Improving Digital Side Channel Protection with Static Analysis Techniques

A Turing Scholar Honors Thesis by:
Josh Eversmann
josh.eversmann@utexas.edu
May 15, 2016

Honors Thesis Committee:
Calvin Lin (Advisor)
İşıl Dillig
Alan Cline

Abstract
Side-channel attacks monitor running programs to extract sensitive information from them. Escort combats a broad class of such attacks by performing compile-time transformations which hide the data processed by the resulting binary. Escort focuses on protecting floating point operations. We extend Escort to be able to transform C programs with pointers or complicated control flow that were formally unusable.

1 Introduction

Even with the growing strength and availability of encryption, it remains difficult to fully ensure the security of the data in a program, since sophisticated attackers can still gain access to secure information through program side channels. A side channel is any way a running program leaks information other than its intended output or effect. Researchers have demonstrated that it is possible to infer things about data within a running program through numerous side channels, such as heat generation [16], timing execution [24,5], or cache behavior [20,4].

Escort [22] is a system which uses automatic program transformations to protect against a broad class of side-channel attacks with minimal developer work. Other defense techniques can require special, secure programming languages [15] or custom hardware [15,28]. Escort can be used to instrument a C program during compilation and produce a safe binary which can run on any x86 system. Escort defends against digital side channels [21] by changing a program’s control flow and memory access behavior in a way which prevents execution traces from revealing information about data internal to the program. By using timing-safe, predicated arithmetic and memory operations during compilation, the modified program can conceal information from several digital side channels, such as cache probing, timing attacks, and instruction pointer monitoring.

The goal of continuing Escort development is to support all language features in C and reduce the performance impact of the security transformations on the resulting program. Progress in both of these directions requires work in the static analysis. Instrumenting complex control flow structures such as loops while maintaining the semantics of the original source code requires information from different types of static analysis, such as pointer analysis and control dependence relationships.

The contribution of this paper is to augment Escort’s static analysis information by building a Program Dependence Graph (PDG) [9] which integrates pointer analysis information from an additional compiler module with several features of Escort. The primary use of the PDG will be in an implicit flow taint analysis, which will allow a programmer to greatly reduce the performance impact of Escort’s transformations by annotating the value(s) in the program which are considered secret; Escort will then only instrument parts of the code which are at risk for leaking information relevant to these values, which will reduce both the runtime and size of the resulting instrumented binary. In addition, the information in the PDG helps transform several cases of program control flow.

The paper is organized as follows: Section 2 goes over important background information in the fields of security and static analysis. Section 3 describes some of the recent research developments similar to Escort, followed by a description of Escort’s internals in Section 4. Next, Section 5 describes the design and implementation of this paper’s contributions to Escort. Section 6 presents an analysis of its impact on Escort’s capabilities, and Section 7 covers directions for the future of the project. Finally, Section 8 offers conclusions based on the work of this paper.
2 Background

This paper is the result of applying ideas from static program analysis to problems being dealt with in computer security, and as such requires a varied basis of information to follow. This section will address general topics in both fields which this paper and many of its related works depend on.

2.1 Static Analysis

Static analysis is the process of reasoning about the structure or behavior of a program based on the source code rather than the program’s execution. While academia has produced static analyzers for all kinds of programming languages, the most useful tooling is found around compiled languages. Compilers have internal data structures which capture information about the behavior of the program that will be used to produce optimized binaries; by integrating with compilers, abstract syntax trees, control flow graphs, and other intermediate representations of the program can be used to assist in different static analysis techniques.

Pointer analysis is an important field of work within the analysis of C programs which aims to determine statically what values in memory could be referred to by pointers at a given stage in the program’s execution. More specific pointer information requires more complex analysis, and pointer analysis techniques are divided up based on the class of information they produce:

1. Flow-sensitive pointer analysis treats reads and writes differently based on their order in the program.

2. Context-sensitive pointer analysis treats the pointer information that flows through a function call as specific to a call site rather than to the function being called.

