of buses (typically with simple techniques to not affect latency), VTROM used

instead of Mask ROM and Flash memory for non-volatile memory (not visible with static reverse engineering), and various defenses against memory remanence. There has been significant work on securing cryptographic implementations and software protection. These techniques are often software based an vulnerable to even simple attacks based on reverse engineering and running through debuggers. When they are microprocessor-assisted, they are vulnerable as microprocessors today to dot protect against sophisticated attackers.

[0012] Examples of micro-architectural techniques include memory architectures with protection like ARM Trust-Zone, randomized clock or various asynchronous designs, circuits based on process variation, etc.

[0013] The ever increasing sophistication of attacks implies that there is a considerable need to enhanced security during processing. Clearly, with a global trade of products and services it will be difficult to address security without establishing trust at the processing layer. No more can one rely on that just because a processing unit is completing a function in hardware it will be able to withstand attacks targeting extracting secret information, getting access to intellectual property, and gaining unauthorized access to system resources.

SUMMARY

[0014] The present invention addresses the foregoing need by providing methods and a processing framework creating an effective defense against the aforementioned security attacks at the digital level. As opposed to many defenses, the approach provides comprehensive security with very low cost and minimal power and performance overhead.

[0015] At the heart of the invention is a novel processor technology for obfuscated and randomized execution that is based on a security-focused compilation and code generation, associated instruction set architecture paradigm, and security-focused microarchitecture approach for allowing randomized and protected execution internally in the processor.

[0016] An aspect is the compiler-driven approach for instruction obfuscation and randomization, where the instruction encodings are randomized and tied together. The microarchitecture component of the invention supports this scrambled instruction execution wherein instructions that execute have their meaning decoded at runtime but remain in obfuscated format even internally in a processor. Another aspect is that this processor has its switching activity decorrelated from the operations it executes as the execution is itself random due to the mechanisms and random encoding.

[0017] Execution in conventional processors is based on a fixed encoding of all instructions. This allows for easy reverse engineering and makes them also vulnerable to a variety of side-channel attacks at runtime. By contrast, the invention proposed here is based on the fact that, with suitable support, the encoding of instructions can be changed at fine granularity and even randomized in chip-unique ways and execution kept obfuscated deep into the processor pipeline.

[0018] This has significant security benefits such as protecting against side-channel attacks like power and electromagnetic analysis, fault injection that would require precise knowledge of the time instances when faults are injected and data remanence attacks in RAM and non-volatile memory. Reverse engineering of the processor in this invention is not sufficient to reveal critical information due to the layered compiler-hardware approach and chip-unique obfuscated execution technology.

[0019] Furthermore, the approach hardens against microprobing attacks by establishing fine-grained secure instruction zones, as small as basic blocks: information extracted from a secure zone is not sufficient to compromise another zone. Instructions in each secure zone are uniquely and randomly encoded. Furthermore, execution can be rendered such that the lifetime of information used to decode an instruction in a secure zone is minimized to the very short durations necessary for the instruction's execution. As soon as decoding of an instruction is completed, the information required for decoding can be discarded.

[0020] The randomization of encoding and execution can be finalized at runtime to achieve a chip unique random execution. Attacking one chip would not help in extracting information that can be used in another chip.

[0021] These features provide considerable benefits in defending against sophisticated security attacks.

[0022] Unless otherwise defined, all technical and scientific terms used herein have the same meaning as commonly understood by one of ordinary skill in the art to which this invention belongs. Although methods and materials similar or equivalent to those described herein can be used in practice or in the testing of the present invention, suitable methods and materials are described below. In addition, the materials, methods, and examples are illustrative only and are not intended to be limiting.

[0023] Other features and advantages of the invention will become apparent from the following description, including the claims and drawings.

DRAWINGS

[0024] FIG. **1** is a block diagram comparing a conventional processor framework (left) with a processor framework relying on invention (right). An embodiment of such a processing device is described in embodiment 1.

[0025] FIG. **2** shows an example microprocessor pipeline diagram implementing embodiment 1.

[0026] FIG. **3** shows an example security mutation instruction encoding in the ISA.

[0027] FIG. **4** shows a block level diagram of protecting an AES cryptographic implementation with security approach (detailed in embodiment 2).

[0028] FIG. **5** shows how a block diagram of how a digital filter can be protected with security approach (detailed in embodiment 3).

