Cape: Compiler-Aided Program Transformation for HTM-Based Cache Side-Channel Defense

Rui Zhang  
Ohio State University  
USA

Michael D. Bond  
Ohio State University  
USA

Yinqian Zhang  
Southern University of Science and Technology  
China

Abstract
Cache side-channel attacks pose real threats to computer system security. Prior work called Cloak leverages commodity hardware transactional memory (HTM) to protect sensitive data and code from cache side-channel attacks. However, Cloak requires tedious and error-prone manual modifications to vulnerable software by programmers. This paper presents Cape, a compiler analysis and transformation that soundly and automatically protects programs from cache side-channel attacks using Cloak’s defense. An evaluation shows that Cape provides protection that is as strong as Cloak’s, while performing competitively with Cloak.


Keywords: cache side-channel defense, compiler analysis and transformation, hardware transactional memory

1 Introduction
In today’s computing infrastructures, physical computing resources such as memory and processor caches are shared among multiple programs that run in parallel on the same physical machine. This situation enables side-channel attacks in which malicious users exploit implementation-specific hardware features to spy on other users’ executions and infer sensitive information. One of these exploits is cache side-channel attacks [6, 23, 30, 35–37, 50], which exploit cache behavior to acquire sensitive information.

Cache side-channel attacks may take place when the attacker program shares the same processor cache, or more specifically, the same cache sets (in Prime-Probe attacks [36]) or the same cache lines (in Flush+Reload attacks [50]), with the victim program. By repeatedly manipulating the states of the shared cache sets or cache lines, the attacker program forces the victim to leave measurable execution traces in the cache. As a result, the attacker can observe secret-dependent memory access patterns of the victim, which leads to complete or partial leakage of the victim’s secrets.

Prior work performs program analysis on application source code or binaries, to identify and mitigate side-channel vulnerabilities [17, 18, 38]. Recent work transforms programs to execute the same instructions for both outcomes of every secret-dependent condition [12, 41, 47]. Although tools like Constantine [12] can protect programs containing indirect memory accesses, provided that enough static information is available to the compiler, that is not always the case.

Cloak [22] is a software-only solution that leverages commodity hardware transactional memory (HTM) [24, 26], such as Intel’s TSX [51], to mitigate cache side-channel attacks. Cloak exploits the fact that transaction read and write sets are implemented in caches; it executes secret-sensitive code in transactions and deterministically preloads memory accesses at transaction start that might reveal secrets. An attacker program cannot observe secret-dependent memory accesses later in the transaction because evictions of accessed lines will abort the transaction.

HTM thus enables a generic and effective means of protecting software against cache side-channel attacks. It is generic, because tools like Cloak can be applied to any program, and it is effective as it invalidates the fundamental assumption of known cache side-channel attacks. While Cloak’s HTM-based defense is promising, it requires human effort to modify programs to protect sensitive data and code in transactions, which is tedious and error prone for complex programs. Therefore, to practically apply Cloak and other HTM-based side-channel protection [15, 40], compiler-assisted tools must be developed to automate the modification of software programs so that sensitive data and code are protected by transactions.

To address this problem, this paper proposes novel compiler support for automated cache side-channel protection, called Cape. (Cape is an acronym for Compiler analysis for protecting executions.) Cape consists of program analysis and instrumentation techniques that identify sensitive data and code, delimit transactions, and insert code to preload...
We evaluate Cape and Cloak on programs that have known side-channel vulnerabilities. A simulation-based evaluation shows that Cape provides protection that is as strong as Cloak’s. An evaluation on real hardware shows that Cape incurs slowdowns that are usually comparable to Cloak’s. Both Cape and Cloak add transactions that are too large to succeed on current hardware, motivating future HTM implementations that provide larger transaction capacity. Cape’s efficient and effective automation of Cloak’s defense represents an improvement in the state of the art.

2 Background and Related Work
This section overviews existing cache side-channel attacks and defenses, as well as transactional memory.

2.1 Cache Side-Channel Attacks
Cache side-channel attacks exploit the timing channels created in processor caches to steal sensitive information. A category of cache side-channel attacks of particular interest is so-called access-driven attacks, where the attacker program and the victim program run on the same physical machine and hence share the same processor. This type of cache side channel is a critical attack vector between programs sharing desktop computers, mobile phones, or cloud servers.

In order to leak information from a shared cache, the attacker program repeatedly prepares the shared cache into a known state and then waits for the victim program to execute a piece of code such that its execution will alter the states of the shared cache. The attacker program can then measure the altered states of the cache using timing channels, and thereby infer the memory access patterns of the victim program. There are two categories of techniques for the attacker program to manipulate the cache states:

- **Flush+Reload** [8, 10, 23, 49, 50, 53]: When physical memory is shared between the two programs (e.g., pages of a shared library), the attacker program can prepare the cache states by using the unprivileged clflush instruction to evict a specific line out of the shared cache, and later measure the access latency of the same line to determine if it has been fetched back to the cache by the victim program. Shorter latency suggests the line has been fetched into the cache.

- **Prime+Probe** [5, 28, 31, 35–37, 42, 52]: When the attacker and victim programs do not share physical memory, they still share the same cache sets. Therefore, to prepare the cache states, the attacker program fills a specific cache set (or multiple cache sets) by reading applicable cache lines. To measure the altered cache states, the attacker program reloads the same previously read cache lines and measures the access latency. Longer access latency implies that one or more lines of the cache set were evicted by the victim.

2.2 Cache Side-Channel Defenses
Attempts to defeat cache side-channel attacks have been considered at the hardware, system, and software levels. As Cape is a compiler-based approach that transforms code, most relevant to our work are software-level defenses. Raccoon partially closes cache side channels by introducing decoy execution paths and dummy data fetches [38]. Crane et al. propose to defeat cache side-channel attacks by diversifying program memory layout at load time [18].

