A Hardware-based Technique for Efficient Implicit Information Flow Tracking

Jangseop Shin[1], Hongce Zhang[2], Jinyong Lee[1], Ingoo Heo[1], Yu-Yuan Chen[2], Ruby Lee[2], and Yunheung Paek[1]
[1]Department of Electrical and Computer Engineering, Seoul National University
{jsshin,jylee,igheo,ypaek}@sor.snu.ac.kr
[2]Department of Electrical Engineering, Princeton University
hongcez@princeton.edu, yctwo.princeton@gmail.com, rblee@princeton.edu

ABSTRACT

To access sensitive information, some recent advanced attacks have been successful in exploiting implicit flows in a program in which sensitive data affects the control path and in turn affects other data. To track the sensitive data through implicit flows, several software and hardware based approaches have been proposed, but they suffer from the non-negligible performance overhead. In this paper, we propose a hardware tracking engine for implicit flow, called the implicit flow tracking unit (IFTU). By adopting the tracking scheme for implicit flow and mapping it to the specialized hardware, our solution can efficiently perform the implicit flow tracking with reasonable area costs.

1. INTRODUCTION

In recent years, computer security has been severely threatened by various malicious attacks that intend to leak sensitive information [10]. The general goal of these attacks is to transfer the critical data from sensitive sources (e.g., SIM card, password list) to output channels like network connections so that the attackers can acquire the sensitive information in the system. To achieve the goal, the malicious program or the victim program being exploited by attackers first accesses the critical data and then copies them from the source to the destination at each instruction execution. When they are finally delivered to the output channel, the attacker can leak the sensitive information out of the system.

One of the most widely used solutions against this type of attacks is *Dynamic information flow tracking* (DIFT) [10]. Generally, DIFT sets up rules to taint internal data of interest and keeps track of their taintness throughout the system. At runtime, whenever an instruction is executed, the taintness of sources is propagated to the destinations, to track the information flow associated with the data transfers (data copying and transformations). An alarm will be triggered as soon as any of the tainted data is involved in potentially illegal activities, such as being included in a data stream on the output channels. In several previous studies [3, 4, 10, 11, 19], it was demonstrated that DIFT is an effective way to detect the attacks which attempt to leak the sensitive information with explicit data transfers.

However, there have been some advanced attacks that can bypass the explicit DIFT approaches by acquiring certain sensitive information only through the execution control flow analysis of a victim program without data transfers. In practice, when a data value affects a conditional branch result, execution flow is altered and it affects other data. Then the affecting value can often be inferred merely by examining the values of the affected ones. In this case, we can see that although there is no *explicit* data copy or transfer, the affecting data value is in effect transferred to other data values via the *implicit flow* along the execution control path. In the previous studies [6, 7], they presented empirical evidence that the attackers can leak the sensitive information by exploiting the implicit flow. Thus, in order to deal with such advanced attacks, a DIFT solution should track the taintness of the sensitive data tags not only through the explicit information flow associated with data copy and transfer operations, but also through the implicit flow associated with conditional branch operations.

For the tracking of implicit flow, several software solutions have been proposed [6, 18]. In these works, they analyze the program code and find the control flow that might be related to the implicit information flow at runtime. Then, they augment the original application with the additional code to keep track of the implicit flow as well as the explicit flow. In spite of their effectiveness, the main drawback of these solutions is that they incur too much runtime overhead, since it takes up to 20 instructions to emulate a single tag propagation operation per instruction.

To reduce the performance overhead for implicit flow tracking, RIFLE [16] resorts to a hardware technique. Although they have shown an impressive improvement on the overall DIFT computation, their experiment also reveals that they still suffer from the non-negligible performance overhead for implicit flow tracking. This is primarily because their hardware has been designed originally for the information tracking with explicit data transfers. Therefore, to utilize their hardware for implicit flow tracking, they had to convert the implicit flow problem to the equivalent explicit one. For this reason, they instrumented their binary code to transform all implicit information flow operations across conditional branches into explicit data copy operations. According to their experiments, the performance degrades by a factor of two in the worst case, mainly due to the instrumented instructions.

Motivated by previous work, we have developed a dedicated hardware unit to efficiently tackle the implicit flow tracking problem. In this paper, we introduce our hardware engine for implicit flow tracking, called the *implicit flow tracking unit* (IFTU), and the implicit flow tracking scheme designed to work on IFTU. We have built IFTU as an external hardware module attached to the host processor via the system interconnect. To evaluate its effectiveness, we have implemented our solution on an FPGA board. In our experiments, we show that our proposed approach with IFTU successfully tracks the implicit information flow on the system with negligible performance overhead, while the additional logic required for the implicit flow tracking is also small.

