

# SeMPE: Secure Multi Path Execution Architecture for Removing Conditional Branch Side Channels

Andrea Mondelli\*, Paul Gazzillo† and Yan Solihin‡

Department of Computer Science

University of Central Florida

Orlando, FL

\*mondelli@knights.ucf.edu, †paul.gazzillo@ucf.edu, ‡yan.solihin@ucf.edu

**Abstract**—One of the most prevalent source of side channel vulnerabilities is the secret-dependent behavior of conditional branches (SDBC). The state-of-the-art solution relies on Constant-Time Expressions, which require high programming effort and incur high performance overheads. In this paper, we propose SeMPE, an approach that relies on architecture support to eliminate SDBC without requiring much programming effort while incurring low performance overheads. The key idea is that when a secret-dependent branch is encountered, the SeMPE microarchitecture fetches, executes, and commits both paths of the branch, preventing the adversary from inferring secret values from the branching behavior of the program. To enable that, SeMPE relies on an architecture that is capable of safely executing both branch paths sequentially. Through microbenchmarks and an evaluation of a real-world library, we show that SeMPE incurs near ideal execution time overheads, which is the sum of the execution time of all branch paths of secret-dependent branches. SeMPE outperforms code generated by FaCT, a constant-time expression language, by up to a factor of 18×.

## I. INTRODUCTION

As more computation is performed in the cloud, secure and private computation becomes more and more critical. Sharing of hardware resources in the cloud is crucial to keeping their utilization rate high, but it opens the way for side channel vulnerabilities where an application may leak secret data through the usage patterns it exhibits on the shared hardware. Applications that share a hardware resource can then observe the resource usage pattern to infer secrets.

An important and prevalent source of side channels is the *secret-dependent behavior of conditional branches (SDBC)*. Code such as `if (secret) {if-path} else {else-path}` reveals information to the attacker through secret-dependent differences in the performance characteristics of the two paths resulting from the conditional. For instance, the leak is a *timing channel* when two paths differ in execution time, a *cache access channel* if the paths differ in cache access counts or occurrences, a *memory access pattern channel* if the memory accesses occur to different addresses in the two paths, and a *branch predictor channel* when the branch predictor state captures the past outcomes of the branch. Rather than designing an architecture support to close each different side channel, in this paper we propose an architecture that removes a sources of these side channels.

Figure 1 is classic example of a side-channel attack, the modular exponentiation routine from RSA public-key cryptography. The secrets are the bits of the key ( $e$ ), tested on line 4 ( $e_i = 1$ ). The attacker can indirectly infer the value of  $e_i$  by observing the time it takes to execute the operation. Closing secret-dependent conditional branches as a source of side channels is critical and challenging: the routines have to be carefully rewritten manually to eliminate secret-dependent conditionals [1]–[3], [5], [6], [9], [41].

```

1: for  $i = n - 1$  to 0 do
2:    $r \leftarrow \text{square}(r)$ 
3:    $r \leftarrow \text{modulo}(r, m)$ 
4:   if  $e_i = 1$  then
5:      $r \leftarrow \text{multiply}(r, b)$ 
6:      $r \leftarrow \text{modulo}(r, m)$ 
7:   end if
8: end for

```

Fig. 1: Modular exponentiation in RSA with  $e_i$  as secret.

The large human resources involved in manually rewriting code means only the most sensitive software are protected, leaving secret user data in general-purpose applications unprotected. Currently, there are several approaches to eliminating the secret-dependent behavior of conditional branches (SDBC). A popular software technique, used in many cryptographic libraries, is to use *Constant Time Expression (CTE)*. CTE eliminates conditional statements by manually converting the conditions into arithmetic expressions used in the branch paths.

Figure 2a shows an example of a nested `if-else` statement that operates on secret user data  $A$ ,  $B$ , and  $C$ . Figure 2b is the same program after a CTE transformation. Each secret condition ( $A$ ,  $B$ , and  $C$ ) is converted into a binary value ( $bA$ ,  $bB$ , and  $bC$ ). Each statement is converted into an expression that includes the logical combination of the binaries that produces the statement. For example, in line 3,  $j$  is assigned the value of  $j + 1$  when  $bA$  and  $bB$  are true,  $bA$  is true and  $bB$  is false, or when  $bA$  is false and  $bB$  is true. Otherwise, when both  $bA$  and  $bB$  are false,  $j$  is assigned its old value, i.e., it is not mutated.

Other approaches have also been proposed. Memory Trace Obliviousness (MTO) [31] and GhostRider [30] transform

<pre> 1: @secret A, B, C 2: if A ∨ B then 3:   j ← j + 1 4: else 5:   if C then 6:     k ← k + 1 7:   else 8:     k ← k - 1 9:   end if 10: end if           </pre> <p style="text-align: center;">(a)</p>	<pre> 1: @secret A, bA, B, bB, C, bC 2: bA ← (bool)A 3: bB ← (bool)B 4: j ← (bA × bB + bA 5:   × (1 - bB) + (1 - bA) × bB) 6:   × (j + 1) + (1 - bA) × (1 - bB) × j 7: bC ← (bool)C 8: k ← (1 - bA) × (1 - bB) × bC × (k + 1) 9: k ← k + (1 - bA) × (1 - bB) 10:  × (1 - bC) × (k - 1)           </pre> <p style="text-align: center;">(b)</p>
--	--

Fig. 2: Examples: (a) code with conditional statements, and (b) its constant-time version. A, B, and C are secrets.

code in order to equalize memory accesses in both branch paths and obfuscates their addresses using ORAM [22], [23], [38]. Instead of equalizing the execution of both branch paths, Raccoon [42], a software approach built on top of transactional memory hardware, executes both branch paths. Raccoon transforms code so that both branch paths are executed, converts every load and store to transactions (using transactional memory support), and relies on a conditional move instructions (CMOV) to ensure that true-path values are written to memory.

We introduce *Secure Multi-Path Execution (SeMPE)*, an approach that extends existing microarchitecture to eliminate SDBC. Table I compares the three prior approaches with *SeMPE* across four categories important for protecting private user data in the cloud: (1) *programming complexity* to encourage its use, (2) *low performance overhead* for real-world scalability, (3) *architectural simplicity* to ease adoption, and (4) *backwards compatibility* for binary compatibility with non-*SeMPE* architectures. *SeMPE* provides the best tradeoffs between security and performance, while remaining backwards compatible and simple to program.

Like Raccoon, *SeMPE* works by executing both paths of branch instructions to eliminate the secret-dependent behavior, thereby preventing an adversary from inferring secret values. Unlike Raccoon, however, our approach uses new hardware extensions that require minimal compiler support. *SeMPE* repurposes and builds on *dual-path execution*, originally proposed for improving the performance of hard-to-predict branches by speculatively executing both paths of a branch. Similarly, *SeMPE* fetches and executes all paths of a secret conditional branch. But, unlike prior dual-path execution architectures, *SeMPE* ensures that the execution of both paths is indistinguishable from running either path alone, thereby preventing a side channel leak of secret values. Achieving this security property requires major differences in the architectural design compared to traditional dual-path execution: an indistinguishable memory access pattern, an execution order independent of the branch condition, and the commit of all instructions of both paths.

*SeMPE* introduces a new branching instruction, the Secure Jump (sJMP). When executed, the sJMP instruction pushes the

Aspects	CTE	GhostRider	Raccoon	SeMPE
Approach	elim. cond. branch	equalize path	execute both paths	<b>execute both paths</b>
Technique	SW	HW/SW	SW	<b>HW/SW</b>
Programming complexity	High	Low	Low	<b>Low</b>
Reported Overheads	187.3×	1,987×	452×	10.6×
Simple architecture	Yes	No	Yes	<b>Yes</b>
Backward compatible?	Yes	No	No	<b>Yes</b>

TABLE I: Comparing approaches to eliminate SDBC: constant time expression (CTE), GhostRider [30], [31], Raccoon [42], and our *SeMPE* Architecture.

destination address into a hardware Last-In-First-Out (LIFO) structure. When all the subsequent instructions have been committed, the pushed address is popped and used to set the Next Program Counter (nextPC), automatically executing the other branch of a secret-dependent conditional.

