Raccoon: Closing Digital Side-Channels through Obfuscated Execution

Ashay Rane, Calvin Lin
Department of Computer Science,
The University of Texas at Austin
{*ashay,lin*} *@cs.utexas.edu*

Mohit Tiwari
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
tiwari@austin.utexas.edu

Abstract

Side-channel attacks monitor some aspect of a computer system's behavior to infer the values of secret data. Numerous side-channels have been exploited, including those that monitor caches, the branch predictor, and the memory address bus. This paper presents a method of defending against a broad class of side-channel attacks, which we refer to as *digital side-channel attacks*. The key idea is to obfuscate the program at the source code level to provide the illusion that many extraneous program paths are executed. This paper describes the technical issues involved in using this idea to provide confidentiality while minimizing execution overhead. We argue about the correctness and security of our compiler transformations and demonstrate that our transformations are safe in the context of a modern processor. Our empirical evaluation shows that our solution is 8.9× faster than prior work (GhostRider [20]) that specifically defends against memory trace-based side-channel attacks.

1 Introduction

It is difficult to keep secrets during program execution. Even with powerful encryption, the values of secret variables can be inferred through various side-channels, which are mechanisms for observing the program's execution at the level of the operating system, the instruction set architecture, or the physical hardware. Side-channel attacks have been used to break AES [26] and RSA [27] encryption schemes, to break the Diffie-Hellman key exchange [15], to fingerprint software libraries [46], and to reverse-engineer commercial processors [18].

To understand side-channel attacks, consider the pseudocode in Figure 1, which is found in old implementations of both the encryption and decryption steps of RSA, DSA, and other cryptographic systems. In this function, s is the secret key, but because the Taken branch is computationally more expensive than the Not Taken branch, an adversary who can measure the time it takes to execute an iteration of the loop can infer whether the branch was Taken or Not Taken, thereby inferring the value of s one bit at a time [31, 5]. This particular block of code has also been attacked using side-channels involving the cache [44], power [16], fault injection [3, 41], branch predictor [1], electromagnetic radiation [11], and sound [32].

```
1:  function SQUARE_AND_MULTIPLY(m, s, n)
2:      z ← 1
3:      for bit b in s from left to right do
4:          if b = 1 then
5:              z ← m · z^2 mod n
6:          else
7:              z ← z^2 mod n
8:          end if
9:      end for
10: return z
11: end function
```

Figure 1: Source code to compute $m^s \bmod n$.

Over the past five decades, numerous solutions [20, 30, 21, 42, 35, 22, 40, 14, 43, 37, 39, 38, 23, 45, 25, 34, 9, 33, 10] have been proposed for defending against side-channel attacks. Unfortunately, these defenses provide point solutions that leave the program open to other side-channel attacks. Given the vast number of possible side-channels, and given the high overhead that comes from composing multiple solutions, we ideally would find a single solution that simultaneously closes a broad class of side-channels.

In this paper, we introduce a technique that does just this, as we focus on the class of *digital side-channels*, which we define as side-channels that carry information over discrete bits. These side-channels are visible to the adversary at the level of both the program state and the instruction set architecture (ISA). Thus, address traces, cache usage, and data size are examples of digital side-

channels, while power draw, electromagnetic radiation, and heat are not.

Our key insight is that *all* digital side-channels emerge from variations in program execution, so while other solutions attempt to hide the symptoms—for example, by normalizing the number of instructions along two paths of a branch—we instead attack the root cause by executing extraneous program paths, which we refer to as *decoy* paths. Intuitively, after obfuscation, the adversary's view through any digital side-channel appears the same as if the program were run many times with different inputs. Of course, we must ensure that our system records the output of only the real path and not the decoy paths, so our solution uses a transaction-like system to update memory. On the real paths, each `store` operation first reads the old value of a memory location before writing the new value, while the decoy paths read the old value and write the same old value.

The only distinction between real and decoy paths lies in the values written to memory: Decoy and real paths will write different values, but unless an adversary can break the data encryption, she cannot distinguish decoy from real paths by monitoring digital side-channels. Our solution does *not* defend against non-digital side-channel attacks, because analog side-channels might reveal the difference between the encrypted values that are stored. For example, a decoy path might "increment" some variable x multiple times, and an adversary who can precisely monitor some non-digital side-channel, such as power-draw, might be able to detect that the "increments" to x all write the same value, thereby revealing that the code belongs to a decoy path.

Nevertheless, our new approach offers several advantages. First, it defends against almost all digital side-channel attacks.[1] Second, it does not require that the programs themselves be secret, just the data. Third, it obviates the need for special-purpose hardware. Thus, standard processor features such as caches, branch predictors and prefetchers do not need to be disabled. Finally, in contrast with previous solutions for hiding specific side channels, it places few fundamental restrictions on the set of supported language features.

This paper makes the following contributions:

1. We design a set of mechanisms, embodied in a system that we call Raccoon,[2] that closes digital side-channels for programs executing on commodity hardware. Raccoon works for both single- and multi-threaded programs.

2. We evaluate the security aspects of these mechanisms in several ways. First, we argue that the obfuscated data- and control-flows are correct and are always kept secret. Second, we use information flows over inference rules to argue that Raccoon's own code does not leak information. Third, as an example of Raccoon's defense, we show that Raccoon protects against a simple but powerful side-channel attack through the OS interface.

3. We evaluate the performance overhead of Raccoon and find that its overhead is 8.9× smaller than that of GhostRider, which is the most similar prior work [20].[3] Unlike GhostRider, Raccoon defends against a broad range of side-channel attacks and places many fewer restrictions on the programming language, on the set of applicable compiler optimizations, and on the underlying hardware.

This paper is organized as follows. Section 2 describes background and related work, and Section 3 describes our assumed threat model. We then describe our solution in detail in Section 4 before presenting our security evaluation and our performance evaluation in Sections 5 and 6, respectively. We discuss the implications of Raccoon's design in Section 7, and we conclude in Section 8.

2 Background and Related Work

Side-channel attacks through the OS, the underlying hardware, or the processor's output pins have been a subject of vigorous research. Formulated as the "confinement problem" by Lampson in 1973 [19], such attacks have become relevant for cloud infrastructures where the adversary and victim VMs can be co-resident [29] and also for settings where adversaries have physical access to the processor-DRAM interface [46, 22].

Side-Channels through OS and Microarchitecture. Some application-level information leaks are beyond the application's control, for example, an adversary reading a victim's secrets through the `/proc` filesystem [13], or a victim's floating point registers that are not cleared on a context switch [2]. In addition to such *explicit* information leaks, *implicit* flows rely on *contention* for shared resources, as observed by Wang and Lee [39] for cache channels and extended by Hunger et al. [37] to all microarchitectural channels.

Defenses against such attacks either partition resources [40, 14, 43, 37], add noise [39, 38, 23, 45], or

[1] Section 3 (Threat Model) clarifies the specific side-channels closed by our approach.

[2] Raccoons are known for their clever ability to break their scent trails to elude predators. Raccoons introduce spurious paths as they climb and descend trees, jump into water, and create loops.

[3] GhostRider [20] was evaluated with non-optimized programs executing on embedded CPUs, which results in an unrealistically low overhead (∼10×). Our measurements instead use a modern CPU with an aggressively optimized binary as the baseline.

normalize the channel [17, 20] to curb side-channel capacity. Raccoon's defenses complement prior work that modifies the hardware and/or OS. Molnar et al. [25] describe a transformation that prevents control-flow side-channel attacks, but their approach does not apply to programs that contain function calls and it does not protect against data-flow-based side-channel attacks.

Physical Access Attacks and Secure Processors. Execute-only Memory (XOM) [36] encrypts portions of memory to prevent the adversary from reading secret data or instructions from memory. The AEGIS [35] secure processor provides the notion of tamper-evident execution (recognizing integrity violations using a merkle tree) and tamper-resistant computing (preventing an adversary from learning secret data using memory encryption). Intel's Software Guard Extensions (SGX) [24] create "enclaves" in memory and limit accesses to these enclaves. Both XOM and SGX are only partially successful in prevent the adversary from accessing code because an adversary can still disassemble the program binary that is stored on the disk. In contrast, Raccoon permits release of the transformed code to the adversary. Hence Raccoon never needs to encrypt code memory.