2. RELATED WORK

There has been much prior work that focuses on explicit information flow tracking [3, 4, 8, 10, 11, 13, 19]. Software approaches in [10, 11] suggest the use of a binary instrumentation technique, which mainly inserts additional instructions to the target code to keep track of the tainted data at runtime. During the program execution, the taintness of data is propagated according to the data dependency, and any misuse of data (e.g. information leak) is detected by their proposed solutions. Other works introduced

ICCAD '16 November 07-10, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2966986.2966991.

in [3,4,13] suggest the use of specialized hardware logic for DIFT mainly to reduce the performance overhead caused by the DIFT computation. In [3, 13], for instance, they augmented the host processor internals directly, including register files and caches. In [4, 8], they proposed a decoupled DIFT hardware that can be attached to the outside the host. These previous approaches, implemented either in software or hardware, show their effectiveness in the tracking of explicit flows. However, a critical limitation they have is that they do not consider the implicit flows, which can result in the *under-tainting problem* where the values that should be tainted are not tainted [6].

To resolve the under-tainting problem, several software solutions for implicit flow tracking have been proposed. In [5], the authors use dynamic analysis to keep track of the flow of sensitive information processed by the web browser application. To handle the implicit flows, their *taint engine* examines all conditional branch instructions that are encountered during execution. If such an instruction has at least one tainted operand, the taint engine identifies all instructions whose execution is conditionally dependent on the direction of the branch and then it taints the results of those instructions. In [6], another software solution, called DTA++, is proposed. To achieve efficiency in the tracking, instead of examining all conditional branches, DTA++ focuses only on the implicit flows within certain code patterns (i.e., the information-preserving transformations), based on the observation that under-tainting usually occurs at just a few locations. With the proposed tracking strategy, DTA++ can achieve effectiveness and efficiency in implicit flow tracking. Nevertheless, the main drawback of these software approaches is that they still suffer from performance degradation mainly due to the additional code instrumented with the binary translation. For example, although DTA++ only applies the tracking technique to certain cases, the performance overhead is around 1.5X even with the parallel execution of the binary translation.

For this reason, several hardware techniques [15,16] have been proposed to enhance the tracking performance. RIFLE [16] is a hybrid approach that uses compiler-assisted binary rewriting to change the program to turn implicit information flows due to condition flags into explicit tag assignments. However, as discussed, the main problem of RIFLE is that it relies on the hardware architecture designed for the explicit flow tracking and thus requires code transformation to convert implicit flows to explicit ones. Since too many additional instructions are added to the original program binary to utilize the hardware, the efficiency is severely degraded and the performance overhead reaches up to 1.5X in the worst case (when the data cache of the system is duplicated to store the tags). On the other hand, in our approach, we propose a hardware engine specialized for implicit flow tracking and thus can overcome the limitation of RIFLE. GLIFT [15] and Leases [14] are interesting hardware solutions that track information flow at the gate level to build a system with strong noninterference properties which can be used to eliminate all forms of information leak, including those from timing and storage channels. While this is a potentially promising approach, all the hardware has to be re-designed from the gates up, requiring unproven new hardware design methodologies and tools. On the other hand, our IFTU can be connected to the commodity processor with an external interface, not requiring the redesign of the off-the-shelf processor architecture, since it is designed as an external module.

3. OUR APPROACH FOR IMPLICIT FLOW TRACKING

We now discuss our approach for efficient implicit flow tracking, inspired by [2] and improved. After briefly explaining the tracking scheme implemented in our work, we will describe our code analysis and transformation technique whose purpose is to enable the scheme to work correctly in real programs.

3.1 Implicit Flow Tracking Scheme with Program Counter Tag

The code snippet in Figure 1 shows a simple example of implicit information flow in a program. In this example, the value of `x` can be changed to either `0` or `1` according to the branch result that is affected by the signedness of variable `s`. Clearly, there is a flow of information between the two variables since the value of `s` affects the value of `x`; however, it is not the result of direct data transfers, but rather the result of the branch outcome affected by setting the condition flag through the comparison.

```
x := 2
if s <= 0 then x := 0 else x := 1
```

Figure 1: An example code with implicit flow

To handle these implicit flows correctly, language-based static techniques [12] use a tracking scheme that introduces the program counter tag (denoted as t_{PC}), which indicates whether the control flow path is affected by tainted data or not. In this scheme, for every conditional branch, the taintness of data that is used for the condition checking is propagated to t_{PC}. Then, for the instructions after the branch, the value of t_{PC} is propagated to the tags of their destinations to indicate that the values are affected by the branch result. Now, assume that the example described in Figure 1 is tracked with this scheme. In this example, the variable `s` is used for the condition checking of the branch. Thus, if the variable `s` contains the sensitive information and its tag is tainted, t_{PC} is also set to `1`, to indicate that the branch result is affected by the sensitive information. Then, by propagating the value of t_{PC} to the tag for variable `x` when it is set, the implicit flow along the branch can be tracked.

In our approach, to handle the implicit flow as well as the explicit ones, we combine the tracking scheme introduced above with the conventional DIFT technique that tracks the data flow [10]. To denote tagging, every location for storing data such as registers and memory is augmented with a tag bit. Then, the tags are propagated during the program execution, based on a set of tag propagation rules that are specified for each basic operation type such as arithmetic, logical, or conditional branch.