Early approaches transform programs to eliminate secret-dependent control and data flow [17, 33]. More recently, SC-Eliminator, Lif, and Constantine automatically transform programs to eliminate timing channels, by transforming code that executes conditionally on a secret to execute the same instructions for both sides of the condition [12, 41, 47]. The transformation that these approaches perform is called linearization, which ensures that every program execution accesses a secret-independent sequence of addresses. (The program analyses that enable linearization are related to our secret dependence analysis, but our transformations that add transactions and preloading are novel to the best of our knowledge.) However, linearization cannot handle two kinds of secret dependencies: (1) indirect accesses and array accesses in which the addressed address depends on a secret and (2) loops in which the termination condition depends on a secret. In contrast, our approach can handle these cases; Section 7 compares our approach empirically with Lif.

2.3 Cloak and Hardware Transactional Memory
Cloak leverages hardware transactional memory (HTM) to hide secret-dependent memory accesses from attackers [22].

**Hardware Transactional Memory.** HTM enables atomic execution of a sequence of memory reads and writes on shared data [24, 26]. Intel Transactional Synchronization Extensions (TSX) is an implementation of HTM on Intel processors [51]. TSX implements speculation-based atomic execution by maintaining a read set and a write set for each transaction. The read set is implemented by adding one bit to each entry of the LLC (approximated with a Bloom filter [11]), so that a memory read loads the corresponding cache line in the read set. The write set is implemented in a core’s private data cache by setting a write bit for cache lines written by an executing transaction [15]. When cache lines in the read set or the write set are evicted, the transaction aborts: Memory stores in the write set are discarded without committing to memory, and all read and write bits are cleared. As such, execution inside a transaction can be rolled back without making any architecturally observable changes.

Although TSX was not designed as a security defense, the special software–hardware contract enabled by TSX has been exploited in several security research projects. Most relevant to ours is Cloak, which utilizes TSX to prevent cache side-channel attacks [22] (Section 2.2). Analogously, T-SGX uses TSX to prevent controlled-channel attacks on SGX [40].

TSX provides several instructions for managing transactions. XBEGIN starts a transaction, and XEND commits a transaction. If a transaction aborts for any reason, the hardware rolls back all effects of the transaction, and control returns to a program-defined abort handler specified as a parameter to XBEGIN. An abort handler can choose to retry the
transaction or to execute some other fallback. TSX provides
XTEST to check whether the current thread is executing a
transaction, and XABORT to abort a transaction explicitly.

How Cloak’s HTM-Based Defense Works. Cloak’s de-
fense executes secret-dependent memory accesses inside
of transactions. Just after a transaction starts, inserted code
preloads all secret-dependent data and instruction addresses
that might be accessed in the transaction. Subsequent secret-
dependent accesses are effectively invisible to the attacker.
If the transaction aborts due to eviction of an accessed line—
caused by an active attack or a capacity overflow—the core
rolls back the transaction immediately by invalidating dirty
lines written by the transaction, leaving the cache in a state
independent of secret values. If the transaction succeeds, the
cache (which retains all data accessed in the transaction) is
likewise in a state that is independent of secret values.

Cloak requires programmers to manually identify code
regions to be encapsulated in transactions and to insert de-
terministic preloading operations at transaction start.

3 Overview

The goal of this research is to provide a compiler-based ap-
proach to transform a program so that its memory access
trace is insensitive to secret values, by using HTM to enforce
secret-oblivious memory access traces.

Threat Model. Our threat model follows that of Cloak [22].
Specifically, we consider an adversary that is able to run an
attacker program on the same physical machine as the victim
program. This could happen in the case of multi-user work-
stations or cloud data centers, where programs or virtual
machines belonging to different users can run side-by-side.
Attacks against programs running inside of a trusted ex-
ecution environment (TEE), such as Intel Software Guard
Extensions (SGX), are also under consideration, though in
such cases the adversary controls the entire operating system
rather than an individual program.

The commonality of these considered scenarios is that the
victim programs targeted by cache side-channel attacks are
immune to direct memory inspection and modification. To
breach the confidentiality of the victim program, the attacker
program can only perform Prime+Probe or Flush+Reload
cache side-channel attacks (or their variants) to learn the
memory access patterns of the victim’s execution and then
infer secrets indirectly. As such, we only consider access-
driven cache attacks, which are the primary concern in prior
work [17–19, 33, 38, 43, 46, 48].

Requirements of Effective Defenses. As in Cloak [22],
the transformed program must satisfy the following require-
ments. The memory access trace (i.e., the list of accessed
cache line addresses) must be secret oblivious, i.e., not de-
pendent on secret values. Importantly, in a memory access
trace that uses hardware transactions, repeated accesses to
a cache line in a transaction are effectively invisible to an
attacker program. Thus, if a transaction preloads all cache

\[
\text{seq} ::= \text{stmt} \cdot \text{seq} \mid \text{stmt}
\]

\[
\text{stmt} ::= \text{var} := \langle \text{expr} \rangle;
\]

\[
\text{expr} ::= \text{const}
\]

Figure 1. Context-free grammar for a simple language that
we use to define secret dependence analysis.

lines that it might access irrespective of secret values, the
memory access trace will be secret oblivious. As a result, a
defense should add transactions and preloading such that (1)
every secret-dependent memory access is in a transaction,
and (2) a transaction must not be secret control dependent.

While Cape is an automatic approach, a programmer must
identify secret values (e.g., cryptographic keys) for Cape.

4 Cape’s Secret Dependence Analysis

This section introduces Cape’s secret dependence analysis,
which identifies memory accesses and other instructions that
are dependent on secret values and thus need to be protected.

Definitions of control-flow graph, dominator, and depend-
ence graph are from the literature [7, 20].

4.1 A Simple Language

To define secret dependence analysis clearly, we define a
simple language with memory accesses and control statements.
Figure 1 shows the grammar for the language.

A program is a sequence of statements: assignments to
a variable, memory accesses, and control flow. Statements
operate on expressions; an expression can be a constant, a
local variable, a secret input (represented by getSecret()), an
allocation that returns an address, a load from a memory
address, or an arbitrary pure function (e.g., arithmetic or
bitwise operation) on two subexpressions.

A single word type is used for all values including ad-
dresses. The value of every variable and memory location
is undefined until it is written to.