We rely on a pointer analysis with tunable precision (TPA) written by Jia Chen [6] which processes programs in LLVM IR and provides information about memory objects and points-to relationships. We use the library in Semi-Sparse Pointer Analysis mode for the work in this paper.

Much of static analysis depends on properties of graphs that represent the program being analyzed. The control flow graph of a program is made up of nodes called basic blocks, which contain instructions which can be executed in sequence followed by a single jump or branch, and edges which represent the relationship between basic blocks created by the jumps that go between them. We are concerned in this paper with an important property called dominance: A basic block X is referred to as the dominator of another basic block Y if all paths through the program that go through Y must go through X. There is also mirrored relationship called post-dominance: X post-dominates Y if all paths from X to the end of the program must go through Y.

Program Dependence Graphs (PDGs) are a static analysis tool proposed by Ferrante et al. [9] to better encode the interdependence of control and data flows in a program. The primary components of any PDG are nodes representing program points with edges representing control dependences and edges representing data dependences. Both types of edge can be integral in propagating information through a program, as we discuss in Section 5.2. PDGs can then further be separated into regions by inserting special nodes into the graph. Some advances have been made in computing control dependence [8] which benefit applications that use PDGs for slicing or optimization and only require the computation of region nodes in the control graph. Escort cannot take advantage of these changes to the PDG because we depend on storing the all edges from each basic block that terminates in a conditional branch to every basic block which is control dependent on it.

This paper builds off of the work of the LLVM framework [14], which provides an Intermediate Representation (IR) which program analysis tools can operate on. LLVM’s IR is a small instruction set (31 opcodes) which acts as an in-between state for assembly languages and higher level languages such as C, C++, Go, and Haskell. The IR represents values in a program in Static Single Assignment (SSA) form, which means that no named value in the program is assigned to more than once. To represent memory the IR reduces all memory operations to load and store instructions which use typed pointers to memory objects, and the IR uses $\phi$-functions to merge values which should have the same name but have assignments along different control paths, making it possible to produce SSA form for assignments in arbitrary control flow locations in the code.

```
01:  unsigned gcd(unsigned x, unsigned y) {
02:      if(x == y) {
03:          return x;
04:      } else if(x < y) {
05:          return gcd(x, y - x);
06:      } else {
07:          return gcd(x - y, y);
08:      }
09:  }
```

Figure 1: Example code for calculating the greatest common denominator of two numbers in C.

Figure 2 is a visual representation of the LLVM IR that would be generated by compiling the C function in Figure 1. LLVM treats instructions, basic blocks, and functions as objects and provides an API for writing static
2.2 Side Channels

Side-channel attacks are ways that an attacker can reason about the secrets of a program by observing something other than the program’s output. Such attacks may use physical side channels, such as monitoring the heat generation of a processor during a secret computation [24, 5] or the time required for a program to respond to an input [20, 4]. Attacks may also use digital side channels [21], which are information available to a malicious program such as the instruction pointer of another running program. Physical side channels usually require the attacker to have physical access to the device running the program with secrets, but digital side channels only require the attacker to be able to run their own malicious programs on the same hardware as the victim program.

Researchers have demonstrated digital side channel attacks which steal keys from encryption libraries such as AES [19, 4] and RSA [20]. These attacks exploited memory access patterns in popular encryption algorithms to develop models that allowed them to accurately predict the keys being used based on data observable from a thread running malicious code at the same time and on the same machine as their victim. Implementations of the same algorithms have also been shown to be vulnerable to attacks based on the timing patterns of individual arithmetic instructions [10, 3]. Other researchers have identified problems with OS process isolation [12] which unintentionally leak information about programs running on the same system.