Rather than introducing a new opcode to use sJMP, the programmer or compiler prefixes a normal branch instruction with a special byte at the beginning and end of the branch. For optimization purposes, a programmer can omit the byte to use a non-secure branch for code not working with secret values. This design simplifies conversion of both hand-written and automatically generated assembly code. In contrast, writing a CTE algorithm has been cited as notoriously difficult [1]–[3], [5], [6], [9], [11], [12], [15], [17], [21], [41] and carries hefty performance overheads. While domain specific languages have been proposed to reduce the programming effort [18], [19], such solutions require rewriting software in a new language.

The byte chosen for sJMP is ignored on non-*SeMPE* architectures, enabling backwards compatibility of *SeMPE* assembly code, therefore *SeMPE* code can run on existing architectures without modification, albeit without the same security guarantees. This is in contrast with Raccoon which requires processors that have hardware transactional memory support. For a feasible security solution to have widespread adoption in commercial systems, the performance overhead should be minimal. Both MTO and Raccoon report overheads of 195× and 22×, respectively, on average, and 1,987× and 452× in the worst case [42]. While that direct comparisons between these overheads with each other and with *SeMPE* is not feasible due to differences in benchmarks and machine assumptions, our evaluation of *SeMPE* found an overhead of only 10.6× even in the case of conditionals nested ten deep and on a real-world case of a side-channel vulnerability.

We evaluated the performance of our proposed architecture with both a set of microbenchmarks and a real-world software library for image conversion called *libjpeg* [7] that contains a side channel leak that reveals an image visual details during decompression. The use of microbenchmarks allows for targeted stress testing of *SeMPE* performance by controlling the number and nesting depth of multiple secret-dependent branches. The *libjpeg* evaluation demonstrates *SeMPE*'s ability

to remove a side channel using a variety of image types of sizes. Our evaluation shows that the execution time with SeMPE is near ideal: execution time increases linearly with the number of secret branch paths, independent of the size of the workload executed. When compared against CTE code derived using the state of the art CTE language and compiler (FaCT), SeMPE outperforms CTE substantially, by a factor of  $1.6 - 18\times$ .

The rest of the paper is organized as following. Section II covers background and related work. Section III discusses the threat model. Section IV discusses the proposed architecture design, and Section V limitations and compiler support. Section VI and Section VII discuss the evaluation methodology and evaluation results. Finally, Section VIII concludes the paper.

## II. BACKGROUND AND RELATED WORK

Several techniques have been proposed for eliminating the secret-dependent behavior of conditional branches, including constant time expressions, memory trace obliviousness, and hardware transactional memory. SeMPE, however, draws inspiration from multi path execution in order to provide high performance while providing security guarantees.

### A. Techniques to Remove SDBCB

*a) Constant Time Expression:* Manual programming effort for Constant Time Expression (CTE) is currently standard practice technique for eliminating SDBCB. While popular, it has two substantial drawbacks that limit its use to simple code structures, such as in some crypto libraries. First, it involves a large manual effort, because it prohibits programmers from using conditional statements that use secret data. Moreover, programmers need to inspect the resulting assembly after each compilation to ensure that the compiler has not added conditional branches. Compilers are known to have inserted conditional branches even when the source code contains know conditional constructs [18], [19]. Writing a constant-time algorithm is notoriously difficult [11], [17], [21], [41]. Part of the reason is that the complexity of CTE code increases super-linearly with the nesting depth of conditional branches. In response, a domain specific language, Flexible Constant-Time Programming Language (FaCT) [18], [19], has been proposed to simplify CTE programming. While simpler to program, substantial restrictions exist at least in the latest version, e.g. no manual memory allocation, no function pointers, no function calls, no floating point, etc. can be used. FaCT is a new programming language, making it difficult to use for existing production software. Finally, performance overheads incurred by CTE are very high. In Figure 2a, the original code contains three additions, but the constant-time version in Figure 2b contains 28 additions or multiplications, nearly an order of magnitude higher. Since CTE is standard practice today, we compare SeMPE against CTE.

*b) Memory Trace Obliviousness:* Memory Trace Obliviousness [31] and the compiler and architecture for it (GhostRider [30]) transform code in order to balance memory

accesses in both branch paths and obfuscate their addresses using ORAM. For example, if in the *if* path there is an array access, then a new array access is added to the *else* path. ORAM is used to randomize memory addresses so that the two array accesses are indistinguishable from the point of view of the address stream.

*c) Raccoon:* Raccoon [42] is a software approach that uses hardware transactional memory to executes both branches of a secret-dependent conditional. Raccoon works by modifying code so that both branch paths are executed, converting every load and store to a transaction, and relying on a CMOV instruction to ensure that the only true-path store value is written to memory. While also executing both branch paths, SeMPE does so directly through new hardware mechanisms without depending on code transformation to transactions and, moreover, does not incur the overhead of transactions wrapping every load and store.

### B. Multi Path Execution

Dual/Multi Path Execution is a class of microarchitecture techniques previously proposed to reduce branch misprediction penalties by executing instructions from all paths of a conditional branch instruction [13], [25], [48]. Once the branch outcome is discovered, the false path instructions are squashed while the true path instructions are allowed to commit. The Dual Path Instruction Processing (DPIP) [13] allows false-path instructions to be fetched, decoded, renamed, but not executed, while predicted-path instructions are executed. The Selective Dual Path Execution (SDPE) [25] selectively forks a second path when a low confidence branch prediction is encountered. Threaded Multi-Path Execution (TME) [48] allows the alternative path instructions to execute in a separate thread context of an Simultaneous Multi-threading (SMT) processor.

While the goal of previous multi-path execution architectures was to reduce the branch misprediction penalty for hard to predict branches, the goal of SeMPE is to eliminate the secret-dependent behavior of conditional branches (SDBCB). Consequently, while sharing some similarities, SeMPE is fundamentally different from traditional multipath execution in several ways. First, the execution of instructions from both branch paths must be indistinguishable to the observer in SeMPE. That means that instructions from both paths must commit, instead of having one of them squashed. Otherwise, the multi path execution may still leak a secret value. Second, traditional multi path execution only handles one conditional branch, stalling at nested conditionals. In contrast, SeMPE must be able to handle nested conditional branches because a secure region may cover nested conditional branches that are both secret and non-secret. Finally, the scope of traditional multipath execution is limited to the instruction window of the processor. In contrast, SeMPE must handle an instruction count within secure conditional branch paths that often exceeds the processor instruction window.

### III. THREAT MODEL

We assume a threat model that is realistic for cloud computing where distinct applications share hardware. We assume that physical security is strong hence we do not protect against physical side channels (such as power usage) or other physical attacks. The victim and the attacker are assumed to run as separate processes in the same or different virtual machines that are scheduled to run on the same server, either in different cores sharing a cache, or in the same core through simultaneous multi-threading or time sharing.

We assume that the hypervisor and OS are trusted, and that they correctly enforce address space isolation, so the attacker cannot directly read secret data of the victim. We assume the attacker can measure timing at a coarse granularity, but has no access to hardware counters that track the victim’s execution characteristics. The attacker can prime the cache and branch predictor state through its own execution and can infer the victim’s working set, i.e., addresses of past reads and writes to memory, through a shared cache. The attacker knows or can guess the code that the victim is running. We do not focus on eliminating specific side channels. Instead, we focus on eliminating a common source of various side channels: SDBCB. We do not consider secret leaking through general memory access pattern. If such leakage is present, we assume techniques such as Oblivious RAM [22] are used for protection, which are orthogonal to our work. We only seek to protect memory access patterns that leak a secret as a result of different conditional branch paths. We note that SeMPE does not address Spectre/Meltdown-style attacks [28], [29], because they do not involve leaks due to secret-dependent branch behavior. Techniques for preventing Spectre/Meltdown are orthogonal to SeMPE.

We rely on the same input program assumptions used by Raccoon [42], i.e. (1) the program does not contain bugs that will induce application crashes, (2) the program does not exhibit undefined behavior, and (3) if multi-threaded, the program is data-race free. Because the proposed architecture executes all paths of a secure branch, an instruction in a false path may incur an exception, such as due to operating on incorrect value (e.g. divide-by-zero). Such situations are normally acceptable even in a bug-free program, if the programmer assumed always-taken or always-not taken branch behavior for a specific secret.