4.2 The Control-Flow Graph

A control-flow graph (CFG) is a directed graph in which each
node represents an instruction. An instruction is any of the
production alternatives for \( \text{stmt} \) in Section 4.1—except that
if statements and while loops are split into multiple instruc-
tions: one for the condition and one for each statement in
its body. All instructions of a legal program are bijectively
mapped to nodes of the program’s CFG, so the rest of the
paper uses the term “instruction” to refer to a CFG node.
Each edge in a CFG represents possible flow of control from one instruction to another. In addition, a condition or loop header instruction has an outgoing edge to the instruction immediately following the last instruction of its body; the last instruction in a loop has only a single outgoing edge to the loop header; and a break has an outgoing edge to the instruction immediately following the last instruction of the innermost loop body that contains the break.

A CFG has two special nodes, ENTRY and EXIT, which are the entry and exit points of the CFG, respectively. Every CFG node is on a path from ENTRY to EXIT.

4.3 Dominate and Post-dominate

**Dominators.** In a CFG, a node \(d\) dominates a node \(n\) (i.e., \(d\) is a dominator of \(n\)) if every directed path from ENTRY to \(n\) (not including \(n\)) contains \(d\). The immediate dominator of a node \(n\), i.e., \(iPdom(n)\), is the unique node that dominates \(n\) but does not dominate any other node that dominates \(n\).

The **dominator tree** of a CFG is a tree with root ENTRY such that a node \(m\) is a child of a node \(n\) if and only if \(n = iPdom(m)\).

**Post-Dominators.** A node \(p\) post-dominates a node \(n\) if every directed path from \(n\) to EXIT (not including \(n\)) contains \(p\). The **immediate post-dominator** of a node \(n\), i.e., \(iPdom(n)\), is the unique node that post-dominates \(n\) but does not post-dominate any other node that post-dominates \(n\).

A CFG’s **post-dominator tree** is a tree with root EXIT such that a node \(m\) is child of a node \(n\) if and only if \(n = iPDom(m)\).

4.4 The Dependence Graph

A **dependence graph** (DG) is a directed graph in which each node is a CFG node (i.e., a program instruction), and each edge represents a dependence relation between a pair of instructions [20, 27].

Each DG edge has one of two types, control dependency and data dependency, denoted \(\rightarrow^{cd}\) and \(\rightarrow^{dd}\). We denote a path from \(\text{inst}_1\) to \(\text{inst}_2\) as \(\text{inst}_1 \rightarrow^{(dd, cd)^*} \text{inst}_2\), indicating that \(\text{inst}_2\) is transitively dependent on \(\text{inst}_1\) through data and/or control dependencies. If a path in the DG starts with an instruction that contains an \(\text{expr}\) that includes \(\text{getSecret()}\), we call the path a secret dependency path, which is a critical concept in Cape’s secret dependence analysis.

**Data Dependency.** Let \(\text{inst}_1\) and \(\text{inst}_2\) be nodes in a CFG. Then \(\text{inst}_1 \rightarrow^{dd} \text{inst}_2\) if and only if \(\text{inst}_1\) stores a value (either in a variable or memory location) that \(\text{inst}_2\) loads.

**Control Dependency.** Let \(\text{inst}_1\) and \(\text{inst}_2\) be nodes in a CFG. Then \(\text{inst}_1 \rightarrow^{cd} \text{inst}_2\) if and only if

1. \(\text{inst}_1\) is not post-dominated by \(\text{inst}_2\); and
2. there exists a path \(p\) from \(\text{inst}_1\) to \(\text{inst}_2\) such that any node in \(p\) (excluding \(\text{inst}_1\) and \(\text{inst}_2\)) is post-dominated by \(\text{inst}_2\).

This definition does not handle so-called loop-carried dependencies, in which an instruction in a loop is dependent on whether the loop exits from its body.\(^1\) Consider Listing 1, in which the the outcome of \(\text{inst}_5\) (i.e., the instruction at line 5) determines whether \(\text{inst}_4\) executes again—accessing additional memory addresses. Therefore, we should consider \(\text{inst}_5\) to be control dependent on \(\text{inst}_4\).

We thus extend the definition of control dependency to include loop-carried control dependencies: For instructions \(\text{inst}_i\) and \(\text{inst}_j\) in a loop \(L\), \(\text{inst}_i \rightarrow^{cd} \text{inst}_j\) if

\[ \exists \text{inst}' : (\text{inst}' \text{ is a break} \land \text{inst}_i \rightarrow^{cd} \text{inst}' \land L \text{ is innermost loop containing inst'}) \]

Accordingly, dependence analysis marks all instructions of a loop as control dependent on a condition that is not the loop header but controls whether the loop exits.

4.5 Address Dependency

Using data and control dependencies, an analysis can identify all instructions that are transitively dependent on secret values. However, not all instructions that are data and control dependent on secret values need to be protected by Cape; for example, consider instruction \(*\text{addr} := \text{val}\); when only \(\text{val}\) is secret dependent. Here we define a concept that helps with defining whether a memory access needs protection.

A memory access is **address dependent** on an instruction if the memory address used for the access is data dependent on the instruction. That is, \(\text{inst}_1 \rightarrow^{ad} \text{inst}_2\) if

\[ \text{inst}_1 \rightarrow^{dd} \text{inst}_2 \land \text{inst}_1 \text{ is var:=}\_\_\land \text{inst}_2 \text{ contains a } *\text{expr} \text{ such that expr includes var} \]

4.6 Secret Dependency

Memory stores \(*\langle\text{expr}\rangle:=...\rangle\) and loads (any use of \(*\langle\text{expr}\rangle\) may leak secrets due to revealing patterns of memory accesses for data. In contrast, we assume accesses to local variables (reads and writes of \(\text{var}\)) do not leak secrets. Any secret-dependent instruction may leak secrets by serving as an instruction cache side channel.

