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

Andrea Mondelli
Dept. of Computer Science
University of Central Florida
mondelli@knights.ucf.edu

Paul Gazzillo
Dept. of Computer Science
University of Central Florida
paul.gazzillo@ucf.edu

Yan Solihin
Dept. of Computer Science
University of Central Florida
yan.solihin@ucf.edu

***Abstract*—One prevalent source of side channel vulnerabilities is the secret-dependent behavior of conditional branches (SDBCB). 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 architecture support to eliminate SDBCB without requiring much programming effort while incurring low performance overheads. When a secret-dependent branch is encountered, SeMPE fetches, executes, and commits both paths of the branch, preventing the adversary from inferring secret values from the branching behavior of the program. SeMPE outperforms code generated by FaCT, a constant-time expression language, by up to 18×.**

***Index Terms*—side channel, conditional branch, multipath execution, microarchitecture**

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.

Techniques to address side channels have been proposed, in general using two separate approaches. One approach is to close each *type* of side channel. Each architecture component shared by multiple applications has the potential of leaking information through the use of the component, e.g. cache, branch predictor, memory controller, DRAM rows/banks, etc. (Figure 1). Techniques such as partitioning, randomization, etc. have been proposed for each type [1]–[4]. Since every shared component potentially leaks information, closing side channels with this approach requires all components to be protected simultaneously; thus it is hard to guarantee protection completeness. An alternative approach is to close the *source* of side channels. There are several sources in the code that cause side channel leakage, including conditional branches and access patterns. The general solution for closing the source of side channels is to make application behavior secret-independent. In general, closing the source is more powerful than closing the types of side channels as it provides broad based protection. However, efficient mechanisms for closing side channel source are still elusive. ORAM, for example, is known to slow down applications by orders of magnitude [5].

In this paper, we take the approach of addressing an important and prevalent *source* of side channels, which we refer to as the *the secret-dependent behavior of conditional branches (SDBCB)*. Code such as **if** *(secret)* {*if-path*} **else** {*else-path*} reveals information to the attacker through differences in behavior of the two paths of the conditional branch. 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 to support closing each type of side channel, we propose an architecture that removes a source of these side channels.

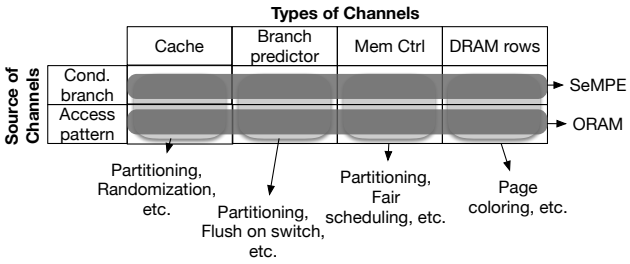

Fig. 1: Approaches to deal with side channels.

```
1: for i = n − 1 to 0 do
2:     r ← square(r)
3:     r ← modulo(r, m)
4:     if e_i = 1 then
5:         r ← multiply(r, b)
6:         r ← modulo(r, m)
7:     end if
8: end for
```

Fig. 2: Modular exponentiation in RSA with e_i as secret.

To illustrate an example, Figure 2 shows SDBCB leakage for the modular exponentiation function from RSA public-key cryptography. The secrets are the bits of the key (e), tested on line 4 ($e_i = 1$). By observing the difference in behavior of the branch paths (e.g. difference in execution time), the attacker can infer e_i. Protecting against this attack is challenging: the function has to be rewritten to eliminate the secret-dependence while preserving the same functional behavior [6]–[12].

Currently, there are only *software* approaches to eliminate the secret-dependent behavior of conditional branches (SDBCB). A popular 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. Memory Trace Obliviousness (MTO) [13] and GhostRider [14] transform code in order to equalize memory accesses in both branch paths and obfuscates their addresses using ORAM [15]–[17]. Raccoon [18] wraps a branch with functions that ensure both branch paths are executed. Due to reliance on software, they incur very high performance overheads; two to three orders of magnitude slowdown have been reported [13], [14], [18].