Figure 2 shows an example code at the assembly level and the associated tag propagation operations. Basically, the tag propagation rules applied in our approach are based on the data dependency, as in the previous works [10, 11]. For example, when the `ldr` instruction at line 1 is executed, the tags of sources (`%i0` register and the memory location pointed by the register value) are propagated to the tag for register `%g2`. In addition to the basic rules, we add new rules to track the implicit flow along the control path. In principle, a conditional branch has its condition code, such as `equal`, `not equal` or `less than`. When the branch is executed, the processor checks the condition code register (CCR) which generally consists of several condition bits (e.g., N,Z,V,C in SPARC machines), and determines the control path based on the value of CCR. That is, the result of a conditional branch is affected by the value of CCR. In practice, CCR is configured by an arithmetic instruction like `sub`, or a specialized comparison instruction like `cmp`, as in the code at line 3. For this reason, in our solution, when CCR is set by these instructions, the tags of their sources are propagated to t_{CCR}, which is the tag for CCR (see the right column of line 3). Then, when a conditional branch is executed later, the value of t_{CCR} is propagated to t_{PC} (see line 4). (If an unconditional branch is executed, such tag propagation is not performed since the branch is not affected by CCR (see line 6).) Thus, t_{PC} can indicate whether the control path is affected by tainted data or not. Since the value of t_{PC} is propagated to the destination tags (marked in boldface at the right column) at each ordinary instruction execution, we can track the implicit flow along the control dependency.

In spite of its effectiveness, there is a challenge in correctly

	Original Code	Tag Propagation
1	ldr [%i0], %g2	tag[%g2] = tag[%i0] or tag[mem[%i0]] **or t_{pc}**
2	sub %g2, %g3, %g1	tag[%g1] = tag[%g2] or tag[%g3] **or t_{pc}**
3	cmp %g1, #0	tag[%ccr] = tag[%g1] **or t_{pc}**
4	be L1 // **branch equal**	tag[%pc] = **t_{ccr} or t_{pc}**
5	mov #2, %g2	tag[%g2] = **t_{pc}**
6	b L2 // **unconditional branch**	none
7	L1: mov '1', %g2	tag[%g2] = **t_{pc}**
8	L2: add %g5, %g2, %g3	tag[%g3] = tag[%g5] or tag[%g2] **or t_{pc}**

Figure 2: Example of tag propagation rules

tracking implicit flow with the propagation rules introduced above. In principle, the taintness of t_{PC} set for a conditional branch should be propagated only to the instructions whose execution is conditionally dependent on the result of the branch according to the definition of implicit flow. Otherwise, the taintness of t_{PC} would be propagated to the tag for the data that is not affected by the branch or the tag that should be tainted would not be tainted. Thus, it is necessary to analyze the code in order to determine the exact scope of every conditional branch, which is a set of instructions that are affected by the branch result. In section 3.2, we will discuss a code analysis technique to identify the scopes of conditional branches and the management scheme for correctly clearing t_{PC} based on the analysis.

```
x := 2
if s <= 0 then x := 0
```

Figure 3: An example code with implicit flow through the untaken path

Also, from the tag propagation rules in Figure 2, once t_{PC} is tainted, all the instructions executed after that will be affected, because their execution is decided by a tagged condition. However, information flow can also exist between the condition of a branch and the instructions that are not executed. For example, in Figure 3, if the condition for the `if` statement is true,then x will be tainted according to the propagation rules. However, if the condition is false, x will not be tainted even though the value of x can leak the information about the branch condition. This example shows that only propagating tags according to the executed instructions is not enough, and there is the necessity for tag compensation of the untaken path. In section 3.3, we will describe our tag compensation scheme.

3.2 t_{PC} Management Technique

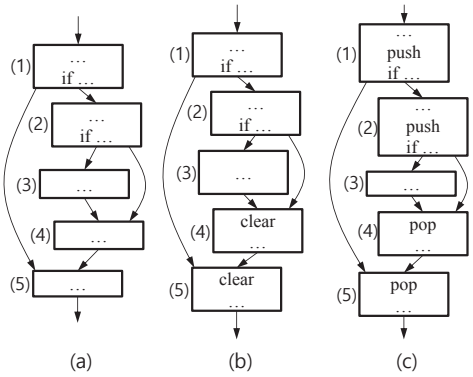

Figure 4: t_{PC} setting and clearing example

In principle, the result of a conditional branch determines the control path which in turn determines the instructions executed by the processor. For example, in the control flow graph (CFG) shown in Figure 4(a), the execution of block (2) is determined depending on the result of the conditional branch in block (1). However, at a certain point in a program, the control path is no longer affected by the conditional branch. In general, the influence of a conditional branch ends at the *immediate post-dominator* of the branch. In our example, block (5) is the immediate post-dominator of block (1) because all paths from block (1) to the exit must pass block (5). Thus, the value of t_{PC} set at a conditional branch should be cleared upon the entrance of the immediate post-dominator (5) of the branch in (1) (Figure 4(b)).