Oblivious RAM. AEGIS, XOM, and Intel SGX do not prevent information leakage via memory address traces. Memory address traces can be protected using Oblivious RAM, which re-encrypts and re-shuffles data after each memory access. The Path ORAM algorithm [34] is a tree-based ORAM scheme that adds two secret on-chip data structures, the *stash* and *position map*, to piggyback multiple writes to the in-memory data structure. While Raccoon uses a modified version of the Path ORAM algorithm, the specific ORAM implementation is orthogonal to the Raccoon design.

The Ascend [9] secure processor encrypts memory contents and uses the ORAM construct to hide memory access traces. Similarly, Phantom [22] implements ORAM to hide memory access traces. Phantom's memory controller leverages parallelism in DRAM banks to reduce overhead of ORAM accesses. However, both Phantom and Ascend assume that the adversary can only access code by reading the contents of memory. By contrast, Raccoon hides memory access traces via control flow obfuscation and software ORAM while still permitting the adversary to read the code. Ascend and Phantom rely on custom memory controllers whereas Memory Trace Oblivious systems that build on Phantom [20] rely on a new, deterministic processor pipeline. In contrast, Raccoon protects off-chip data on commodity hardware.

Memory Trace Obliviousness. GhostRider [20, 21] is a set of compiler and hardware modifications that transforms programs to satisfy Memory Trace Obliviousness (MTO). MTO hides control flow by transforming programs to ensure that the memory access traces are the same no matter which control flow path is taken by the program. GhostRider's transformation uses a type system to check whether the program is fit for transformation and to identify security-sensitive program values. It also pads execution paths along both sides of a branch so that the length of the execution does not reveal the branch predicate value.

However, unlike Raccoon, GhostRider cannot execute on generally-available processors and software environments because GhostRider makes strict assumptions about the underlying hardware and the user's program. Specifically, GhostRider (1) requires the use of new instructions to load and store data blocks, (2) requires substantial on-chip storage, (3) disallows the use of dynamic branch prediction, (4) assumes in-order execution, and (5) does not permit use of the hardware cache (it instead uses a scratchpad memory controlled by the compiler). GhostRider also does not permit the user code to contain pointers or to contain function calls that use or return secret information. By contrast, Raccoon runs on SGX-enabled Intel processors (SGX is required to encrypt values on the data bus) and permits user programs to contain pointers, permits the use of possibly unsafe arithmetic statements, and allows the use of function calls that use or return secret information.

3 Threat Model and System Guarantees

This section describes our assumptions about the underlying hardware and software, along with Raccoon's obfuscation guarantees.

Hardware Assumptions. We assume that the adversary can monitor and tamper with any digital signals on the processor's I/O pins. We also assume that the processor is a sealed chip [35], that all off-chip resources (including DRAM, disks, and network devices) are untrusted, that all read and written values are encrypted, and that the integrity of all reads and writes is checked.

Software Assumptions. We assume that the adversary can run malicious applications on the same operating system and/or hardware as the victim's application. We allow malicious applications to probe the victim application's run-time statistics exposed by the operating system (*e.g.* the stack pointer in `/proc/pid/stat`). However, we assume that the operating system is trusted, so Iago attacks [7] are out of scope.

The Raccoon design assumes that the input program is free of errors, *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, then the program is data-race free. Under these assumptions, Raccoon does not introduce new termination-channel leaks, and Raccoon correctly obfuscates multi-threaded programs.

Raccoon statically transforms the user code into an obfuscated binary; we assume that the adversary has access to this transformed binary code and to any symbol table and debug information that may be present.

In its current implementation, Raccoon does not support all features of the C99 standard. Specifically, Raccoon cannot obfuscate I/O statements[4] and non-local `goto` statements. While `break` and `continue` statements do not present a fundamental challenge to Raccoon, our current implementation does not obfuscate these statements. Raccoon cannot analyze libraries since their source code is not available when compiling the end-user's application.

As with related solutions [30, 20, 21], Raccoon does not protect information leaks from loop trip counts, since naïvely obfuscating loop back-edges would create infinite loops. For the same reason, Raccoon does not obfuscate branches that represent terminal cases of recursive function calls. However, to address these issues, it is possible to adapt complementary techniques designed to close timing channels [42], which can limit information leaks from loop trip counts and recursive function calls.

Raccoon includes static analyses that check if the input program contains these unsupported language constructs. If such constructs are found in the input program, the program is rejected.

System Guarantees. Within the constraints listed above, Raccoon protects against all digital side-channel attacks. Raccoon guarantees that an adversary monitoring the digital signals of the processor chip cannot differentiate between the real path execution and the decoy path executions. Even after executing multiple decoy program paths, Raccoon guarantees the same final program output as the original program.

Raccoon guarantees that its obfuscation steps will not introduce new program bugs or crashes, so Raccoon does not introduce new information leaks over the termination channel.

Assuming that the original program is race-free, Raccoon's code transformations respect the original program's control and data dependences. Moreover, Raccoon's obfuscation code uses thread-local storage. Thus, Raccoon's obfuscation technique works seamlessly with multi-threaded applications because it does not introduce new data dependences.

[4]Various solutions have been proposed that allow limited use of "transactional" I/O statements through runtime systems [6], operating systems [28], or the underlying hardware [4].

```
1: p ← &a;
2: if secret = true then
3:     ...                                  ▷ Real path.
4: else
5:     ...                                  ▷ Decoy path.
6:     p ← &b;        ▷ Dummy instructions do not update p.
7:     *p ← 10;       ▷ Accesses variable a instead of b!
8: end if
```

Figure 2: Illustrating the importance of Property 2. This code fragment shows how solutions that do not update memory along decoy paths may leak information. If the decoy path is not allowed to update memory, then the dereferenced pointer in line 7 will access ***a*** instead of accessing ***b***, which reveals that the statement was part of a decoy path.

4 Raccoon Design

This section describes the design and implementation of Raccoon from the bottom-up. We start by describing the two critical properties of Raccoon that distinguish it from other obfuscation techniques. Then, after describing the key building block upon which higher-level oblivious operations are built, we describe each of Raccoon's individual components: (1) a taint analysis that identifies program statements that require obfuscation (Section 4.3), (2) a runtime transaction-like memory mechanism for buffering intermediate results along decoy paths (Section 4.4), (3) a program transformation that obfuscates control-flow statements (Section 4.5), and (4) a code transformation that uses software Path ORAM to hide array accesses that depend on secrets (Section 4.6). We then describe Raccoon's program transformations that ensure crash-free execution (Section 4.7). Finally, we illustrate with a simple example the synergy among Raccoon's various obfuscation steps (Section 4.8).

4.1 Key Properties of Our Solution

Two key properties of Raccoon distinguish it from other branch-obfuscating solutions [20, 21, 25, 8]:

- **Property 1:** Both real and decoy paths execute actual program instructions.
- **Property 2:** Both real and decoy paths are allowed to update memory.

Property 1 produces decoy paths that—from the perspective of an adversary monitoring a digital side-channel—are indistinguishable from from real paths.

Without this property, previous solutions can close one side-channel while leaving other side-channels open. To understand this point, we refer back to Figure 1 and consider a solution that normalizes execution time along the two branch paths in the Figure by adding NOP instructions to the Not Taken path. This solution closes the timing channel but introduces different instruction counts along the two branch paths. On the other hand, the addition of dummy instructions to normalize instruction counts will likely result in different execution time along the two branch paths, since (on commodity hardware) the NOP instructions will have a different execution latency than the multiply instruction.

Property 2 is a special case of Property 1, but we include it because the ability to update memory is critical to Raccoon's ability to obfuscate execution. For example, Figure 2 shows that if the decoy path does not update the pointer ***p***, then the subsequent decoy statement will update ***a*** instead of ***b***, revealing that the assignment to ****p*** was part of a decoy path.

4.2 Oblivious Store Operation

Raccoon's key building block is the oblivious `store` operation, which we implement using the CMOV x86 instruction. This instruction accepts a condition code, a source operand, and a destination operand; if the condition is true, it moves the source operand to the destination. When both the source and the destination operands are in registers, the execution of this instruction does not reveal information about the branch predicate (hence the name *oblivious* store operation).[5] As we describe shortly, many components in Raccoon leverage the oblivious `store` operation. Figure 3 shows the x86 assembly code for the CMOV wrapper function.