### IV. SEMPE DESIGN

#### A. Foundation of Security

The foundation for security of SeMPE is that executing both paths of a conditional branch that depends on secret is necessary to hide the secret. Assume a conditional branch with the following form, if (secret) P1 else P2 . Suppose that P1 and P2 *exclusive*, i.e. do not share common instructions, and *minimal*, i.e. removing any instruction from P1 (or P2) changes the live out values of P1 (P2). Also suppose that P1 and P2 are bug-free and do not incur any terminating exceptions. We claim that:

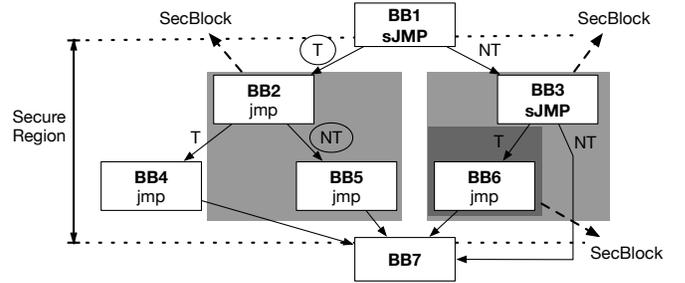


Fig. 3: Illustrating key concepts used in SeMPE: secure region, secure block, and secure branches.

**Claim.** *For the secret to be not inferrable from the execution of P1 or P2, the minimum execution needed is all instructions of P1 and all instructions of P2.*

To support the claim, consider the cases below. If only one of P1 or P2 is executed, secret is inferrable due to the behavior reflecting only one of them. If both P1 and P2 are executed entirely, secret cannot be inferred as execution behavior no longer depends on secret. Now suppose that we execute both P1 and P2 minus one instruction from P1. Since P1 is minimal, the correctness of P1 is affected. If P1 is the correct path, the execution of the code following the paths is affected and the change is observable by the attacker. If P1 is the wrong path, the execution of the code following the paths is not affected, but the observer expects change in behavior. Hence, no instructions can be removed from P1 and P2.

The important implication of the claim is that the execution time for execution of both paths of a secret branch represents the *ideal* overheads. If there are  $N$ -deep nested conditionals, and each path incurs  $T$  time, the ideal execution time in theory is  $2^N \times T$ . Any secure execution must be evaluated against that ideal.

#### B. Terminology

In order to be practical, SeMPE design must meet the following criteria. First, the architecture modification to the processor core must be *simple* (low complexity). Second, it must be *bidirectionally backward compatible*: traditional code must run correctly on the new architecture, and modified code must run correctly on traditional architecture albeit without security guarantees. Third, it must incur *low programming effort* and preferably code transformation should be automatable. Finally, it needs to be **fast**; *excessive* overheads are unlikely tolerable in production systems. To clarify the last point, the execution time must be as close as possible to the ideal case of the sum of execution time of all paths.

Before continuing, let us first discuss several terms. Suppose we have a control flow graph shown in Figure 3, containing seven basic blocks. The true branch outcomes and paths are shown circled. Two basic blocks BB1 and BB3 contain secret-dependent conditional branches, denoted as sJMP. For convenience, we will refer them as “*secret branches*”. Other basic blocks contain either non-secret conditional branches

or non-conditional branches, together denoted as `jmp`. All instructions in the path of a secret branch are referred to as SecureBlock (SecBlock). The figure shows both paths of BB1’s `sJMP` as SecBlocks. In contrast, BB2’s branch is not a secure branch hence BB4 and BB5 do not form SecBlocks. SecBlock can be nested, for example, BB6 is a SecBlock contained within the larger (BB3, BB6) SecBlock. The significance of SecBlock is that all instructions in SecBlock must be executed. For example, in the figure, the execution must cover BB1, BB2, BB5, BB3, BB6, and finally BB7. The only basic block that does not need to be executed is BB4, because it is not SecBlock. The encapsulating (i.e. outermost) code starting from the secure branch to the joint point of its paths is referred to as the *secure region*. For a secret branch with two SecBlocks, we refer to the true path as valid block.

### C. Expressing Secure Regions

a) *Instruction Set support*: SeMPE needs to be able to identify a secure branch. To ensure backward compatibility, we add prefix to existing branch instructions instead of introducing entirely new branch instructions. For this discussion, we will assume x86\_64 Instruction Set Architecture (ISA) [20], but a similar approach can be applied to other ISAs. The x86\_64 is chosen because it was the most challenging to add new extensions and instructions, due to the variety and the large number of instructions [10].

To support the SeMPE, the ISA is extended by adding a new instruction (`eosJMP`), and a unique prefix for branch instructions, called Secure Execution Prefix (SecPrefix). Branch instructions are coded as `sJMP` using the SecPrefix. We use byte `0x2e`, which is normally interpreted as hints of static branch prediction to the compiler.

The second modification is the addition of a new instruction that will be inserted as the first instruction in common between the two branch paths of the secure jump. The compiler inserts this instruction displacing the instruction that used to be the joint point of both branch paths. We refer to the new instruction as End-of-SecureJump (`eosJMP`). The instruction works as a backward jump to return the execution to the branch and the other branch path. We implement it using bytes `0x2e,0x90`. This instruction will be interpreted as a NOP in regular processors.

By using prefix and NOP, the binaries that are backward compatible with regular processors, keeping the execution overhead-free when running on a legacy architecture. The instructions added are interpreted as secure branches only by the microprocessor described in this paper.

### D. Challenges to Multi-Path Execution

Multi-path execution introduces challenges in designing the pipeline. Consider a code example in Figure 4 with four basic blocks with several instructions in each basic block. Suppose that a secure branch’s true path is not taken. Note that we have read after write (RAW) dependence between instructions B and F ( $B \rightarrow_{RAW} F$ ), and between G and H ( $G \rightarrow_{RAW} H$ ). If BB2 is also executed, phantom dependences

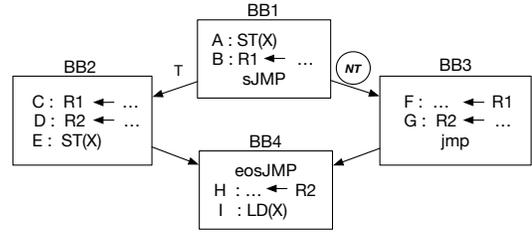


Fig. 4: Basic Blocks with Phantom Dependencies. Secure branch’s true path is not taken (NT).

may be introduced. An execution sequence of BB1, BB2, BB3, and BB4 will introduce the following phantom dependences:  $B \rightarrow_{WAW} C$ ,  $C \rightarrow_{RAW} F$ , and  $D \rightarrow_{WAW} G$ . Likewise, if the execution sequence is BB1, BB3, BB2, and BB4, phantom dependences are also introduced. The dependences obviously affect the correctness of the execution. Phantom memory dependences are also possible, with  $A \rightarrow_{MEM} I$  or  $E \rightarrow_{MEM} I$  being phantom.

Unlike past multi-path architectures, SeMPE’s goal is not lowering branch misprediction penalty. A secret branch must execute and *commit* both branch paths regardless of the branch predictor. While instructions from both paths can be executed in parallel, it increase architecture complexity significantly, in particular the outcome of register renaming may be unpredictable due to phantom register dependences, and restoring state becomes complicated. To keep the hardware support simple, we choose to execute the paths sequentially: a secret branch is evaluated twice, as true for the first SecBlock and as false for the second SecBlock.

Phantom dependences are still introduced with sequential execution of SecBlocks. When a false-path SecBlock is executed, the architecture state such as the rename table and register file will be changed. Thus, when `eosJMP` is encountered and the execution needs to go to the alternate path, the architecture state prior to the SecBlock needs to be restored. Similarly, the architecture state corresponding to the true SecBlock must be in place (or restored) prior to exiting the secure region. Section IV-F discusses our approach to this problem.

A similar phenomena exists for memory dependences, except that memory values are not part of the micro-architectural state, so saving and restoring memory values is out of the scope of SeMPE’s capabilities. We assume that programs are written or compiled with memory dependences already disambiguated.

