# Architecture-Independent Dynamic Information Flow Tracking *

Ryan Whelan[1], Tim Leek[2], and David Kaeli[1]

[1] Department of Electrical and Computer Engineering
Northeastern University, Boston, MA USA
{rwhelan, kaeli}@ece.neu.edu
[2] Cyber System Assessments Group
MIT Lincoln Laboratory, Lexington, MA USA
tleek@ll.mit.edu

**Abstract.** Dynamic information flow tracking is a well-known dynamic software analysis technique with a wide variety of applications that range from making systems more secure, to helping developers and analysts better understand the code that systems are executing. Traditionally, the fine-grained analysis capabilities that are desired for the class of these systems which operate at the binary level require tight coupling to a specific ISA. This places a heavy burden on developers of these systems since significant domain knowledge is required to support each ISA, and the ability to amortize the effort expended on one ISA implementation cannot be leveraged to support other ISAs. Further, the correctness of the system must carefully evaluated for each new ISA.

In this paper, we present a *general* approach to information flow tracking that allows us to support multiple ISAs without mastering the intricate details of each ISA we support, and without extensive verification. Our approach leverages binary translation to an intermediate representation where we have developed detailed, architecture-neutral information flow models. To support advanced instructions that are typically implemented in C code in binary translators, we also present a combined static/dynamic analysis that allows us to accurately and automatically support these instructions. We demonstrate the utility of our system in three different application settings: enforcing information flow policies, classifying algorithms by information flow properties, and characterizing types of programs which may exhibit excessive information flow in an information flow tracking system.

**Keywords:** Binary translation, binary instrumentation, information flow tracking, dynamic analysis, taint analysis, intermediate representations

* This work was sponsored by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

# 1 Introduction

Dynamic information flow tracking (also known as dynamic taint analysis) is a well-known software analysis technique that has been shown to have wide applicability in software analysis and security applications. However, since dynamic information flow tracking systems that operate at the binary level require fine-grained analysis capabilities to be effective, this means that they are generally tightly coupled with the ISA of code to be analyzed.

In order to implement a fine-grained analysis capability such as information flow tracking for an ISA of interest, an intimate knowledge of the ISA is required in order to accurately capture information flow for each instruction. This is especially cumbersome for ISAs with many hundreds of instructions that have complex and subtle semantics (e.g., x86). Additionally, after expending the work required to complete such a system, the implementation only supports the single ISA, and a similar effort is required for each additional ISA. To overcome this challenge, we've elected to take a compiler-based approach by translating architecture-specific code into an architecture-independent intermediate representation where we can develop, reuse, and extend a single set of analyses.

In this work, we present Pirate: a Platform for Intermediate Representation-based Analyses of Tainted Execution. Pirate decouples the tight bond between the ISA of code under analysis and the additional instrumentation code, and provides a general taint analysis framework that can be applied to a large number of ISAs. Pirate leverages QEMU [4] for binary translation, and LLVM [14] as an intermediate representation within which we can perform our architecture-independent analyses. We show that our approach is both *general* enough to be applied to multiple ISAs, and *precise* enough to provide the detailed kind of information expected from a fine-grained dynamic information flow tracking system. In our approach, we define detailed byte-level information flow models for 29 instructions in the LLVM intermediate representation which gives us coverage of thousands of instructions that appear in translated guest code. We also apply these models to complex guest instructions that are implemented in C code. To the best of our knowledge, this is the first implementation of a binary level dynamic information flow tracking system that is general enough to be applied to multiple ISAs without requiring source code.

The contributions of this work are:

- A framework that leverages dynamic binary translation producing LLVM intermediate representation that enables architecture-independent dynamic analyses, and a language for precisely expressing information flow of this IR at the byte level.
- A combined static/dynamic analysis to be applied to the C code of the binary translator for complex ISA-specific instructions that do not fit within the IR, enabling the automated analysis and inclusion of these instructions in our framework.
- An evaluation of our framework for x86, x86_64, and ARM, highlighting three security-related applications: 1) enforcing information flow policies, 2)

characterizing algorithms by information flow, and 3) diagnosing sources of state explosion for each ISA.

The rest of this paper is organized as follows. In Section 2, we present background on information flow tracking. Sections 3 and 4 present the architectural overview and implementation of Pirate. Section 5 presents our evaluation with three security-related applications, while Section 6 includes some additional discussion. We review related work in Section 7, and conclude in Section 8.

## 2 Background

Dynamic information flow tracking is a dynamic analysis technique where data is labeled, and subsequently tracked as it flows through a program or system. Generally data is labeled and tracked at the byte level, but this can also happen at the bit, word, or even page level, depending on the desired granularity. The labeling can also occur at varying granularities, where each unit of data is also accompanied by one bit of data (tracked or not tracked), one byte of data (accompanied by a small number), or a data structure that tracks additional information. Tracking additional information is useful for the cases when *label sets* are propagated through the system. In order to propagate the flow of data, major components of the system need a shadow memory to keep track of where data flows within the system. This includes CPU registers, memory, and in the case of whole systems, the hard drive also. When information flow tracking is implemented for binaries at a fine-grained level, this means that propagation occurs at the level of the ISA where single instructions that result in a flow of information are instrumented. This instrumentation updates the shadow memory accordingly when tagged information is propagated.

Information flow tracking can occur at the hardware level [7, 26, 28], or in software through the use of source-level instrumentation [12, 15, 31], binary instrumentation [10, 13, 22], or the use of a whole-system emulator [9, 20, 24]. In general, hardware-based approaches are faster, but less flexible. Software-based approaches tend to have higher overheads, but enable more detailed dynamic analyses. Source-level approaches tend to be both fast and flexible, but are sometimes impractical when source code is not available. These techniques have proven to be effective in a wide variety of applications, including detection and prevention of exploits, malware analysis, debugging assistance, vulnerability discovery, and network protocol reverse engineering.