4.3 Taint Analysis

Raccoon requires the user to annotate secret variables using the `__attribute__` construct. With these secret variables identified, Raccoon performs inter-procedural taint analysis to identify branches and data access statements that require obfuscation. Raccoon propagates taint across both implicit and explicit flow edges. The result of the taint analysis is a list of memory accesses and branch statements that must be obfuscated to protect privacy.

[5]Contrary to the pseudocode describing the CMOV instruction in the Intel 64 Architecture Software Developer's Manual, our assembly code tests reveal that in 64-bit operating mode when the operand size is 16-bit or 32-bit, the instruction resets the upper 32 bits regardless of whether the predicate is true. Thus the instruction does not leak the value of the predicate via the upper 32 bits, as one might assume based on the manual.

```
01: cmov(uint8_t pred, uint32_t t_val, uint32_t f_val) {
02:     uint32_t result;
03:     __asm__ volatile (
04:         "mov    %2, %0;"
05:         "test   %1, %1;"
06:         "cmovz  %3, %0;"
07:         "test   %2, %2;"
08:         : "=r" (result)
09:         : "r" (pred), "r" (t_val), "r" (f_val)
10:         : "cc"
11:     );
12:     return result;
13: }
```

Figure 3: CMOV wrapper

4.4 Transaction Management

To support Properties 1 and 2, Raccoon executes each branch of an obfuscated if-statement in a transaction. In particular, Raccoon buffers `load` and `store` operations along each path of an if-statement, and Raccoon writes values along the real path to DRAM using the oblivious `store` operation. If a decoy path tries to write a value to the DRAM, Raccoon uses the oblivious store operation to read the existing value and write it back. At compile time, Raccoon transforms `load` and `store` operations so that they will be serviced from the transaction buffers. Figure 4 shows pseudocode that implements transactional loads and stores. Loads and stores that appear in non-obfuscated code do not use the transaction buffers.

4.5 Control-Flow Obfuscation

To obfuscate control flow, Raccoon forces control flow along both paths of an obfuscated branch, which requires three key facilities: (1) a method of perturbing the branch outcome, (2) a method of bringing execution control back from the end of the `if`-statement to the start of the `if`-statement so that execution can follow along the unexplored path, and (3) a method of ensuring that memory updates along decoy path(s) do not alter non-transactional memory. The first facility is implemented by the `obfuscate()` function (which forces sequential execution of both paths arising out of a conditional branch instruction). Although Raccoon executes both branch paths, it evaluates the (secret) branch predicate only once. This ensures that the execution of the first path does not unexpectedly change the value of the branch predicate. The second facility is implemented by the `epilog()` function (which transfers control-flow from the post-dominator of the `if`-statement to the beginning of the `if`-statement). Finally the third facility is implemented using the oblivious `store` operation described earlier. The control-flow obfuscation functions

```
// Writes a value to the transaction buffer.
tx_write(address, value) {
    if (threaded program)
        lock();

    // Write to both the transaction buffer
    // and to the non-transactional storage.
    tls->gl_buffer[address] = value;
    *address = cmov(real_idx == instance,
            value, *address);

    if (threaded program)
        unlock();
}

// Fetches a value from the transaction buffer.
tx_read(address) {
    if (threaded program)
        lock();

    value = *address;
    if (address in tls->gl_buffer)
        value = tls->gl_buffer[address];

    value = cmov(real_idx == instance,
            *address, value);

    if (threaded program)
        unlock();

    return value;
}
```

Figure 4: Pseudocode for transaction buffer accesses. Equality checks are implemented using `XOR` operation to prevent the compiler from introducing an explicit branch instruction.

(`obfuscate()` and `epilog()`) use the libc `setjmp()` and `longjmp()` functions to transfer control between program points.

Safety of setjmp() and longjmp() Operations. The use of `setjmp()` and `longjmp()` is safe as long as the runtime system does not destroy the activation record of the caller of `setjmp()` prior to calling `longjmp()`. Thus, the function that invokes `setjmp()` should not return until `longjmp()` is invoked. To work around this limitation, Raccoon copies the stack contents along with the register state (identified by the `jmp_buff` structure) and restores the stack before calling `longjmp()`. To avoid perturbing the stack while manipulating the stack, Raccoon manipulates the stack using C macros and global variables.

As an additional safety requirement, the runtime system must not remove the code segment containing the call to `setjmp()` from instruction memory before the call to `longjmp()`. Because both `obfuscate()`—which calls `setjmp()`—and `epilog()`—which calls `longjmp()`—are present in the same program module, we know that that the code segment will not vanish before calling `longjmp()`.

Obfuscating Nested Branches. Nested branches are obfuscated in Raccoon by maintaining a stack of transaction buffers that mimics the nesting of transactions. Unlike traditional transactions, transactions in Raccoon are easier to nest because Raccoon can determine whether to commit the results or to store them temporarily in the transaction buffer *at the beginning of the transaction* (based on the secret value of the branch predicate).

4.6 Software Path ORAM

Raccoon's implementation of the Path ORAM algorithm builds on the oblivious `store` operation. Since processors such as the Intel x86 do not have a trusted memory (other than a handful of registers) for implementing the *stash*, we modify the Path ORAM algorithm from its original form [34]. Raccoon's Path ORAM implementation cannot directly index into arrays that represent the *position map* or the *stash*, so Raccoon's implementation streams over the *position map* and *stash* arrays and uses the oblivious `store` operation to selectively read or update array elements. Raccoon implements both recursive [33] as well as non-recursive versions of Path ORAM. Our software implementation of Path ORAM permits flexible sizes for both the *stash memory* and the *position map*.

Section 6.3 compares recursive and non-recursive ORAM implementations with an implementation that streams over the entire data array. Raccoon uses AVX vector intrinsic operations for streaming over data arrays. We find that even with large data sizes, it is faster to stream over the array than perform a single ORAM access.

4.7 Limiting Termination Channel Leaks

By executing instructions along decoy paths, Raccoon might operate on incorrect values. For example, consider the statement `if (y != 0) { z = x / y; }`. If $y = 0$ for a particular execution and if Raccoon executes the decoy path with $y = 0$, then the program will crash due to a division-by-zero error, and the occurrence of this crash in an otherwise bug-free program would reveal that the program was executing a decoy path (and, consequently, that $y = 0$).

To avoid such situations, Raccoon prevents the program from terminating abnormally due to exceptions. For each integer division that appears in a transaction (along both real and decoy paths), Raccoon instruments the operation so that it obliviously (using `cmov`) replaces

```
/* Sample user code. */
01: int array[512] __attribute__((annotate ("secret")));
02: if (array[mid] <= x) {
03:     l = mid;
04: } else {
05:     r = mid;
06: }

/* Transformed pseudocode. */
07: r1 = stream_load(array, mid);
08: r2 = r1 <= x;
09: key = obfuscate(r2, r3);

10: if (r3) {
11:     tx_write(l, mid);
12: } else {
13:     tx_write(r, mid);
14: }

15: epilog(key);
```

Figure 5: Sample code and transformed pseudocode.

the divisor with a non-zero value. To prevent integer division overflow, Raccoon checks whether the dividend is equal to `INT_MIN` and whether the divisor is equal to -1; if so, Raccoon obliviously substitutes the divisor to prevent a division overflow. Raccoon also disables floating point exceptions using `fedisableexcept()`. Similarly, array `load` and `store` operations appearing on the decoy path are checked (again, obliviously, using `cmov`) for out-of-bounds accesses. Thus, to ensure that the execution of decoy paths does not crash the program, Raccoon patches unsafe operations. Section 5.3 demonstrates that this process of patching unsafe operations does not leak secret information to the adversary.

4.8 Putting It All Together

We now explain how Raccoon transforms the code shown in Figure 5. Here, the `secret` annotation informs Raccoon that the contents of `array` are secret.

Static taint analysis then reveals that the branch predicate (line 2) depends on the secret value, so Raccoon obfuscates this branch. Similarly, implicit flow edges from the branch predicate to the two assignment statements (at lines 3 and 5) indicate that Raccoon should use the oblivious `store` operation for both assignment statements.