### E. SeMPE Microarchitecture

In this section, we describe the architecture to enable secure execution of both branch paths of a secret branch. In traditional architectures, when a conditional branch instruction is encountered, the nextPC is set to either the following instruction (if the branch is not taken) or the target branch address (if the branch is taken). The branch predictor outcome sets the nextPC based on the predicted outcome.

In SeMPE, sJMP must execute both paths, hence the branch predictor does not need to generate prediction. Hence, the nextPC is set to the following instruction address, as if the branch condition is not verified. The not-taken SecBlock is executed entirely, while the target address of the sJMP instruction is calculated. Once the target address is calculated, it will be saved and used by the eosJMP instruction to set up the nextPC, which corresponds to the first instruction in the second SecBlock. Not-taken path is always executed first hence no secret-dependent behavior can be observed by the attacker, including order of memory accesses and behavior of prefetcher. We also assume the attacker does not alter the code.

The target address is managed in a LIFO hardware structure, called a Jump-Back Table (jbTable), shown in Figure 5. The jbTable consists of multiple entries to support nested secret branches, with each entry containing the nextPC address, the branch outcome (T/NT), a valid bit (Valid), and a Jump-Back (jb) bit. When a sJMP is issued (Step ①), a new entry in the jbTable is created, with the Valid and jb reset. When the sJMP is committed, the calculated target address is written to the jbTable, and the Valid bit is set (Step ②). A sJMP instruction can only be issued if the prior jbTable entry has its Valid bit set, otherwise it must stall from issuing. In this way, the jbTable will be faithful to LIFO to ensure that the correct Valid bit is set for the correct sJMP.

At the end of the first SecBlock, the eosJMP is executed and committed (Step ③). At that time, the most recent jbTable entry is looked up. If the jb is not set (when the eosJMP is encountered for the first time), the address field of the most recent entry is copied to the nextPC (Step ④), and the jb is set (Step ⑤). If, instead, the jb is already set, this indicates that the second SecBlock of the sJMP has been executed and the corresponding entry of the jbTable can be removed.

The existing issue queue presents a valid bit for each source operand, called V1 and V2 [24]. In the simplified issue queue entry in Figure 5, assuming two source operands, the V1 and V2 bits are set when the corresponding operand are ready, or ignored when the operand is not used. In existing microarchitectures, the V2 bit remains unset for conditional branches and not used. SeMPE set it when the Valid bit of the jbTable is set. A nested sJMP can be issued (Step ⑥) only if the jbTable is empty or the last sJMP in the LIFO is executed, i.e., the Valid bit is set and copied in V2. We don't need to modify the existing issue queue.

The use of a LIFO structure allows the handling of nested sJMP with low hardware complexity, without the need of a more complex random-access structures, and without adding address comparison logic. When running a SecBlock, we may encounter non-secret branch instructions. In contrast to sJMP instructions, they will consult (and update) the branch predictor.

The sJMP does not need to use the branch predictor, because we know in advance we will execute both path despite the value of the secret. If the pipeline is flushed due to a branch misprediction, the flushing works as follows. For each sJMP squashed in the Reorder Buffer (ROB), from the newest to

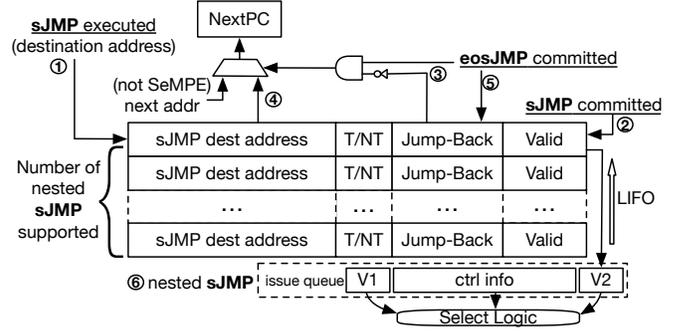


Fig. 5: Micro-architecture support for SeMPE. The branch outcome is saved in the T/NT bit field, where T is *Taken* and NT is *NotTaken*

the oldest, the most recent jbTable entry is deleted. The ROB will contain, at any time, the sJMP instructions representing SecBlock whose Program Counter (PC) has not “jumped back” yet, i.e. jb is still invalid. Since the address contained in the jbTable will be used as nextPC only when the eosJMP is committed the first time, we can guarantee the correctness after the pipeline flush.

Since each entry of the table deals with one sJMP instruction in a secure region, the number of jbTable entries is equal to the maximum number of nested sJMP the architecture can handle.

The total size of jbTable is small. Each jbTable entry equals to the size of a register (64 bits) + two bits (jb and Valid bits). Even with 30 entries, jbTable has less than 256 bytes. We believe a few dozen entries should be sufficient, because outside of recursion, deeply nested secure branches are rare. Our investigation reveals that the degree of sJMP nesting on a cryptographic algorithm is likely much less than a dozen. Dealing with secret user data may require a higher nesting degree, but unlikely to be beyond 30 in most situations.

Furthermore, the compiler can reduce the nesting degree by collapsing multiple conditionals into a single one with larger expression. For example, if (A) {if (B) ...} can be converted into if (A and B) {...}. Recursion may be either rejected at compile time, or made to trigger exception at run time. It is up to the exception handler whether to stop program execution, or to continue execution of the branch as non-secure. We note that such restrictions are also common in CTE.

### F. Dealing with Phantom Register Dependences

Phantom register dependences are false register dependences that occur between both paths of a secure branch. To manage them, we consider several architecture solutions. The first solution considered was the Lazy Register Spill (LRS). LRS uses a cache-like rename table with tags, similar to [36]. The tag identifies the SecBlock, allowing to spill only modified registers. Unfortunately, LRS complicates the rename table and affects instructions not belonging to SecBlock. Our goal is to keep hardware changes low without impacting the performance of the rest of the program. The second technique we considered was the use of a Physical Register Snapshot

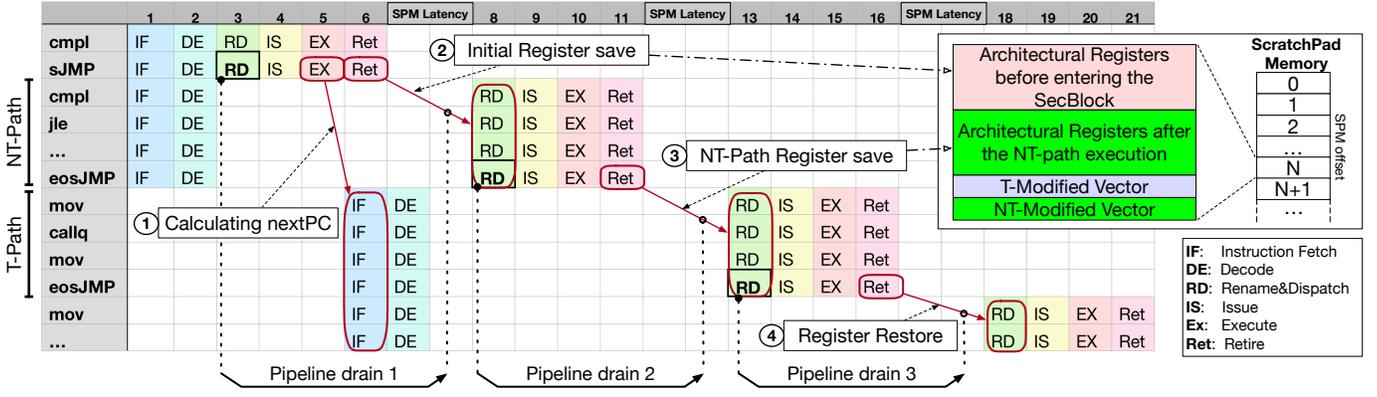


Fig. 6: Example of SemPE pipeline. Before entering the SecBlock and after the NT-Path, the pipeline is drained and the instruction rename is stopped. The pipeline is also drained at the end of the T-Path. The address of the first T-Path instruction is available after the sJMP execution ①. The SPM snapshots contain the registers saved before entering the SecBlock ② and after the NT-Path ③. The two bit-vector, updated during the execution, are used to restore the correct register value at the end of both paths ④. The address of the first instruction in the T-Path is available after the sJMP execution.

(PhyRS) mechanism to restore the contents of the register file and the Register Alias Table (RAT) at the end of both paths, depending on the secret.