Due to the popularity of the x86 ISA, and the tight bond of these binary instrumentation techniques with the ISA under analysis, many of these systems have been carefully designed to correctly propagate information flow only for the instructions that are included in x86. This imposes a significant limitation on dynamic information flow tracking since a significant effort is required to support additional ISAs. Pirate solves this problem by decoupling this analysis technique from the underlying ISA, without requiring source code or higher-level semantics.

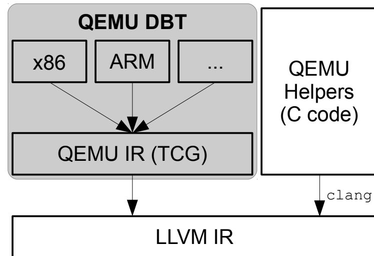



**Fig. 1.** Lowering code to the LLVM IR with QEMU and Clang

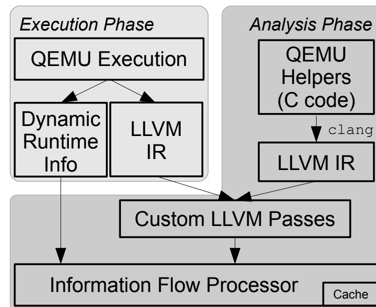



**Fig. 2.** System architecture split between execution and analysis

## 3 System Overview

At a high level, PIRATE works as follows. The QEMU binary translator [4] is at the core of our system. QEMU is a versatile dynamic binary translation platform that can translate 14 different guest architectures to 9 different host architectures by using its own custom intermediate representation, which is part of the Tiny Code Generator (TCG). In our approach, we take advantage of this translation to IR to support information flow tracking for multiple guest architectures. However, the TCG IR consists of very simple RISC-like instructions, making it difficult to represent complex ISA-specific instructions. To overcome this limitation, the QEMU authors implement a large number of guest instructions in *helper functions*, which are C implementations of these instructions. Each guest ISA implements hundreds of instructions in helper functions, so we have devised a mechanism to automatically track information flow to, from, and within these helper functions.

### 3.1 Execution and Analysis

We perform dynamic information flow tracking on Linux binaries using the following approach, which is split between execution and analysis. In the execution phase, we run the program as we normally would under QEMU. We have augmented the translation process to add an additional translation step, which translates the TCG IR to the LLVM IR and then executes on the LLVM just-in-time (JIT) compiler. This translation occurs at the level of basic blocks of guest code. In Figure 1, we show the process of lowering guest code and helper function code to the LLVM IR to be used in our analysis.

Figure 2 shows the architecture of our system. Once we have the execution of guest code captured in the LLVM IR, we can perform our analysis over each basic block of guest code with custom IR passes we have developed within the LLVM infrastructure. The first part of our analysis is to derive information flow operations to be executed on our abstract information flow processor. This is

```
void glue(helper_pshufw, SUFFIX) (Reg *d, Reg *s, int order){
    Reg r;
    r.W(0) = s->W(order & 3);
    r.W(1) = s->W((order >> 2) & 3);
    r.W(2) = s->W((order >> 4) & 3);
    r.W(3) = s->W((order >> 6) & 3);
    *d = r;
}
```

**Fig. 3.** QEMU helper function implementing the `pshufw` (packed shuffle word) MMX instruction for x86 and x86_64 ISAs

an automated process that emits information flow operations that we've specified for each LLVM instruction. After deriving the sequence of information flow operations for a basic block of code, we execute them on the information flow processor to propagate tags and update the shadow memory. This allows us to keep our shadow memory updated on the level of a guest basic block. When we encounter system calls during our processing, we treat I/O-related calls as sources or sinks of information flow. For example, the `read()` system call is a source that we begin tracking information flow on, and a `write()` system call is a sink where we may want to be notified if tagged information is written to disk.

### 3.2 Optimizations

QEMU has a *translation block cache* mechanism that allows it to keep a copy of translated guest code, avoiding the overhead of re-translating frequently executed code repeatedly. This optimization permeates to our analysis phase; a guest basic block that is cached may also be executed repeatedly in the LLVM JIT, but it only appears once in the LLVM module that we analyze. This optimization in QEMU also provides us the opportunity to cache information flow operations. As we analyze translated guest code in the representation of basic blocks that may be cached, we can also perform the derivation of information flow tracking operations once, and then cache; we refer to these as *taint basic blocks*. Once we derive a taint basic block for a guest basic block of code, we deposit it into our *taint basic block cache*.

### 3.3 Static Analysis of QEMU Helper Functions

Since QEMU helper functions perform critical computations on behalf of the guest, we need to include them in our analysis as well. To do this, we use Clang [1] to translate helper functions to the LLVM IR. From there, we can use the exact same analysis that we apply to translated guest code to derive information flow operations. Since this is a static analysis, we can emit the corresponding information flow operations for each helper function into a *persistent cache*. Figure 3 shows the helper function that implements the `pshufw` MMX instruction. Automatically analyzing functions like these takes a significant burden off of developers of information flow tracking systems.

## 4 Implementation

Next, we present implementation details of PIRATE. The implementation consists of several major components: the execution and trace collection engine, the information flow processor which propagates tagged data, the shadow memory which maintains tagged data, the analysis engine which performs analysis and instrumentation passes over the LLVM IR, and the caching mechanism that caches information flow operations and reduces overhead. In this paper, we support information flow tracking for the x86, x86_64, and ARM ISAs on Linux binaries, but our approach can be extended in a straightforward manner to other ISAs that are supported by QEMU. Our system is implemented with QEMU 1.0.1 and LLVM 3.0.