[0029] FIG. **6** shows an example of applying mutation instruction in a basic block of a computer program consisting of instructions and how mutation is applied to each instruction. The figure shows how the information coming in can be used to decode the instruction at runtime. The information encoding allows using randomly selected encodings. In other embodiments, the approach can be used to convert from one fixed ISA to another ISA targeting a flexible hardware implementation as opposed to security.

DESCRIPTION

Embodiment 1

Security Microprocessor with Randomized Encoding and Execution

[0030] A security processor in this embodiment is based on a suite of innovative technologies across compiler, instruction set architecture, and micro-architecture layers (see FIG. **1** for

a comparison with a conventional processor). A key aspect is the compiler-driven approach **104** for instruction obfuscation, where instruction encodings **106** are randomized. The micro-architecture supports this scrambled instruction execution **105**.

[0031] Execution in conventional processors is based on a fixed encoding of all instructions **103** and a compiler **101** that focuses on generating the sequences of instructions for a computer program. This allows for easy reverse engineering, easily identifiable internal points for microprobing, and a variety of side-channel attacks at runtime like Differential Power Analysis (DPA) in the processor **102**. DPA is based on correlating the instructions with operations completed using power measurements and statistical analysis. By contrast, the processor embodiment described here is based on the fact that, with suitable support, the encoding of instructions can be changed at fine granularity and even randomized, and instructions can be executed in this format.

[0032] The basic idea of the encoding approach is to add security control instructions during compile-time code-generation; these control instructions embed guidance or hints related to how subsequent instructions should be decoded at runtime. The actual encoding of instructions can then be generated randomly: the instructions during execution would be still decodable with the help of the embedded hints in the control instructions. Of course the requirement is that the associated hints are available at runtime at the time a particular instruction is decoded. Each instruction in an executable can be encoded with an encoding scheme described or mutated by such a security control instruction. This is achieved by a security-focused code generation that can be completed at compile time or runtime.

[0033] The encoding of the control instructions themselves is similarly randomly generated and their decoding is completed with the help of other earlier control instructions. The embedded compile-time structures and built-in code-generation also support a final step of code-generation at runtime. A chip-unique encoding scheme can be created during the first power-on of the chip by randomly modifying the payload of the security/mutation instructions and rewriting the code based on the new mutations. This runtime step is enabled by symbolic information inserted into the binary by the compiler. The root of a runtime chip-unique modification can be based on a scheme leveraging a non-deterministic Random Number Generator and on-chip persistent memory cells. Other schemes can be based on codes derived with a die-specific deterministic circuit or the RTL state created by a randomly generated initialization sequence of instructions stored in persistent memory. This initialization sequence can be created at runtime inside a chip to make the sequence unique across chips.

[0034] Another aspect is that the code-generation in this embodiment introduces ambiguous control-flow between blocks fundamentally breaking up the code into secure zones: as each zone is uniquely obfuscated, compromising one zone would not make breaking into another zone simple.

Security Mutation Instructions and Secure Zones:

[0035] Before discussing the different types of mutations, FIG. **5** shows an example of using security mutations. In the figure, shown for a basic block **615**, there is an incoming instruction encoding template called M_i. This template is randomly generated and possibly mutated randomly prior to this basic block. All instructions following in the BB615 are using the template when they are decoded unless the template is changed in the block.

[0036] The M_i shown in the figure can be changed with inserted security mutation instructions ssi referred to with **501**. The region following the ssi instruction changes the encoding to M_{i+1} referred to as area **504**.

[0037] This way, instructions can be having an encoding that is randomly created and encoding is continuously mutated whenever ssi instructions are encountered. The code is generated and organized in such a way that decoding is made possible during execution. The mutation instructions, like ssi, are also randomly encoded. For example, ssi in the example is encoded with template M_i.

[0038] As shown, in addition to mutation instructions, other mutations based on the instruction address can be used and combined with mutations with instructions or otherwise. This allows a modification of an encoding on potentially every instruction.

[0039] There are three types of instruction mutations that occur in this embodiment. Implicit mutations are hardware-generated mutations that are expected but not explicit in the software. Example of usage includes the initialization phase of these processing cores. A second type of ISA mutation is through static security/mutation instructions based on immediates. This type is shown in FIG. **3**: opcode is **301** defines how the payload should be interpreted and payload **302** defines the mutation payload.