\(^1\)The stated definition of control dependency does not handle loop-carried dependencies in our non-SSA program representation. However, if programs are represented in SSA form, then the stated definition will handle loop-carried dependencies implicitly (cf. [9, 16, 39]).
An instruction is secret dependent if it is address or control dependent on either (1) a secret value directly or (2) an instruction that is transitively dependent on a secret value. That is, an instruction inst is secret dependent if and only if there exists an instruction inst3 that evaluates an expression that includes getSecret() and

\[ \text{inst} \xrightarrow{\text{getSecret}()} \text{inst} \lor \exists \text{inst}': \text{inst} \xrightarrow{\text{inst}'} \text{inst} \]

**Example.** Listing 2 shows a program in the simple language that has secret-dependent instructions.

**Types of Secret Dependencies.** We can refine the above definition. An instruction inst is secret address dependent if and only if there exists an instruction inst that evaluates an expression that includes getSecret() and

\[ \text{inst} \xrightarrow{\text{getSecret}()} \text{inst} \lor \exists \text{inst}': \text{inst} \xrightarrow{\text{inst}'} \text{inst} \]

An instruction inst is secret control dependent if and only if there exists an instruction inst that evaluates getSecret() and

\[ \text{inst} \xrightarrow{\text{getSecret}()} \text{inst} \lor \exists \text{inst}': \text{inst} \xrightarrow{\text{inst}'} \text{inst} \]

Only memory access instructions can be secret address dependent, but any instruction can be secret control dependent.

**Secret-Dependent Loops.** We note that, according to the dependency definitions, all instructions of a loop are secret dependent if its header is secret dependent or it is the innermost loop containing a secret-dependent break instruction.

**5 Cape’s Compiler Instrumentation**

This section presents Cape’s compiler transformation that places transactions and adds preloading instructions.

Listing 3 shows a program with secret-control-dependent accesses, after instrumentation by Cape. Instructions inst3 and inst4 are secret dependent because both instructions are control dependent on inst2, which is data dependent on the secret-loading instruction inst1. Cape automatically places a transaction around the if statement, and it inserts code to preload sensitive data and code accessed by inst3 and inst4.

**5.1 Placing Transactions**

To protect secret-dependent instructions from attacker programs, Cape must ensure that (1) all secret-dependent instructions are surrounded by transactions and (2) whether a transaction executes is not dependent on the value of any secret. More specifically, Cape adheres to the following rules for transaction placement:

- A secret-dependent instruction must be in a transaction.
- A transaction start cannot be secret control dependent.
- The transaction start must dominate its end—unless the end utilizes XTEST to end the transaction conditionally. We abbreviate "if XTEST then XEND" as XEND_if_XTEST.
- A transaction end must post-dominate its start.

To help identify where to place transaction boundaries, we define the head and tail of a secret-dependent instruction as follows. If the instruction is in a secret-dependent loop, then its head is the preheader of the loop and its tail is the preheader’s closest post-dominator that is not in the loop; otherwise, its head and tail are the instruction itself. Secret-dependent loops require special handling for transaction placement. As defined in Section 4.6, an entire loop is secret dependent if its header is secret dependent or it is the innermost loop containing a secret-dependent break instruction.

**Placing Transaction Boundaries.** To ensure each secret-dependent instruction is in exactly one transaction and and no transaction is secret dependent, Cape places transactions according to the following rules:

- Every static transaction has a unique pair of XBEGIN and XEND_if_XTEST instructions.
- Every instruction executes in at most one transaction (no nested transactions).
- Every path from XBEGIN to EXIT goes through exactly one XEND_if_XTEST with no intervening XBEGIN.
- For any instruction inst that is secret control dependent, if there exist instructions inst’, inst” such that inst \xrightarrow{\text{getSecret}()} \text{inst} \rightarrow inst’ and XEND_if_XTEST is prepended to inst” and XEND_if_XTEST is prepended to if (xtest()){}[xend()].
- Otherwise (for a secret address dependency), XBEGIN is prepended to head(inst) and XEND_if_XTEST is appended to tail(inst).
- An XBEGIN (or XEND_if_XTEST) is added at most once to a code location.

**5.2 Preloading Data and Code**

Cape modifies the compiler to insert code at the start of each inserted transaction to preload secret-dependent data and code into the transaction’s read set. Since TSX implements...
read sets for lines in the shared LLC, Cape provides protection with respect to the LLC, which satisfies our assumed attack scenarios. It would be straightforward to extend Cape’s protection to private caches by using both write and read sets for preloading as presented by Cloak [22].

In Listing 3, since inst$_3$ and inst$_4$ are secret dependent, Cape preloads both the data that the instructions may access and the two instructions themselves.

6 Implementation

We implemented Cape in LLVM [29]. We have made Cape’s source code publicly available.

The implementation first performs secret dependence analysis (Section 6.1) and then uses the analysis results to transform LLVM bitcode with transactions and preloading (Section 6.2).

Since the implementation deals with LLVM bitcode, instructions described in this section refer to those defined for the LLVM IR, rather than those defined in our simple language (Sections 4.1 and 4.2). Of particular importance for our implementation are the LLVM memory access and branch instructions: LOAD, STORE, and BR. Note that for memory accesses, we only deal with LOAD and STORE in this section for the sake of brevity, but other LLVM instructions and intrinsics that access memory, including memset and memcpy, are also handled by the implementation.

6.1 Implementing Cape’s Analysis

We implemented Cape’s secret dependence analysis using dg [2, 14], which builds program dependence graphs [20] and performs static program slicing [45]. To compute data and control dependencies, dg implements control dependence analysis, pointer analysis, and reaching definition analysis.

Computing Secret Dependency Paths. Our Cape implementation uses dg to construct dependence graphs and perform forward slicing to mark all instructions in a dependence graph that are reachable from any instruction inst, that loads from a programmer-annotated secret variable. The reachable instructions constitute secret dependency paths (Section 4.4). Consequently, all instructions in the resulting secret dependency paths are transitively dependent on inst, through one or more data and control dependencies.

Identifying Secret-Dependent Instructions. For each instruction inst that is in any secret dependency path but not in any secret-dependent loop, Cape checks if inst is secret control dependent, i.e., if inst is control dependent on an instruction that is in a secret dependency path. If so, inst is control dependent on a secret. Otherwise, if inst is a LOAD or STORE, Cape checks if inst is secret address dependent. The check iterates over all instructions in the secret dependency paths that inst is data dependent on, and determines inst is secret address dependent if there exists an instruction inst’ whose result is used as the address operand of inst.