### 4.1 Dynamic Binary Translation to LLVM IR

At the core of our approach is the translation of guest code to an ISA-neutral IR. Much like a standard compiler, we want to perform our analyses in terms of an IR, which allows us to decouple from the ISA-specific details. We take advantage of the fact that QEMU's dynamic binary translation mechanism translates guest code to its own custom IR (TCG), but this IR is not robust enough for our analyses. In order to bridge the gap between guest code translated to the TCG IR and helper functions implemented in C, we chose to perform our analysis in the LLVM IR. Since both the TCG and LLVM IR consist of simple RISC-like instructions, we have a straightforward translation from TCG to LLVM. For this translation, we leverage the TCG to LLVM translation module included as part of the S2E framework [8]. LLVM also enables us to easily translate helper functions to its IR (through the Clang front end), and it provides a rich set of APIs for us to work with.

By performing information flow tracking in the LLVM IR, we abstract away the intricate details of each of our target ISAs, leaving us with only 29 RISC-like instructions that we need to understand in great detail and model correctly for information flow analysis. These 29 LLVM instructions describe all instructions that appear in translated QEMU code, and all helper functions that we currently support. Developing detailed information flow tracking models for this small set of LLVM instructions that are semantically equivalent to guest code means that our information flow tracking will also be semantically equivalent. Additionally, since the system actually executes the translated IR, we can rely on the correctness of the translation to give us a degree of assurance about the completeness and correctness of our analysis. While formally verifying the translation to LLVM IR is out of scope of this work, we assume that this translation is correct since we can execute programs on the LLVM JIT and obtain correct outputs.

### 4.2 Decoupled Execution and Analysis

In PIRATE, we decouple the execution and analysis of code in order to give us flexibility in altering our analyses on a single execution. We capture a compact

**Table 1.** Information flow operations

| Operation | Semantics |
|---|---|
| `label(a,l)` | $L(a) \leftarrow L(a) \cup l$ |
| `delete(a)` | $L(a) \leftarrow \emptyset$ |
| `copy(a,b)` | $L(b) \leftarrow L(a)$ |
| `compute(a,b,c)` | $L(c) \leftarrow L(a) \cup L(b)$ |
| `insn_start` | Bookkeeping info |
| `call` | Begin processing a QEMU helper function |
| `return` | Return from processing a QEMU helper function |

dynamic trace of the execution in the LLVM bitcode format, along with dynamic values from the execution that include memory access addresses, and branch targets. We obtain these dynamic values by instrumenting the IR to log every address of loads and stores, and every branch taken during execution. The code we capture is in the format of an LLVM bitcode module which consists of a series of LLVM functions, each corresponding to a basic block of guest code. We also capture the order in which these functions are executed. Our trace is compact in the sense that if a basic block is executed multiple times, we only need to capture it once.

Once we've captured an execution, we leverage the LLVM infrastructure to perform our analysis directly on the LLVM IR. Our analysis is applied in the form of an LLVM analysis pass, where we specify the set of *information flow operations* for each LLVM instruction in the execution. We perform this analysis at the granularity of a guest basic block, and our analysis emits a *taint basic block*. Our abstract information flow processor then processes these taint basic blocks to update the shadow memory accordingly.

### 4.3 Shadow Memory, Information Flow Processor

*Shadow Memory.* Our shadow memory is partitioned into the following segments: virtual memory, architected registers, and LLVM registers (which includes multiple calling scopes). The virtual memory portion of the shadow memory keeps track of information flow through the process based on virtual addresses. The architected state portion keeps track of general purpose registers, program counters, and also some special purpose registers (such as MMX and XMM registers for x86 and x86_64) – this is the only architecture-specific component of our system. The LLVM shadow registers are how we keep track of information flow between LLVM IR instructions, which are expressed in static single assignment form with infinite registers. Currently, our shadow memory models 2,000 abstract registers, which is sufficient for our analysis. We maintain multiple scopes of abstract LLVM registers in our shadow memory to accommodate the calling of helper functions, which are explained in more detail in Section 4.5. The shadow memory is configurable so data can be tracked at the binary level (tagged or untagged), or positionally with multiple labels per address, which we refer to

as a *label set*. Since we are modeling the entire address space of a process in our shadow memory, it is important that we utilize an efficient implementation. For 32-bit ISAs, our shadow memory of the virtual address space consists of a two-level structure that maps a directory to tables with tables that map to pages, similar to x86 virtual addressing. For 64-bit ISAs, we instead use a five-level structure in order to accommodate the entire 64-bit address space. To save memory overhead, we only need to allocate shadow guest memory for memory pages that contain tagged information.

*Deriving Information Flow Operations.* On our abstract information flow processor, we execute information flow operations in order to propagate tags and update the shadow memory. These operations specify information flow at the byte level. An address can be a byte in memory, a byte in an architected register, or a byte in an LLVM abstract register. The set of information flow operations can be seen in Table 1. Here, we describe them in more detail:

- `label`: Associate label $l$ with the set of labels that belong to address $a$.
- `delete`: Discard the label set associated with address $a$.
- `copy`: Copy the label set associated with address $a$ to address $b$.
- `compute`: Address $c$ gets the union of the label sets associated with address $a$ and address $b$.
- `insn_start`: Maintains dynamic information for operations. For loads and stores, a value from the dynamic log is filled in. For branches, a value from the dynamic log is read to see which branch was taken, and which basic block of operations needs to be processed next.
- `call`: Indication to process information flow operations for a QEMU helper function. Shift information flow processor from caller scope to callee scope, which has a separate set of shadow LLVM registers. Retrieve information flow operations from the persistent cache. If the helper function takes arguments, propagate information flow of arguments from caller scope to callee scope.
- `return`: Indication that processing of a QEMU helper function is finished. Shift information flow processor from callee scope to caller scope. If the helper function returns a value, propagate information flow to shadow return value register.

The information flow models we've developed allow us to derive the sequence of information flow operations for each LLVM function using our LLVM analysis pass. In this pass, we iterate over each LLVM instruction and populate a buffer with the corresponding information flow operations. In Figure 4, we show sequences of information flow operations for the LLVM `xor` and `load` instructions.