Molnar et al. [18] defined a formal model for dealing with side channels called the Program Counter Model. In this threat model the attacker has full access to the value of the program counter at each step of a program’s execution; this is feasible for an attacker monitoring process information through a digital side channel, but is also shown to cover other side channels such as some timing attacks. They then introduced Program Counter Security which requires that such information be of no use to an attacker. This model of security does not defend against all side channels, or even all timing attacks, since it relies on the assumption that the hardware does not leak information during execution, which has been demonstrated to be false [10, 3, 4]; however, their security model is still useful for reasoning about the nature of side channel protections. In this paper we are concerned with execution traces, which refer generally to which instructions are being executed, the order they are executed in, how long each instruction takes, and the memory access pattern observable by the hardware. An attacker with access to an execution trace has strictly more information than an attacker with access to the program counter.

2.3 Taint Analysis

Taint analysis is a class of program analysis often of interest to security which tries to identify which values affect other values in a program. Typical taint analysis is centered around three sets: sources, sanitizers, and sinks. Sources of taint are methods in a program whose return values are untrusted, usually functions related to user input. Sinks are methods which perform security-sensitive computation and should not act on tainted data. Sanitizers are methods that act on tainted data and produce taint-free output. Taint “flows” from sources through unmarked functions and operations either as the tainted data is used or at places where control flow depends on tainted information. Most security focused taint analyzers detect locations where taint is allowed to flow into a sink along a path that has no sanitizer.

This paper is concerned with a different model of taint which is particular to side channels. Since Escort is concerned with closing side channels, it is not enough to prevent untrusted data from entering security critical computations; rather, taint in Escort flows from secure values. There are many computations which act as points for data leakage through side channels and Escort considers each of these sinks into which tainted data cannot flow. Sanitizers in Escort’s security model are functions through which taint cannot flow; instead of being used to process data as in the typical model, they act as barriers in the call stack through which taint does not flow. This is useful for protecting the behavior of high level, non-secret computations from unnecessary instrumentation.

It is worth noting that TPA provides its own implementation of taint analysis, which is not used in favor of the version built on top of Escort’s PDG because it does not match with Escort’s model of taint.

3 Related Work

Program Dependence Graphs have been used to detect security problems by many other tools [26, 11, 13, 25].
often with the goal of locating code paths where attacker controlled inputs can impact security critical functions. Rodrigues et al. [24] created a sparse representation of a PDG specifically to track information flow, which has performance benefits over the algorithm implemented in Escort, but relies on properties of SSA’s $\phi$-functions which Escort does not preserve.

Dynamic taint analysis tools such as TaintEraser [27] allow users and developers to detect when information is being leaked, but using dynamic tooling during development requires a security test suite that provides full path coverage in order to give confidence that the application is secure. This means that they are not ideal tools for the production of secure programs.

GhostRider [15] is a security system that prevents programs from leaking information through memory access patterns, but it requires both conversion to a security-typed language and custom hardware to execute instrumented programs. Similarly, HIDE [28] introduced a cache design that allowed cache accesses to protect from leaking memory access information, which incurs a less significant slowdown than GhostRider but still requires custom hardware to deploy.

Molnar et al. [18] introduced and proved the security of a source-to-source C transformation that ensures Program Counter Security. Their transformation relies on conditional assignments similar to what Escort produces, but their system does not close side channels which rely on timing of individual operations or memory performance. Crane et al. [7] introduced an LLVM module pass which inserts randomly placed loads and creates dynamic control-flow diversity to reduce the extent that an attacker can conclude information based on a statistical analysis of a program’s execution. Their approach presents a tunable trade-off between runtime performance and information hiding, but cannot guarantee that a transformation will provide no information through side channels as Escort does in cases when there is no variance in the execution trace of a program.