978-1-6654-3274-0/21/$31.00 ©2021 IEEE

To avoid the prohibitively high performance overheads of software solutions, we introduce *Secure Multi-Path Execution (SeMPE)*, a microarchitecture to eliminate SDBCB. SeMPE has the following unique benefits: (1) *low programming complexity* as only source annotations are required, (2) *low performance overhead*, (3) *simple architecture support*, and (4) *backwards compatibility* for binary compatibility with non-*SeMPE* architectures. SeMPE introduces architectural support for executing both paths of branch instructions to eliminate the secret-dependent behavior, thereby preventing an adversary from inferring secret values using any side channels.

Our approach uses new hardware extensions that can be utilized with minimal compiler support. SeMPE repurposes and builds on *dual-path execution* [19]–[21], 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 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.

We evaluated the performance of our proposed architecture with both a set of microbenchmarks, for stress testing and a real-world software image conversion library (*libjpeg* [22]) for realistic evaluation. *libjpeg* contains a side channel vulnerability that leaks visual details of an image during decompression. 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 [23], SeMPE outperforms CTE substantially, by a factor of $1.6 - 18\times$.

II. Background and Related Work

A. Techniques to Remove SDBCB

Several software techniques have been proposed for eliminating the secret-dependent behavior of conditional branches, including constant time expressions [23], memory trace obliviousness [13], [14], and Racoon [18]. Table I compares the three prior approaches with *SeMPE* across four categories important for protecting private user data in the cloud.

Aspects	CTE	GhostRider	Raccoon	SeMPE
Approach	elim. cond. branch	equalize path	execute both paths	**execute both paths**
Technique	SW	HW/SW	SW	**HW/SW**
Prog. complexity	High	Low	Low	**Low**
Overheads	Superlinear	Superlinear	Superlinear	**Linear**
Simple arch	Yes	No	Yes	**Yes**
Back-compatib.	Yes	No	No	**Yes**

TABLE I: Comparing approaches to eliminate SDBCB: constant time expression (CTE), GhostRider [13], [14], Raccoon [18], and our SeMPE architecture.

a) Constant Time Expression: CTE works by removing conditional branches by converting branch paths to full arithmetic operations. Figure 3a shows an example of a nested `if-else` statement that operates on secret user data A, B, and C. Figure 3b shows the resulting CTE transformation. Secrets (A, B, and C) are converted into binary values (bA, bB, and bC), and each statement is converted into an expression that includes the logical combination of the binaries that produces the statement.

```
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
```

(a)

```
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)
```

(b)

Fig. 3: Examples: (a) code with conditional statements, and (b) its constant-time version. A, B, and C are secrets.

Constant Time Expression (CTE) is currently the standard practice technique for eliminating SDBCB in some crypto libraries. However, it involves a large manual effort, both for code transformation and for verifying the resulting assembly code is free of conditional branches (which may be inserted by the compiler [23], [24]). Finally, the complexity of CTE code increases super-linearly with the nesting depth of conditional branches. For our test case of 11-path nested branches, CTE incurs $187.3\times$ slowdown vs. only $10.6\times$ with SeMPE.

b) Memory Trace Obliviousness and Racoon: Memory Trace Obliviousness [13] and the compiler and architecture for it (GhostRider [14]) transform code in order to balance memory accesses in both branch paths and obfuscate their addresses using ORAM. Raccoon [18] executes both branch paths and converts every load and store to a transaction to ensure that the false branch path does not affect program state. Both MTO and Raccoon incur huge execution time slowdowns of up to $1,987\times$ and $452\times$, respectively [13], [18].

Considering that MTO and Raccoon's overheads are higher than CTE, and CTE is the standard practice today, we choose CTE to compare against SeMPE.