The implementation needs two snapshots per nested SecBlock, containing the register file and the Register Alias Table (RAT). The first snapshot is taken prior to the execution of a SecBlock, right after the sJMP is committed. The second snapshot is taken at the end of the execution of the not-taken path, when the eosJMP is committed for the first time. At the end of the SecBlock, the register file and the RAT are rebuilt using the correct snapshot, according to the branch outcome. For saving snapshots, we considered the combined of scratchpad memory and register spilling. The Scratchpad Memory (SPM) was used as a temporary buffer to mitigate register spilling before any nested SecBlock.

This solution solves the problem of false register dependencies between paths but introduced an excessive performance overhead during the memory spilling of the content of the SPM. In modern architecture, it is common to have hundreds of physical registers [8]. Saving all physical registers and the RAT [49] produce too much snapshot spilling to memory, especially for deeply nested conditional branches.

Therefore, we choose a third design based on Architectural Register Snapshot (ArchRS) mechanism instead. The main difference is that only architectural registers are saved in the Scratchpad Memory (SPM), the number of which is much lower than the physical registers. Figure 6 shows the composition of a SecBlock snapshot at nesting level  $N$ . The nesting level is used as an offset to access the SPM during saving and restore.

Along with the two architectural register states, one before entering the SecBlock and another after the NT-Path execution, the SPM contains two bit-vectors. Each vector contains many bits to the number of architectural registers. The vectors track the architectural register modified during the two paths, Taken Path (T-Path) and NotTaken Path (NT-Path), and will be used

to restore the correct content of the architectural register at the end of SecBlock. A pipeline drain is added at the beginning of SecBlock. All the registers are saved when the sJMP is committed, and only modified registers are saved when the first eosJMP is committed. After the NT-Path the contents of the registers are restored from SPM. After the T-Path, the content of the architectural registers is updated with the correct value according to the secret.

At the end of a SecBlock, the register restore phase takes place. The registers modified in at least one of the two paths are read from the SPM. Depending on the branch outcome contained in the corresponding jbTable entry, the register is overwritten with the correct value. Figure 6 shows the sequence of executed instructions and when registers are saved or restored. The order of execution is independent of the secret. When the NT-Path is the true path, the value restored depends by the bit-vectors. For register modified in the NT-Path, the correct value comes from the NT-Path snapshot. For register modified in the T-Path but not in the NT-Path, the correct value is the one saved before entering the SecBlock. When the T-Path is the true path, all the modified register values are still read by the SPM but not used to restore the register contents. Instead, the current value is overwritten by itself. This behavior prevents the attacker from deducing the secret with a timing attack [15], [17], [32], [39], [50].

The execution of secret blocks is never interleaved, so one secret branch is always completely executed until the eosJMP, which occurs just before the CMOV. The pipeline is drained after each eosJMP. This pipeline drain allows that (1) the instruction window does not contain instructions from both paths at the same time, and (2) the instructions after the SecBlock observe the correct state of memory and registers.

The ArchRS mechanism introduces a third pipeline drain before entering the SecBlock, so that only the contents of valid registers are saved without introducing an additional level of complexity in the reconstruction of the RAT. This pipeline

drain is less expensive than a normal branch misprediction because the instructions are still fetched and decoded correctly, until their queues are full. Registers modified in at least one of the two paths are always read by the SPM, even if not used to restore the corresponding register value.

### G. Security Analysis

SeMPE eliminates SDBCBC through the execution of both branch paths (SecBlocks) in an order not related to the secret. The branch predictor channel, where the branch predictor state captures the past outcomes of the branch, is eliminated since there is no use of the predictor branch for sJMP. The compiler needs to reject any SecBlocks that have a potential hardware exception, e.g., a divide-by-zero error, removing any potential leaks due to exceptions. The user can decide whether to risk such code or not.

The combined use of Shadow Memory Locations (ShadowMemory) and CMOV hides the cache access to the attacker. The attacker, therefore, is not able to leak secret through the cache utilization analysis.

## V. DISCUSSION AND LIMITATIONS

### A. Phantom Memory Dependences

Executing both paths of a branch may cause the same memory locations to be written or read, creating phantom memory dependences, which are more difficult to address because memory values are beyond the architecture state of the pipeline. A store cannot be rolled back easily once it has been committed. To obtain the effects of fully executing and committing both paths of a secure branch, we considered several solutions. First, we could design the cache to keep versions of data from taken and not-taken paths and discard one of them at the conclusion of secure branch execution. However, keeping multiple versions of data in the cache creates complication with addressing and cache coherence, since one address may correspond to multiple data values. Furthermore, the manner in which values are discarded in the cache may cause a new side channel if not implemented carefully.

To keep SeMPE simple, to deal with phantom memory dependences, we duplicate any memory-allocated data modified in the SecBlocks for each secret branch, disambiguating the memory and preventing conflicting reads and writes to the same memory location by the false path. We refer to the duplicated memory as ShadowMemory. At the joint point (i.e. the postdominator block), a conditional move instruction CMOV is used to copy one of the values to the original copy. This completely avoids having to depend on a memory snapshot or use a memory state rollback. The approach has some similarity with Raccoon [42], but with substantial differences: (1) we apply this only to memory locations, as phantom register dependences are handled differently (Section IV-F), and (2) we do not use the transactional memory and transaction buffers that Raccoon uses. As a result, SeMPE overheads come only from the expansion of memory footprint due to

privatization, and the instruction execution overheads that come from executing CMOVs.

### B. Compiler Support for SeMPE

The benefits of SeMPE depend on correct usage of the ISA's two new instructions, SecureJump and End-of-SecureJump. These instructions mark the beginning and end of secure branches due to conditional branches on secret values. Such usage can be automated in the compiler, however, using a combination of information flow algorithms that track secrets and existing control- and data-flow analyses available in modern compiler frameworks, e.g., LLVM.

Using SeMPE correctly requires identifying the branches of secret values. Automatic identification is possible by leveraging existing work on information flow analysis [27], [35], [40], [43], [44], [47]. Information flow can be used to check for leaks of secret values from a *source*, e.g., input from a protected database, to a non-secret *sink*, e.g., an attacker-accessible output channel. In the case of secret-dependent branches, the sinks are all branch statements.

Once the compiler has identified which conditional branches involve secrets, the compiler can identify which basic blocks of the control-flow graph are the secure blocks. The secure blocks are successors of blocks that have secret-dependent branches, e.g., BB2, BB3, and BB6 from Figure 3. These secure blocks can then be transformed automatically for use with SeMPE. The compiler need only insert the secret-dependent branch with an sJMP where it would normally insert a JMP, and insert a eosJMP at the join point of the branch's two paths. For instance, in Figure 3, the end point for the sJMP in BB3 is the beginning of BB7, the first point after finishing the execution of any the resulting secret blocks. In a control-flow graph, this point is the immediate postdominator of BB3, i.e., the first block through which any path from BB3 must enter [34].

For a single secret-dependent branch, the eosJMP will end up being the successor of all secret blocks due to a secret-dependent branch. Conditional statements may have nested conditionals, either secret or non-secret. BB3 in Figure 3 is due to a secret branch nested inside of the branch resulting in BB1. Each branch has its own set of secret blocks. For each, the immediate postdominator indicates where the insert the eosJMP, handling the effects of any nesting. In this case, both BB3 and BB1's postdominator is the same block, BB7. While the postdominator need not be the same in all cases, one eosJMP is needed per sJMP. In this case, when the postdominator is the same, the compiler needs to insert both End-of-SecureJumps (eosJMPs) at the beginning of BB7.

The ISA and its accompanying system software (assemblers, linkers, etc), require very little change. Only two additional instruction types are needed. The sJMP to indicate a jump into a secure block and eosJMP to indicate the end of a secure block. The sJMP instructions have the same semantics as JMP, and are merely a signal to the hardware that both sides of the branch should be executed. The eosJMP is equivalent to a NOP and is a signal that the secure block is complete. The compiler

toolchain need only emit these mnemonics and assemble them into the appropriate machine code.

A compiler can also help automate SeMPE’s requirement of memory disambiguation in some cases. The simplest solution for stack-allocated variables is to create additional stack frame entries for each variable used in both branches an sJMP. A conditional move (CMOV) can then be inserted to select between the copies of the variable, as is done with hand-written SeMPE code.