6.2 Implementing Cape’s Transformation

After identifying secret-dependent instructions and loops, Cape inserts code into the LLVM bitcode to start and end transactions and preload sensitive data and code.

6.2.1 Placing Transactions. For a secret-dependent instruction inst that does not belong to any secret-dependent loops, Cape finds all instructions that inst is (transitively) secret control dependent on, by computing a backward transitive closure of control dependencies along secret dependency paths. Let inst$_{top}$ be the earliest instruction in the backward closure, i.e., the instruction that all other instructions are secret control dependent on. If inst$_{top}$ is a BR, Cape inserts a transaction start immediately before inst$_{top}$ and inserts the matching transaction end immediately after inst$_{top}$’s closest post-dominator that is outside the loop.

For a secret-dependent loop, Cape performs the same backward transitive closure starting from the loop’s preheader. If inst$_{top}$ is not a BR—so the secret-dependent loop is not secret control dependent on any instruction outside the loop—the implementation inserts a transaction start before inst$_{top}$, but inserts the matching transaction end immediately after inst$_{top}$’s closest post-dominator that is outside the loop.

6.2.2 Preloading. Once a secret-dependent loop or instruction is wrapped in a transaction, Cape inserts code at transaction start to preload sensitive data and code.

Data Preloading. The actual address accessed by a secret-dependent memory access is unknown at instrumentation time, especially if the memory access is through a pointer. Therefore, Cape uses dg’s pointer analysis to get points-to information for the address operand of each secret-dependent access. More precisely, for the address operand of a secret-dependent access inst, the pointer analysis generates a set containing all LLVM instructions that allocate memory locations that may be accessed by inst.

LLVM uses three types of allocation: static, dynamic, and automatic. Static allocations allocate static variables (such as global variables) whose lifetime is the program’s lifetime. Cape can preload these variables directly by inserting LOADs of these always-live, globally visible variables.

A dynamic allocation represents a call to malloc(), calloc(), or realloc() to allocate space in heap memory at run time. Since a dynamic allocation instruction can have multiple dynamic instances at run time, to support preloading data from all possible locations allocated by the instruction, Cape saves

\[\text{http://github.com/PLaSTicity/Cape-implementation} \]
dynamic allocation information (i.e., addresses and sizes) in a custom data structure that we call the allocation information buffer (AIB). Each static allocation instruction has one dedicated AIB. To preload data from dynamically allocated sensitive locations, Cape inserts code at the transaction start to load memory at the address ranges indicated by the information in the AIB. To avoid preloading freed addresses, Cape instruments calls to free() to remove the address from the AIB(s) corresponding to the dynamic allocation.

The last type of allocation is for local variables (ALLOCA instructions in LLVM IR), which are stored in stack locations and CPU registers in the generated machine code. If we applied our LLVM transformation to local variables, then all local variables would be stored in stack locations, harming performance significantly. Instead, for ease of implementation, both Cape and Cloak do not protect local variables—which is unsafe since secret-dependent stack accesses are a potential cache side-channel attack vector. With engineering effort, Cape could protect secret-dependent local variables without inadvertently forcing all secret-dependent local variables to be stored in stack locations.

Code Preloading. A simple approach for preloading secret-dependent code addresses would identify secret-dependent basic blocks in the analyzed LLVM bitcode, and generate code to touch addresses corresponding to the basic blocks. However, downstream LLVM optimizations reorder the linear order of basic blocks in memory and also make other changes such as merging, splitting, copying, and deleting basic blocks, leading to incorrect code preloading instrumentation.

To support correct code preloading in the face of downstream optimizations, our implementation inserts instrumentation that preloads an entire function’s code. Cape identifies functions that include secret-dependent basic blocks, and it generates code to preload the entire function at transaction start. Cape obtains each function’s address range after compilation, and the instrumented program loads and uses these address ranges at run time.

### Table 1. The evaluated programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Data size (in elements)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes</td>
<td>256</td>
<td>AES T-table implementation in OpenSSL [1]</td>
</tr>
<tr>
<td>bsearch</td>
<td>10–10,000</td>
<td>Binary search implementation in the C standard library</td>
</tr>
<tr>
<td>dtree</td>
<td>10–5,000</td>
<td>Pointer-based decision tree classification adapted by us from the Cloak paper [22]</td>
</tr>
<tr>
<td>mdtree</td>
<td>10–5,000</td>
<td>RSA square-and-multiply textbook implementation from the Cloak paper [22]</td>
</tr>
<tr>
<td>rsa</td>
<td>n/a</td>
<td>A wolfSSL-provided wrapper that uses wolfSSL’s RSA implementation to sign binary data [4]</td>
</tr>
<tr>
<td>signature</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

### 7.1 Evaluation Methodology

To measure effectiveness, we compute whether memory traces of vulnerable programs, collected using Pin [32], are secret oblivious. To evaluate performance, we execute programs natively with Intel TSX.

**Programs.** Table 1 shows the evaluated programs, which are known to be vulnerable to cache side-channel attacks. Each program is either a real program (aes from OpenSSL [1], bsearch from glibc, and signature from wolfSSL [4]), was evaluated by Cloak [22] (rsa and dtree), or was derived by us from dtree (mdtree).

We obtained aes (which uses the vulnerable AES T-table implementation from OpenSSL [1]) and rsa (which provides a textbook implementation of the RSA square-and-multiply algorithm) from the Cloak authors. We removed cache side-channel attacks that these programs’ harnesses performed. The code for dtree is listed in the Cloak paper [22]. The most complex program we evaluate is signature, which signs binary data using a realistic implementation of the RSA algorithm provided by the wolfSSL library [4]. We compiled each program with LLVM’s -O3 optimizations.