[0040] A third type of mutation instruction has a register-defined payload. These instructions can be used and inserted in a number of places in safe zones. When inserted at the top of the zone they modify the encoding of the following instructions of the zone but their encoding is happening with an incoming mutation defined in another safe zone. Mutations can also be added elsewhere as the only condition is that they must be available at the time a particular safe zone (they enable decode) is decoded at runtime.

[0041] There are two typical usage scenarios for the register-defined mutations: 1) a constant payload is moved to the register in a previous secure zone; or 2) the payload is made dependent on a memory-mapped location that could be either internally-generated or external to the processing core in the embodiment (memory-mapped IO).

[0042] These mutations allow implementing schemes where a mutation is tied to a different secure zone than where the mutation instruction resides or depends on outside events.

[0043] In addition to mutation instructions, the processing core in the embodiment also uses an address-based obfuscation scheme with rotating keys: this, in combination with the mutation instructions, creates a unique encoding for almost every instruction in a binary.

[0044] The mutation payload in an explicit mutation instruction is randomly generated at compile-time and/or runtime; instructions in the affected zone are transformed accordingly during compile-time and/or runtime.

[0045] A mutation instruction encodes a bit permutation such as an XOR operation and rotation of bits as defined by its payload. Because the bit permutations are simple operations, the decoding of instructions is done on-the-fly in the processor pipeline.

[0046] Each secure zone is based on a random ISA encoding and ends with an ambiguous branch. There is no correlation between the encodings used. Secure zones are linked together in a random order at compile-time, creating a fully

random layout. A binary in the embodiment is protected against differential binary analysis as every compilation would result in a different set of random mutations and layout.

[0047] Pipeline Design: A pipeline design is shown in FIG. **2**. The different types of mutations on instruction encodings are resolved in the decode stage **201** in hardware blocks **203** and **204**. **203** represents decoding due h/w based implicit mutations such as discussed above. The block **204** represents mutations due to the ssi security mutation instructions. Any given time there is a mutation Mi available to be used. This Mi can be changed in different ways as mentioned earlier as instructions are decoded and executed. The actual mutation operations are fine grained and therefore can be kept simple so the impact on the decode stage to set up control signals is minimized. This pipeline implementation is not intended to be limiting. Other pipeline implementations are possible including compiler-driven approaches as well as single and multiple issue designs based on speculative implementations with Reservation Stations, Reorder Buffer, Result Shift Registers, virtual registers, etc.

[0048] First Power On: During the first power-on, additional randomization of a software binary executing on the processor in the embodiment can be supported, making each binary chip-unique without requiring a separate compilation for each chip. During the first startup some or all of the mutation payloads and the rotating keys can be replaced with (runtime) chip-unique random numbers that are persistent across power-on cycles; instructions in the affected secure zones are rewritten at the same time. The compiler embeds enough symbolic information to make this step computationally efficient and straightforward at runtime. A chip-unique encoding is enabled with the help of die-specific circuitry such as based on process-variation. Another approach is based on encoding the die-specific access latency (similarly due to process-related variation) in SRAM arrays. Another alternative is to have a few persistent memory cells on the die, written once by the processing core's non-deterministic random number generator. At the end of the initial boot even the startup code can be modified such that its decoding is based on a chip-unique implicit mutation.

Protection Provided by the Processing Cores in the Embodiment

[0049] At the heart of the embodiment is a unique randomized encoding and execution approach: 1) these processing cores execute instructions whose encodings can be randomly generated; 2) instructions' encodings can be further randomized at runtime in a chip-unique manner; 3) associated code-generation creates secure zones—compromising one zone would not make breaking into another zone easy; 4) this processing core's execution and switching activity cannot be correlated with the operations it executes because its execution remains obfuscated deep into its pipeline; 5) several techniques across compiler-architecture layers are used to additionally mask the power profile of operations during execution in addition to the inherent masking due to obfuscated execution.

[0050] The randomization affects all state in the processor including buses, caches, branch address tables and branch target address caches (BTAC) and register files. In the case of BTACs its content is randomly kept with the same encoding as the branch instruction's encoding. That means that when the branch instruction is decoded, even BTAC information becomes accessible for the specific branch. Other branch targets in the BTAC would, however, be protected as they are encoded with another branch's encoding that is independent from the current encoding. In the case the register file what registers are used is randomly set up at the initialization time. Content can be similarly mutated. Instruction memory is automatically protected due to the obfuscated encoding. Additional techniques can be used to protect data memory. The compiler maps each temporary memory access statically to a consumer-producer group called a location set; these are extracted by the compiler and/or rely on additional user information. As both memory reads and writes belonging to a location set would use the same obfuscation, correctness of execution is maintained. At runtime, random keys are read in and masking happens in the software uniquely for each location set. The masking varies after each power on or reset. All persistent memory (on-chip as well as off-chip) can be encrypted with a DPA-resilient AES leveraging similarly the obfuscated execution. A protection example of an AES module is presented in a subsequent embodiment.