Accordingly, Raccoon replaces direct memory stores for `l` and `r` with function calls that write into transaction buffers in lines 11 and 13 of the transformed pseudocode. The access to `array` in line 1 is replaced by an oblivious streaming operation in line 7. Finally, the branch in line 2 is obfuscated by inserting the `obfuscate()` and `epilog()` function calls. The `epilog()` and `obfuscate()` function calls are coordinated over the `key` variable. To prevent the compiler from deleting or optimizing security-sensitive code sections, Raccoon marks security-sensitive functions, variables, and memory access operations as `volatile` (not shown in the transformed IR).[6]

At runtime, the transformed code executes the following steps:

1. Line 7 streams over the array and uses ORAM to load a single element (identified by `mid`) of the array.

2. Line 8 calculates the actual value of the branch predicate.

3. The key to this obfuscation lies in the `epilog()` function on line 15, which forces the transformed code to execute twice. The first time this function is called, it transfers control back to line 9. The second time this function is called, it simply returns, and program execution proceeds to other statements in the user's code.

4. Line 9 obfuscates the branch outcome. The first time the `obfuscate()` function returns, it stores 0 in `r3`, and control is transferred to the statement at line 13, where the `tx_write()` function call updates the transaction buffer. Non-transactional memory is updated only if this path corresponds to the real path.

   The second time the `obfuscate()` function returns, it stores 1 in `r3`, and control is transferred to the statement at line 11, again calling the `tx_write()` function to update the transaction buffer. Again, non-transactional memory is updated only if this path corresponds to the real path.

5 Security Evaluation

In this section, we first demonstrate that the control-flows and data-flows in obfuscated programs are correct and that they are independent of the secret value. Then, using type-rules that track information flows, we argue that Raccoon's own code does not leak secret information. We then illustrate Raccoon's defenses against termination channels by reasoning about exceptions in x86 processors. Finally, we evaluate Raccoon's ability to prevent side-channel attacks via the `/proc` filesystem.

5.1 Security of Obfuscated Code

In this section, we argue that the obfuscated control-flows and data-flows (1) preserve the original program's

[6]The C99 standard states that any "any expression referring to [a volatile object] shall be evaluated *strictly* according to the rules of the abstract machine", and the abstract machine is defined in a manner that considers that "issues of optimization are irrelevant".

dependences and (2) do not reveal any secret information. We only describe scalar loads and stores, since all array-loads and array-stores are obfuscated by simply streaming over the array. To simplify the explanation, the following arguments describe a top-level (*i.e.* a non-nested) branch. The same arguments can be extended to nested branches by maintaining a stack of transaction buffers.

Correctness of Obfuscated Data-Flow. To ensure correct data-flow, Raccoon uses a combination of transaction buffers and non-transactional storage (*i.e.* main memory). Raccoon sets up a fresh transaction buffer for each thread that executes a new path. Figure 4 shows the implementation of buffered `load` and `store` operations for use with transactions. The `store` operations along real paths write to both transaction buffers and non-transactional storage (since threads cannot share data that is stored in thread-local transaction buffers).

Consider a non-obfuscated program that stores a value to a memory location *m* in line 10 and loads a value from the same location in line 20. We now consider four possible arrangements of these two `load` and `store` operations in the obfuscated program, where each operation may reside either inside or outside a transaction. Our goal is to ensure that the `load` operation always reads the correct value, whether the correct value resides in a transactional buffer or in non-transactional storage.

- **`store` outside transaction, `load` inside transaction:** This implies that there is no `store` operation on *m* within the transaction. Thus, the transaction buffer does not contain an entry for *m*, and the `load` operation reads the value from the non-transactional storage.
- **`store` inside transaction, `load` inside transaction:** Since the transaction has previously written to *m*, the transaction buffer contains an entry for *m*, and the `load` operation fetches the value from the transaction buffer.
- **`store` inside transaction, `load` outside transaction:** This implies that the `store` operation must lie along the real path. Real-path execution updates non-transactional storage. Since `load` operations outside of transactions always fetch from non-transactional storage, the `load` operation reads the correct value of *m*.
- **`store` outside transaction, `load` outside transaction:** Raccoon does not change `load` or `store` operations that are located outside of the transactions. Hence the non-obfuscated reaching definition remains unperturbed.

Raccoon correctly obfuscates multi-threaded code as well. In programs obfuscated by Raccoon, decoy paths only update transactional buffers. Thus, only the `store` operations on real path affect reaching definitions of the obfuscated program. Furthermore, `store` (or `load`) operations along real path immediately update (or fetch) non-transactional storage and do not wait until the transaction execution ends. Thus, memory updates from execution of real paths are immediately visible to all threads, ensuring that inter-thread dependences are not masked by transactional execution. Finally, all transactional `load` and `store` operations use locks to ensure that these accesses are atomic. Put together, `load` and `store` operations on real paths are atomic and globally-visible, whereas `store` operations on decoy paths are only locally-visible and get discarded upon transaction termination. We thus conclude that the obfuscated code maintains correct data-flows for both single- and multi-threaded programs.

Concealing Obfuscated Data-Flow. Raccoon always performs two `store` operations for every transactional write operation, regardless of whether the write operation belongs to a real path or a decoy path. Moreover, by leveraging the oblivious `store` operation, Raccoon hides the specific value written to the transactional buffer or to the non-transactional storage. Although the `tx_read()` function uses an `if`-statement, the predicate of the `if`-statement is not secret, since an adversary can simply inspect the code and differentiate between repeated and first-time memory accesses. Thus, we conclude that the data-flows exposed to the adversary do not leak secret information.

Concealing Obfuscated Control-Flow. Raccoon converts control flow that depends on secret values into static (*i.e.* deterministically repeatable) control-flow that does not depend on secret values. Given a conditional branch instruction and two branch targets in the LLVM Intermediate Representation (IR), Raccoon always forces execution along the first target and then the second target. Thus, the sequence of executed branch targets depends on the (static) encoding of the branch instruction and not on the secret predicate.

5.2 Security of Obfuscation Code

Raccoon's own code should never leak secret information, so in this section, we demonstrate the security of the secret information maintained by Raccoon. Because the Raccoon code exposes only a handful of APIs (Table 1) to user applications, we can perform a detailed analysis of the code's entry- and exit-points to ensure that these

$$\text{T-LOAD}\quad \frac{l_r(p) = \mathtt{L}, A = pts(p), m = \max_{a \in A} l_a(a)}{\langle x = \textbf{load} p; c, l_a, l_r\rangle \rightarrow \langle c, l_a, l_r[x \mapsto m]\rangle}$$

$$\text{T-STORE}\quad \frac{l_r(p) = \mathtt{L}, A = pts(p)}{\langle \textbf{store}(x,p); c, l_a, l_r\rangle \rightarrow \langle c, \bigcup_{a \in A} l_a[a \mapsto max(l_a(a), l_r(x))], l_r\rangle}$$

$$\text{T-BINOP}\quad \frac{}{\langle v = \textbf{binary-op}(x,y); c, l_a, l_r\rangle \rightarrow \langle c, l_a, l_r[v \mapsto \textbf{max}(l_r(x), l_r(y))]\rangle}$$

$$\text{T-UNOP}\quad \frac{}{\langle v = \textbf{unary-op}(x); c, l_a, l_r\rangle \rightarrow \langle c, l_a, l_r[v \mapsto l_r(x)]\rangle}$$

$$\text{T-BRANCH}\quad \frac{l_r(p) = \mathtt{L}, \langle c_t; c, l_a, l_r\rangle \rightarrow \langle c, l_a{}', l_r{}'\rangle, \langle c_f; c, l_a, l_r\rangle \rightarrow \langle c, l_a{}'', l_r{}''\rangle}{\langle \textbf{branch}(p, c_t, c_f); c, l_a, l_r\rangle \rightarrow \langle c, M(l_a{}', l_a{}''), M(l_r{}', l_r{}'')\rangle}$$

$$\text{T-CMOV}\quad \frac{}{\langle v = \textbf{cmov}(p,t,f); c, l_a, l_r\rangle \rightarrow \langle c, l_a, l_r[v \mapsto \mathtt{L}]\rangle}$$

$$\text{T-SKIP}\quad \frac{}{\langle v = \textbf{skip}; c, l_a, l_r\rangle \rightarrow \langle c, l_a, l_r\rangle}$$

$$\text{T-SEQUENCE}\quad \frac{\langle c_0, l_a, l_r\rangle \rightarrow \langle c_0{}', l_a{}', l_r{}'\rangle}{\langle c_0; c_1, l_a, l_r\rangle \rightarrow \langle c_0{}'; c_1, l_a{}', l_r{}'\rangle}$$

$$M(l', l'') = \forall_{x \,\in\, \{K(l') \,\cup\, K(l'')\}}\ (x,\ max(l'(x), l''(x)))\qquad K(l) = \{x \mid (x,s)\ \in\ l\}$$

Figure 6: Typing rules and supporting functions that check security of Raccoon's code.