Our experiments run some programs with multiple data sizes, which are the sizes of the data structure used in the program, as shown in Table 1. Large data sizes stress transaction cache capacity limits. For aes, the data size is fixed to the size of the T-table: 256 table elements, which take up 2,048 bytes. The experiments run on data sizes of 10–10,000 elements (40–40,000 bytes) for bsearch and 10–5,000 elements (240–120,000 bytes) for dtree and mdtree. rsa and signature have no secret-dependent data accesses—only secret-control-dependent instructions.

We harness each program to generate and use a random secret value as input each time it runs.

**Evaluating Cape.** Listings 4–6 show code for aes, mdtree, and rsa, with instrumentation added by Cape for transactions and preloading. For space and simplicity we omit dtree and bsearch, which have code structure similar to mdtree; and signature, which, like rsa, has secret control-dependent instructions, but signature’s secret-dependent code is much larger than rsa’s. For aes (Listing 4), inst7–inst10 and inst13 are secret address dependent, so Cape wraps them in transactions and preloads all data that might be accessed by them. No code

---

1 Preloading sizes are limited by a transaction’s read set, which is often tied to the L1 size and is bounded by the LLC size (cf. [13, 25]).
preloading is required since the instructions themselves are guaranteed to be executed independent of the secret value. The T-table in aes is allocated as a global variable when the program starts (omitted in Listing 4), so Cape preloads data in the T-table directly without the need to save any allocation information in AIBs. For mdtree (Listing 5), since there are instructions that are secret control dependent, Cape inserts code to preload both data accessed by the instructions and the instructions themselves. To facilitate its data preloading, Cape inserts code immediately after the allocation instructions \(\text{inst}_8\) to save allocation information in AIBs. For rsa (Listing 6), Cape inserts code to preload \(\text{inst}_8\) since the instruction is secret control dependent.

**Evaluating Cloak.** We evaluate (prior work) Cloak by manually instrumenting the programs’ source to wrap security-sensitive code in transactions and perform preloading operations, based on the relevant descriptions in the Cloak paper [22] and the code obtained from the Cloak authors.

The resulting code puts a single transaction around the entire AES computation for aes, around each iteration of the loop of the square-and-multiply algorithm for rsa, around the entire search for bsearch, and around the entire tree traversal for dtree and mdtree. The inserted code performs data preloading of aes’s T-table, bsearch and dtree’s arrays, and mdtree’s linked-list-based tree; and instruction preloading of functions that include secret-dependent code.

Given the complexity of signature, our understanding of its instrumentation needs was admittedly aided by referring to the results of Cape’s instrumentation and of our Pintoool-based dynamic analysis (Section 7.1). Like Cape, our manual Cloak instrumentation adds a single transaction that encompasses virtually the entire executed program, since nearly all instructions are secret control dependent. While Cape adds both data and code preloading at transaction start, our manual Cloak instrumentation adds only code preloading because we determined that data preloading is unnecessary, and is only added by Cape as a result of analysis imprecision.

**Evaluating Lif.** The Lif authors used Lif to transform the evaluated programs. (Lif [41] is publicly available, but the Lif authors offered to run Lif for us to save time.) For rsa, the Lif authors sent us Lif-transformed LLVM bitcode. For the other five programs, the Lif authors reported that the code uses secret-dependent access patterns that Lif’s linearization approach cannot defend against (Section 2.2).

**Configurations.** We evaluate four main configurations:
- **Base** — the unmodified, vulnerable program
- **Cloak** — the program modified by manual instrumentation based on prior work’s approach [22]
- **Cape** — the program modified by this paper’s approach to execute transactions and perform preloading
- **Lif** — the program modified by Lif [41]

In addition, we break down the costs of Cape by evaluating two partial (unsecure) Cape configurations:
To evaluate whether Cape-instrumented programs produce memory access traces that are insensitive to secret values, we wrote a Pintool [32] that records and processes traces of events, which are memory accesses or transaction boundaries, for executing programs. To avoid aborts, these experiments do not actually execute hardware transactions: The Pintool intercepts XBEGIN and XEND instructions and generates corresponding events but skips executing the instructions.

For each memory access outside a transaction, the Pintool logs the accessed cache line’s address. For each memory access inside a transaction, the Pintool logs the accessed line’s address only on the transaction’s first access to the line. This behavior captures the fact that repeat accesses to a cache line in a transaction must be cache hits and are invisible to attacker programs. (The Pintool does not need to model cache capacities because a transaction whose working set size is too large to fit in caches will abort by design, preventing any information leakage through cache side channels.)

We detect secret-sensitive accesses by comparing the address traces from two executions using different secret inputs. Note that this methodology of comparing address sequences checks a strong, cache-model-oblivious security guarantee.

For a given program and data size, we execute 10 trials of the Cape-instrumented program using the Pintool to generate and compare address traces. Each trial independently generates a random secret input. If all 10 address traces are identical, then this result provides evidence that the program’s memory access trace is insensitive to secret values. Otherwise, the discrepancy indicates that the memory access trace is sensitive to secret values.

Methodology for Evaluating Performance. The experiments run on native TSX-enabled hardware. For all programs except rsa, experiments run on an Intel Xeon Gold 5218 16-core processor running Linux. For rsa, experiments run on an Intel Xeon E5-2683 14-core processor running Linux. We used the second machine for rsa to add results for Lif.

To account for startup time and noise, we harness each program to execute the program’s code 5,000,000 times in a loop. For example, bsearch performs 5,000,000 binary searches (each using a new randomly generated key). As a result, each program executes for at least 0.1 seconds and as much as a few seconds. To account for run-to-run variation, every reported result is the arithmetic mean from 10 trials.

For configurations that use transactions (Cloak, Cape, and Cape preld only), a repeatedly aborting transaction retries 200 times before finally failing by terminating the program.

7.2 Results

7.2.1 Effectiveness. Our Pin-based experiments compare memory access traces to determine if they are insensitive to secret values. For unmodified programs (Base) across all data sizes, the memory access traces are sensitive to secret values, implying that the unmodified programs are vulnerable to cache side-channel attacks. In contrast, for every program and data size, Cloak and Cape each execute memory access traces that are identical across different secret inputs, implying that the instrumented programs are not exploitable by cache side-channel attacks.