[0051] Protection Against Black-Box Reverse Engineering Attacks: A brute-force attack against the instruction obfuscation in this embodiment would consist of quickly running through all possible scrambling permutations and filtering out those which are obviously wrong. To give an approximate idea of breaking this encoding one would need to try **232** permutations (for a 32-bit ISA) for each instruction and try to combine variable length sequences of such instructions into valid instruction sequences. The processor ISA opcodes are mapped uniformly across operations making all bit permutations valid. Furthermore, it would be impossible to distinguish real security instructions from permutations of other ordinary instructions. It is easy to show that brute-force attacks against this scheme would be therefore too complex (from the point of view of computational and storage complexity) to be practical. The reason is that all possible bit patterns in the instruction set are legal and all possibilities would have to be considered. Note that the solution does not in fact require that all bit permutations are valid and another embodiment might choose to reserve instruction space for future extensions. The reason is that if an extremely high fraction of the possible bit patterns in the instruction set is legal, simply filtering out permutations that are syntactically incorrect would not greatly reduce the number of possibilities that would have to be considered. Moreover, in practice the length of a safe zone is not known so different lengths would need to be tried.

[0052] Protection against Side-Channel Attacks: DPA is based on statistically correlating differences in power profile across instruction sequences at key points. This embodiment, works by breaking up the correlation necessary for successful DPA attacks. By decoupling encoding from execution and combining it with other compiler-driven architecture techniques to randomize the power profile of operations—note that the control instructions are hidden by the obfuscated instruction encoding—the processing core can be protected against side-channel attacks like DPA.

[0053] Because the processing core's execution in the embodiment is kept obfuscated, the actual switching activity on internal buses, logic and memory structures cannot be correlated with the instructions. Moreover, the same type of instruction has many different encodings during execution so probing the system with different instructions would not work. The only activity that could provide a power signature of the operation is the switching activity in the Arithmetic

Logic Unit (ALU) stage. The embodiment has special techniques and ISA to defend against power-analysis based on ALU power traces. These techniques can be turned on in sections of code that are security-sensitive against DPA during the security focused compilation.

[0054] Examples of techniques in the embodiment for ALU masking are operation masking and phase masking.

[0055] 1) Operation Masking—It is known that the power consumption varies with each arithmetic and logic operation (for example, an AND will not consume the same power as an ADD operation). A variety of techniques are used to normalize/randomize the power profile, including: Randomly switching ON various arithmetic and logic units even when they are not used by the instruction being executed—the added additional power consumption helps mask the actual operations; Randomly switching input operands to arithmetic and logic units being used by the instruction being executed changes the power consumed by the operation by activating different transistor paths in the circuit. By doing this one can mask the actual input data values to each arithmetic and logic unit. Both this and the previous technique are fairly easy to support and do not affect performance;

[0056] Some operations, like multiplication, consume significantly more power than other operations, and it is important to mask these operations since attackers can use the power peaks created by these operations as a pivot to find patterns in the execution flow. Letting these units consume power throughout the execution in order to mask actual usage might not always be a good solution since the overall power consumption will increase significantly. The processing core in this embodiment employs a solution to mask the power consumption of these operations by randomly replacing these operations, at runtime, with SWIs (Software Interrupts).

[0057] These SWIs invoke performance-optimized code to perform requested operation in an alternate way.

[0058] Another technique is based on multiple path executions—these are equivalent implementations with different power profiles that are randomly selected during runtime.

[0059] Phase Masking is based on randomly inserting pipeline stalls during execution of security-sensitive codes the boundaries of these phases can be further masked.

[0060] Another side-channel attack described in the literature is based on injecting faults. Fault-injection attacks would be practically impossible as the encoding and execution of instructions is kept confidential: an attacker cannot find meaningful attack points to inject faults.

[0061] Protection against Advanced Micro-probing: A processing core in this embodiment has an effective protection against sophisticated micro-probing attacks such as those based on Focus Ion Beam (FIB). In this attack scenario, we assume that the attacker has the ability to understand the design after reverse engineering some of its circuits with Scanning Electron Microscopy (SEM)—note that the randomized execution makes it considerably harder even to find useful probing points compared to conventional designs.