However, this scheme does not work if the code has multiply-nested branches. For instance, we have a doubly-nested branch in block (2). According to the above scheme, t_{PC} set in block (1) would be cleared in block (4) although it should have remained set until block (5). In order to remedy this, we maintain a new t_{PC} *stack* that is used to save and restore the value of t_{PC} at each nested branch level. Basically, we save the current t_{PC} value by pushing it onto the stack just before a conditional branch, and later when we need to clear t_{PC}, we simply overwrite the current value in t_{PC} with the value popped from the stack (Figure 4(c)).

Algorithm 1: Algorithm for inserting push/pop and compensation code

Input : Control flow graph of a function
Output: Control flow graph with push/pop operations and compensation tag set operations inserted for correct implicit flow tracking

```
1  foreach loop l do
2      Insert push at the end of the preheader of l;
3      Insert pop at the start of the (common) immediate
       post-dominator of exiting block(s) of l;
4  end
5  foreach conditional branch block t do
6      Find t's immediate post-dominator block p;
7      if t is inside a loop and p is outside then continue;
8      if t is inside a loop and p does not dominate blocks with
       loop back edge then continue;
9      Insert push before conditional branch in t;
10     Insert pop at the beginning of p;
11     R = set of basic blocks that is reachable from t before
       reaching p;
12     foreach block b in R ∪ {p} do
13         foreach predecessor block pred of b do
14             if pred is not in R ∪ {t, p} then
15                 Insert push between pred and b;
16             end
17         end
18     end
19     foreach live-in register r of p do
20         D = set of basic blocks in R that defines r;
21         R_D = set of basic blocks in R that are reachable
           from basic blocks in D before reaching p ∪ {p};
22         G = set of basic blocks in R that are guaranteed to
           pass at least one basic block in D before reaching p;
23         foreach edge e connecting a block in R - R_D - G
           with a block in R_D - G do
24             Insert tag compensation for r;
25         end
26     end
27 end
```

In Algorithm 1, we illustrate our algorithm that finds and marks the places in the code where such push/pop operations should be performed. Basically, as explained above, a push operation is inserted before a conditional branch and a pop operation is inserted at the immediate post-dominator of a conditional branch (see lines 9-10). However, real application codes have some exceptional cases that need a more complex algorithm such as ours.

In Figure 5(a), we can see one of these exceptional cases that should be handled in our algorithm. In the figure, push/pop oper-

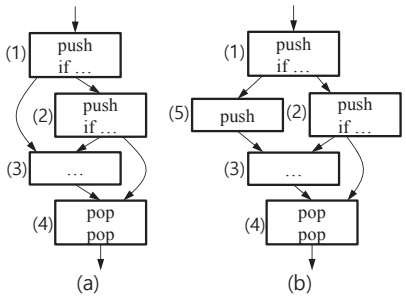

Figure 5: Solving push/pop imbalance

ations are marked according to the naive algorithm. Now suppose that the control flow takes the path (1)-(3)-(4) at runtime. If so, a push operation will be performed at block (1) and two pop operations will be processed at block (4). This obviously causes an error in the stack management because more entries are popped from the stack than are pushed onto the stack. In general, this problem arises when there is a path that reaches a pop operation inserted for some other conditional branch without passing that conditional branch.

To avoid this problem, for a conditional branch block *t* and its immediate post-dominator *p*, we first define a set *R* consisting of the basic blocks that are reachable from *t* before reaching *p*. Then, among the paths that reach *p*, let *P* be a path (p_1)-...-(p_{r-1})-(p_r)-...-(*p*) that does not pass *t*, where p_r is the first basic block in the path that is in R. Then, we insert a new basic block that contains a dummy push operation, between p_{r-1} and p_r. (For the example in Figure 5(a), such a new block is inserted between blocks (1) and (3) as shown in Figure 5(b).) In this way, we can make sure that the number of push and pop operations are equal along any path. This process corresponds to lines 12-18 of Algorithm 1.