For rsa, the Lif-transformed program executes memory access traces that are insensitive to secret values. For the other five programs, the Lif-transformed programs would execute memory access traces that are sensitive to secret values, implying that the programs use secret-dependent access patterns that Lif’s approach cannot defend against (Section 7.1).

7.2.2 Performance. Tables 2 and 3 present performance results for Cape, compared with Base, Cloak, and Lif. Table 2 shows counts of memory accesses, obtained from Pin-based experiments on compiled code. It breaks down Cape and Cloak’s total memory accesses into those executed outside transactions (Outside txn), due to preloading (Preloading), and inside transactions but not for preloading (Inside txn).

For aes, Cape’s preloading operations incur many more accesses than Cloak’s since Cape inserts transactions and preloading operations inside a loop and thus incurs repeated executions of the operations, while Cloak wraps the entire loop in a single transaction and only preloads sensitive data once. For the other programs, compared with Base, both Cape and Cloak incur significant access overhead for data preloading, especially with large data sizes. For rsa, Cape and Cloak execute the same number of accesses, and significantly more than Lif. For bsearch and dtree, Cape incurs comparable, but slightly more, accesses than Cloak. Cape’s preloading operations incur more accesses due to fetching allocation information from the ALBs.

The impact of data preloading is particularly significant for mdtree since the program uses malloc to allocate contiguous memory space for its secret-sensitive data structure (a pointer-based tree). Cape stores the allocation information as separate elements in the ALBs. Iterating over these ALB elements and preloading contiguous locations are costly. Cloak performs significantly more preloading accesses than Cape for mdtree. Our manual Cloak instrumentation initially
Table 2. Memory accesses outside of transactions and inside of transactions, from preloading and the original program, for Base, Cape, Cloak, and Lif (rsa only). Inside txn counts only non-preloading accesses inside transactions. Each count is the average from 10 trials, rounded to the nearest integer. The ‘-’ entries reflect that Base and Lif execute no transactions or preloading.

<table>
<thead>
<tr>
<th></th>
<th>Data size = 10</th>
<th>Data size = 100</th>
<th>Data size = 1000</th>
<th>Data size = 5000</th>
<th>Data size = 10000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Base Cape Cloak</td>
<td>Base Cape Cloak</td>
<td>Base Cape Cloak</td>
<td>Base Cape Cloak</td>
<td>Base Cape Cloak</td>
</tr>
<tr>
<td>Outside txn</td>
<td>35 33 33</td>
<td>38 33 33</td>
<td>41 33 33</td>
<td>44 33 33</td>
<td>45 33 33</td>
</tr>
<tr>
<td>Preloading</td>
<td>– 13 2</td>
<td>– 23 14</td>
<td>– 137 126</td>
<td>– 637 626</td>
<td>– 1,261 1,250</td>
</tr>
<tr>
<td>Inside txn</td>
<td>– 8 8</td>
<td>– 11 11</td>
<td>– 14 14</td>
<td>– 17 17</td>
<td>– 18 18</td>
</tr>
<tr>
<td>Total</td>
<td>35 54 43</td>
<td>38 67 58</td>
<td>41 164 173</td>
<td>44 687 676</td>
<td>45 1,312 1,301</td>
</tr>
</tbody>
</table>

(a) bsearch

<table>
<thead>
<tr>
<th></th>
<th>Data size = 10</th>
<th>Data size = 100</th>
<th>Data size = 1000</th>
<th>Data size = 5000</th>
<th>Data size = 10000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Base Cape Cloak</td>
<td>Base Cape Cloak</td>
<td>Base Cape Cloak</td>
<td>Base Cape Cloak</td>
<td>Base Cape Cloak</td>
</tr>
<tr>
<td>Outside txn</td>
<td>35 24 23</td>
<td>46 24 23</td>
<td>57 24 23</td>
<td>60 24 23</td>
<td>71 24 23</td>
</tr>
<tr>
<td>Preloading</td>
<td>– 19 8</td>
<td>– 87 76</td>
<td>– 387 376</td>
<td>– 761 750</td>
<td>– 3,761 3,750</td>
</tr>
<tr>
<td>Total</td>
<td>35 60 49</td>
<td>46 139 128</td>
<td>57 450 438</td>
<td>60 827 815</td>
<td>71 3,838 3,826</td>
</tr>
</tbody>
</table>

(b) dtree

<table>
<thead>
<tr>
<th></th>
<th>Data size = 10</th>
<th>Data size = 100</th>
<th>Data size = 1000</th>
<th>Data size = 5000</th>
<th>Data size = 10000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Base Cape Cloak</td>
<td>Base Cape Cloak</td>
<td>Base Cape Cloak</td>
<td>Base Cape Cloak</td>
<td>Base Cape Cloak</td>
</tr>
<tr>
<td>Outside txn</td>
<td>34 23 21</td>
<td>46 23 21</td>
<td>55 23 21</td>
<td>59 23 21</td>
<td>69 23 21</td>
</tr>
<tr>
<td>Preloading</td>
<td>– 42 103</td>
<td>– 402 1,003</td>
<td>– 2,002 5,003</td>
<td>– 4,002 10,003</td>
<td>– 20,002 50,003</td>
</tr>
<tr>
<td>Inside txn</td>
<td>– 17 17</td>
<td>– 29 30</td>
<td>– 38 38</td>
<td>– 42 41</td>
<td>– 52 53</td>
</tr>
<tr>
<td>Total</td>
<td>34 82 141</td>
<td>46 454 1,054</td>
<td>55 2,063 5,062</td>
<td>59 4,067 10,065</td>
<td>69 20,077 50,077</td>
</tr>
</tbody>
</table>

(c) mtree

<table>
<thead>
<tr>
<th></th>
<th>Data size = 256</th>
<th>Data size = n/a</th>
<th>Data size = n/a</th>
<th>Data size = n/a</th>
<th>Data size = n/a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Base Cape Cloak</td>
<td>Outside txn</td>
<td>Outside txn</td>
<td>Outside txn</td>
<td>Outside txn</td>
</tr>
<tr>
<td>Outside txn</td>
<td>303 172 151</td>
<td>292 392 328 458</td>
<td>13,441 K &lt;1 K</td>
<td>13,441 K &lt;1 K</td>
<td></td>
</tr>
<tr>
<td>Insidetxn</td>
<td>– 307 228</td>
<td>Inside txn</td>
<td>– 13,441 K 13,442 K</td>
<td>– 13,441 K 13,442 K</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>303 1,119 411</td>
<td>Total</td>
<td>Total</td>
<td>Total</td>
<td>Total</td>
</tr>
</tbody>
</table>

