Optimizing Memory Side-Channel Defenses through Amortized Execution

A Turing Scholar Honors Thesis by:
Gregory McDonald
gregorymcdonald@utexas.edu
May 25, 2018

Honors Thesis Committee:
Calvin Lin (Advisor)
Mohit Tiwari
Brent Waters

Abstract

Side-channel attacks observe the behavior of a program to infer the value of secret data. For instance, monitoring the addresses of load and store operations that interact with random-access memory can allow an adversary to gain secret information about the execution of a program. Compiler-based side channel defenses, such as Escort, access all memory locations within a region for every load and store operation, thus preventing an adversary from gaining information about the secret address. However, such defenses are prohibitively slow.

This paper presents a solution which improves the performance of Escort up to 2-4× depending on the data size. Our key insight is that we can eliminate redundancy in Escort’s secret memory accesses by reordering memory accesses while still maintaining security. We demonstrate that our transformations preserve the original function of the code and do not introduce new side channels.

1 Introduction

Even in the presence of encryption, isolation, and access control, side channel vulnerabilities, through which an adversary can infer secret information, can leak secret information such as encryption keys [6], private user information [25], and intellectual property [7].

To understand side channel attacks, consider the vulnerable code in Figure 1 that serves as a login program. Our example program accepts a password that is 10 characters long and composed of lower-case alphabets. This program compares the input password with a secret true password one character at a time and immediately rejects a password once a character does not match. There are 26 possible lowercase alphabets, and 10 possible positions, so there are $10^{26}$ unique passwords. Any adversary attempting to crack this password with brute force would have to attempt all $10^{26}$ possible passwords.

```c
01: string true_password = "abcdefghijklmnopqrstuvwxyz";
02: login(char* password) {
03:     for (int i = 0; i < 10; i++) {
04:         if (password[i] != true_password[i]) {
05:             return false;
06:         }
07:     }
08:     return true;
09: }
```

Figure 1: Vulnerable code for a login program. This function returns true if the input matches the true password; otherwise it returns false.

However, the execution time of this program can depend on the input password. For instance, if given inputs ‘aaaaaaaaaa’ and ‘bbbbbbbbbb’, the program will take more time to reject ‘aaaaaaaaaa’. This difference is due to an additional iteration of the for loop because the first character matches the true password. The first character of input ‘bbbbbbbbbb’ does not match the true password, and is rejected sooner. Thus we can see that execution time of the program provides clues about the correct password.

Using these clues, an adversary can determine the correct password far faster than with brute force. More precisely, an adversary does not need to try all $10^{26}$ possible unique passwords. Instead, the adversary can attempt only $10\times 26$ passwords. This type of attack is known as a timing side channel attack, because it exploits the execution time of a program to infer secret information. Furthermore, the code above is also vulnerable through other side channels such as instruction caches and data caches [12, 13] and branch predictors [2].

Multiple solutions exist that seek to close digital [16, 25, 12] and non-digital side channels [14]. Escort [16] is one such solution which closes all digital side channels using a single solution. However, Escort’s defense imposes up to a $10,000\times$ performance penalty because of redundant memory accesses. In particular, Es-
cort replaces all memory accesses that dereference a secret pointer with code that accesses the entire array for every load and store operation which results in significant redundancy. We refer to this behavior as a streaming operation.

In this solution, we significantly reduce the performance overhead of closing digital side channels. Our key insight is to reduce redundant memory accesses by coalescing streaming operations. For instance, instead of streaming over an array \( k \) times for \( k \) different secret accesses, our solution streams over the array once while fetching \( k \) values.

However, naively coalescing streaming operations has the potential to produce incorrect results due to dependencies. We address this problem by optimistically assuming independence between array accesses at compile-time, and resolving any occurring dependencies at runtime. Correctly and securely implementing such a transformation requires compile-time analysis and transformations, as well as microarchitecture support.

Effectively, our solution improves performance by accessing memory in chunks, thus reducing redundancy and amortizing the high cost of the streaming operation. Our solution’s performance depends on the program’s data sizes—the larger the data size, the higher the performance improvement.