### 4.4 Caching of Information Flow Tracking Operations

One of the main optimizations that QEMU implements is the *translation block cache* which saves the overhead of retranslating guest code to host code for frequently executed basic blocks. We took a similar approach for our *taint basic*

**LLVM Instruction:**

```
%32 = xor i32 %30, %31;
```

**Information Flow Operations:**

```
compute(%30[0], %31[0], %32[0]);
compute(%30[1], %31[1], %32[1]);
compute(%30[2], %31[2], %32[2]);
compute(%30[3], %31[3], %32[3]);
```

**LLVM Instruction:**

```
%7 = load i32* %2;
```

**Information Flow Operations:**

```
// get load address from dynamic
// log, and fill in next
// four operations
insn_start;

// continue processing operations
copy(addr[0], %7[0]);
copy(addr[1], %7[1]);
copy(addr[2], %7[2]);
copy(addr[3], %7[3]);
```

**Fig. 4.** Examples of byte-level information flow operations for 32-bit `xor` and `load` LLVM instructions

*blocks* and developed a caching mechanism to eliminate the need to repeatedly derive information flow operations. This means we only need to run our pass once on a basic block, and as long as it is in our cache, we simply process the information flow operations.

Our caching mechanism works as follows. During our analysis pass, we leave dynamic values such as memory accesses and taken branches empty, and instead fill them in at processing time by using our `insn_start` operation, as illustrated in Figure 4. In the case of a branch, the `insn_start` operation tells the information flow processor to consult the dynamic log to find which branch was taken, and continue on to process that *taint basic block*. This technique enables us to process cached information flow operations with minor preprocessing to adjust for dynamic information.

### 4.5 Analysis and Processing of QEMU Helper Functions

*Instrumentation and Analysis.* Because important computations are carried out in helper functions, we need some mechanism to analyze them in a detailed, correct way. Because there are hundreds of helper functions in QEMU, this process needs to be automated. We have modified the QEMU build process to automatically derive information flow operations for a subset of QEMU helper functions, and save them to a *persistent cache*. Here, we describe that process in more detail:

1. **Translate helper function C code to LLVM IR using Clang.**
   The Clang compiler [1], which is a C front end for the LLVM infrastructure, has an option to emit a LLVM bitcode file for a compilation unit. We have modified the QEMU build process to do this for compilation units which contain helper functions we are interested in analyzing.
2. **Run our LLVM function analysis pass on the helper function LLVM.**
   Once we have helper function code in the LLVM format, we can compute

information flow operations using the same LLVM analysis pass that we have developed for use on QEMU translated code.

3. **Instrument the LLVM IR to populate the dynamic log.**
   In order for us to perform our analysis on the helper function LLVM, we need this code to populate the dynamic log with load, store, and branch values. We have developed a simple code transformation pass that instruments the helper function IR with this logging functionality.
4. **Emit information flow operations into a persistent cache.**
   Helper function information flow operations can be emitted into a persistent cache because they are static, and because runtime overhead will be reduced by performing these computations at compile time. This cache is now another by-product of the QEMU build process.
5. **Compile and link the instrumented LLVM.**
   Since the instrumented IR should populate the dynamic log during the trace collection, we create an object file that can be linked into QEMU. Again, we can use Clang to translate our instrumented LLVM bitcode into an object file, and then link that file into the QEMU executable during the QEMU build process.

*Processing.* Integration of helper function analysis into PIRATE works as follows. During analysis of QEMU generated code, we see a call to a helper function, arguments (in terms of LLVM registers, if any), and return value (in terms of a LLVM register, if any). When we see a call instruction in our analysis pass on translated code, we propagate the information flow of the arguments to the callee's scope of LLVM registers, if necessary. For example, assume in the caller's scope that there is a call to `foo()` with values `%29` and `%30` as arguments. In the scope of the helper function, the arguments will be in values `%0` and `%1`. So the information flow of each argument gets copied to the callee's scope, similar to how arguments are passed to a new scope on a stack. We then insert our `call` operation, which tells the information flow processor which function to process, and the pointer to the set of corresponding taint operations that are in the cache. The information flow processor then processes those operations until return. On return, a helper function may or may not return a value to the previous scope. For return, we emit a `return` operation to indicate that we are returning to the caller's scope. If a value is returned, then its information will be present in the LLVM return value register in our shadow memory so if there are any tags on that value, they will be propagated back to the caller's scope correctly.

## 5 Evaluation

In our evaluation, we show that PIRATE is decoupled from a specific ISA, bringing the utility of information flow tracking to software developers and analysts regardless of the underlying ISA they are targeting. We demonstrate the following three applications for x86, x86_64, and ARM: enforcing information flow policies, algorithm characterization, and state explosion characterization. We

Table 2. Functions which operate on tagged data

| Program | x86 | x86_64 | ARM |
|---|---|---|---|
| Hello World | 10/104 (9.62%) | 11/93 (11.83%) | 10/100 (10.00%) |
| Gzip Compress | 17/150 (11.33%) | 15/147 (10.20%) | 17/150 (11.33%) |
| Bzip2 Compress | 16/167 (9.58%) | 16/153 (10.46%) | 17/165 (10.30%) |
| Tar Archive | 2/391 (0.51%) | 2/372 (0.54%) | 2/361 (0.55%) |
| OpenSSL AES Encrypt | 8/674 (1.19%) | 7/655 (1.07%) | 7/671 (1.04%) |
| OpenSSL RC4 Encrypt | 4/672 (0.59%) | 4/653 (0.61%) | 5/679 (0.73%) |
| Kernighan-Lin Graph Partition | 29/132 (21.97%) | 63/122 (51.64%) | 32/134 (23.88%) |

performed our evaluation on Ubuntu 64-bit Linux 3.2, and in each case, we compiled programs with GCC 4.6 with default options for each program.

### 5.1 Enforcing Information Flow Policies

One important application of dynamic information flow tracking is to define information flow policies for applications, and ensure that they are enforced within the application. For example, one may define a policy that a certain subset of program data is not allowed to be sent out over the network, or that user-provided data may not be allowed to be passed to security-sensitive functions. A universal information flow policy that most programs enforce is that user-provided data may not be used to overwrite the program counter. However, there is a lack of information flow tracking systems that support embedded systems employing ARM, MIPS, PowerPC, and even x86_64, so defining and verifying these policies without modifying source code is difficult or impossible with existing information flow tracking systems.