[0062] Nevertheless, let us assume that an attacker would somehow find the encoding of an instruction I_k and also uncover the mutation used for the instruction, S_k, and has access to the binary. The embodiment would still limit the information this attacker can extract to a few instructions, typically less than the size of a basic block (or secure zone). If the attacker tries to reverse engineer instructions going backwards in the address space from I_k, it would after a few instructions enter another secure zone based on a different encoding not related to the current uncovered mutation S_k (because mutations are randomly picked for each secure zone). If the attacker were to try to go forward, he will always reach an ambiguous, e.g., register based, branch instruction at the end of the zone with a branch address that is defined in a previous secure zone and therefore protected.

[0063] The microarchitecture in the embodiment can also use static-instruction-based implicit branches that can be inserted in an earlier zone effectively replacing a conditional branch from the binary. Static instruction are control instructions containing control information of various sort. Implicit branching would mean that the control instruction would contain information for a branch at the end of the basic block often in addition to other information. This allows removing the actual branch instruction and completing the branch prediction ahead of time; encoding of the implicit branching can be made differ from the encoding of the safe zone where the branch it replaces normally resides. Secure zones end with an ambiguous unconditional branch with their target address defined in a different secure zone. This enables separation between the encoding used in zones and also creates a randomized layout. The performance overhead of the two branches per secure zone is mitigated by one of them often being an implicit branch, which is a zero-cycle branch in terms of execution because branch prediction is performed ahead of the control-flow it needs to encode.

[0064] The fact that application codes are based on secure zones increases the hurdles for an attacker because as many successful microprobings as secure zones would be required on many points to even have a chance to gain access to IP hidden in a processing core in this embodiment. The processing core in this embodiment has a number of techniques and a layered defense making this extremely difficult to attack.

[0065] First, each mutation has a very short lifetime of just a few cycles and is discarded after use (the next secure zone is at an unknown address that is ambiguous and will use a different random mutation key). This is not the case during instruction execution in a conventional processor where if the instructions are encrypted, the same key is used typically every time an instruction is decrypted.

[0066] Second, the very first mutation in this core is created at randomized times measured from reset—this is accomplished, e.g., by inserting random stalls during the initialization—and is implicit and chip-unique, re-generated at every power-on.

[0067] In addition, dynamic mutations (these are mutation instructions which are register-based with the register loaded from a memory-mapped IO location in a previous zone) can be correlated with either external or on-chip time-specific events—the attacker would need to capture those events and monitor many points simultaneously to have a chance to bypass the associated secure zones.

Protection against Reverse Engineering with RTL Simulation:

[0068] The attack in this scenario assumes accurate-enough extraction of the design such that an RTL-level simulation can be attempted where instructions can be executed and probed. The embodiment can protect against this attack similarly with a layered defense. First, a core in this embodiment requires comprehensive reverse engineering and additional factors would need to be true for an attacker to have a chance to succeed with simulation: conventional execution would not necessarily require a complete RTL model to simulate most

of the instructions—a core in this embodiment would require that because its decoding/ISA of instructions in some secure zones, including the initial one, is tied to a comprehensive RTL state derived from many areas of the design and state that would normally not be required for instruction execution. Secondly, these cores use die-specific (due to process variation) circuits like [41] and similar techniques to make some of the encoding sequence invisible with invasive imaging alone, such as Scanning Electron Microscopy (SEM). Additional protection is introduced by adding a small persistent on-chip memory with its content filled at first power-on with the help of a non-deterministic hardware RNG. An attacker would need to be able to bypass these with microprobing and complete microsurgery to read content by generating the addresses, in addition to also successfully reverse-engineering the entire chip. After reverse engineering, a memory model would need to be constructed at the RTL level to simulate execution of instructions. One key aspect is that even if there is only a small discrepancy in the created RTL for the processor in this embodiment, the instructions would likely not decode at all as decoding is tied to a fairly accurate RTL state across the whole chip. This means that if there is a tamper-protection mechanism in place that would prohibit a fully accurate reverse engineering (even a very small fraction of the die), the RTL simulation would likely not work despite the other micro-probing requirements for a successful attack being all met.