The paper is organized as follows. Section 2 describes our threat model. We provide background in Section 3 before commenting on related work in Section 4, and presenting our solution in Section 5. We evaluate our solution in Section 6. Finally, we conclude and propose future in Section 7.

2 Threat Model

We now describe our threat model and our assumptions about the capabilities of the adversary.

Our solution prevents an attacker from gaining information about secret values by observing the digital side channels produced by a program. We assume that the adversary can observe the address and pattern of all accesses to off-chip resources (i.e. disk storage, random-access memory, cache, etc.). We assume the contents of memory are encrypted, and therefore the adversary cannot read the true value of any memory location.

We assume that the contents of the processor registers are trusted and cannot be read by the adversary. We assume that the compiler, assembler, and operating system are trusted. We assume that the adversary has access to the source code of the program being transformed, as well as access to the source code of our solution. Lastly, we assume that there exists an on-chip microarchitecture scratchpad memory that does not expose a memory address trace to the adversary. Despite the limited size of scratchpad memory, our solution’s security guarantees extend to arbitrarily-large data structures.

2.1 Restrictions on the Input Program

The Escort compiler, which our solution extends, accepts input programs that can be compiled using the LLVM compiler infrastructure [8]. The Escort compiler cannot transform recursive programs or programs that contain system calls. However, standard compiler techniques [11] can convert recursive programs to non-recursive programs, and system call usage can be eliminated or minimized by Library operating systems [3]. The Escort compiler, and by extension our solution, assumes that the input program does not contain inline assembly instructions, undefined behavior, race conditions, or code that can throw exceptions.

In addition to the restrictions placed by the Escort compiler, our solution is unable to transform programs where the index of the secret memory access is a loop-carried dependence. Our solution is also currently unable to transform any load, store, or load-store pair whose dependent instructions span multiple basic blocks. However, Escort can convert code containing branches to branchless code, which then can be transformed by our solution.

3 Background

We now describe the background information relevant to our solution.

3.1 Side Channels

A side channel is broadly defined as any way that a program exposes information that is not through its output. A side channel can be a physical symptom of computation, like the heat generated by a processor, or a digital effect of a program’s execution, like the amount of time it takes to for a program to terminate. Exploitation of physical side channels most often requires physical access to the machine running the program, but digital side channel attacks can be performed by any entity with the ability to execute their own malicious code within the same environment as the victim program.

In this paper we focus on memory address traces, which refer to which memory address locations are accessed (i.e. read or written) and the order they are accessed in.

3.2 Escort

Our solution extends Escort [16], which is a solution to defend digital side channel attacks. Escort modi-
fies memory accesses, math operations, and control flow to prevent information leakage through side channels. More precisely, Escort changes the control flow of the program to produce code which executes the same sequence of instructions regardless of secret values while maintaining its original functionality. Similarly, Escort also transforms memory accesses and floating-point operations so that the address trace and timing of the program are independent of any secret values.

Escort is built using the LLVM [8] compiler which provides an intermediate representation and a framework for modifying programs. Escort relies on this functionality, and is composed of a series of LLVM passes.

3.2.1 Escort’s Detection of Instructions that Leak Information

Escort detects instructions that may leak information through digital side channels by identifying which values are dependent on secret information through taint tracking [5]. Escort then determines a subset that may leak information through digital side channels, including load and store operations which use a secret value as the location operand. Our solution uses taint tracking to detect instructions that may leak information through the address-trace side channel which become candidates for our solution’s transformation.

3.2.2 Transformed Scalar Variable Access

Escort’s scalar variable accesses prevent the program from leaking whether any specific store operation succeeds. Escort transforms store operations to first load the value at the target location, and then a predicate determines if the updated value is written back to memory (see Figure 2).

```c
uint64_t result = 0;
__asm__ volatile (
    "mov %2, %0;"
    "test %1, %1;"
    "cmov %3, %0;"
    : "+r" (result)
    : "r" (predicate & 0x1), "r" (t_val), "r" (f_val)
    : "cc"
); return result;
```

Figure 2: Code for the conditional move operation. This function returns t_val when predicate is true; otherwise it returns f_val. This operation does not leak information over digital side channels.