B. Multi Path Execution

Dual/Multi Path Execution is a technique proposed to reduce branch misprediction penalties by executing instructions from all paths of a conditional branch instruction [19]–[21]. Once the branch outcome is discovered, the false path instructions are squashed while the true path instructions are allowed to commit. SeMPE differs from traditional multipath execution in several ways. First, to eliminate side channels, the execution of instructions from both branch paths must be indistiguishable to the observer in SeMPE. That means that instructions from both paths must *commit*, instead of having one of them squashed as in prior multipath techniques. Second, traditional multi path execution only handles one conditional branch, stalling at nested conditionals. In contrast, SeMPE handles nested conditional branches. Finally, traditional multipath's scope is tiny as it applies only to instruction window of the processor. In contrast, SeMPE's scope is arbitrary and not limited by the instruction window.

III. Threat Model and Assumptions

We assume a cloud computing platform where distinct applications share hardware. We assume that physical security is strong hence

we do not protect against physical attacks or physical side channels (such as power usage). The victim and the attacker may be separate processes in the same or different virtual machines scheduled to run on the same server. We assume a trusted hypervisor and OS that enforce address space isolation. 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 only focus on SDBCB sources of side channels, as access pattern source is already dealt with by Oblivious RAM [15]. We note that SeMPE does not address Spectre/Meltdown-style attacks [25], [26], 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 [18], 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 an incorrect value (e.g. divide-by-zero). Such situations are normally acceptable even in a bug-free program, if the programmer assumes 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 are *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:

Claim. *For the secret to not be inferrable from the execution of P1 or P2, the minimum execution needed is all instructions of P1 plus 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, the secret cannot be inferred as execution behavior no longer depends on the 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. The important implication of the claim is that the execution time for the 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 execution time is at least $2^N \times T$ when secure.

B. Terminology

In the presence of a control flow, a basic block contains the instructions of all the possible branch outcomes. All the instructions in the path of a secret branch are referred to as SecureBlock (SecBlock). The significance of SecBlock is that all instructions in SecBlock must be executed. 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.

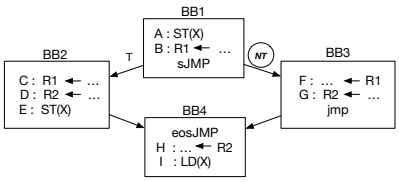

Fig. 4: Basic Blocks with Phantom Dependencies. Secure branch's true path is not taken (NT).

C. Expressing Secure Regions

a) Instruction Set support: To support the SeMPE, the Instruction Set Architecture (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. 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 and displaces the instruction that used to be the join 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 take the other branch path.

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 a 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 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 affect the correctness of the execution when both paths are executed. Phantom memory dependences are also possible, e.g, $A \rightarrow_{MEM} I$ or $E \rightarrow_{MEM} I$.

In SeMPE, a secret branch must execute and *commit* both branch paths regardless of the branch predictor. 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 phenomenon 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 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

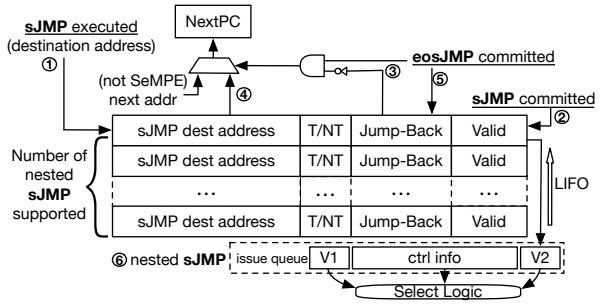

Fig. 5: Micro-architecture support for SeMPE. The branch outcome is saved in the T/NT bit field, where T is $Taken$ and NT is $NotTaken$

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 a prediction. Hence, the nextPC is always first set to the following instruction address, as if the branch condition is not true. 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 assume the attacker does not alter the code at runtime.) 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 bit 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 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 [27]. 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 sets 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 a nested sJMP with low hardware complexity, without the need of 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 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 a 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 [28]. 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 (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 combination 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 dependences between paths but introduced an excessive performance overhead during the memory spilling of the content of the SPM. In modern architectures, it is common to have hundreds of physical registers [29]. Saving all physical registers and the RAT [30] 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. Assuming N-nesting level, 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.

G. 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 [31]–[36].

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 compiler 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.