Our system enables software developers to define and enforce these information flow policies, regardless of the ISA they are developing for. In one set of experiments, we carried out a buffer overflow exploit for a vulnerable program and our system was able to tell us exactly which bytes from our input were overwriting the program counter for x86, x86_64, and ARM.

In addition to telling the developer where in the program these information flow policies are violated, Pirate can also tell the developer each function in the program where tagged data flows. This can assist the developer in identifying parts of the program that operate directly on user input so they can more clearly identify where to focus when ensuring the security of their program. In Table 2, we present results for the ratio of functions in several programs that operate on tagged data. These ratios indicate the percentage of functions in the program that operate on tagged data compared with every function executed in the dynamic trace. For most of the programs we evaluated, these ratios are under 25%. The exception is KL graph partition for x86_64, which shows effects of state explosion. This is addressed in more detail in Section 5.3.

With this enhanced security capability, software developers can more easily identify parts of their programs that may be more prone to attacks. This ca-

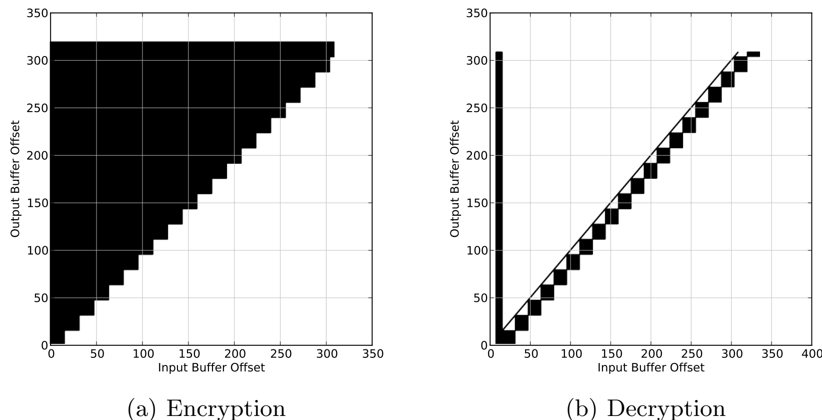



**Fig. 5.** AES CBC mode input/output dependency graphs

pability can also be used in the context of vulnerability discovery when source code isn't available in binaries compiled for different ISAs.

### 5.2 Algorithm Characterization

Recent work has shown that dynamic analysis techniques such as information flow tracking can help malware analysts better understand the obfuscation techniques employed by malware [6, 17]. However, these approaches suffer the same limitations as other systems, where they are tightly-coupled with a single ISA (x86). As embedded systems are becoming increasingly relevant in the security community, it is becoming more desirable for analysts to leverage the power of dynamic information flow tracking for these embedded ISAs.

Here, we highlight the capability of our system to characterize encryption algorithms based on *input/output dependency graphs*. We generate these graphs by positionally labeling each byte in the buffer after the `read()` system call, and tracking how each byte of the input is propagated through the encryption algorithm. By subsequently interposing on the `write()` system call, we can inspect each byte in the buffer to see the set of input bytes that influences each output byte. For these experiments, we chose OpenSSL 1.0.1c [2] as a test suite for two modes of AES block cipher encryption, and RC4 stream cipher encryption for x86, x86_64, and ARM. The OpenSSL suite is ideal for demonstrating our capability because most of the encryption algorithms have both C implementations and optimized handwritten assembly implementations.

*AES, Cipher Block Chaining Mode.* AES (Advanced Encryption Standard) is a block encryption algorithm that operates on blocks of 128 bits of data, and allows for key sizes of 128, 192, and 256 bits [25]. As there are a variety of

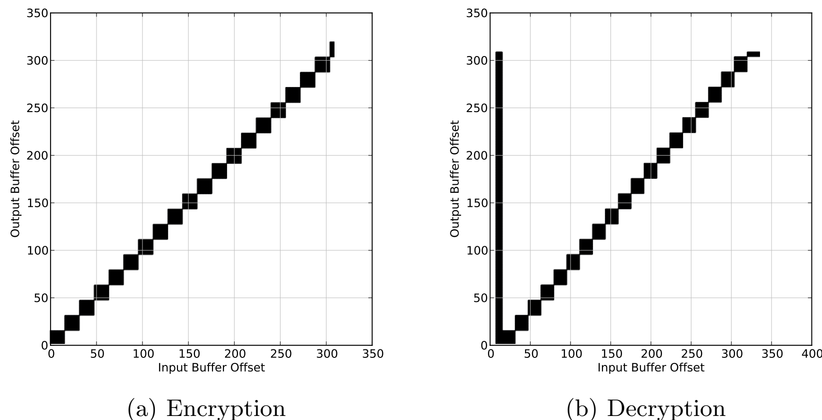
encryption modes for AES, cipher block chaining mode (CBC) is one of the stronger modes. In CBC encryption, a block of plaintext is encrypted, and then the resulting ciphertext is passed through an exclusive-or operation with the subsequent block of plaintext before that plaintext is passed through the block cipher. Inversely, in CBC decryption, a block of ciphertext is decrypted, and then passed through an exclusive-or operation with the previous block of ciphertext in order to retrieve the plaintext.

Figure 5 shows our input/output dependency graphs for AES encryption and decryption. In these figures, we can visualize several main characteristics of the AES CBC cipher: the block size (16 bytes), and the encryption mode. In Figure 5(a), the first block of encrypted data is literally displayed as a block indicating complicated dependencies between the first 16 bytes. We see the chaining pattern as each subsequent block depends on all blocks before it in the dependency graph. In Figure 5(b), we can see that each value in the output is dependent on the second eight bytes in the input; this corresponds to the salt value, which is an element of randomness that is included as a part of the encrypted file. We can also see the chaining dependency characteristic of CBC decryption, where each block of ciphertext is decrypted, and then passed through an exclusive-or operation with the previous block of ciphertext. This series of exclusive-or operations is manifested as the diagonal line in Figure 5(b). With PIRATE, we were able to generate equivalent dependency graphs for x86, x86_64, and ARM, for both handwritten and C implementations.