[0069] The embodiment has additional defense enabled by its dynamic mutation instructions at the boundary between certain secure zones. These mutations are fine-grained core-external or die-specific; they are equivalent to execution authorizations required to enter certain zones, i.e., by allowing correct instruction decoding in those zones. If this authorization is externally provided and in a time-specific manner (e.g., by another sub-system), the RTL simulation would fail as it is considerably slower than the silicon chip, and as a result, the decoding of the instructions executing on the core would fail.

[0070] An attacker cannot use multiple chips to complete an attack. This is because there is no secret shared across the chips. That means that every chip would need to be attacked separately and information gained from one chip would not help in attacking any other chip.

[0071] Protection against Cloning: Cloning attacks would require copying the design transistor-by-transistor and associated software bit-by-bit. By executing a uniquely generated code, of which decoding is tied to chip or die-unique aspects, effective defense against cloning can be provided. Even if a chip incorporating a processing core such as described above would be replicated exactly at the transistor level and a copy of the software binary is available, the software would not run on the new chip and the chip would not function.

Embodiment 2

Protecting Cryptographic Implementations Against High-Order Differential Power Analysis

[0072] An embodiment showing protecting a cryptographic implementation is shown below. As mentioned in the standard and noted in the Advanced Encryption Standard (AES) literature, AES is susceptible to differential power analysis (DPA) attacks.

[0073] The embodiment is based on a software-hardware approach; it is based on the microprocessor technology described earlier for randomization of execution and internal microprocessor switching activity. The objective is to provide high-order DPA protection with minimal area overhead and performance impact on AES.

[0074] AES is a round-based symmetric block cipher, working on 128 bit chunks of data. The AES algorithm is based on 4 different operations per round, as well as some pre- and post-processing. These operations are SubBytes, ShiftRows, MixColumn, and AddRoundKey. More details can be found in the standard outlining document.

[0075] One of the main concerns with the AES algorithm is its susceptibility to DPA attacks. Side-channel attacks, such as DPA, work due to the fact that correlation exists between physical measurements taken during execution and the internal state of the algorithm being executed.

[0076] In FIG. **4** a standard AES algorithm **401** is shown at the top. The microprocessor core with the techniques outlined in the patent, including randomization of encoding and execution, is referred to as TGM.

[0077] In the AES algorithm an attacker may target the time at which the input data and key are operated on for the first time (see highlighted point **402** in the figure). By monitoring the average power consumption at this point, a correlation can be made between the input data (known to the attacker) and the secret key, to eventually find the key. In order to combat this DPA attack, approaches based on masking the input data have been introduced. Data masking is used to remove the power-trace related correlation between the (known) input data and the data used in the algorithm with the key. Mask correction must be performed during the algorithm (as SBox lookups in the SubBytes stage are non-linear operations) to ensure that the masking will not affect the output cipher-text and that the cipher-text can still be decrypted with the same key. Although various approaches, based on either using separate SBox table(s) for each possible mask or by replacing the SBox lookup with logic to perform equivalent transformation, have been proposed and offer protection against first-order DPA, scaling such a solution to higher order DPA is extremely difficult.

[0078] See for example the middle implementation **403** in FIG. **4** that uses data masking: while it protects against first-order DPA it is vulnerable to second-order DPA at point **410**. In a second-order DPA attack, the attacker monitors the power profile when the mask is exclusive-or-ed with the (known) input data. Capturing traces for both this point and the point when the masked data is used with the key in stage A is sufficient for an attacker to correlate the mask, the input data, and the secret key bit by bit.

[0079] The proposed third AES implementation **406** shown in the bottom sub-figure in FIG. **4** is leveraging the strength of TGM security core **408** that is based on an embodiment of the randomized encoding and execution approach.

[0080] During AES encryption the TGM calculates a reversible function, f, in software that takes as inputs the key, the data to be encrypted and a chip-unique random number Z shown as **409** (persistent across power-on cycles). The TGM execution is resistant to high-order DPA as switching activity in TGM buses, memory, etc, is randomized by the random encoding and execution model and by operation masking techniques presented before. Due to the high-order DPA protection in TGM that de-correlates data d from dtgm and key k from ktgm (see the bottom part of FIG. **4**), the AES module is now protected against DPA.