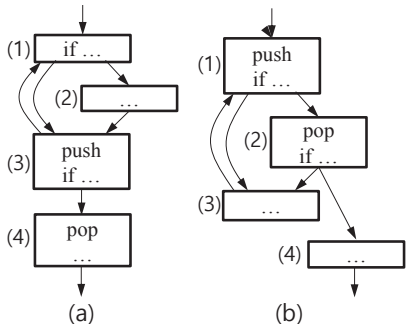

Figure 6: Incorrect push/pop insertion for a loop

When the host code includes a loop (e.g., (1), (2), and (3) in Fig 6), we must handle a few other exceptional cases. If the immediate post-dominator of a conditional branch block is out of the loop, the push operation marked before the conditional branch is repeatedly executed while the corresponding pop operation is not performed during the loop iteration. In Figure 6(a), the immediate post dominator of the conditional branch block (3) is out of the loop, so the push operation may be performed many times while the pop operation will only be executed once in (4). Also, even if the immediate post dominator is in the loop, there can be push/pop imbalances if the block(s) with the backward edge of the loop is not dominated by the immediate post dominator. For example, in Figure 6(b), the immediate post dominator of the conditional branch block (1) is block (2). However, there is a path from block (1) that leads to the block with the loop back-edge (block (3)) without crossing block (2). Therefore, the push operation may be performed more than the corresponding pop operation. To handle these exceptional cases, we insert a push operation in the preheader for the loop so that the push is performed only once upon the entrance of the loop, and insert a pop operation in the (common) immediate post dominator of the loop-exiting block(s) of the loop. We rely on this push-pop pair to handle all the conditional branch blocks that correspond to the above exceptional case. Note that push operations do not affect the t_{PC} value so moving the push operation in front of the loop does not change the correctness of the result. This process is described at lines 1-4 and 7-8 in Algorithm 1.

3.3 Compensation for the Untaken Path

To compensate for the untaken path, we analyze the code to find out which register tag needs to be set in which location. There can also be implicit flow through the memory location, but our implementation does not compensate for the memory locations since memory addresses could be determined in the runtime which means that we will have to actually execute the path to determine which memory tag to apply compensation. This could possibly introduce false negatives, but the chances are relatively low since it will also be hard for the attacker to reason about the implicit flow through the memory locations.

Among the registers, we only need to consider the ones that are live-in to the immediate post dominator of the conditional branch block. Registers that are not live at the entry point of the post dominator are not used after the immediate post dominator and cannot be used for the propagation of data. The register tag for the live-in register is set at runtime if there is at least one instruction defining the register on the execution path between the conditional branch and its immediate post dominator. Thus, an implicit flow through an untaken branch will occur if there is an instruction defining the register through some execution path but not all execution paths.

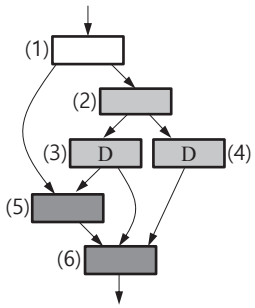

Figure 7: Example CFG for tag compensation

Based on this idea, we find the minimum number of program points where the tag compensation is needed for each register. We first define three sets of basic blocks as described in lines 20-22 in Algorithm 1. In Figure 7, we show these sets of basic blocks where block (1) is the conditional branch block currently concerned with. Blocks which define the register are marked with `D`. Lightly shaded blocks are the blocks which pass at least one basic block which defines the register. Darker blocks are the ones that are reachable from blocks defining the register. After determining these sets of blocks, we start from the conditional branch block (block (1)) and traverse the CFG in depth-first manner. If we encounter a lightly shaded block, we do not need to set the register tag since there will be at least one register definition along that path. If we encounter a dark shaded block, we put the register tag set operation on the edge between the current block (block (1)) and the dark block (block (5)). In this way, we can make sure that the register tag is set for all execution path between the conditional branch block and its immediate post dominator. The entire process corresponds to lines 19-26 in Algorithm 1.

We implemented the code analysis and transformation tech-

nique described in Algorithm 1 on the LLVM compiler framework. Our transformation tool inserts the pop and push operations in the host code, which are implemented as special instructions whose encodings are not used by the ISA of the host processor architecture. Thus, at runtime, the host processor regards such marked operations as `nop` operation. Our IFTU processes these operations to manage the t_{PC} stack. For the register tag compensation, we used a dummy `add` instruction that adds 0 to the target register and sets the register to that value. It does not change the semantic meaning of the original program, but it can set the register tag if the PC tag is set at that time.

4. ARCHITECTURE DESIGN OF IFTU

In this section, we will discuss the hardware architecture of our solution. After introducing an architectural overview of our solution, we will discuss the detailed structure of IFTU.

4.1 Overall System

Figure 8 shows the overall system design for our solution, which mainly consists of the host processor and IFTU. In our implementation, as introduced in Section 1, we design our IFTU as an external hardware module and integrate it with the host processor through the system bus, instead of embedding the dedicated hardware logic internally in the host processor [2]. The main advantage achievable from this design strategy is that our proposed solution can be easily adopted by existing commercial platforms such as application processor (AP) SoC platforms for smartphones. Generally in these platforms, the host processors are typically the commodity processors that are quite difficult to modify the internals without tremendous cost and labor from the vendors. Thus, our solution would be adoptable in these platforms as it does not require such modification.

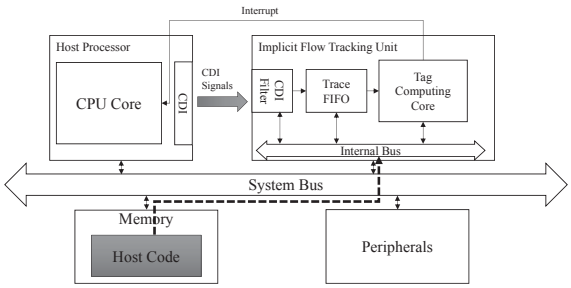

Figure 8: Overall system design

However, such design strategy raises a challenge. As discussed in Section 3, in order to track the information flow of a program, the taintness of tags should be propagated during the program execution according to the propagation rules. Since the rules are dependent on the instruction type and operands, it is necessary for IFTU to know about the instructions executed by the host. For this reason, our IFTU reads the host program code in main memory and extracts the propagation rules as shown in Figure 8. Nevertheless, the problem is that, from only the host code, IFTU cannot obtain some essential information required for the correct flow tracking, which is only resolved during code execution. In particular, such information includes (1) an execution path of the original program and (2) memory addresses of load/store instructions. Without this information, our IFTU cannot perform the tag operations correctly, while following the execution of the host program.