V. Evaluation Methodology

To evaluate our scheme, we use three sets of workloads: crypto benchmark (for functional testing), microbenchmarks (for stress testing), and a real-world application (for performance testing). The crypto benchmark is an asymmetric encryption algorithm **RSA**, which encrypts and decrypts plaintext for a set amount of times with different keys.

The microbenchmarks are computation kernels with customizable nested conditional branches that depend on secret, with several different workloads on the Taken path of the branch. The 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; each combination runs for at least 100 million instructions.

issue (micro-ops)	8 μops
retire	12 μops / cycle
physical registers	256 INT, 256 FP
load/store queue	32+32 entries
cache	DL1 32KB, IL1 16KB, L2 256KB, 2w
page size	4MB
SPM size	216KB (up to 30 snapshots supported)
SPM throughput	64 Bytes/cycle R/W

TABLE II: Haswell-like Baseline microarchitecture model.

The real world benchmark is a library `djpeg` from *libjpeg* library that converts JPEG images into one of PPM, GIF, and BMP; each case varies in behavior. The secrets for this benchmark are matrices that represent the color and intensity of image pixels.

We compare SeMPE against CTE written using the state of the art domain-specific language Flexible Constant-Time Programming Language (FaCT) [23], [24] and its compiler. The benchmark were compiled with clang/llvm on GNU/Linux, with secret-dependent code into its own compilation unit and manually checked against side channel inadvertantly reintroduced at compile time. FaCT was not used for `djpeg` due to many limitations of the language and compiler (e.g. no floating point support, etc.).

We use gem5 simulator [37] modelling the baseline as an out-of-order processor configured similar to the Intel Haswell [38] microarchitecture. The baseline differs from recent microarchitectures in terms of the cache size, to adjust for the benchmarks' smaller working set. The major differences with Haswell are reported in Table II.

VI. Evaluation Results

a) Real-World Application Results: Figure 6 depicts the performance of djpeg of SeMPE over the baseline architecture with no security protection, for three output formats and input file sizes. The overheads vary between 31% and 87% across image output formats, but not much across image sizes. Across the three image types, the number and type of instructions differ and the number of decode steps for each file type also differ, hence they produce different performance overheads. The overheads are quite reasonable considering the large amount of secrets being protected, i.e. large image data.

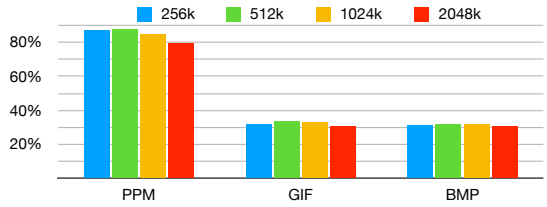

Fig. 6: Execution time overhead for libjpeg with different image output format, varying input size.

b) Crypto and Microbenchmarks Results: Figure 7a shows the slowdown factor (in log scale) of SeMPE (solid lines) and CTE (dashed lines) versus non-secure baseline, as the nesting depth W increases along the x-axis. SeMPE executes both paths of a secret-dependent branch, hence we expect that the execution time with SeMPE will be roughly linearly proportional to the number of branch paths executed. The figure mostly confirms it: SeMPE slows down the benchmarks by $8.4 - 10.6\times$ when $W = 10$ (11 branch paths). CTE, on the other hand, slows down the execution between $12.9 - 187.3\times$, as it executes all branch paths *plus* unrolling all the expressions that were part of the conditional statements (Figure 3b).

Figure 7b compares the slowdown (geometric mean across benchmarks), against theoretical ideal in which the execution time is equal to the number of branch paths (ideal). The figure shows that CTE is usually over $5\times$ slower vs. ideal. Surprisingly, SeMPE outperforms ideal when the nesting level is deep. This can be attributed to the *prefetching effect*, where both paths of a branch may have overlapped working set. Executing one path warms up the cache for the alternate path, accelerating the alternate path when it executes.