This operation copies the contents of one memory location to another memory location if the given condition is true. If the given condition is false, the operation gives the appearance of reading and writing to the location but the value is unchanged. However, regardless of the condition, this operation consumes a fixed amount of time, executes the same set of instructions, and does not access application memory.

3.2.3 Redundancy in Escort’s Array Access

The memory instruction transformations Escort uses to access arrays avoid leaking information to the adversary about the secret pointer. To accomplish this, code transformed by Escort accesses every index in the array for each load and store operation, in an operation we refer to as streaming. The Escort array operations use the scalar memory access instructions shown in Figure ?? to read or write only the index of the array, but maintains the appearance of reading or writing to every index within the array (see Figure 3).

```c
int[] array;
int secret_index = 10;
// Array Access
int x = array[10];
// Escort Array Access (equivalent to above)
int x;
for (int i = 0; i < array.length; i++) {
    uint8_t predicate = i == secret_index;
    x = cmove(predicate, array[i], x);
}
```

Figure 3: A code comparison between a standard non-secure array access and a secured array access generated by Escort.

To stream across the entire array on every access is prohibitively expensive. Figure 4 shows an example.

```c
int[] values = new int[10000];
...  
int[] secret_indices = { 0, 1, 2, 3, 4 }; // Secret.
int sum = 0;
for (int i = 0; i < 5; i++) {
    uint8_t predicate = i == secret_index;
    int secret_index = secret_indices[i];
    int value = values[secret_index]; // This leaks.
    sum += value;
}
return sum;
```

Figure 4: A code sample that computes the sum of values determined by secret indices.

The example code above computes the sum of five values. The values are determined by memory accesses using secret indices, and thus may leak information through digital side channels. Escort’s defense replaces each
load operation with a stream over the entire values array, and thus the transformed code streams over the values array five separate times. For large array sizes, or for applications that load and store from memory often, the performance impact of this transformation renders the resulting code impractically slow.

4 Related Work

Prior research has yielded many solutions that close side channels. These defenses are compiler-based [16, 10, 15] or use novel microarchitecture behavior [12, 24]. However, the majority of these defenses only close a limited number of side channels, and can leave vulnerabilities to others. For example, various solutions exist that close the memory address trace side channel [21, 9, 18, 17], but do not completely close the timing side channel, nor the control flow side channel, and therefore are still vulnerable to side channel attacks.

Escort [16], the work we expand upon in this paper, closes a broad class of side channels with a single solution. However, the broad nature of Escort’s defense is detrimental to its performance. This performance penalty makes Escort’s transformations infeasibly slow for common programs. This paper also relies on static taint analysis [5] to detect instructions which may leak memory through the address trace side channel.

The design of our solution uses an optimization technique referred to as loop tiling or loop blocking [1, 22]. Applications of this technique have been used to improve the performance of image processing [1], parallel execution [23], cache temporal locality [20], dataflow programming [4], and sparse tensor-matrix multiplication [19]. In these applications, loop tiling is used to increase the locality of cache accesses, therefore reducing expensive cache misses and improving performance. Similar to prior work, our solution also uses tiling to increase the locality of memory access operations. However, our solution assumes fewer dependencies at compile time and resolves detected dependencies at runtime, thus being applicable to a broader set of applications. Additionally, our solution has the added requirement of maintaining Escort’s security guarantees when introducing the loop tiling optimization.

5 Design

We now give an description of the design of our solution.

5.1 Overview

Our solution is composed of (1) dependence analysis to determine what computation is dependent on instructions that may leak information through the address-trace side channel, and (2) a runtime function that replaces individual memory accesses with secure chunk memory accesses while preserving the original functionality of the code. We demonstrate our methodology and motivation through analysis of the example shown in Figure 5.

```c
int[] histogram;
histogram_update(int[] secret_indices) {
    for (int i = 0; i < secret_indices.length; i++) {
        int secret_index = secret_indices[i];
        histogram[secret_index] += 1;
    }
}
```

Figure 5: A code sample which increments the value of a secret bucket in the histogram array.