Escort [22] is the successor of a tool called Raccoon [21], whose goal was to provide developers with a way to defend against a broad class of data leaking digital side channels without altering the behavior of the code or introducing significant slowdowns. Raccoon’s protection did not include Escort’s support for timing safe floating point operations; it focused entirely on preventing data leakage from malicious access to memory performance and the instruction pointer. The approach Raccoon took to defend against these was to introduce predicated memory instructions which replaced calls to load and store, and to calculate the necessary predicates and store them in a series of temporary buffers. Raccoon would then modify the program’s control flow to introduce jumps that surrounded every branch in the program, ensuring that each branch was executed enough times for all paths to be taken exactly once; each time one of these jumps is followed the correct predicates for all memory operations along the current path are loaded so that only the real path’s values are written to memory, preserving the semantics of the program. Escort uses the same system of memory operation predication, but stores the predicates in a different way and uses a different system for control flow protection that ensures that every basic block that is part of a branch is executed exactly once.

4 Workings of Escort

Escort is a simple, unified, and automatic solution to closing a wide class of digital side channel attacks. It provides safe versions of low-level memory and math operations which are known to leak data through either timing or memory related side channels. In addition to these, Escort modifies the control flow of the program to produce code with consistent execution traces that maintain the semantics of the original source code. A modified program performs all of the operations it would have during normal execution, as well as additional dummy operations to prevent attackers from drawing inferences about program secrets from the execution trace.

4.1 Timing-Safe Instructions

The safe operations that Escort provides can be separated into two categories, arithmetic operations and memory operations. The memory operations can further be divided into direct access operations and array access operations. The arithmetic operations are versions of floating point operations which are safe from timing attacks but do not depend on predication and are therefore far simpler than the other means of instrumentation. The safe array operations stream across the entire array in order to prevent leaking tainted array index values to adversaries with access to information about memory performance, but the particulars of array operations are not needed to follow the work of this paper, and as such we will focus on simple direct access predicated memory instructions.

The predicated memory instructions that Escort provides allow the program to write to memory conditionally without leaking whether or not the write actually occurred. To do this, every write in Escort begins by reading the value at the target location of the store and then using a predicated copy, shown in Figure 3 which allows the program to choose to write either the old or new value to memory based on the operation’s predicate. This way an adversary with access to memory side channels such as cache behavior will not be able to tell whether a new value was committed to memory or not. In the case
4.2 Control Flow Obfuscation

In addition to replacing store instructions with predicated writes, Escort changes the targets of branches in the program to reduce the variance in the execution traces the program will produce. Since the primary goal is to reduce the number of different paths through the program, Escort replaces as many conditional branches between basic blocks with unconditional jumps as possible. Preserving the behavior of the program without relying on conditionals requires computing predicates at any point in the program which was formerly on a path that was only sometimes taken due to a conditional branch. Escort uses the algorithm presented in Figure 5 to assign predicates to each basic block based on the conditions required for that basic block to be reached.

1: for each basic block $bb$ in function do
2:   if entry block($bb$) then
3:     $pred[bb] \leftarrow$ true
4:   else
5:     $pred[bb] \leftarrow$ false
6:   end if
7: end for
8: for each basic block $bb$ in function do
9:   $br \leftarrow$ branch($bb$)
10:   if unconditional branch($br$) then
11:     $\{s\} \leftarrow$ successors($bb$)
12:     $pred[s] \leftarrow pred[s] \lor pred[bb]$
13:     $pred[s] \leftarrow$ simplify($pred[s]$)
14:   else $\triangleright$ Conditional Branch.
15:     $\{s_1, s_2\} \leftarrow$ successors($bb$)
16:     if loop condition branch($br$) then
17:       $\triangleright$ Skip branches that represent loops.
18:       $pred[s_1] \leftarrow pred[s_1] \lor pred[bb]$
19:       $pred[s_2] \leftarrow pred[s_2] \lor pred[bb]$
20:     else
21:       $p \leftarrow$ condition($br$)
22:       $pred[s_1] \leftarrow pred[s_1] \lor (pred[bb] \land p)$
23:       $pred[s_2] \leftarrow pred[s_2] \lor (pred[bb] \land \lnot p)$
24:     end if
25:     $pred[s_1] \leftarrow$ simplify($pred[s_1]$)
26:     $pred[s_2] \leftarrow$ simplify($pred[s_2]$)
27:   end if
28: end for

Figure 5: Algorithm for predicating basic blocks.