As with stack-allocated variables, phantom dependences can occur between two secret blocks due to accesses to the same heap location. Such dependences are more difficult to detect at compile-time precisely and in general, in particular, for complex heap structures and pointer arithmetic. There are some solutions to handling phantom heap dependences that could be employed to ensure correct usage of SeMPE, e.g., shape analysis [33], [37], but may be too imprecise for some programs. In the worst-case, a library for intercepting memory referencing could ensure disambiguation at runtime, albeit at the expense of substantial overhead.

## VI. EVALUATION METHODOLOGY

To evaluate our scheme, we use two sets of workloads: microbenchmarks and a real-world application. The microbenchmarks are designed to stress test SeMPE across a wide range of code characteristics. They are also useful due to the scarcity of real world applications that have been implemented with CTE; CTE is currently only used in crypto libraries.

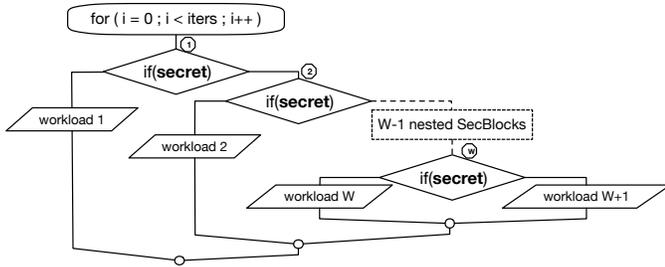


Fig. 7: Microbenchmark structure. The number of nested sJMP depends by  $W$ . The total number of sJMP per iteration is  $W$ , and the number of of nested sJMP is  $W - 1$

The microbenchmark has a customizable nested conditional branches that depend on secret, with several different workloads, as shown in Figure 7. The workload is one of the following: 1) **Fibonacci**, which calculates Fibonacci series up to the specified term, 2) **Ones**, which allocates a vector of integer, filling it with random numbers, and deleting the vector on exit, 3) **Quicksort**, which utilises a divide-and-conquer strategy to sort a large array [26], and 4) Eight **Queens** problem [14], which places eight queens on an  $8 \times 8$  chessboard such that none of them attacks one another. As can be seen in Figure 7, the two main parameters of the microbenchmark are (1) the number of iterations of the entire secure region ( $I$ ) and (2) the nesting depth and width of each iteration ( $W$ ). We vary  $I$  and  $W$  to produce over 700 combinations in order to test

clock frequency	2.0 GHz
branch predictor	31KB TAGE [45], 6KB ITTAGE [46]
fetch	8 instructions / cycle
decode	8 $\mu$ ops / cycle
rename	8 $\mu$ ops / cycle
issue (micro-ops)	8 $\mu$ ops
load issue	2 loads / cycle
retire	12 $\mu$ ops / cycle
reorder buffer (ROB)	192 $\mu$ ops
physical registers	256 INT, 256 FP
issue buffers	60 INT / 60 FP $\mu$ ops
load/store queue	32+32 entries
DL1 cache	32KB, 2-way assoc.
IL1 cache	16KB, 2-way assoc.
L2 cache	256KB, 2-way assoc.
prefetcher	stride pref. (L1), stream pref. (L2)
page size	4MB
SPM size	216KB (up to 30 snapshots supported)
SPM throughput	64 Bytes/cycle R/W

TABLE II: Baseline microarchitecture model.

the effect of nesting depth and eliminate measurement noise. We report a range of configurations for nesting depth: from  $W = 1$  (1-deep) to  $W = 10$  (10-deep). In all configurations, the number of instructions executed is at least 100 million, run to completion.

The second benchmark is a real-world library `djpeg`, which is an application from the `libjpeg` library that converts JPEG images into one of PPM, GIF, and BMP <sup>1</sup>. The secret value for this benchmark is the input array that holds the image by representing the color and intensity of each pixel. The core of the processing involves conditional branches that depend on each input array element. In contrast to the crypto library for which only tiny data (the key) is secret, input array in `djpeg` is substantially larger, e.g., a high-resolution photograph.

The three output file types (PPM, GIF, and BMP) differ in the number and type of instructions that are independent of the secret image. Even the number of secret-dependent instructions is not the same due to the different number of decode steps each file type has. The overall impact on the memory and the execution time depends on the output file type, so we use them as three separated workloads for the following analysis.

We compare SeMPE against de facto technique for SDBCB elimination: CTE. Specifically, we choose CTE version of the microbenchmarks written using FaCT [18], [19]. FaCT offers a domain-specific language which greatly simplifies microbenchmark conversion to CTE. However, we did not apply FaCT to `djpeg` because FaCT has many limitations that prevents this, e.g. supporting only boolean and integers, lack of memory allocator support, lack of support for function pointers and lack of support for global variable (macros or multiple file inclusion). It took us approximately three weeks to convert the microbenchmark using FaCT, which is a substantial programming effort. In contrast, with SeMPE, the

<sup>1</sup>JPEG stands for Joint Photographic Experts Group, GIF stands for Graphics Interchange Format, PPM stands for Portable Pixmap Format, and BMP represents Device Independent Bitmap.

programmer only needs to insert directives into the code that specify the secret.

The benchmarks were compiled with clang/llvm on Debian GNU/Linux, separating the secret-dependent code into its own compilation unit. The secret-dependent code was compiled with optimizations disabled to ensure that optimization does not inadvertently reintroduce a side channel. The rest of the benchmark code used the default optimization level, i.e.,  $-O2$ .

Each SecBlock was manually instrumented with sJMP and eosJMP instructions. Local variables were manually privatized (ShadowMemory) as described in Section V-A, adding additional local variables and inline assembly to use CMOV after the secret branches. Both register allocated and memory allocated variables are privatized, so we can consider the worst case. The ArchRS mechanism described in Section IV-F allow to limit privatization and CMOVs to memory allocated variables only.

For baseline architecture, we model a processor with parameters shown in Table II. We use gem5 simulator [16] with an out-of-order processor configured similar to the Intel Haswell [20] microarchitecture. The baseline differs from recent microarchitectures in terms of the cache size, to adjust for the benchmarks’ smaller working set.

We used a Scratchpad Memory that supports up to 30 register snapshots (one for each nested sJMP). Each snapshot contains two architectural register states and two bit-vectors with one bit per register each (Figure 6). Each register state contains the 48 architectural registers [4]. The total size of a snapshot for each SecBlock is 7392 bytes.

## VII. EVALUATION RESULTS

### A. Real-World Application Results

To evaluate SeMPE performance, we display its execution time overheads over the baseline architecture that has no security protection (Figure 8), for three output formats and input file sizes. The figure shows that the overheads vary between 31% and 87% across image output formats, but are not much affected by different image sizes. The overheads are much smaller than  $2\times$ , because the secure region only contributes to a fraction of the total instruction count. This factor also explains the variation across image output formats: the secure region in PPM contributes to much higher instruction count than GIF and BMP. On the other hand, the size of the input image does not affect the instruction count in the secure region because, on *djpeg*, the input array is decomposed into blocks, and each block performs several decompression steps depending on the type of output file-type produced. The SeMPE affects only the execution of SecBlocks, allowing a consistent behavior that is largely independent of the input size.

The pipeline drain described in Figure 6 produces “holes” in the pipeline, similar to a branch misprediction. This tends to increase Cycles Per Instruction (CPI). Factors that tend to decrease CPI include not having branch misprediction (the branch predictor is not used for sJMP) and parallelism

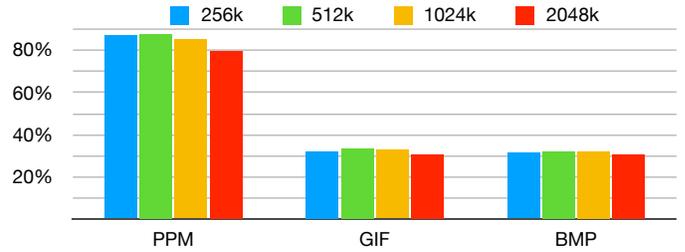


Fig. 8: Execution time overhead for libjpeg with different image output format, varying input size.

increases between branch paths due to the use of ShadowMemory. Executing both branch paths may increase or decrease cache spatial and temporal locality. If the total working set of both branch paths increases beyond the cache capacity, locality may decrease and cache miss rates increase. On the other hand, if the working set of both paths overlap substantially, executing one path produces *prefetching effect* for the alternate path, accelerating it. Apart from ShadowMemory for privatization of local variables written in branch paths and used outside the secure region, different branch paths of a secure branch share all other memory locations, which improves the cache temporal locality.