The Escort compiler transforms the code in Figure 5 so that each loop iteration streams over the entire histogram array twice, once to load the value of histogram[secret_index] and again to store the updated value. However, such repeated streaming introduces significant redundancy in memory access operations since the transformed code must access the entire array for every secret access.

Our solution eliminates redundancies by coalescing multiple memory access operations into one streaming operation. Given k buckets that must be accessed in the histogram array, the transformed code can perform all load and store operations to the k buckets in one streaming operation. Effectively the transformed code performs k load operations, followed by k computations, followed by k store operations. Since the k load and store operations can be performed in one streaming operation, the transformed code can be expected to perform O(k) times faster.

However, naively implementing such a transformation can produce incorrect results due to dependence between loop iterations. In the example in Figure 6 we expand on the code shown in Figure 5 to show a simple case which produces incorrect output under naive independence assumptions.

```c
histogram = {10, 20};
int[] secret_indices = {0, 0};
histogram_update(secret_indices);
// Expected Result: histogram == [12, 20];
// Actual Result: histogram == [11, 20];
```

Figure 6: A code sample which produces incorrect output under naive independence assumptions.

The actual result shown in Figure 6 differs from the expected result because of the ordering of the load, compute, and store operations. All load operations are
completed before any computation is executed, therefore all load operations on index 0 will return 10. The result of histogram update on the value in bucket 0 is 11, which is then stored twice into bucket 0 resulting in an incorrect answer of 11. Our solution addresses this issue at runtime, which we describe in Section 5.4.

5.2 Case-by-case Transformation

We now describe the specific patterns of memory access that our solution transforms.

5.2.1 Store

We now describe the transformation of a store instruction that may leak information through digital side channels. Figure 7 shows a program that contains such a store instruction.

```c
int[] secret_indices;
int[] values;
...
for (int i = 0; i < 10; i++) {
    int secret_index = secret_indices[i];
    // Store instruction.
    values[secret_index] = 0;
}
```

Figure 7: A program containing a store instruction that may leak information through digital side channels.

The store instruction in Figure 7 is an array access that uses a secret value as an index, and therefore may leak information. A store instruction returns no value, so the set of dependent instructions is trivially empty, and no computation need be delayed. However, in order to be able to perform a chunk store operation the index and value of this store operation must be remembered until a configurable number of delayed operations have been reached, and a stream operation occurs. Pseudocode resulting from the transformation of the code in Figure 7 is shown in Figure 8.

Our solution’s transformation removes the store instruction and replaces it with instructions that remember the necessary operands. Our solution’s transformation also adds instructions to perform chunk memory access once a configurable number of delayed operations has been reached.

5.2.2 Load

We now describe the transformation of a load instruction that may leak information through digital side channels. Figure 9 shows a program that contains such a load instruction.

```c
int[] secret_indices;
int[] values;
// How many delayed operations until a stream operation.
int request_size = 5;
int free_index = 0;
int[] values_to_store;
int[] store_indices;
...
for (int i = 0; i < 10; i++) {
    int secret_index = secret_indices[i];
    // Remember the index of the store.
    store_indices[free_index] = secret_index;
    // Remember the value of the store.
    values_to_store[free_index] = 0;
    free_index++;
    // If the number of delayed operations has been reached, perform the stream operation.
    if (free_index == (request_size - 1)) {
        stream_store(values, store_indices, values_to_store);
        free_index = 0;
    }
}
stream_store(values, store_indices, values_to_store);
stream_store(int[] array, int[] indices, int[] values) {
    // Stream over the entire array and store only the input values in their corresponding indices.
}
```

Figure 8: Pseudocode resulting from our solution’s transformation of the code in Figure 7. stream_store is a special case of the runtime function described in Section 5.4.

The load instruction in Figure 9 is an array access that uses a secret value as an index, and therefore may leak information. A load instruction does return a value; in the example this is the variable loaded_value. The set of dependent instructions includes all instructions that depend on the returned value of the load operation. The computation that depends on the value of loaded_value is 'sum += loaded_value', and must be delayed. To perform a chunk store operation the index of the load operation must be remembered until a configurable number of delayed operations have been reached, and a stream operation occurs. Pseudocode resulting from the transformation of the code in Figure 9 is shown in Figure 10, this pseudocode omits the handling of dependent computation.