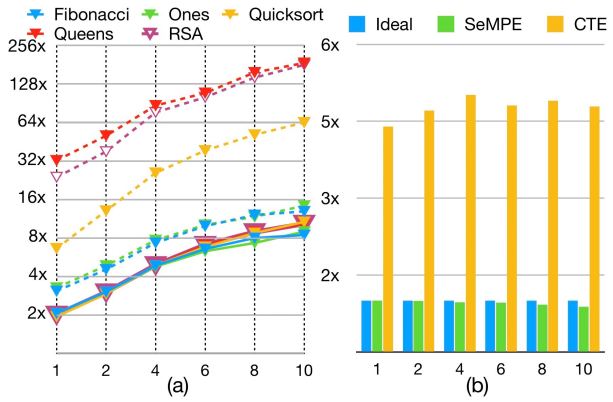

Fig. 7: Execution time overheads affected by the nesting depth W (X-axis): (a) SeMPE slowdown (solid line) vs. the slowdown due to CTE using FaCT (dashed line), and (b) Average slowdown normalized to ideal case.

VII. Conclusion

We introduced SeMPE, an architecture support for eliminating secret-dependent conditional branch behavior source of side channels. SeMPE executes both paths of a conditional branch, making the branch behavior secret-independent. SeMPE achieves near-ideal performance overheads without requiring high programming effort. It allows programmers to annotate secret branches in their program, and the architecture executes both paths automatically. 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. 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] O. Aciicmez, K. Koc, and J. Seifert, "On the power of simple branch prediction analysis," in *Symposium on Information, Computer and Communication Security*, ser. AsiaCCS, 2007.

[2] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *26th USENIX Security Symposium*, 2017.

[3] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *International Symposium on Microarchitecture*, ser. MICRO, 2016.

[4] D. Evtyushkin, N. Abu-Ghazaleh, R. Riley, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *International Symposium on Architecture Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2018.

[5] E. Stefanov, M. v. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, *Path ORAM: An Extremely Simple Oblivious RAM Protocol*, 2012.

[6] "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."

[7] "Added more constant-time code / removed biases in the prime number generation routines," https://github.com/ARMmbed/mbedtls/pull/182.

[8] "Aes timing attack countermeasures," https://github.com/weidai11/cryptopp/commit/c8e2f8959414846031634477b2a0614434843ca3.

[9] "Bearssl," https://bearssl.org/gitweb/?p=BearSSL.

[10] "Coding rules," https://cryptocoding.net/index.php/Coding_rules.

[11] "Why not use '¡', '¿' or '==' in constant time comparison?" https://crypto.stackexchange.com/a/39432.

[12] T. Pornin, "Why Constant-Time Crypto?" [Online]. Available: https://www.bearssl.org/constanttime.html

[13] 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, 2013.

[14] 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*, 2015.

[15] 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*, New York, NY, USA, 1987.

[16] R. Ostrovsky, "Efficient computation on oblivious RAMs," in *STOC '90*.

[17] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM*, vol. 43, pp. 431–473, 1996.

[18] 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., 2015.

[19] T. H. Heil and E. F. Smith, "Selective Dual Path Execution," 1996.

[20] J. L. Aragón, J. M. H. Gonzalez, A. González, and J. E. Smith, "Dual path instruction processing," in *ICS*, 2002.

[21] S. Wallace, B. Calder, and D. M. Tullsen, "Threaded Multiple Path Execution," in *ISCA*, 1998, 00192.

[22] "Libjpeg Library," http://libjpeg.sourceforge.net.

[23] 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)*, 2017.

[24] S. Cauligi, G. Soeller, and et al, "Fact: A dsl for timing-sensitive computation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2019.

[25] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[26] 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.

[27] A. González, F. Latorre, and G. Magklis, "Processor microarchitecture: An implementation perspective," *Synthesis Lectures on Computer Architecture*, 2010.

[28] 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.

[29] "Software Optimization Guide for AMD Family 17h Models."

[30] 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.

[31] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *16th ACM Symp. on Operating System Principles (SOSP)*, October 1997.

[32] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, 2003.

[33] 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.

[34] 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.

[35] 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*, 2016.

[36] 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*.

[37] J. Lowe-Power and et al., "The gem5 simulator: Version 20.0+," 2020.

[38] I. Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual," 2016.