(d) aes

<table>
<thead>
<tr>
<th></th>
<th>Data size = 256</th>
<th>Data size = n/a</th>
<th>Data size = n/a</th>
<th>Data size = n/a</th>
<th>Data size = n/a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Base Cape Cloak</td>
<td>Outside txn</td>
<td>Outside txn</td>
<td>Outside txn</td>
<td>Outside txn</td>
</tr>
<tr>
<td>Outside txn</td>
<td>548 328 458</td>
<td>292 392 328 458</td>
<td>13,441 K &lt;1 K</td>
<td>13,441 K &lt;1 K</td>
<td></td>
</tr>
<tr>
<td>Insidetxn</td>
<td>– 307 228</td>
<td>Inside txn</td>
<td>– 13,441 K 13,442 K</td>
<td>– 13,441 K 13,442 K</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>548 1,119 411</td>
<td>Total</td>
<td>Total</td>
<td>Total</td>
<td>Total</td>
</tr>
</tbody>
</table>

(e) rsa

used a non-recursive depth- or breadth-first search to traverse the tree, but we found that transactions consistently aborted due to a compiler-generated instruction (XSAVEC) that is unsupported in transactions. So we instead used a recursive depth-first search, which allows transactions to finish through data sizes of 1,000 (a data size of 5,000 consistently causes Cloak to abort for cache capacity reasons).

For the most complex program, signature, preloading incurs modest overhead. Cape performs more preloading accesses than Cloak primarily because only Cape performs data preloading (because of analysis imprecision; see Section 7.1).

Table 3 shows run time and other statistics from native execution for all programs except signature (explained shortly). Table 4(a) evaluates Cape’s impact on execution time compared with Cloak and Lif (for rsa only); run times are wall-clock times normalized to Base. The table breaks down preloading and transaction costs with Cape txonly and Cape preld only. Tables 4(b) and 4(c) help explain the HTM-related aspects of performance results.

Cape slows aes by 8.76× because of repeated executions of transactions and preloading. In contrast, Cloak slows aes by only 1.86× because it adds only one transaction wrapping the entire AES computation.

For rsa, Cape and Cloak incur modest 1.06–1.09× slowdowns despite repeated executions of transactions and preloading, because the original program provided by the Cloak authors includes loops to slow its execution, hiding overhead of transactions and preloading. In contrast, Lif slows rsa by 1.82×, despite executing fewer instructions than Cape and Cloak (Table 2(e)), for reasons that are unknown to us but may be related to the Lif-transformed code’s potentially heavy use of conditional moves.

For bsearch and dtree, Cape and Cloak incur comparable, moderate slowdowns (2.06×–9.57×) for data sizes that are not too large (<5,000 elements). Relatively high slowdowns are incurred for the two programs with large data sizes and for mtree for most data sizes, mainly because of data and code preloading. For these programs, as the data sizes increase, Cape and Cloak preload more data in transactions and incur higher abort rates, leading to higher slowdowns. For mtree, Cloak incurs much higher slowdowns than Cape, presumably because Cloak uses recursive DFS to perform Cloak’s data preloading operations.

We note that Cape adds overhead to all of the evaluated programs because they have secret-dependent accesses. In contrast, Cape would perform no transformations and thus
add no overhead to programs that have no secret-dependent accesses, unless Cape’s conservative static analysis falsely identified some memory accesses as secret dependent.

Table 3 lacks results for signature because the program repeatedly aborts a transaction due to a cache capacity limitation, whether the transaction is added by Cape or Cloak. The transaction consistently aborts because of code in signature that writes to two 64KB arrays, which exceed the 32KB L1 cache available for tracking transaction write sets. Prior work reports that transaction write set sizes in existing implementations of TSX are bounded by the L1 cache size [13, 21, 22, 25, 34, 44, 51]. By using Intel’s Software Development Emulator (SDE) [3] and modifying its default cache configuration, we found that Cape-instrumented signature completes successfully using a 128KB L1 cache size. Thus the transactions could commit successfully with modifications to future hardware implementations. In particular, Cape-instrumented signature would complete successfully if transaction write sets were extended to private L2 caches (typically 256KB in contemporary Intel processors). Alternatively or additionally, future work could hybridize Cape with another software approach such as Raccoon’s decoy paths [38] (Section 2), enabling transactions with smaller working set sizes.

Validating Cloak Results. For aes and rsa, Table 3’s performance results for Cloak are inconsistent with those reported in the Cloak paper [22]. To understand such discrepancies, we evaluated the performance results for Cloak using the instrumented AES and RSA square-and-multiply programs provided by the Cloak authors (minus their attacks; Section 7.1), but were unable to reproduce the results reported in the Cloak paper in our environment. (We ran multiple trials to verify that our results are repeatable.) The differences are likely due to microarchitecture differences, especially different processors’ TSX implementations, but further effort would be needed to verify this hypothesis.

8 Conclusion
Cape provides automatic HTM-based cache side-channel protection using novel compiler analyses and transformations. An evaluation shows that Cape is as effective and generally as efficient as an existing manual approach, but with minimal human effort. This work opens up opportunities to apply HTM-based cache side-channel protection more widely.

Acknowledgments
We thank the paper’s shepherd, Fernando Magno Quintão Pereira, and the anonymous reviewers for insightful suggestions that improved the paper. Thanks to Luigi Soares, Michael Canesche, and Prof. Pereira for valuable help with evaluating Lif on the evaluated programs. Thanks to Daniel Gruss for sharing the programs evaluated in the Cloak paper.

This work was supported by NSF grants XPS-1629126, CAREER-1253703, and CCF-1421612.

References