This process extends to arbitrarily nested conditional statements and complex predicates by using the Z3 theorem prover [17] to remove redundancy in predicates. Because the new write operations are executed unconditionally and the transformed code contains mostly unconditional branches, code from any basic blocks which have been instrumented can all be combined into a single basic block for efficient, straight-line execution. Escort does not combine these into a larger basic block, but leaves the opportunity for additional compiler optimizations which are part of the LLVM optimization suite.

4.3 Protection for Loops

Instrumenting loops with Escort requires additional complexity on top of typical predication because not all loops can be straightened into code with a uniform execution trace. Rather than attempting to unroll loops by default, Escort performs other transformations to ensure the
safety of the loop’s execution which may then result in
code which the compiler can unroll in a later optimiza-
tion pass. LLVM provides a simplified representation of
loops by using basic blocks connected by branch instruc-
tions which may include back edges, and while this sim-
plifies some of the concerns of having different looping
control structures, detecting and instrumenting loops re-
mains a difficult problem. The ability of Escort to in-
crease the safety of a loop while preserving its behav-
ior will be improved by the addition of the PDG and its
pointer information, which will be discussed in Section

5.2.4.

Our goal is for Escort to protect loops whose num-
ber of iterations, or trip count, depends on a secret value
from leaking information about that secret. This could
happen either because the termination condition of the
loop involves a tainted value or because the loop uses
a control flow construct such as break or return on a
control tainted path. (The continue operation generates
LLVM IR which can be handled by Escort using the rules
put forth for the general cases of conditionals.) When
there is an upper bound on the iteration count and tainted
control flow could cause the loop to exit early, Escort en-
ures that the loop’s trip count is always the maximum
by adding dummy iterations after the iteration in which
the loop would have exited. When the termination con-
tion is tainted, there is no way to ensure a common up-
per bound for all loop executions, so Escort performs an
exponential back off, checking the termination condition
only once every $2^n$ iterations; this pads the trip count with
dummy iterations to the nearest power of two, which is a
compromise between leaking data and preserving a rea-
sonable runtime.

4.4 Function Level Transformations

Since an entire function could be called but not be al-
lowed to update memory, each instrumented function is
given its own predicate which is passed to it from its
caller and is used to calculate all of its internal predi-
cates. To do this, every function instrumented by Escort
is cloned into a new function with an additional argument
in order to pass the predicate, and the previous function
is replaced with a call to the clone with a predicate of
ture. This way, Escort can replace calls to the origi-
nal function with predicated calls in other functions that
it transforms, but the interface provided by the original
function is preserved for uninstrumented call sites.

5 Escort’s Program Dependence Graph

The PDG implementation in Escort is a simplification of
the structure originally proposed by Ferrante et al. In
Escort, the control dependences are calculated and
stored for each basic block in each tainted function using
the algorithm from the original paper. Data dependences
are provided through the integration with TPA and Es-
cort does not store local copies of them. The PDG then
traverses these edges to propagate taint to each instruc-
tion in the function. Additionally, Escort calculates and
stores its own version of the call graph, which is lever-
aged by the PDG for simple interprocedural taint propa-
gation. As discussed in Section 5.2.4, the PDG is traversed ad-
ditional times to calculate the loop induction values for
any loop marked for instrumentation.

5.1 Constructing the PDG

5.1.1 Control Dependences

The algorithm to construct control dependences creates
edge lists in $O(N^2)$ time in the number of basic blocks
in the function. Informally, a basic block $X$ is control
dependent on another basic block $Y$ if the terminating
instruction of $X$ chooses between a path where $X$ is exe-
cuted and a path where $X$ is not executed. The algorithm
described in the original paper is based around the more
technical definition of control dependence: A basic block
$Y$ is control dependent on another basic block $X$ if and
only if,

1. $X$ is not post-dominated by $Y$ and
2. there is a path in the control flow graph from $X$ to $Y$
   which goes through another basic block, $Z$, which
   is post-dominated by $Y$.