Category	Functions	Secret info.
Control-flow obfuscation.	`obfuscate()`, `epilog()`.	Predicate value
Wrapper functions for unsafe operations.	`stream_load()`, `stream_store()`, `div_wrapper()`.	Array index, division operands.
Registering stack and array information.	`reg_memory()`, `reg_stack_base()`.	-
Initialization and clean-up functions.	`init_handler()`, `exit_handler()`.	-

Table 1: Entry-points of Raccoon's library.

interfaces never spill secret information outside of Raccoon's own code.

Type System for Tracking Information Flows. Figure 6 shows a subset of the typing rules used for checking the IR of Raccoon's own code. These rules express small-step semantics that track security labels. We assume the existence of a functions $l_r : v \rightarrow \gamma$ and $l_a : \Delta \rightarrow \gamma$ that map LLVM's virtual registers (v) and addresses (Δ) to security labels (γ), respectively. Security labels can be of two types: **L** represents low-context (or public) information, while **H** represents high-context (or secret) information. Secret information listed in Table 1 is assigned the **H** security label, while all other information is assigned the **L** security label. We also assume the existence of a function $pts : r \rightarrow \{\Delta\}$ that returns the points-to set for a given virtual register r.

Our goal is to ensure that Raccoon does not leak secret information either through control-flow (`branch` instructions) or data-flow (`load` and `store` instructions). The typing rules in Figure 6 verify that information labeled as secret never appears as an address in a `load` or `store` instruction and never appears as a predicate in a `branch` instruction. Otherwise, the typing rules would result in a stuck transition. To prevent information leaks, Raccoon passes the secret information through the declassifier (`cmov`) before executing a `load`, `store`, or `branch` operation with a secret value. Due to its oblivious nature, the `cmov` operation resets the security label of its destination to **L**.

Security Evaluation of the `cmov` Operation. The tiny code size of the `cmov` operation (Figure 3) permits us to thoroughly inspect each instruction for possible information leaks. We use the Intel 64 Architecture Software Developer's Manual to understand the side-effects of each instruction.

Since the code operates on the processor registers only and never accesses memory, it operates within the (trusted) boundary of the sealed processor chip. The secret predicate is loaded into the `%1` register. The `mov` instruction in line 4 initializes the destination register with `t_val`. The `test` instruction at line 5 checks if `pred` is zero and updates the Zero flag (`ZF`), Sign flag (`SF`), and the Parity flag (`PF`) to reflect the comparison. The subsequent `cmovz` instruction copies `f_val` into the destination register *only if* `pred` is zero. At this point, `ZF`, `SF`, and `PF` still contain the results of the comparison. The `test` instruction at line 7 overwrites these flags by comparing known non-secret values.

Since none of the instructions ever accesses memory, these instructions can never raise a General Protection Fault, Page Fault, Stack Exception Fault, Segment Not Present exception, or Alignment Check exception. None of these instructions uses the `LOCK` prefix, so they will never generate an Invalid Opcode (#UD) exception. As per the Intel Software Developer's Manual, the above instructions cannot raise any other exception besides the ones listed above. Through a manual analysis of the descriptions of 253 performance events[7] supported

[7]Intel 64 and IA-32 Architectures Software Developers Manual, Section 19.5.

by our target platform, we discovered that only two performance events are directly relevant to the code in Figure 3: PARTIAL_RAT_STALLS.FLAGS_MERGE_UOP and UOPS_RETIRED.ALL. The first event (FLAGS_MERGE_UOP), which counts the number of performance-sensitive flags-merging micro-operations, produces the same value for our code, no matter whether the predicate is true or false. The second event (UOPS_RETIRED.ALL) counts the number of retired micro-operations. Since details of micro-operation counts for x86 instructions are not publicly available, we used an unofficial source of instruction tables[8] to verify that the micro-operation count for a cmov instruction is independent of the instruction's predicate. We thus conclude that the code in Figure 3 does not leak the secret predicate value.

Category	Interrupt list
Arithmetic errors	Division by zero, invalid operands, overflow, underflow, inexact results.
Memory access interrupts	Stack exception fault, general protection fault, page fault.
Debugging interrupts	Single-step, breakpoint.
Privileged operations	Invalid TSS, segment not present.
Coprocessor (legacy) interrupts	No coprocessor, coprocessor overrun, coprocessor error.
Other	Non-maskable interrupt, invalid opcode, double-fault abort.

Table 2: Categorized list of x86 hardware exceptions.

5.3 Termination Leaks

In Section 4.7, we explained how Raccoon patches division operations and memory access instructions to prevent the program from crashing along decoy paths. We now explain why these patches are sufficient in preventing the introduction of new termination leaks. Table 2 shows a categorized list of exception conditions arising in Intel x86 processors[9] that may terminate programs. Among these interrupts, Raccoon transparently handles arithmetic and memory access interrupts.

Debugging interrupts are irrelevant for the program safety discussion because they do not cause the program to terminate. Our threat model does not apply obfuscation to OS or kernel code. Since we do not expect user programs to contain privileged instructions, Raccoon does not need to mask interrupts from privileged operations. Coprocessor interrupts are relevant to Numeric Processor eXtensions (NPX), which are no longer used today. Non-maskable interrupts are not caused by software events and thus need not be hidden by Raccoon. Branches in Raccoon always jump to the start of valid basic blocks, so invalid opcodes can never occur in an obfuscated version of a correct program. A double-fault exception occurs when the processor encounters an exception while invoking the handler for a previous exception. Aborts due to double-fault need not be hidden by Raccoon because none of the primary exceptions in an obfuscated program will leak secret information. In conclusion, Raccoon prevents abnormal program termination, thus guaranteeing that Raccoon's execution of decoy paths will never cause information leaks over the termination channel.

5.4 Defense Against Side-Channel Attacks

We have argued in Sections 5.1 and 5.2 that Raccoon closes digital side-channels. We now show a concrete example of a simple but powerful side-channel attack, and we use basic machine-learning techniques to visually illustrate Raccoon's defense against this attack. We model the adversary as a process that observes the instruction pointer (IP) values of the victim process. Both the victim process and the adversary process run on the same machine. The driver process starts the victim process and immediately pauses the victim process by sending a SIGSTOP signal. The driver process then starts the adversary process and sends it the process ID of the paused victim process. This adversary process polls for the instruction pointer of the victim process every 5ms via the kstkeip field in /proc/pid/stat. When the victim process finishes execution, the driver process sends a SIGINT signal to the adversary process, signalling it to save its collection of instruction pointers to a file. We run the victim programs with various secret inputs and each run produces a (sampled) trace of instruction pointers. Each such trace is labelled with the name of the program and an identifier for the secret input. We collect 300 traces for each label. For the sake of brevity, we show results for only three programs from our benchmark suite.

The labelled traces are then passed through a Support Vector Machine for k-fold cross-validation (we choose $k = 10$) using LIBSVM v3.18. Using the prediction data, we construct a confusion matrix for each program, which conveys the accuracy of a classification system by counting the number of correctly-predicted and mis-predicted values (see Figure 7). The confusion matrices show that for the non-secure executions, the classifier is able to label instruction pointer traces with high accuracy. By contrast, when using traces from obfuscated execution, the classifier's accuracy is significantly lower.