This result highlights the versatility of our approach based on the wide variety of implementations of AES in OpenSSL. In particular, the x86 handwritten version is implemented using instructions from the MMX SIMD instruction set. Our automated approach for deriving information flow operations for these advanced instructions allows us to support these instructions without the manual effort that other systems require.

*AES, Electronic Code Book Mode.* Electronic Code Book (ECB) mode is similar to CBC mode, except that it performs block-by-block encryption without the exclusive-or chaining of CBC mode [25]. The input/output dependency graphs we've generated to characterize this algorithm can be seen in Figure 6. Here, we see that our system can accurately tell us the block size and the encryption mode, without the chaining dependencies from the previous figures. We again see the dependence on the bytes containing the salt value in Figure 6(b).

For AES in ECB mode, we were able to generate equivalent dependency graphs for each ISA (x86, x86_64, and ARM) and implementation (handwritten and C implementation), with the exception of the ARM C implementation for decryption, and the x86 handwritten assembly implementation for encryption. In these exceptional cases, we see a similar input/output dependency graph with some additional apparent data dependence. This highlights a design decision of our system, where we over-approximate information flow transfer of certain LLVM instructions in order to prevent the incorrect loss of tagged information. This over-approximation can manifest itself as additional information flow spread, but we've made the decision that it is better to be conservative rather




(a) Encryption

(b) Decryption

**Fig. 6.** AES ECB mode input/output dependency graphs

than miss an exploit, especially in the context of security-critical applications of this system.

*RC4.* RC4 is a stream cipher where encryption occurs through byte-by-byte exclusive-or operations. The algorithm maintains a 256 byte state that is initialized by the symmetric key [25]. Throughout the encryption, bytes in the state are swapped pseudo-randomly to derive the next byte of the state to be passed through an exclusive-or operation with the next byte of the plaintext.

The input/output dependency graphs for RC4 encryption can be seen in Figure 7. Since we only track information flow of the input data and not the key, we can see from these figures that there is a linear dependence from input to output, based on the series of exclusive-or operations that occur for each byte in the file. As with the previous figures for decryption, we can see the dependence on the salt value that is in the beginning of the encrypted file in Figure 7(b). For RC4 encryption and decryption, we were able to generate equivalent dependency graphs for encryption and decryption for each ISA (x86, x86_64, and ARM) and implementation (handwritten and C implementation) with the exception of x86_64 encryption and decryption handwritten implementations. For these cases, we see an equivalent dependency with additional information, again due to the conservative approach we take in terms of information flow tracking.

### 5.3 Diagnosing State Explosion

One limitation of information flow tracking systems is that they are subject to state explosion where tagged data spreads (i.e., grows) uncontrollably, increasing the amount of data that needs to be tracked as it flows through the system. This is especially true when pointers are tracked in the same way as data [23]. Despite

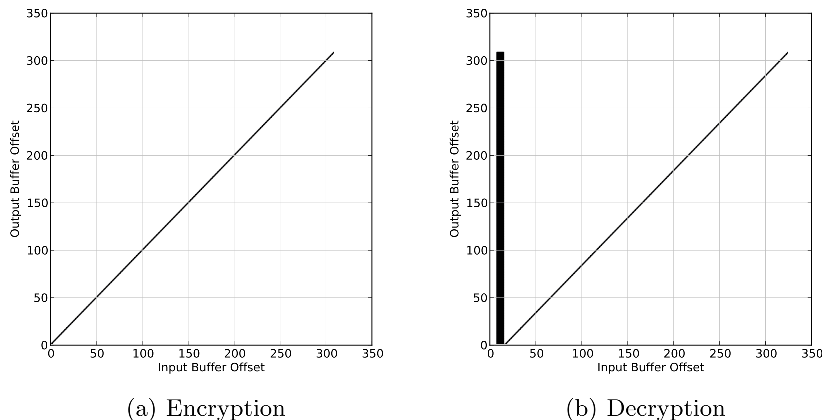
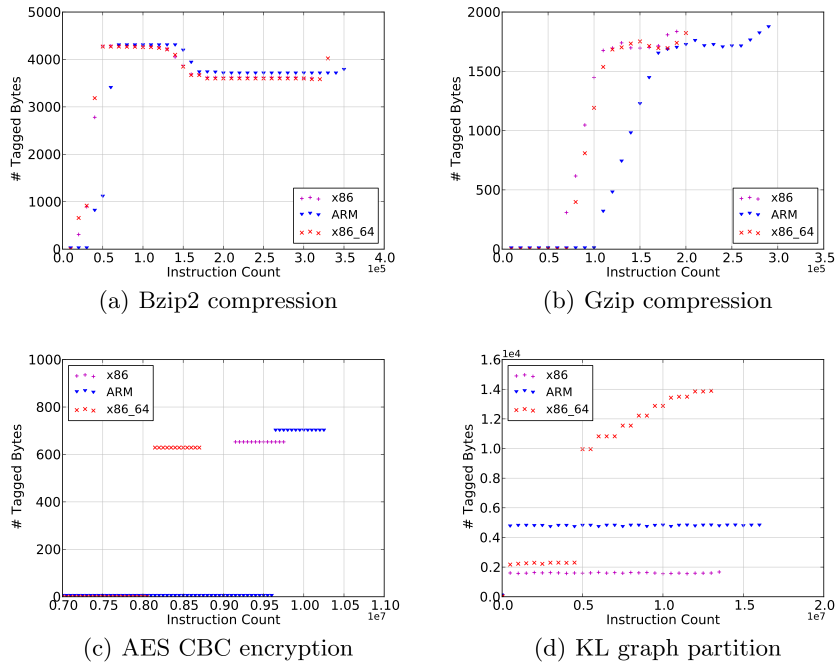



**Fig. 7.** RC4 input/output dependency graphs

this limitation, it is necessary to track pointers to detect and prevent certain kinds of attacks, such as those involved in read or write overflows but where no control flow divergence occurs [7], or those that log keyboard input which is passed through lookup tables [9]. In PIRATE, we've implemented tagged pointer tracking as a configurable option. When this option is turned on, we propagate information for loads and stores not only from the addresses that are accessed, but also from the values that have been used to calculate those addresses. PIRATE allows us to evaluate the effects of state explosion between CISC and RISC ISAs since we support x86, x86_64, and ARM. It also allows us to evaluate the rate of state explosion for different software implementations of the same application.