Our solution’s transformation removes the load operation and the computation dependent on the load operation. Our solution’s transformation replaces the removed load operation with instructions that remember the necessary operands, and inserts instructions that perform chunk memory access once a configurable number of delayed operations has been reached.
int[] secret_indices; // Secret.
int[] values;
...
int sum = 0;
for (int i = 0; i < secret_indices.length; i++) {
    int secret_index = secret_indices[i];
    // Load instruction.
    int loaded_value = values[secret_indices];
    sum += loaded_value;
}

5.3 Microarchitecture Changes

In addition to compile-time transformations, our solution also relies on microarchitecture support of on-chip memory controlled by the instruction-set architecture (ISA). We refer to this on-chip memory as scratchpad memory. Scratchpad memory contains only a small amount of memory, but memory access in the scratchpad does not leak information, as scratchpad memory access does not leave the processor. The motivation for the requirement of scratchpad memory is that the runtime function employed by our solution to perform chunk memory access will leak information if the memory it uses is stored off-chip. We implement scratchpad memory into our solution by using a set of custom pseudo-instructions in the Gem5 processor simulator to mimic the function of a physical scratchpad memory system.

5.4 Runtime Function

In order to improve performance, our solution modifies the behavior of Escort’s memory address trace side channel defense. Instead of streaming over the entire array on each array access, our solution uses a runtime function that performs a single stream operation to handle multiple secret indices in one chunk. Our runtime function accepts as input an array address, a collection of indices, and information about computation dependent on transformed memory access operations. This function is composed of four distinct components: (1) sort input indices into ascending order, so they may all be accessed in one streaming operation; (2) load values from the input indices, if necessary; (3) perform computation dependent on the loaded values; and (4) store the resulting values into the input indices. The memory access is chunked by introducing code transformations that accumulates indices that may leak information, and then executing them in one stream once a configurable amount of operations has been accumulated. To help understand the behavior of our solution’s runtime function Figure 11 shows an example use.

6 Evaluation

We now demonstrate the security of our solution and test its performance using the Gem5 microarchitecture simulator and several sample programs.

6.1 Experimental Setup

We run all experiments on a 4-core Intel Core i7-4770K processor. The processor is clocked at 3.50 GHz. Each core on this processor has a 32 KB private L1 data cache, a 32 KB private L1 instruction cache, and a 256 KB private L2 cache. A single 8 MB L3 cache is shared among all four cores. The host operating system is Ubuntu 16.04 running kernel version 4.13. We implement compiler transformations using the LLVM compiler framework [8] version 4.0. We gather cycle counts using the Gem5 microarchitecture simulator that models 1 GHz out-of-order x64, ARM32, and ARM64 processors with 32 KB L1 instruction and data caches, a 256 KB L2 cache, and an 8 MB L3 cache.
int main() {
    // Inputs to the runtime function.
    int[] array = {0, 10, 20, 30, 40};
    int[] secret_indices = {3, 1, 0, 4};
    int (*computation)(int) = int increment(int value) {
        return value + 1;
    }
    runtime_function(array, secret_indices, computation);
    // Expected Result: array == [1, 11, 20, 31, 40]
}

// Simplified form of the runtime function.
void runtime_function(int array[], int secret_indices[],
    int (*computation)(int)) {
    // (1) Sort secret indices, and remove duplicates.
    sort(secret_indices);
    // secret_indices == [0, 1, 3, 4]
    // (2) Load the array contents at each secret index.
    int[] loaded_values = stream_load(array, secret_indices);
    // loaded_values == [0, 10, 30, 40]
    // (3) Perform computation on loaded values.
    int[] computed_values = compute(loaded_values, computation);
    // computed_values == [1, 11, 31, 41]
    // (4) Store the computed values.
    stream_store(array, secret_indices, computed_values);
    // array == [1, 11, 20, 31, 40]
}

Figure 11: An example execution of the runtime function
our solution uses to perform chunk memory access.