6 Performance Evaluation

Methodology. Raccoon is implemented in the LLVM compiler framework v3.6. In our test setup, the host op-

[8] http://www.agner.org/optimize/instruction_tables.pdf
[9] http://www.x86-64.org/documentation/abi.pdf

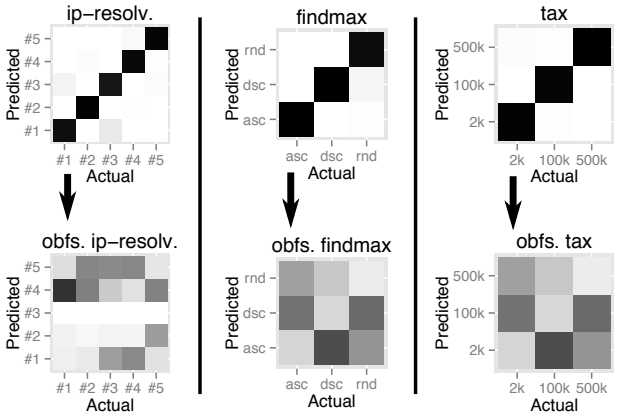


Figure 7: Confusion matrices for `ip-resolv`, `find-max` and `tax`. The top matrices describe original execution. The bottom matrices describe obfuscated execution.

Name	Lines	Data size
Classifier	86	5 features, 5 records
IP resolver	247	3,500 records
Medical risk analysis	92	3,200 records
CRC32	76	10 KB
Genetic algorithm	446	pop. size = 1 KB
Tax calculator	350	-
Radix sort	675	256K elements
Binary search	35	10K elements
Dijkstra	50	1K edges
Find max	27	1K elements
Heap add	24	1K elements
Heap pop	42	10K elements
Histogram	40	1K elements
Map	29	1K elements
Matrix multiplication	28	500 x 500 values

Table 3: Benchmark programs used for performance evaluation of Raccoon. The bottom eight programs are also used to evaluate GhostRider. The remaining seven programs cannot be transformed by GhostRider because these programs use pointers and invoke functions in the secret context.

erating system is CentOS 6.3. To evaluate performance, we use 15 programs (eight small kernels and seven small applications). Table 3 summarizes their characteristics and the associated input data sizes. The bottom eight programs in the table are the same programs used to evaluate GhostRider [20, 21], and we use these to compare Raccoon's overhead against that of GhostRider. To simplify the comparison between Raccoon and GhostRider, we use data sizes that are similar to those used to evaluate GhostRider [20]. Raccoon uses the `__attribute__` construct to mark secret variables—which mandates that the input programs are written in C/C++. However the rest of Raccoon operates entirely on the LLVM IR and does not use any source-language features. Thus, Raccoon can easily be ported to work with any language that can be compiled to the LLVM IR. All tests use the LLVM/Clang compiler toolchain.

We run all experiments on a machine with two Intel Xeon (Sandy Bridge) processors and with 32 GB (8 × 4 GB) DDR3 memory. Each processor has eight cores with 256 KB private L2 caches. The eight cores on a processor chip share a 20 MB L3 cache. Streaming encryption/decryption hardware makes the cost of accessing memory from encrypted RAM banks almost the same as the cost of accessing a DRAM bank. The underlying hardware does not support encrypted RAM banks, but we do not separately add any encryption-related overhead to our measurements because the streaming access cost is almost the same with or without encryption.

Performance measurements of our simulated ORAM use the native hardware performance event—`UNHALTED_CORE_CYCLES`. We measure overhead using `clock_gettime()`. Our software Path ORAM implementation is configured with a block size of 64 bytes. Each node in the Path ORAM tree stores 10 blocks. The stash size is selected at ORAM initialization time and is set to $\frac{ORAM\ block\ count}{100}$ or 64 entries, whichever is higher.

6.1 Obfuscation Overhead

There are two main sources of Raccoon overhead: (1) the cost of the ORAM operations (or streaming) and (2) the cost of control-flow obfuscation (including the cost of buffering transactional memory accesses, the cost of copying program stack and CPU registers, and the cost of obliviously patching arithmetic and memory access instructions). We account for ORAM/streaming overhead over both real and decoy paths. Of course, the overhead varies with program characteristics, such as size of the input data, number of obfuscated statements, and number of memory access statements. Figure 8 shows the obfuscation overhead for the benchmark programs when compared with an aggressively optimized (compiled with `-O3`) non-obfuscated binary executable. The geometric mean of the overhead is $\sim$16.1×. Applications closer to the left end of the spectrum had low overheads due to Raccoon's ability to leverage existing compiler optimizations (if-conversion, automatic loop unrolling, and memory to register promotion). In most applications with high obfuscation overhead, a majority of the overhead arises from transactional execution in control-flow obfuscation.

6.2 Comparison with GhostRider

To place our work in the context of similar solutions to side-channel defenses, we compare Raccoon with the

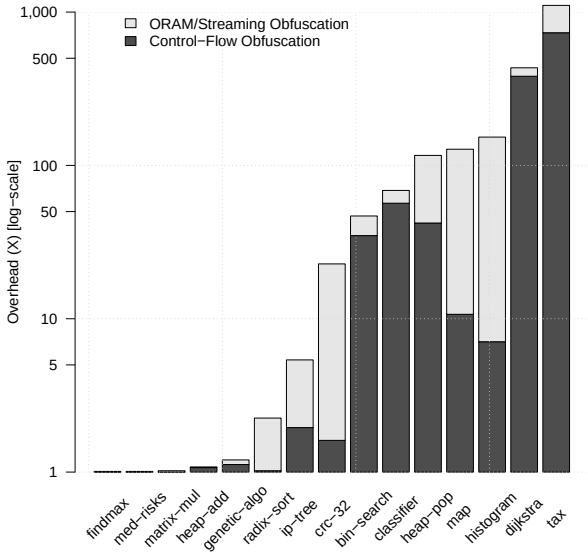


Figure 8: Sources of obfuscation overhead.

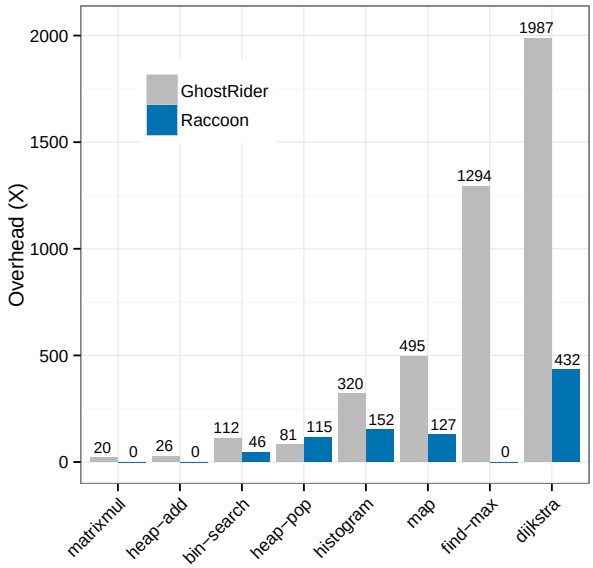


Figure 9: Overhead comparison on GhostRider's benchmarks. Even when we generously underestimate GhostRider's overhead, GhostRider sees an average overhead of 195×, while Raccoon's overhead is 21.8×.

GhostRider hardware/software framework [20, 21] that implements Memory Trace Obliviousness. This section focuses on the performance aspects of the two systems, but as mentioned in Section 2, Raccoon provides significant benefits over GhostRider beyond performance. First, Raccoon provides a broad coverage against many different side-channel attacks. Second, the dynamic obfuscation scheme used in Raccoon strengthens the threat model, since it allows the transformed code to be released to the adversary. Third, Raccoon does not require special-purpose hardware. Finally, since GhostRider adds instructions to mimic address traces in both branch paths, it requires that address traces from obfuscated code be known at compile-time, which significantly limits the programs that GhostRider can obfuscate. Raccoon relaxes this requirement by executing actual code, so Raccoon can transform more complex programs than GhostRider.

Methodology. We now describe our methodology for simulating the GhostRider solution. As with our Raccoon setup, we compare GhostRider's obfuscated program with an aggressively optimized (compiled with `-O3`) non-obfuscated version of the same program. Various compiler optimizations (dead code elimination, vectorization, constant merging, constant propagation, global value optimizations, instruction combining, loop-invariant code motion, and promotion of memory to registers) interfere with GhostRider's security guarantees, so we disable optimizations for the obfuscated program. We manually apply the transformations implemented in the GhostRider compiler. We simulate a processor that is modelled after the GhostRider processor, so we use a single-issue in-order processor that does not allow prefetching into the cache.