To perform this evaluation, we've experimentally measured the amount of tagged information throughout executions of four programs that make extensive use of pointer manipulations with our tagged pointer tracking turned on. These programs are bzip2, gzip, AES CBC encryption, and the Kernighan-Lin (KL) graph partitioning tool (obtained from the pointer-intensive benchmark suite [3]). The bzip2 and gzip programs make extensive use of pointers in their compression algorithms. Part of the AES encryption algorithm requires lookups into a static table, known as the S-box. For these three programs, tagged pointer tracking is required to accurately track information flow through the program, or else this tagged information is lost due to the indirect memory accesses that occur through pointer and array arithmetic and table lookups. In addition, the KL algorithm implementation utilizes data structures like arrays and linked lists extensively for graph representation.

The measurements of information spread for pointer-intensive workloads can be seen in Figure 8 for x86, x86_64, and ARM. Figures 8(a), 8(b), and 8(c) show similar results for each ISA in terms of the number of tagged bytes in memory,




(a) Bzip2 compression (b) Gzip compression

(c) AES CBC encryption (d) KL graph partition

**Fig. 8.** Tagged information spread throughout execution

but we can see offsets in instruction counts that highlights one of the differences of these CISC vs. RISC implementations. Figures 8(a) and 8(b) show the results of compressing the same file, which was approximately 300 bytes. These figures show the extent of information spread for these workloads; the peak number of tagged bytes reaches 14x the original file size for bzip2 on average across each ISA, and 6x the original file size for gzip on average across each ISA. For bzip2, the drastic growth in tagged data occurs as soon as the block sorting starts in the algorithm. For gzip, there is a more gradual increase in tagged data as soon as the compression function begins compressing the data. These patterns are indicative of the complex manipulations that are made on files as the tagged data flows through these compression algorithms. On the contrary, while many complex manipulations occur on files through AES encryption, Figure 8(c) shows that the amount of tagged data increases just over 2x for each ISA. Overall, these three pointer-intensive algorithms show similar patterns of state explosion for information flow tracking, regardless of the underlying ISAs that we've evaluated.

Figure 8(d) on the other hand shows major discrepancies in the amount of tagged information across the various ISAs. For this experiment, we processed a file of size 1260 bytes. For x86 and ARM, we can see an initial increase of tagged information followed by a gradual increase up to a maximum of 1.5x

and 4.1x the original tagged data, respectively. For x86_64, it is clear that a form of state explosion occurs causing the amount of tagged information to spread dramatically, reaching 11x the amount of original tagged data. Looking more closely at the x86_64 instance, we found that this initial explosion occurs inside of C library functions. One reason for this state explosion is that tagged data propagated to a global variable or base pointer, resulting in subsequent propagation with every access of that variable or base pointer. The fact that this explosion occurs inside of the C library implementation explains why we see the discrepancies across ISAs.

## 6 Discussion

Currently, our system provides the capability to perform dynamic information flow tracking for several of the major ISAs supported by the QEMU binary translator. It is straightforward to support more of these ISAs since we already have the ability to translate from the TCG IR to the LLVM IR. Additional work required for this involves properly tracking changes to CPU state (architecture-specific general purpose registers), and modeling those registers in the shadow memory. We plan to support more of the ISAs included in QEMU as future work.

We also plan to extend our decoupled execution and analysis approach to work with systems at runtime. This will enable us to perform dynamic detection and prevention of exploits, as well as other active defenses. Additionally, having a runtime system will allow us to perform a detailed performance evaluation, identify major sources of overhead, and optimize accordingly. Developing optimization passes over information flow operations is one way that we hope will help to improve performance.

One limitation of the QEMU user mode emulator is that there is limited support for multi-threaded programs. To deal with this, we plan to extend our system to support the QEMU whole-system emulator. With this enhancement, we will have the ability to perform detailed security analyses for entire operating systems, regardless of the ISA that they are compiled to run on. This will enable studies in the area of operating system security, including exploit detection and vulnerability discovery. Our architecture-independent approach will allow us to perform important analyses for embedded systems ISAs, where support for dynamic information flow tracking is limited.

## 7 Related Work

Dynamic information flow tracking has been shown to have a wide variety of real world applications, including the detection and prevention of exploits for binary programs [7, 20, 31] and web applications [27]. Applications to malware analysis include botnet protocol reverse engineering [6, 30], and identifying cryptographic primitives in malware [17]. For debugging and testing assistance, dynamic information flow tracking can be used to visualize where information flows in complex

systems [18], or to improve code coverage during testing [15]. Additionally, this technique can be used for automated software vulnerability discovery [12, 29].

Information flow tracking has been implemented in a variety of ways, including at the hardware level [26, 28], in software through the use of source-level instrumentation [12, 15, 31], binary instrumentation [10, 13, 22], or the use of a whole-system emulator [9, 20, 24]. Additionally, it can also be implemented at the Java VM layer [11]. Between all of these different implementations, the most practical approach for analyzing real-world software (malicious and benign) when source code is not available is to perform information flow tracking at the binary level. PIRATE is the first information flow tracking system that operates at the binary level, supports multiple ISAs, and can be extended in a straightforward manner to at least a dozen ISAs.