To resolve the problem, in our solution, we utilize the core debug interface (CDI) in the host processor, as was done in the hardware-based solution introduced in [8]. CDI is an interface placed in recent commodity processors, whose main role is to provide the external debug modules with the processor's internal status information required for debug/trace, without affecting the performance of the host. Based on the specification of CDI in commercial processors and the prior works that utilize CDI [1, 8, 9, 17], in our prototype, we assume that CDI provides a set of signals as follows: instruction address, current context ID (or process ID), data address/value of memory access instructions, branch type/source address/target address, exception and privilege mode information. Since the set of signals includes the necessary runtime information for flow tracking, our IFTU can follow the execution of the host and perform the tag operations correctly.

As shown in Figure 8, IFTU consists of three components: the *CDI filter*, the *trace FIFO* and the *tag computing core* (TCC). Although the CDI in the host processor provides plenty of signals, our IFTU needs only a subset of those signals. The role of the CDI filter is to filter out unnecessary signals and leave the ones that are necessary for the tracking: the current process ID (PID), the address of memory data accessed by a load/store and the target address of a branch. (The current PID is necessary to recognize the active process running on the host. If the monitored program goes into sleep mode, the main controller informs the trace FIFO to ignore the traces from the CDI filter.) IFTU consumes the traces containing such information to obtain necessary information and store them in the trace FIFO in order at runtime.

4.2 Tag Computing Core

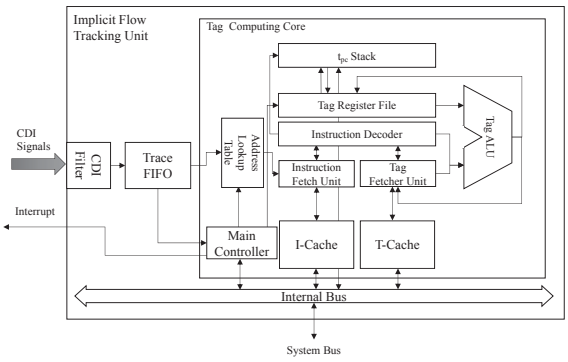

Figure 9: Tag computing core architecture design

Figure 9 shows the microarchitecture of TCC, whose main role is to manage all tags and perform the tag operations. The overall operation of IFTU is controlled by the *main controller* in TCC. It contains several configuration registers and the host processor can control the functions of IFTU by setting the registers, such as the tag initialization that marks the location of tainted data. To track the flow of information, we augment the tags to the processor registers and memory locations, as in other previous approaches [10, 13]. The tags for registers are stored to a special register file in TCC called the *tag register file* (TRF). Each entry of TRF represents the 1-bit tag for the corresponding processor register. We also add two register tags t_{PC} and t_{CCR} to the basic structure of TRF, which are used only for the implicit flow tracking. For the memory tags, we reserve a special region called the *tag space* in the main memory. Each bit of the tag space represents the tag for a memory word (32-bit). The *T-cache* is employed in our TCC design to reduce the access latency of tag fetching.

The branch target addresses transferred from CDI are consumed by TCC in order to follow the execution path of the host program. However, since the addresses stored in the trace FIFO are virtual addresses, they cannot be used directly for fetching the host code from main memory. To resolve this problem, TCC includes the *address lookup table* (ALT) where an entry of ALT is comprised of the process ID and the virtual-to-physical address

mapping information [8]. At runtime, the host OS kernel updates the entries whenever a code page is allocated on the host, and by using the information the *instruction fetcher unit* reads the host code from the memory with the translated physical addresses. To reduce the access latency required for the instruction fetch, we employ the *I-cache* in TCC as done in previous work [8].

After the host code is fetched, it is delivered to the *instruction decoder* which extracts the propagation rules from the opcode and operands of the instruction. TCC accesses TRF to fetch the tag values if the rule requires register tags. If the operand is the memory address for a load/store, TCC firstly accesses the trace FIFO to acquire the exact address. (Since all load/store instructions generate the CDI signals for the access addresses and the trace containing such information is stored in the trace FIFO in order, it is guaranteed that TCC can obtain the address for the memory instruction.) Then, TCC loads the memory tag corresponding to the address from the T-cache. If a miss occurs, the *tag fetcher unit* accesses the tag space to handle the miss. Finally, once all the tags are prepared, the *tag ALU* performs the tag propagation with the tags and the resulting values are written to TRF or the T-cache.