We analyze the impact of SeMPE on cache memory in Figure 9, which shows miss rates of the Instruction Cache (IL1), Data Cache (DL1), and the Second-Level Cache (L2), across image output formats and image sizes for each format. Observe that the impact on instruction cache is unrelated to the size of the input image. *djpeg* divides the input into multiple sub-blocks, and the decompression work-flow is applied to each sub-block. The image size has an impact on the total number of SecBlocks executed, but not on the number of instruction executed within a given SecBlock. The IL1 miss rate is low overall. Despite the reduced IL1 size used in our simulations, it is enough to contain the instructions that need to be fetched.

The situation changes when we dig into the DL1 miss rate analysis, shown in Figure 9b. The two SecBlocks within a single decoding step of *djpeg* are, in all cases, small enough to fit the DL1. The ShadowMemory used during the SeMPE play a fundamental role to take advantage of the principle of locality described earlier. Despite the execution of all the path of the sJMP, each path works on memory allocated (by the compiler) very close each other. This memory is just a copy of the memory allocated before the secure region, that will be written only after the eosJMP by the CMOV instruction. The benefits of the DL1 miss-rate for ShadowMemory have, as a consequence, a relevant impact on the already low global miss rate. We perform a similar analysis on the L2 miss rates (Figure 9c). The L2 miss rates are overall higher than for the data cache. However, the miss rates exhibit similar behavior as ones from the DL1, even if this time changing the output file type, and consequently the number and type of instructions executed outside the SecBlocks, has a much bigger impact on

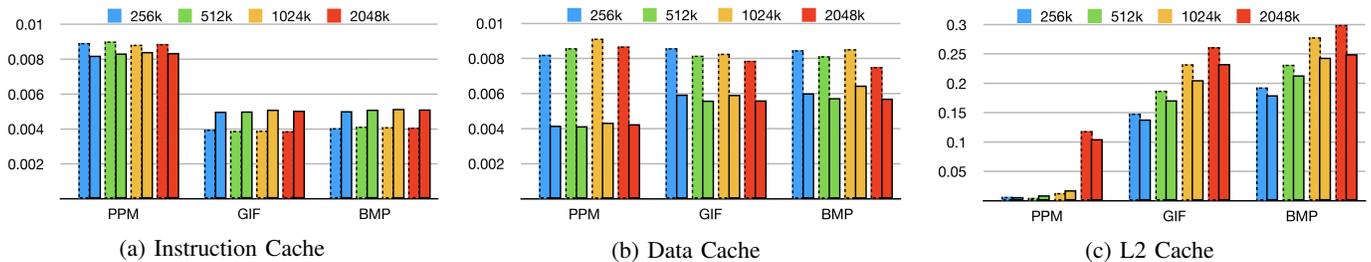


Fig. 9: Cache miss rates. Group of 2 columns: baseline (left, dashed line) and SeMPE (right, solid line). Lower is better.

the total miss rates.

### B. Microbenchmarks Results

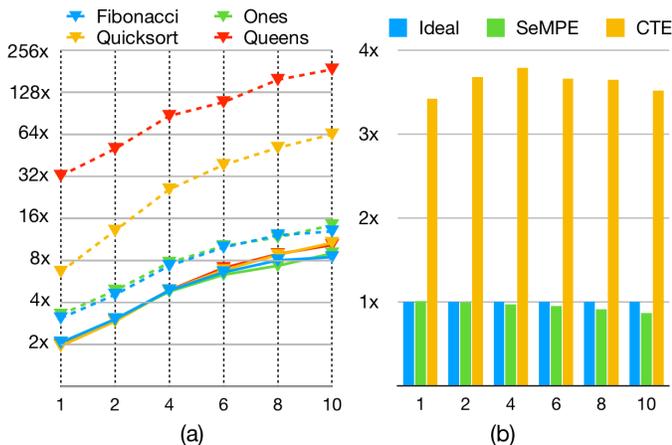


Fig. 10: Execution time overheads affected by the nesting depth  $W$  (X-axis): (a) SeMPE slowdown (solid line) vs. the slowdown due to CTE using FaCT (dashed line), and (b) Average slowdown normalized to ideal case.

SeMPE execute both paths of a secret-dependent branch, with instructions in each branch path unmodified, except for prefixing sJMP and inserting eosJMP instructions. Hence, we expect that the execution time with SeMPE should be roughly linearly proportional to the number of branch paths executed. While other factors affect the execution time as well, such as the overhead of the multi-path jump back mechanism and the improvement obtained by the better usage of cache memories, they also scale proportionally to the number of branch paths executed. In contrast, we cannot expect that the execution time of CTE to be linearly proportional to the number of branch paths in the original program. CTE requires each statement in a branch path to be modified to include the unrolling of all the expressions that were part of the conditional statements (Figure 2b). Hence, not only all statements in all branch paths are executed, but each statement takes longer to execute. Furthermore, the deeper the nesting level, and the more complex each conditional expression, the more expressions need to be unrolled and the more complex each statement becomes. The execution time ratio to baseline for the microbenchmarks shown in Figure 10a confirm these

expectations. The trends from no nesting ( $W = 1$ ) to deep nesting ( $W = 10$ ) are shown in Figure 10a for SeMPE (solid line) and FaCT (dashed line). Firstly, SeMPE shows much lower overheads vs. CTE (note that the y-axis is in logarithmic scale). Furthermore, since each statement becomes more complex and translates to a higher instruction count, the execution time overheads are higher when the original code has more instructions. The slowdown of FaCT, with  $W = 1$ , ranges from  $3\times$  for Fibonacci to  $32\times$  for Queens (Figure 10a).

Figure 10a also shows that as the nesting depth is increased up to ten, execution time slowdown increase for both SeMPE and CTE. When  $W = 10$ , SeMPE increases execution time by roughly  $8.4 - 10.6\times$ , consistent with the total number of branch paths of 11. CTE, on the other hand, slows down the execution between  $12.9 - 187.3\times$  (Figure 10a). Such slowdowns render CTE impractical for use in user code which, unlike crypto library, may be executed frequently. Overall, CTE can be up to  $18\times$  slower than SeMPE. This is on top of CTE’s substantially higher programming effort. To remove side channel leakage from secret-dependent-conditional-branches, the execution must be indistinguishable for any secret value. Thus, unless two paths can be merged, the ideal overhead is the sum of execution time of all paths, which is exponential to nesting depth. SeMPE beats this ideal overhead thanks to the *prefetching effect*, hence it is low vs. ideal (Figure 10b). In contrast, CTEs overheads are super-exponential.

## VIII. CONCLUSION

We introduced a hardware/software approach, SeMPE, that eliminates SDBC without incurring high performance overheads or requiring high programming effort. SeMPE allows programmers to annotate secret branches in their program. The ISA support is backward compatible. The architecture when encountering the new branch instruction executes both paths of the branch (one after the other) without consulting the branch predictor, thereby preventing the adversary from inferring secret from the executed path. SeMPE requires secret branches to be tracked using a hardware table that is small and simple (e.g. using LIFO instead of random access structure), and a small Scratchpad Memory to avoid the false register dependences that occur between both paths of a secure branch.

Hardware changes allow SeMPE code to run on processor supporting multipath securely, or running on processor not

supporting multipath fast. With hardware support, CMOV is only needed for phantom memory (but not register) dependencies. Code complexity of crypto code is low, so we evaluated SeMPE using a real world application and microbenchmarks. We shown that the execution time with SeMPE is near ideal; it increases linearly with the number of secret branch paths, independent from the size of workload executed in the SecBlock. Our experiments also show a slight positive cache locality benefit from multi-path execution. When compared against CTE code derived using the state of the art CTE language and compiler (FaCT), SeMPE outperforms CTE substantially, by a factor of  $1.6 - 18\times$ .