Existing systems that are the most similar to ours are Argos [20], BitBlaze [24], Dytan [10], and Libdft [13]. These systems have contributed to significant results in the area of dynamic information flow tracking. Argos and BitBlaze are implemented with QEMU [4], while Dytan and Libdft are implemented with Pin [16]. Even though QEMU and Pin support multiple guest ISAs, each of these information flow tracking systems are tightly coupled with x86, limiting their applicability to other ISAs.

Intermediate representations have been shown to be useful not only in compilers, but also in software analyses. Valgrind [19] employs an intermediate representation, but it is also limited to user-level programs which would prevent us from extending our work to entire operating systems. Valgrind also employs a shadow memory, but no tools exist that perform information flow tracking with the detail that we do in an architecture-neutral way. BAP [5] defines an intermediate representation for software analysis, but that system currently can only analyze x86 and ARM programs, and it doesn't have x86_64 support. CodeSurfer/x86 [21] shows how x86 binaries can be statically lifted to an intermediate representation enabling various static analyses on x86 binaries.

## 8 Conclusion

In this paper, we have presented PIRATE, an architecture-independent information flow tracking framework that enables dynamic information flow tracking at the binary level for several different ISAs. In addition, our combined static and dynamic analysis of helper function C code enables us to track information that flows through these complex instructions for each ISA. PIRATE enables us to decouple all of the useful applications of dynamic information flow tracking from specific ISAs without requiring source code of the programs we are interested in analyzing. To demonstrate the utility of our system, we have applied it in three security-related contexts; enforcing information flow policies, characterizing algorithms, and diagnosing sources of state explosion. Using PIRATE, we can continue to build on the usefulness of dynamic information flow tracking by bringing these security applications to a multitude of ISAs without requiring

extensive domain knowledge of each ISA, and without the extensive implementation time required to support each ISA.

## References


1. Clang: A C language family frontend for LLVM. http://clang.llvm.org
2. OpenSSL cryptography and SSL/TLS toolkit. http://openssl.org
3. Austin, T., Breach, S.E., Sohi, G.S.: Efficient detection of all pointer and array access errors. Tech. rep., University of Wisconsin-Madison (1993)
4. Bellard, F.: QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference (2005)
5. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: A binary analysis platform. In: Proceedings of the 23rd International Conference on Computer Aided Verification (2011)
6. Caballero, J., Poosankam, P., Kreibich, C., Song, D.: Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In: Proceedings of the 16th ACM Conference on Computer and Communications Security (2009)
7. Chen, S., Xu, J., Nakka, N., Kalbarczyk, Z., Iyer, R.K.: Defeating memory corruption attacks via pointer taintedness detection. In: International Conference on Dependable Systems and Networks (2005)
8. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: A platform for in-vivo multi-path analysis of software systems. In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (2011)
9. Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., Rosenblum, M.: Understanding data lifetime via whole system simulation. In: Proceedings of the 13th USENIX Security Symposium (2004)
10. Clause, J., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis (2007)
11. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (2010)
12. Ganesh, V., Leek, T., Rinard, M.: Taint-based directed whitebox fuzzing. In: IEEE 31st International Conference on Software Engineering (2009)
13. Kemerlis, V.P., Portokalidis, G., Jee, K., Keromytis, A.D.: libdft: Practical dynamic data flow tracking for commodity systems. In: Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (2012)
14. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (2004)
15. Leek, T., Baker, G., Brown, R., Zhivich, M., Lippman, R.: Coverage maximization using dynamic taint tracing. Tech. rep., MIT Lincoln Laboratory (2007)
16. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (2005)

17. Lutz, N.: Towards Revealing Attackers' Intent by Automatically Decrypting Network Traffic. Master's thesis, ETH Zurich (2008)
18. Mysore, S., Mazloom, B., Agrawal, B., Sherwood, T.: Understanding and visualizing full systems with data flow tomography. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (2008)
19. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (2007)
20. Portokalidis, G., Slowinska, A., Bos, H.: Argos: an emulator for fingerprinting zero-day attacks. In: Proceedings of the 1st ACM SIGOPS European Conference on Computer Systems (2006)
21. Reps, T., Balakrishnan, G., Lim, J.: Intermediate-representation recovery from low-level code. In: Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (2006)
22. Saxena, P., Sekar, R., Puranik, V.: Efficient fine-grained binary instrumentation with applications to taint-tracking. In: Proceedings of the 6th IEEE/ACM International Symposium on Code Generation and Optimization (2008)
23. Slowinska, A., Bos, H.: Pointless tainting? evaluating the practicality of pointer tainting. In: Proceedings of the 4th ACM European Conference on Computer Systems (2009)
24. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A new approach to computer security via binary analysis. In: Proceedings of the 4th International Conference on Information Systems Security (2008)
25. Stallings, W., Brown, L.: Computer Security: Principles and Practice. Pearson Prentice Hall (2008)
26. Suh, G.E., Lee, J.W., Zhang, D., Devadas, S.: Secure program execution via dynamic information flow tracking. In: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (2004)
27. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: TAJ: Effective taint analysis of web applications. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (2009)
28. Venkataramani, G., Doudalis, I., Solihin, Y., Prvulovic, M.: FlexiTaint: A programmable accelerator for dynamic taint propagation. In: Proceedings of the 14th International Symposium on High Performance Computer Architecture (2008)
29. Wang, T., Wei, T., Gu, G., Zou, W.: TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: Proceedings of the 31st IEEE Symposium on Security & Privacy (2010)
30. Wang, Z., Jiang, X., Cui, W., Wang, X., Grace, M.: ReFormat: Automatic reverse engineering of encrypted messages. In: Proceedings of the 14th European Conference on Research in Computer Security (2009)
31. Xu, W., Bhatkar, S., Sekar, R.: Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In: Proceedings of the 15th USENIX Security Symposium (2006)