There are four reasons why our methodology significantly underestimates GhostRider's overhead. The first three reasons stem from our inability to faithfully simulate all features of the GhostRider processor: (1) We simulate variable-latency instructions, (2) we simulate the use of a dynamic branch predictor, and (3) we simulate a perfect cache for non-ORAM memory accesses. All three of these discrepancies give GhostRider an unrealistically fast hardware platform. The fourth reason arises because our simulator does not support AVX vector instructions, so we are unable to compare GhostRider against a machine that can execute AVX vector instructions.

The non-obfuscated execution uses a 4-issue, out-of-order core with support for Access Map Pattern Matching prefetching scheme [12] for the L1, L2 and L3 data caches. In all other respects, the two processor configurations are identical. Both processors are clocked at 1 GHz. The processor configuration closely matches the configuration described by Fletcher et al. [10], and based on their measurements, we assume that the latency to all ORAM banks is 1,488 cycles per cache line. We run GhostRider's benchmarks on this modified Marss86 simulator and manually add the cost of each ORAM access

to the total program execution latency.

Performance Comparison. Figure 9 compares the overhead of GhostRider on the simulated processor and the overhead of Raccoon. Only those benchmark programs that meet GhostRider's assumptions are used in this comparison. The remaining seven applications cannot be transformed by the GhostRider solution because they use pointers or because they invoke functions in the secret context. We see that Raccoon's overhead (geometric mean of 16.1× over all 15 benchmarks, geometric mean of 21.8× over GhostRider-only benchmarks) is significantly lower than GhostRider's overhead (geometric mean of 195×), even when giving GhostRider's processor substantial benefits (perfect caching, lack of AVX-vector support in the baseline processor, and dynamic branch prediction).

6.3 Software Path ORAM

This section considers choices for Raccoon's ORAM implementation. In particular, to run on typical general-purpose processors, we need to modify the Path ORAM algorithm to assume just a tiny amount of trusted memory, which forces us to stream the *position map* and *stash* multiple times to obliviously copy or update elements.

We thus consider three possible implementations. The first, recursive ORAM [33], places the *position map* in a smaller ORAM until the *position map* of the smallest ORAM fits in the CPU registers. The second is a non-recursive solution that streams over a single large *position map*. The third uses AVX intrinsic operations and streams over the entire array to access a single element.

Figure 10(a) compares the cost of ORAM initialization for different ORAM sizes in our recursive and non-recursive ORAM implementations. On this log-log scale, we see that the non-recursive ORAM is significantly faster than the recursive ORAM for all sizes. Figure 10(b) compares our non-recursive ORAM implementation against the streaming approach. In particular, it measures the cost of accessing a single element and the cost of 64 single-element random accesses using ORAM and streaming. We see that the streaming implementation is orders of magnitude faster than our non-recursive ORAM.

In summary, our software implementation of Path ORAM requires non-trivial changes to the original Path ORAM algorithm. Unfortunately, these changes impose a prohibitively large memory bandwidth requirement, making the modified software Path ORAM far costlier than streaming over arrays. Raccoon's obfuscation technique is compatible with the use of dedicated ORAM memory controllers, and Raccoon's overhead can be further reduced by using such special purpose hardware [22].

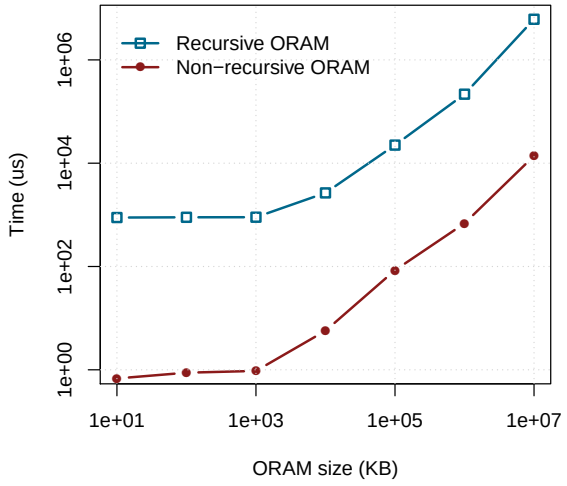
7 Discussion

Closing Other Side-Channels. The existing Raccoon implementation does not defend against kernel-space side-channel attacks. However, many of Raccoon's obfuscation principles can be applied to OS kernels as well. Memory updates in systems such as TxOS [28] can be made oblivious using Raccoon's `cmov` operation. By contrast, non-digital side-channels appear to be fundamentally beyond Raccoon's scope since physical characteristics (power, temperature, EM radiation) of hardware devices make it possible to differentiate between real values and decoy values.

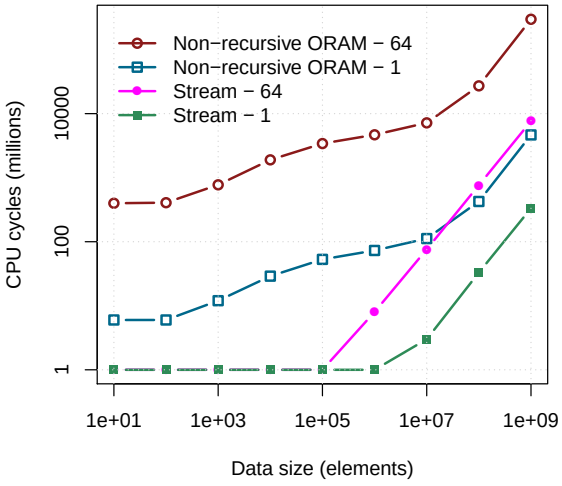
Multi-threaded Programs. Raccoon's data structures are stored in thread-local storage (TLS), so Raccoon can access internal data structures without using locks. Raccoon initializes these data-structures at thread entry-points (identified by `pthread_create()`) and frees them at thread destruction-points (identified by `pthread_exit()`). Raccoon prevents race conditions on the user program's memory by using locks where necessary. Most importantly, as long as the user program is race-free, Raccoon maintains the correct data-flow dependences in both single-threaded and multi-threaded programs, as described in Section 5.1.

Taint Analysis. Raccoon's taint analysis is sound but not complete, so it over-approximates the amount of code that must be obfuscated. For large programs, this over-approximation is a significant source of overhead. Raccoon's taint analysis is flow-insensitive, path-insensitive, and context-insensitive, and Raccoon uses a rudimentary alias analysis technique that assumes two pointers alias if they have the same type. We believe that more precise static analysis techniques can be used to greatly shrink Raccoon's taint graph, thus reducing the obfuscation overhead.

Limitations Imposed by Hardware. Various x86 instructions (`DIV`, `SQRT`, etc.) consume different cycles depending on their operand values. Such operand-dependent instruction execution latency introduces the biggest hurdle in ensuring the security of Raccoon-obfuscated programs. We also believe that the performance overhead of obfuscated programs would be substantially smaller than the current overhead if processors came equipped with (small) scratchpad memory. Based on these conjectures, we plan to explore the impact of modified hardware designs in the near future.



(a) Initialization cost of recursive and non-recursive ORAM implementation (median of 10 measurements for each sample).



(b) Performance comparison of software Path ORAM and streaming over the entire array.

Figure 10: Software ORAM performance.

8 Conclusions

In this paper, we have introduced the notion of digital side-channel attacks, and we have presented a system named Raccoon to defend against such attacks. We have evaluated Raccoon's performance against 15 programs to show that its overhead is significantly less than that of the best prior work and that it has several additional benefits: it expands the threat model, it removes special-purpose hardware, it permits the release of the transformed code to the adversary, and it also expands the set of supported language features. In comparing Raccoon against GhostRider, we find that Raccoon's overhead is 8.9× lower.

Raccoon's obfuscation technique can be enhanced in several ways. First, while the performance overhead of Raccoon-obfuscated programs is high enough to preclude immediate practical deployment, we believe that this overhead can be substantially reduced by employing deterministic or special-purpose hardware. Second, Raccoon's technique of transactional execution and oblivious memory update can be applied to the operating system (OS) kernel, thus paving the way for protection against OS-based digital side-channel attacks. Finally, in addition to defending against side-channel attacks, we believe that Raccoon can be strengthened to defend against covert-channel communication.