The information on dominance and post-dominance is
provided by LLVM’s own internal analysis passes, which
Escort then coerces slightly into several data structures
for efficient access, including a post-dominator tree for
the efficient calculation of control dependence. The first
step in the algorithm is to construct the set $S$ of all edges
$(A,B)$ in the control flow graph such that $B$ is not an an-
cestor of $A$ in the post-dominator tree, shown in Figure 6.

Then, for each of the edges $(A,B)$ in $S$, we traverse back-
wards in the post-dominator tree starting at $B$ until we
reach either $A$ or its parent, marking every node we visit
along the path (including $B$ but not the final node) con-
trol dependent on $A$, as shown in Figure 7. Ferrante et al.
provided a brief proof of correctness of this algorithm in
the presence of arbitrary control flow graphs, along with
the runtime bounds for each step in the algorithm.

5.1.2 Callgraph and Interprocedural Edges

The callgraph in Escort is constructed by iterating across
every instruction in every function searching for LLVM’s
call instruction, and inserts the target of each call to a
set which represents the callees of the current function;
1: for each basic block \( bb \) in function do
2:   for each basic block \( \text{succ} \) in successors of \( bb \) do
3:     if \( \text{succ}.\text{post_dominates}(bb) \) then
4:       \( s_{\text{edges}} \leftarrow (\text{succ},bb) \)
5:     end if
6:   end for
7: end for

Figure 6: Algorithm for populating the set \( S \) of edges.

1: for each edge \((a,b)\) in \( S \) do
2:   \( \text{parent} \leftarrow \text{immediate_postdominator}(a) \)
3:   \( \text{current} \leftarrow b \)
4:   repeat
5:     \( \text{control_deps}[a].\text{insert}(\text{current}) \)
6:     \( \text{current} \leftarrow \text{immediate_postdominator}(\text{current}) \)
7:   until \( \text{current} == a \) \( \text{||} \) \( \text{current} == \text{parent} \)
8: end for

Figure 7: Algorithm for calculating the basic blocks which are control dependent on each basic block.

While interprocedural and intraprocedural taint are calculated at the same time, the primary purpose of interprocedural tainting is to control which functions are added to the worklist of functions to be analyzed for intraprocedural taint. Therefore, we will first consider how taint propagates within a single isolated function and then address what causes taint to flow between functions.

### 5.2.2 Intraprocedural Taint

When a function is removed from the tainted function worklist, the taint within the function is pushed iteratively along both control dependence and data dependence edges in the graph until a fixed point is reached. Starting with the source(s) of taint in the function, a worklist of instructions is created. When an instruction is removed from this list, it represents a tainted operation, and every instruction that depends on it must also be marked as tainted. Different types of instructions represent several different cases:

1. Values are LLVM instructions that compute a number based on inputs; if a value is tainted, then every use of that value is also marked tainted.

2. Branches are terminators of basic blocks; if a branch instruction is tainted that means that its parent basic block represents a branch point in paths of the program which is tainted, and all instructions in all blocks which are control dependant on the parent should be marked tainted.

3. Store instructions are an LLVM primitive which is created to reference a location in memory, such as a pointer struct or array; if a store is tainted, then every instruction which reads a value at an address that the store could update must be marked as tainted.

Case 1 above is provided trivially by the def-use chaining available within LLVM’s SSA IR. Case 2 uses the control dependence edges of the PDG to propagate taint to all basic blocks whose execution could be affected by which path is taken at that branch. Case 3 uses the information from the pointer analysis to determine which instructions are capable of reading the value that is written; TPA is flow-sensitive, so objects in memory behave like the def-use chains in the SSA but with additional potential updates due to the uncertainty created by the pointer’s possible values.