To support the management scheme for t_{PC} introduced in Section 3, TCC includes the hardware for the t_{PC} stack as shown in Figure 9. As discussed, in the instrumented host code, the push/pop operations for the t_{PC} stack are included. When the instruction decoder encounters such operations, TCC takes the corresponding actions: for push operations, TCC reads the value of t_{PC} from TRF and pushes it to the stack. For pop operations, the top entry of the stack is popped and overwrites the t_{PC}. As our current hardware stack implementation has 32 entries, the stack will overflow if the nested level of branches exceeds 32. To cope with this case, we reserve a memory region for the entries to be stored to if the stack is full. Then the tag value for PC is saved or restored from the memory region.

5. PERFORMANCE AND AREA ANALYSIS

To evaluate our approach, we have built a full-system prototype on an FPGA board. In this prototype, we used SPARC V8 processor as the host processor which has separate 4KB instruction/data caches. The AMBA2 AHB compliant bus is used to interconnect the host processor with our IFTU and Linux 2.6 is used as the host OS kernel. IFTU is implemented as described in Section 4 and it includes 4KB I-cache/512B T-cache. Based on the parameters, we synthesized our prototype on to a Xilinx Virtex5 FPGA board. Table 1 presents the design statistics of our implemented hardware. Our experiment shows that our IFTU incurs a hardware resource overhead of 28.18% for LUTs, and the memory requirement is increased by about 4.5 KB (mainly due to the caches) when compared to the baseline system that includes the host core. It is noteworthy that the amount of logic required to perform the implicit flow tracking is very small. In our approach, the two tag registers (i.e., t_{PC}, t_{CCR}) and the t_{PC} stack are the components installed for the implicit flow tracking. In our experiment, the hardware resources for these components are estimated to be about only 5.7% of the overall IFTU. This clearly shows that our approach can be implemented with a small amount of additional logic on top of the DIFT hardware for the explicit flow tracking.

To measure the performance overhead of our approach, we chose eight applications from the *mibench* benchmark suite and executed them on three systems each with different configurations. The first configuration is *Native*, which stands for a system that performs the original application without information flow tracking. In the *Explicit* configuration, our IFTU performs only the explicit flow tracking and therefore the host code is not instrumented. Finally, in *Explicit+Implicit*, IFTU performs the tracking scheme proposed in this paper.

Category	Component	LUTs
Baseline System	Host Processor Core	4876
	Bus components and Memory Controller	844
	Peripherals (TIMER, UART, Interrupt Controller and etc.)	963
	Total Baseline System	**6683**
IFTU	Components for CDI (CDI Filter, Trace FIFO, Address Lookup Table)	826
	Main Controller and Bus Interface	330
	Instruction Cache	293
	Tag Cache	180
	Instruction/Tag Fetcher Unit	97
	Instruction Decoder	35
	Tag ALU	109
	t_{pc} Stack	13
	Total IFTU	**1883**
	% IFTU over Baseline System	**28.18%**

Table 1: Synthesis Result

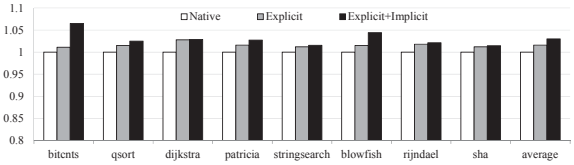

Figure 10: Performance Comparison

Figure 10 shows the execution times for the three configurations normalized to that of *Native*. The results show that the *Explicit* incurs about 1.6% performance overhead although the host code is not instrumented in this configuration. The performance loss is mainly due to the resource competition between the host processor and our IFTU. Since both modules are connected to the same system bus and share the same main memory, the bus transactions of IFTU for accessing the main memory slightly degrades the host performance. *Explicit+Implicit*, which stands for our proposed approach, shows an average performance overhead of about 3%. This shows that the overhead caused by our code instrumentation is negligible. Overall, the performance of our approach is much greater than that of the previous hardware approach, RIFLE [16].

We also measured the code size increase due to the code instrumentation. For the given benchmarks, the code size is increased by 0.3% on average. This shows that the code size overhead of our approach is also negligible.

6. SECURITY ANALYSIS

To evaluate the accuracy of our taint propagation methods, we used a program that has explicit and implicit information flow. We chose a π computing program that calculates the digits of π and uses the `sprintf` library function to put the ASCII representation of π in the specified buffer. While not strictly a security-related program, we found it adequate for evaluating our implicit flow tracking methods. The program is shown in Listing 1. The program calculates the 1002 digits of π, refining the value at each iteration. The final value will be transformed into the ASCII form by the `sprintf` call. To transform the `sprintf` library function to track implicit flow, we have copied the corresponding functions from *dietlibc*. We have slightly modified the core of the `sprintf` function so that it involves implicit flow when translating decimal digits to the ASCII form.

```
long a[337],p,q,k=4000,t=1000;
char buffer[5000];
int j,n=0;
for(;a[j=q=0]+=2,--k;)
  for(p=1+2*k; j<337;
      q=a[j]*k+q%p*t,a[j++]=q/p)
    k!=j>2?:(n+=sprintf(&buffer[n],
                "%.3d",a[j-2]%t+q/p/t));
```