[0081] The additional hardware masking is, in fact, not necessary, since the correlation between the original input data and the data worked on with the key has been removed in the TGM portion of the solution. The flow described above is for encryption; for decryption the initial TGM software layer would pass the data to block A and a TGM software layer will perform the inverse function of f on the data. Furthermore, any DPA would require running the AES in isolation or a modification of the code; however, as the TGM component of AES would not decode correctly without the execution of another secure zone before this code (which in turn requires another secure zone to be decoded and so on) and a modification of that code would essentially mean knowing all the decoding related mutations; a successful attack is therefore extremely unlikely. The performance impact of this scheme is minimal: the TGM-based functionality and the other stages of the AES can be pipelined. Assuming a 256-bit AES, with 16 rounds, the requirement for pipelining without penalty is that the TGM component is completed in less than 16 cycles, assuming each AES round takes one cycle without TGM.

Embodiment 3

Protecting Hardware Intellectual Property by Controlling with Security Processor

[0082] An example is provided in the context of digital filters. Other types of hardware modules could be addressed in a similar way.

[0083] At the heart of modern processing and communication systems are digital filters (DF) that compute a quantized time-domain representation of the convolution of analog signals in digitized form. DFs can be found in almost any military system from avionic to sonar sub-systems and applications such as image recognition and target tracking. The characteristics (i.e. transfer function, amplitude response, etc.) of a DF can leak information about the intended function of the signal processing system to which it belongs, during both the manufacturing and the deployment of the ASIC.

[0084] To protect a DF, the key characteristics must be protected: this includes its type (i.e., whether it is IIR or FIR), order of filter (number of previous inputs and/or outputs used to calculate current output), filter coefficients (weighting function of the filter), and algorithm used to adaptively change the coefficients at runtime—if the DF is adaptive.

[0085] FIG. **5** (top, **501**) shows typical implementations for an adaptive filtering algorithms. Filter coefficients **503** weigh the data shifted down the delay line and are responsible for, in conjunction with the number of taps (delays), the amplitude response of the filter. In a non-adaptive filter, the filter coefficients are generally pre-calculated and stored in non-volatile memory. In adaptive filters, an adaptive algorithm **502** computes these coefficients on the fly in response to changing input samples.

[0086] FIG. **5** (bottom figure) shows an example of how a DF can be protected with TGM. The task of selecting the coefficients in a non-adaptive DF, the algorithm to adaptively compute the coefficients **506** in an adaptive DF (shown as **505**), and controlling the order of the coefficients are moved to the TGM core (see **504**, **505**); these signals are memory-mapped and controlled by secured TGM instructions. To control the programming of the order, support masking, and provide the ability to change on the fly we assume the availability of redundant taps. By transferring key computational steps and the configuration of the DF design to the TGM core, we can harden it against both online and offline attacks.

[0087] In a typical ASIC implementation the interconnection between the adders, multipliers, and delay elements in a DF is predetermined and can be reverse engineered through Scanning Electron Microscopy (SEM).

[0088] When the filter is used with the TGM core (implementing randomized encoding and execution), the interconnection is programmed at start-up and can be changed at regular intervals when the filter is in use. This prevents attackers from knowing how the taps are interconnected with respect to the input, output, and from establishing an order for the filter coefficients. In addition, to thwart micro-probing attacks based on FIB probes, the TGM part could implement coefficient masking: e.g., it can mask the actual filter coefficients sent to the filter hardware (a few at a time depending on the number of redundant taps) with randomly generated mask values in the TGM core.

[0089] To correct the error added to the weighting function of the DF (before it affects the output), the TGM software compensates the weight by altering the coefficients in the redundant taps of the filter accordingly. Masking ensures that the filter coefficients, even for a non-adaptive filter, change constantly, making it extremely difficult for an attacker to figure out whether the filter is adaptive or not or find the coefficients.

[0090] Other masking schemes are possible, e.g., resembling time-hopping, if the component following the DF would be similarly controlled by the TGM. A TGM solution enables occasional integrity checking on the DF hardware: the transfer function of the DF would be run in the TGM in parallel with the DF and outputs checked for matching. As integrity checking can lag the rate at which the hardware components of the DF process the input, the checking mechanism is not on the critical path of the DF.

[0091] Overview TGM Core Microarchitecture used in this embodiment: A TGM core is a 32-bit compiler-driven single-issue (or dual-issue) processor that supports 8-16-32-64-bit operations, has cryptographic hardware acceleration, and sophisticated compiler-driven power management. TGM uses both a hardware-based non-deterministic random number generator (NDRNG) and a deterministic random number generator (DRNG) that is FIPS **140-2** compliant. It has a physically-mapped compiler-managed memory system. It incorporates additional techniques to protect its data memory. The compiler maps each temporary memory access statically to a consumer-producer group called a location set; these are extracted by the compiler and/or rely on additional user information. As both memory reads and writes belonging to a location set would use the same obfuscation, correctness of execution is maintained. At runtime, random keys are read in and masking happens in the software uniquely for each location set. The masking varies after each power on. All persistent memory (on-chip as well as off-chip) is encrypted with a DPA-resilient AES.