## REFERENCES

- [1] "868948 - a patch for nss: a constant-time implementation of the ghash function of aes-gcm, for processors that do not have the aes-ni/pclmulqdq," <https://bugzilla.mozilla.org/showbug.cgi?id=868948>.
- [2] "Added more constant-time code / removed biases in the prime number generation routines," <https://github.com/ARMmbed/mbedtls/pull/182>.
- [3] "Aes timing attack countermeasures," <https://github.com/weidai11/cryptopp/commit/c8e2f8959414846031634477b2a0614434843ca3>.
- [4] "AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions," <http://support.amd.com/TechDocs/24594.pdf>.
- [5] "Bearssl," <https://bearssl.org/gitweb/?p=BearSSL>.
- [6] "Coding rules," [https://cryptocoding.net/index.php/Coding\\_rules](https://cryptocoding.net/index.php/Coding_rules).
- [7] "Libjpeg Library," <http://libjpeg.sourceforge.net>.
- [8] "Software Optimization Guide for AMD Family 17h Models 30h and Greater Processors," [https://www.amd.com/system/files/TechDocs/56305\\_SOG\\_3.00\\_PUB.pdf](https://www.amd.com/system/files/TechDocs/56305_SOG_3.00_PUB.pdf).
- [9] "Why not use 'i', 'i' or '==' in constant time comparison?" <https://crypto.stackexchange.com/a/39432>.
- [10] "X86 Opcode and Instruction Reference," <http://ref.x86asm.net/geek64.html>, 2019.
- [11] O. Açımez, c. K. Koç, and J.-P. Seifert, "Predicting Secret Keys via Branch Prediction," in *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology*, ser. CT-RSA'07. Berlin, Heidelberg: Springer-Verlag, 2006. [Online]. Available: [http://dx.doi.org/10.1007/11967668\\_15](http://dx.doi.org/10.1007/11967668_15)
- [12] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 53–70. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida>
- [13] J. L. Aragón, J. M. H. Gonzalez, A. González, and J. E. Smith, "Dual path instruction processing," in *ICS*, 2002.
- [14] J. Bell and B. Stevens, "A Survey of Known Results and Research Areas for N-queens," *Discrete Math.*, vol. 309, no. 1, Jan. 2009. [Online]. Available: <http://dx.doi.org/10.1016/j.disc.2007.12.043>
- [15] D. J. Bernstein, "Cache-timing attacks on AES," Tech. Rep., 2005.
- [16] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, 2011.
- [17] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, no. 5, Aug. 2005. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S1389128605000125>
- [18] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "FaCT: A Flexible, Constant-Time Programming Language," in *2017 IEEE Cybersecurity Development (SecDev)*. Cambridge, MA, USA: IEEE, Sep. 2017, pp. 69–76. [Online]. Available: <http://ieeexplore.ieee.org/document/8077809/>
- [19] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan, "Fact: A dsl for timing-sensitive computation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: ACM, 2019, pp. 174–189. [Online]. Available: <http://doi.acm.org/10.1145/3314221.3314605>
- [20] I. Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual," 2016.
- [21] N. J. A. Fardan and K. G. Paterson, "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols," in *2013 IEEE Symposium on Security and Privacy*, May 2013.
- [22] O. Goldreich, "Towards a Theory of Software Protection and Simulation by Oblivious RAMs," in *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '87. New York, NY, USA: ACM, 1987. [Online]. Available: <http://doi.acm.org/10.1145/28395.28416>
- [23] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM*, vol. 43, pp. 431–473, 1996.
- [24] A. Gonzalez, F. Latorre, and G. Magklis, "Processor microarchitecture: An implementation perspective," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–116, 2010. [Online]. Available: <https://doi.org/10.2200/S00309ED1V01Y201011CAC012>
- [25] T. H. Heil and E. F. Smith, "Selective Dual Path Execution," 1996.
- [26] T. H. Hoare, "Algorithm 63 partition; algorithm 64 quicksort; algorithm 65 find," 1961.
- [27] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Can't live with 'em, can't live without 'em," in *Information Systems Security, 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings*, 2008, pp. 56–70. [Online]. Available: [https://doi.org/10.1007/978-3-540-89862-7\\_4](https://doi.org/10.1007/978-3-540-89862-7_4)
- [28] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," vol. abs/1801.01203. [Online]. Available: <http://arxiv.org/abs/1801.01203>
- [29] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.
- [30] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '15*. Istanbul, Turkey: ACM Press, 2015, pp. 87–101. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2694344.2694385>
- [31] C. Liu, M. Hicks, and E. Shi, "Memory trace oblivious program execution," in *Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium*, ser. CSF '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 51–65. [Online]. Available: <https://doi.org/10.1109/CSF.2013.11>
- [32] C. Luo, Y. Fei, and D. Kaeli, "Side-channel timing attack of rsa on a gpu," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 3, pp. 32:1–32:18, Aug. 2019. [Online]. Available: <http://doi.acm.org/10.1145/3341729>
- [33] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay, "Automatic numeric abstractions for heap-manipulating programs," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '10. New York, NY, USA: ACM, 2010, pp. 211–222. [Online]. Available: <http://doi.acm.org/10.1145/1706299.1706326>
- [34] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.
- [35] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *16th ACM Symp. on Operating System Principles (SOSP)*, October 1997, pp. 129–142. [Online]. Available: <http://www.cs.cornell.edu/andru/papers/iflow-sosp97/paper.html>
- [36] D. Oehmke, N. Binkert, T. Mudge, and S. Reinhardt, "How to Fake 1000 Registers," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. Barcelona, Spain: IEEE, 2005. [Online]. Available: <http://ieeexplore.ieee.org/document/1540944/>
- [37] P. O'Hearn, "Separation logic," *Commun. ACM*, vol. 62, no. 2, pp. 86–95, Jan. 2019. [Online]. Available: <http://doi.acm.org/10.1145/3211968>
- [38] R. Ostrovsky, "Efficient computation on oblivious rams," in *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, ser. STOC '90. New York, NY, USA: ACM, 1990, pp. 514–523. [Online]. Available: <http://doi.acm.org/10.1145/100216.100289>
- [39] C. Percival, "Cache missing for fun and profit," in *Proc. of BSDCan 2005*, 2005.
- [40] J. Planul and J. C. Mitchell, "Oblivious program execution and path-sensitive non-interference," in *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013*,

- 2013, pp. 66–80. [Online]. Available: <https://doi.org/10.1109/CSF.2013.12>
- [41] T. Pornin, “Why Constant-Time Crypto?” <https://www.bearssl.org/constanttime.html>. [Online]. Available: <https://www.bearssl.org/constanttime.html>
- [42] A. Rane, C. Lin, and M. Tiwari, “Raccoon: Closing Digital Side-Channels through Obfuscated Execution,” in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 431–446. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/rane>
- [43] B. Rodrigues, F. M. Quintão Pereira, and D. F. Aranha, “Sparse representation of implicit flows with applications to side-channel detection,” in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: ACM, 2016, pp. 110–120. [Online]. Available: <http://doi.acm.org/10.1145/2892208.2892230>
- [44] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE Journal on Selected Areas in Communications*, vol. 21, p. 2003, 2003.
- [45] A. Seznec, “A new case for the TAGE branch predictor,” in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec. 2011, pp. 117–127.
- [46] A. Seznec, “A 64-Kbytes ITTAGE indirect branch predictor,” 2011.
- [47] G. Smith, “Principles of secure information flow analysis,” in *Malware Detection*, M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, Eds. Boston, MA: Springer US, 2007, pp. 291–307.
- [48] S. Wallace, B. Calder, and D. M. Tullsen, “Threaded Multiple Path Execution,” in *ISCA*, 1998, 00192.
- [49] J. Xiao, M. Lou, W. Li, and Y. Cui, “Implementing fast recovery for register alias table in out-of-order processors,” *2013 2nd International Symposium on Instrumentation and Measurement, Sensor Network and Automation (IMSNA)*, pp. 821–824, 2013.
- [50] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, “A hardware design language for timing-sensitive information-flow security,” *SIGPLAN Not.*, vol. 50, no. 4, pp. 503–516, Mar. 2015. [Online]. Available: <http://doi.acm.org/10.1145/2775054.2694372>