When an instruction is marked as tainted, it is added to a set of instructions to prevent it from being marked multiple times and is then added to the instruction worklist in order to push its taint further into the function. Once the worklist of instructions empties, it means that no recently
visited instruction marked any instruction with taint for the first time, and a fixed point has been reached.

5.2.3 Interprocedural Taint

Taint travels through procedures at the points where function calls interact with either tainted data or control flow of the program. There are several ways in which a function call can transfer taint either to its callee or the calling function, and the approach by Escort’s PDG is safe for all of these circumstances. There are two ways that a function can be marked as tainted: control taint or data taint. The differences between these exist in how the PDG handles traversing the function and not in the representation that is stored for access by the rest of the pass.

Data taint can travel in or out of a function. When the return value of the function is tainted by the intraprocedural propagation, the PDG uses the callgraph to find all of the sites that call this function and marks the value which holds the tainted result as tainted. When a function modifies a global value or a memory object that is read outside of the function, the PDG leverages the TPA and the nature of LLVM’s SSA IR in order to find all uses following the tainted write and mark those and their parent functions for further processing. When a function is marked in this way, it is then added to the worklist to be processed in isolation, which may cause functions to be processed more than once if new taint sources are detected, but not if the new taint source was already marked for transformation by previous taint propagation. Tainted data flows into a function when one of the arguments to the function, or a memory object pointed to by one of the arguments, is tainted. Escort assumes that if any argument is tainted then all arguments to the function are considered tainted; this is because Escort’s functional analysis is not context-sensitive and only one instrumented version of each function is produced. This approach is safe since it strictly overestimates the number of tainted arguments, but changing these rules could have performance benefits, which are discussed in Section 7.

A function can also be tainted because it occurs at a call site which is control dependant on a tainted value. In this instance, the function must be instrumented to be run with a function level predicate that influences the behavior of all memory instructions in the function. Because this form of taint requires instrumentation that is a superset of the instrumentation that could be performed after normal taint propagation, once a function is control tainted it receives a special marker that prevents further intraprocedural analysis on it. The PDG only needs to perform a single pass over all instructions in the function to mark everything as tainted, and it also uses the call graph to recursively mark any function that could be called as a result of code in the control tainted function, since those call sites are also by definition control tainted.

5.2.4 Loop Inductive Values

As discussed in Section 4.3, loops require special transformation to reduce the performance impact of the obfuscation. Escort attempts to hide as much information leakage through differences in trip count as possible while guaranteeing that the loop is not prevented from making progress by the transformations and only commits the same values to memory that would have been committed in an uninstrumented execution of the loop. In order for loops to make progress, any load of values that are used in the calculation of the loop termination condition must read the value that would have been written if the store operations in the function were uninstrumented. The PDG enables Escort to determine which values belong to this set and instrument those values with an additional, loop-local, dummy variable.

Marking the values required for a loop’s induction is a backward traversal in the PDG. Starting from each branch that is identified as the head of a loop, the PDG travels backward along both the control dependence edges and the data flow edges to find all instructions that were involved in the calculation of the loop exit condition. These instructions are stored in a set which is mapped to the original loop branch so they can be accessed later in the Escort pass. Each store in this set is converted to two different safe writes, one which is identical to a non-loop inductive variable’s write and one which has predicate local to the loop. All uses of the loop inductive values inside of the loop are swapped to read from the dummy instead of the actual value; this does not change the behavior of the loop on a true path, but on a decoy path it enables the loop induction value to be calculated correctly without updating memory that is live outside of the loop and violating the semantics of the program.

6 Evaluation

6.1 Simplified Test Cases

The primary means of ensuring the correctness of Escort’s behavior is a suite of regression tests inherited from Raccoon. These tests were developed by hand to demonstrate particular issues that could be encountered while instrumenting small programs. Each test abuses the interaction between Escort and functions with I/O side effects to produce output to stdout which contains information on paths through the program’s execution equivalent to what could be observed by an attacker in the Program Counter Security model. There are tests
covering a range of possible program constructs, such as different configurations of nested conditionals and programs with different types of threading behavior. During the PDG integration process, additional tests were added to test the success of intraprocedural and interprocedural tainting, as well as different loop constructs with different tainted exit conditions.