6.2 Security Evaluation

In this section we evaluate whether the compile-time
transformations introduced by our solution prevent an
adversary from gaining information from the memory
address trace side channel. We also evaluate whether
our solution runs in fixed time, and therefore does not
leak information through a timing side channel. We can
precisely evaluate digital side channel leakage using
our compiler transformation, but the compiler has no
information about the timing of instructions. In order
to perform our timing channel evaluation, we test
our transformed programs using a microarchitecture
simulator.

6.2.1 Memory Address Trace Security Assurance

In order to assure that our transformations produce code
that does not allow an adversary to gain information by
the memory address trace side channel we demonstrate
that the programs produced by our solution do not ac-
cess memory in a different pattern when the secret value
varies. To this end, we created a LLVM pass to instru-
ment our transformed code and record the location and
timing of each access to memory. We then ran this instru-
mented code and randomly selected a new secret value
for each execution. We compared the record of memory
accesses across all different executions and found them
to be identical. We observe that our solution produces
code which does not allow an adversary to infer infor-
mation about the secret value from the memory address
trace side channel.

6.2.2 Timing Security Assurance

In order to assure that our transformations produce code
that does not allow an adversary to learn secrets through
the timing side channel we demonstrate that the code
produced by our solution runs in a fixed time for vary-
ing values of secret information. To accomplish this we
executed sample programs transformed by our solution
on the Gem5 microarchitecture simulator. We executed
each sample program multiple times, and provided dif-
f erent, randomized secret input each execution. We com-
pared the cycle count of each execution and found them
to be identical. We observe that our solution produces
code which does not take a variable amount of time to
complete depending on input, and thus does not leak in-
formation through the timing side channel.

6.3 Performance Evaluation

We now evaluate the end-to-end application performance
impact of our solution’s transformations to close the
memory address trace side channel and compare it to the
state of the art.

We use a sample program similar to
histogram_update shown in Figure 5 and the Gem5
microarchitecture simulator. We transform the sample
program using Escort and our solution. We generate
several transformations of the sample program using our
solution—each with a different value of M, where M is
the size of the memory chunk operation buffer. We then
execute the transformed sample programs within Gem5
and record the number of processor cycles until the
program terminates. We execute the transformed sample
programs using randomly-generated inputs of varying
size.

Figure 12 shows that our solution performs better than
Escort for large number of input elements. We expect our
solution to perform better yet than Escort as input size
increases, since the growth of Escort’s performance time
is roughly exponential while the growth of our solution’s
performance time is roughly linear.

Figure 12 also shows that our solution’s performance
cost decreases as the size of the memory chunk buffer
decreases (M). We attribute this to the cost of the sorting operation used by our runtime function to perform chunk memory access. Our solution currently uses a sorting operation that is $O(n^2)$, and therefore larger sizes of M produce an exponentially increasing performance penalty.

We are currently in the process of evaluating our solution using other benchmark applications.

7 Conclusion and Future Work

Our solution’s compiler-level transformations and chunk memory access capability provide a more performant protection against digital side channel attacks. The addition of amortized memory access through runtime dependence analysis progresses Escort towards being able to practically transform common programs. The work in this paper is limited to a subsection of all possible cases, but the capability of our solution has advanced the overall state of the project. Ideally, once the proposed future work is completed, our solution will prove additionally useful in reducing the performance impact of Escort’s instrumentations.

The most present next steps for this work are as follows: to improve the ability to detect code that may leak information through the memory address trace side channel and to perform dependence analysis that correctly navigates control flow. The aim of both these improvements is to expand the cases that can be secured by our solution. In our current solution, an access to memory with the potential to leak secret information is not detected nor transformed if it is not contained within one basic block. In addition, our current solution does not contain the level dependence analysis necessary to transform code where the dependencies of an instruction cross basic block boundaries, as often occurs when conditional code is contained within a loop. This limitation hampers our solution’s applicability to larger, more complex, programs. There is no fundamental obstacle preventing the development of these features, and our solution has the potential to grow in scope if work is continued.

8 Acknowledgements

This paper would not have been possible without the continued help of my academic advisor, Dr. Calvin Lin, and his student Ashay Rane. Dr. Lin and Ashay have conducted significant work on side channel defense and without their prior work on Escort and Raccoon, as well as their advice, my work would have no ground to build upon.

References