Listing 1: π computing program

In the program, if the memory location for the array `a` is tagged at the start, the correct explicit and implicit information flow tracking scheme should tag the part of the `buffer` array where the ASCII characters are written. We ran our information tracking hardware after tagging array `a`, and examined the tagged memory locations after the program is finished. A total of 593 words were tagged, of which 337 were for array `a` and 250 for `buffer`. 6 other locations were additionally tagged. We have analyzed the execution trace to find out why those 6 locations were tagged and why there is one missing tag for the `buffer` array. Since there are 1002 digits of π, 251 words should have been tagged in the `buffer` array. We found out that all 6 additionally tagged locations are for temporary data in the stack frame that is destroyed when the `sprintf` function is returned. Those temporary data contained the π digit, its ASCII representation, or the length of the character written for the π digit. Thus, we do not need to regard them as false positives. For the `buffer` array, we found that the tag corresponding to the last word of the character string has been reset at the end because the `sprintf` function has put a $null$ value at the end of the character string. Since there is a 1-bit tag for each 4-byte word, the tag corresponding to the last word was reset even though there were two tainted bytes.

The analysis of the results shows that our implicit information flow tracking scheme effectively catches the implicit information flow without significant false positive rates. Although the `sprintf` function is quite complicated, our t_{PC} stack maintenance technique clears the t_{PC} at the right time so that the tainted tags do not spread throughout memory locations. Although there can also be false positives and false negatives introduced by the granularity of the memory tag, we can expect its impact to be small since character data is usually grouped together.

7. CONCLUSIONS

This paper presented IFTU, our external hardware engine for implicit (and explicit) flow tracking. To keep track of the implicit flows in a program, we employed a tracking scheme which utilizes a t_{PC} register and stack together with the code analysis and instrumentation technique that help us correctly manage the value of t_{PC}. To perform this task efficiently, we have installed within IFTU hardware logic specialized for the task, such as the t_{PC} stack. We have connected IFTU with the host processor via CDI to acquire the runtime information necessary for tracking, while minimizing the host performance degradation. Our experiments on an FPGA prototype showed that our IFTU can perform both the explicit and implicit flow tracking with only about 3% performance loss. In addition, the synthesis result revealed that the hardware resources required for efficient implicit flow tracking are only about 5.7% of the overall resources for IFTU.

8. ACKNOWLEDGMENTS

This work was supported in part by NSF/SRC STARSS 1526493, Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. R0190-16-2010, Development on the SW/HW modules of Processor Monitor for System Intrusion Detection), the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2014R1A2A1A10051792), the Brain Korea 21 Plus Project in 2016, and MSIP, Korea, under the ITRC (Information Technology Research Center) support program (IITP-2016-R0992-16-1006) supervised by the IITP, and IITP grant funded by the Korea government (MSIP) (No. R-20160222-002755, Cloud based Security Intelligence Technology Development for the Customized Security Service Provisioning).

9. REFERENCES

[1] ARM. *Embedded Trace Macrocell Architecture Specification*, 2011.

[2] Y.-Y. Chen. *Architecture for data-centric security*. PhD thesis, Citeseer, 2012.

[3] Y.-Y. Chen, P. A. Jamkhedkar, and R. B. Lee. A software-hardware architecture for self-protecting data. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 14–27. ACM, 2012.

[4] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 482–493. ACM, 2007.

[5] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. X. Song. Dynamic spyware analysis. In *USENIX annual technical conference*, pages 233–246, 2007.

[6] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[7] R. B. P. Laskov. Detection of intrusions and malware & vulnerability assessment. 2006.

[8] J. Lee, I. Heo, Y. Lee, and Y. Paek. Efficient dynamic information flow tracking on a processor with core debug interface. In *Proceedings of the 52nd Annual Design Automation Conference*, page 79. ACM, 2015.

[9] J. Lee, Y. Lee, H. Moon, I. Heo, and Y. Paek. Extrax: Security extention to extract cache resident information for snoop-based external monitors. In *Design Automation and Test in Europe Conference and Exhibition (DATE)*, 2015.

[10] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[11] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 135–148. IEEE, 2006.

[12] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[13] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ACM SIGOPS Operating Systems Review*, volume 38, pages 85–96. ACM, 2004.

[14] M. Tiwari, X. Li, H. M. Wassel, F. T. Chong, and T. Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 493–504. ACM, 2009.

[15] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *ACM Sigplan Notices*, volume 44, pages 109–120. ACM, 2009.

[16] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 243–254. IEEE, 2004.

[17] I. Xilinx. Microblaze processor reference guide v13. 4. *reference manual*, 2011.

[18] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127. ACM, 2007.

[19] Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. *Privacy Scope: A precise information flow tracking system for finding application leaks*. PhD thesis, University of California, Berkeley, 2009.