### 6.3 Discovered Limitations

Using the TPA and the PDG to in an attempt to scale Escort to larger programs revealed significant problems that remain to be solved. In order to prevent any information from leaking due to memory access patterns, a store to a pointer whose memory location is tainted must perform predicted writes to each possible pointee value. While this is a simple extension of Escort’s strategy in theory, once the PDG was used to propagate information interprocedurally we ran into problems with this implementation. A store with a possible pointee which is not local to the function can result in Escort generating writes to values which are not defined on all paths to the writes.

Consider the code provided in Figure 8. Since the value of pointer is dependent on the tainted value secret, any unconditional store to pointer. Therefore, the assignment to the location to x inside of baz must be transformed by Escort to be a predicated write to both a and b. However, since we are not using a context-sensitive pointer analysis, the possible memory locations pointed to by x inside of baz are a, b and c, and three predicated writes are created. Now, no matter what path is taken through the program, when baz is executed it attempts to load from and then store to a location which has not been initialized.

Preventing unsafe writes like this requires knowledge about which memory locations in the points-to set will be initialized at each function invocation; this information is lost in a context-insensitive analysis. Context-sensitivity has additional potential benefits, as discussed in Section 7, but is an unappealing solution this problem because requiring it greatly increases the minimum cost of Escort’s analysis.

### 7 Future Work

There are clear next steps for this project in two different directions: language support and runtime improvements. The addition of the PDG has not expanded Escort’s ability to transform C language features enough to handle common programs. In addition to the case of pointer arguments described in 6.3 several constructs are still unimplemented or untested, such as tainted function pointers. Escort has not run up against any fundamental limitations in its ability to handle these features, and the project should continue to grow in its generality over time. Improvements can also be made to the specificity of the tainting process in order to gain marginal benefits in program execution speed. First, more metadata should be stored per function to allow for greater granularity of argument tainting. In addition, no experiments have been done to assess the impact of a context-sensitive pointer analysis or taint analysis on the size or runtime performance of the resulting binary, but prior security research indicates that context-sensitivity reduces the set of program locations marked as insecure. This means that context-sensitivity could allow for significant per-

```c
01: void foo(bool secret) {
02:   int *pointer, a, b;
03:   if (secret) {
04:     pointer = &a;
05:   } else {
06:     pointer = &b;
07:   }
08:   baz(pointer);
09: }
10: 
11: void bar() {
12:   int c;
13:   baz(&c);
14: }
15: 
16: void baz(int* x) {
17:   *x = 5;
18: }
```

Figure 8: A series of function calls that generates invalid store instructions when instrumented with Escort.
formance boosts at the cost of additional instrumentation time and additional cloned copies of some functions. It is also possible to implement a client-driven approach to this sensitivity which assesses the potential benefit of partially instrumented versions of functions at callsites which are not tainted.

8 Conclusion

Escort’s control flow predication and safe instructions present a protection against a wide range of digital side channel attacks that will become more useful the more it is able to scale. The addition of a PDG is an important step in extending Escort’s ability to correctly instrument complicated C programs. Even though the work in this paper was not able to make it possible to instrument a wide variety of common programs, the information gathered by the analysis and the techniques learned in building the PDG have advanced the overall state of the project. Ideally, once the limitations uncovered during this work have been surpassed, the PDG’s taint analysis may still prove to be useful in reducing the performance impact of Escort’s instrumentations on larger programs.

9 Acknowledgements

I could never have written this paper without the help of my advisor, Dr. Lin, and his student Ashay Rane. Ashay has been working with digital side channel defenses for a long time and without his work on both Escort and Raccoon there would have been no framework for me to apply this analysis to.

References