[0092] Interfacing with Protected Design: An ASIC with built-in TGM might use an interface between the TGM core and the functionality it protects. The TGM core contains a programmable interface which allows software executing on the TGM core to interact with and control hardware components. Since it is possible that the protected hardware components and the TGM core may be operating at different clock speeds, communication between the two will occur via a handshaking protocol. This interface can contain program-

mable IO lines (similar to GPIO) and a special interrupt port through which the ASIC will be able to interrupt the current task being performed on the TGM in order to initiate a higher priority task.

Embodiment 4

Protecting Software Intellectual Property with Add-on Security Processor in Conventional Systems

[0093] In this embodiment instructions on a second processor co-execute with instructions on the security processor. This security processor can be added on add-on card such as PCI, PCI-e, etc. The instructions executing on the security processor, such as TGM, could also be encrypted before sent for execution. By inserting an instruction of which encoding is randomly created, or encrypted, into the stream of instructions on a lesser security processor, such as with a fixed instruction set, the computer program running on a lesser security processor could be protected against reverse engineering and tampering attacks, also due to the voids created in the computer program now containing obfuscated codes executing on a security processor. The codes that execute on the security processor could be coupled with each other, forming a graph, for the purpose of protecting against replay attacks or removal attacks of some of the codes targeted to execute on the security processor.

Other Embodiments

[0094] The invention is not limited to the specific embodiments described herein. Other types of obfuscation or encryption can be used for instructions and data and combined with other techniques, in other embodiments. The invention can be used to implement other types of security services or functionality than described in the embodiments. Other embodiments not described herein are also within the scope of the following claims.

What is claimed is:

1. A method for use with a compiler architecture framework, the method comprising:

performing a compilation process on a computer program;

encoding an instruction with a selected encoding;

encoding the security mutation information in an instruction set architecture of a processor; and

executing a compiled computer program in the processor using an added mutation instruction, wherein executing comprises executing a mutation instruction to enable decoding another instruction.

2. The method of claim **1**, wherein instruction encodings are randomly selected.

3. The method of claim **2**, wherein the compiled program consists of safe zones and each safe zone has independent instruction encoding.

4. The method of claim **1**, wherein a mutation is modified at runtime making an instruction encoding chip unique.

5. The method of claim **1**, wherein an instruction encoding depends on device parameter variation on the die.

6. The method of claim **1**, wherein an instruction encoding depends on a content of a persistent memory.

7. The method of claim **1**, wherein an instruction encoding depends on a hardware state in the processor.

8. The method of claim **1**, wherein an instruction encoding depends on an input output event.

9. A processing framework comprising:

machine storage for storing a compiler that is configured to compile a computer program, the compiler being configured to extract static information about the computer program during compilation, the static information being used to add a mutation instruction in the computer program to help decoding another instruction at runtime;

executing, wherein executing comprises storing of the mutation information encoded in a mutation instruction in the processor such that a subsequent instruction can be decoded by using that mutation.

10. The processing device of claim **9**, wherein the encoding of an instruction is randomly selected;

11. The processing device of claim **9**, wherein an AES module's power profile is protected against information leakage by feeding its input through the processing device.

12. The processing device of claim **9**, wherein a hardware logic is protected by controlling a configuration related to its operation through the processing device.

13. An instruction encoding of claim **9**, wherein the mutation information is in a register.

14. An instruction encoding of claim **9**, wherein the mutation information is in an immediate.

15. An instruction encoding of claim **9**, wherein the mutation information is from an IO device.

16. The system of claim **9** comprising of

the processing framework of claim **9**;

a second processor executing instructions from a computer program wherein at least one of the instructions executes on the processor framework of claim **9**.

17. The processing device of claim **9**, wherein execution time is randomized in time across processor reset cycles with random stall insertion.

18. The processing device of claim **9**, wherein instructions encode control for functional units.

19. The processing device of claim **9**, wherein an operation is replaced with SWIs.

20. The processing device of claim **9**, wherein random stall insertion is controlled by a security mutation instruction.

21. The processing device of claim **9**, wherein an instruction executing is encrypted.

* * * * *