Acknowledgments. We thank our shepherd, David Evans, and the anonymous reviewers for their helpful feedback. We also thank Casen Hunger and Akanksha Jain for their help in using machine learning techniques and microarchitectural simulators. This work was funded in part by NSF Grants DRL-1441009 and CNS-1314709 and a gift from Qualcomm.

References

[1] Aciiçmez, O., Koç, c. K., and Seifert, J.-P. On the power of simple branch prediction analysis. In *Symposium on Information, Computer and Communications Security* (2007), pp. 312–320.

[2] Aciicmez, O., and Seifert, J.-P. Cheap Hardware Parallelism Implies Cheap Security. In *Workshop on Fault Diagnosis and Tolerance in Cryptography* (2007), pp. 80–91.

[3] Bao, F., Deng, R. H., Han, Y., A.Jeng, Narasimhalu, A. D., and Ngair, T. Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults. In *Workshop on Security Protocols* (1998), pp. 115–124.

[4] Blundell, C., Lewis, E. C., and Martin, M. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Tech. rep., University of Pennsylvania, 2006.

[5] Brumley, D., and Boneh, D. Remote timing attacks are practical. In *USENIX Security Symposium* (2005).

[6] Carlstrom, B. D., McDonald, A., Chafi, H., Chung, J., Minh, C. C., Kozyrakis, C., and Olukotun, K. The Atomos transactional programming language. In *Conference on Programming Language Design and Implementation* (2006), pp. 1–13.

[7] Checkoway, S., and Shacham, H. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Architec-*

tural Support for Programming Languages and Operating Systems (2013), pp. 253–264.

[8] CRANE, S., HOMESCU, A., BRUNTHALER, S., LARSEN, P., AND FRANZ, M. Thwarting cache side-channel attacks through dynamic software diversity. In *Network and Distributed System Security Symposium* (2015).

[9] FLETCHER, C. W., DIJK, M. V., AND DEVADAS, S. A Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *ACM Workshop on Scalable Trusted Computing* (2012), pp. 3–8.

[10] FLETCHER, C. W., LING, R., XIANGYAO, Y., VAN DIJK, M., KHAN, O., AND DEVADAS, S. Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *International Symposium on High Performance Computer Architecture* (2014), pp. 213–224.

[11] GANDOLFI, K., MOURTEL, C., AND OLIVIER, F. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems* (2001), pp. 251–261.

[12] ISHII, Y., INABA, M., AND HIRAKI, K. Access map pattern matching for high performance data cache prefetch. *Journal of Instruction-Level Parallelism* (2011), 499–500.

[13] JANA, S., AND SHMATIKOV, V. Memento: Learning secrets from process footprints. In *IEEE Symposium on Security and Privacy* (2012), pp. 143–157.

[14] KIM, T., PEINADO, M., AND MAINAR-RUIZ, G. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *USENIX Conference on Security Symposium* (2012), pp. 11–11.

[15] KOCHER, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology* (1996), pp. 104–113.

[16] KOCHER, P. C., JAFFE, J., AND JUN, B. Differential Power Analysis. In *Advances in Cryptology*. Springer Berlin Heidelberg, 1999, pp. 388–397.

[17] KONG, J., ACIICMEZ, O., SEIFERT, J.-P., AND ZHOU, H. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *High Performance Computer Architecture* (2009).

[18] KUHN, M. G. Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP. *IEEE Transactions on Computers 47*, 10 (1998), 1153–1157.

[19] LAMPSON, B. W. A note on the confinement problem. *Communications of the ACM* (1973), 613–615.

[20] LIU, C., HARRIS, A., MAAS, M., HICKS, M., TIWARI, M., AND SHI, E. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *Architectural Support for Programming Languages and Operating Systems* (2015), pp. 87–101.

[21] LIU, C., HICKS, M., AND SHI, E. Memory Trace Oblivious Program Execution. In *Computer Security Foundations Symposium* (2013), pp. 51–65.

[22] MAAS, M., LOVE, E., STEFANOV, E., TIWARI, M., SHI, E., ASANOVIC, K., KUBIATOWICZ, J., AND SONG, D. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *Conference on Computer and Communications Security* (2013), pp. 311–324.

[23] MARTIN, R., DEMME, J., AND SETHUMADHAVAN, S. TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *International Symposium on Computer Architecture* (2012), pp. 118–129.

[24] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C. V., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. R. Innovative instructions and software models for isolated execution. In *International Workshop on Hardware and Architectural Support for Security and Privacy* (2013).

[25] MOLNAR, D., PIOTROWSKI, M., SCHULTZ, D., AND WAGNER, D. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Information Security and Cryptology* (2006), pp. 156–168.

[26] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: the case of AES. In *RSA conference on Topics in Cryptology* (2006), pp. 1–20.

[27] PERCIVAL, C. Cache missing for fun and profit. In *BSDCan* (2005).

[28] PORTER, D. E., HOFMANN, O. S., ROSSBACH, C. J., BENN, A., AND WITCHEL, E. Operating system transactions. In *Symposium on Operating Systems Principles* (2009), pp. 161–176.

[29] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *Computer and Communications Security* (2009), pp. 199–212.

[30] SABELFELD, A., AND MYERS, A. C. Language-Based Information-Flow Security. *IEEE JSAC* (2003), 5–19.

[31] SCHINDLER, W. A timing attack against RSA with the chinese remainder theorem. In *Cryptographic Hardware and Embedded Systems* (2000), pp. 109–124.

[32] SHAMIR, A., AND TROMER, E. Acoustic cryptanalysis. Online at `http://www.wisdom.weizmann.ac.il/~tromer`.

[33] SHI, E., CHAN, T.-H. H., STEFANOV, E., AND LI, M. Oblivious RAM with O((log n)3) Worst-case Cost. In *International Conference on The Theory and Application of Cryptology and Information Security* (2011), pp. 197–214.

[34] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C., REN, L., YU, X., AND DEVADAS, S. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Conference on Computer and Communications Security* (2013), pp. 299–310.

[35] SUH, G. E., FLETCHER, C., CLARKE, D., GASSEND, B., VAN DIJK, M., AND DEVADAS, S. Author Retrospective AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing. In *International Conference on Supercomputing* (2014), pp. 68–70.

[36] THEKKATH, C., LIE, D., MITCHELL, M., LINCOLN, P., BONEH, D., MITCHELL, J., AND HOROWITZ, M. Architectural Support for Copy and Tamper Resistant Software. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2000), pp. 168–177.

[37] TIWARI, M., HUNGER, C., AND KAZDAGLI, M. Understanding Microarchitectural Channels and Using Them for Defense. In *International Symposium on High Performance Computer Architecture* (2015), pp. 639–650.

[38] VATTIKONDA, B. C., DAS, S., AND SHACHAM, H. Eliminating Fine Grained Timers in Xen. In *Cloud Computing Security Workshop* (2011), pp. 41–46.

[39] WANG, Z., AND LEE, R. B. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *International Symposium on Computer Architecture* (2007), pp. 494–505.

[40] WANG, Z., AND LEE, R. B. A novel cache architecture with enhanced performance and security. In *IEEE/ACM International Symposium on Microarchitecture* (2008), pp. 83–93.

[41] YEN, S.-M., AND JOYE, M. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers* (2000), 967–970.

[42] ZHANG, D., ASKAROV, A., AND MYERS, A. C. Predictive mitigation of timing channels in interactive systems. In *Conference on Computer and Communications Security* (2011), pp. 563–574.

[43] ZHANG, Y., JUELS, A., OPREA, A., AND REITER, M. K. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *IEEE Symposium on Security and Privacy* (2011), pp. 313–328.

[44] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-VM side channels and their use to extract private keys. In *Conference on Computer and Communications Security* (2012), pp. 305–316.

[45] ZHANG, Y., AND REITER, M. K. Duppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *Conference on Computer and Communications Security* (2013), pp. 827–838.

[46] ZHUANG, X., ZHANG, T., AND PANDE, S. HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus. In *Architectural Support for Programming Languages and Operating Systems* (2004), pp. 72–84.