Control Flow Obfuscation with Information Flow Tracking*

Haibo Chen, Liwei Yuan,
Xi Wu, Binyu Zang
Parallel Processing Institute
Fudan University
{hbchen, yuanliwei, wuxi,
byzang}@fudan.edu.cn

Bo Huang
Intel China Software Center
bo.huang@intel.com

Pen-chung Yew
Department of Computer
Science and Engineering
University of Minnesota
yew@cs.umn.edu

ABSTRACT

Recent micro-architectural research has proposed various schemes to enhance processors with additional tags to track various properties of a program. Such a technique, which is usually referred to as information flow tracking, has been widely applied to secure software execution (e.g., taint tracking), protect software privacy and improve performance (e.g., control speculation).

In this paper, we propose a novel use of information flow tracking to obfuscate the whole control flow of a program with only modest performance degradation, to defeat malicious code injection, discourage software piracy and impede malware analysis. Specifically, we exploit two common features in information flow tracking: the architectural support for automatic propagation of tags and violation handling of tag misuses. Unlike other schemes that use tags as oracles to catch attacks (e.g., taint tracking) or speculation failures, we use the tags as flow-sensitive predicates to hide normal control flow transfers: the tags are used as predicates for control flow transfers to the violation handler, where the real control flow transfer happens.

We have implemented a working prototype based on Itanium processors, by leveraging the hardware support for control speculation. Experimental results show that BOSH can obfuscate the whole control flow with only a mean of 26.7% (ranging from 4% to 59%) overhead on SPECINT2006. The increase in code size and compilation time is also modest.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*unauthorized access*

General Terms

Security

Keywords

Control Flow Obfuscation, Information Flow Tracking, Opaque Predicate, Control Speculation

1. INTRODUCTION

Reverse engineering has good applications such as recovering valuable algorithms from legacy software, uncovering protocols and file forms [8] for intrusion detection systems. However, it also has many downsides. For example, software hackers often use reverse engineering tools to discover the software vulnerabilities and inject malicious code, especially for proprietary software. For example, the CodeRed worm [30], was released after "reverse-engineering the binary code" [17] of Microsoft IIS server. Meanwhile, software piracy, enabled by reverse engineering, is a severe threat to software industry.

Program obfuscation [6, 28, 18, 24] is a software protection tool to mitigate the security problems and software piracy raised by reverse engineering. It works by transforming a program into a functionally-equivalent counterpart, which however poses a significant barrier to uncover the high-level semantics and the structure of the program.

Though dynamic analysis virtually can reveal the real execution traces of any obfuscated programs, it is usually difficult to carry out in practice due to the required time and space. Modern processor executes billions of instructions per-second, which requires impractical amount of resources to collect traces and to apply traditional analysis techniques, while deciding when and where to start collecting traces is also difficult on the obfuscated program. Finally, dynamic analysis has the coverage problem in that it may require sufficient test cases to expose many bugs and vulnerabilities, which are usually hidden in rare program paths. This will add to the already time- and resource-consuming process. Thus, there are considerable research interests in providing program obfuscation techniques, to force adversaries to switch to dynamic analysis techniques.

Previous obfuscating approaches mostly use program transformations that rely on opaque predicates[1] [6] to obfuscate the control flow transfers, and then insert bogus code in untaken paths to obfuscate the data flow. Typical program transformations include control flow flattening [28], function pointers [22], branch functions [18], based on the fact that interprocedural alias analysis and pointer analysis are NP-hard [6, 28, 22]. However, these approaches usually come in with notable performance degradation if applied on the whole program level. For example, the control flattening approach [28] can incur performance overhead of more than 5X if

*This research was funded by China National 973 Plan under grant numbered 2005CB321905, National Science Foundation of China under grant numbered 90818015 and a research grant from Intel.

MICRO'09, December 12–16, 2009, New York, NY, USA.
Copyright 2009 ACM 978-1-60558-798-1/09/12 ...$10.00.

[1] A predicate is opaque if the value is known by an obfuscator, but hard to deduce by a deobfuscator.

applied to 80% of the branches. This creates a dilemma for obfuscation: obfuscating only a small region of code might not have good obscurity, while obfuscating the whole program (especially hot code) would significantly degrade the program performance.

To improve the stealthiness of control flow obfuscation, researchers recently propose using signal handling as a mechanism for obfuscation [24]. Specifically, it works by artificially generating exceptions and using the exception handling mechanisms (i.e., signal handling) to hide normal control flow. However, each exception used for obfuscation is standalone and flow-insensitive, which makes the level of obscurity directly rely on the number of exceptions. More importantly, the approach can incur high performance overhead due to the high cost of signal handling (more than 1000 cycles for one signal). For example, it could incur more than 43X performance overhead when obfuscating 90% of the branches. Thus, it can only be used to obfuscate cold traces. However, obfuscating only cold traces is usually not enough. First, valuable algorithms are usually in the hot traces of a program. Second, selective obfuscation might not be robust and resilient enough against sophisticated programming analysis tools such as program slicing [19].

In this paper, we propose a scheme that leverages mechanisms in information flow tracking [25, 7, 9] to do binary obfuscation. To effectively support information flow tracking, researchers have proposed various hardware extensions, including automatic propagation of tags and user-level handling of security violations [13, 9, 1].

This paper explores the architectural support for information flow tracking to enable a low-overhead and resilient control flow obfuscation scheme. The idea is *to treat the propagation of tags as a flow of flow-sensitive opaque predicates*. The proposed scheme, called BOSH[2], explores several features in information flow tracking to convert an artificial flow of taint tags to a dynamic flow of opaque predicates, which can then be used to obfuscate both the control flow and data flow of a program. First, BOSH makes a novel use of benign tag exceptions to generate stealthy taint sources. Second, the tags obtained from the tag exceptions can then be propagated across the program execution path. Finally, the program relies on the existence of the tags to force a normal instruction to raise a tag exception to alter the control flow.

Using the flow of tags as *opaque predicates* for obfuscation enjoys several advantages. First, the flow of tags is inherently a part of the data-flow of a program. Thus, the values of the tags are flow-sensitive instead of being constant values through the program execution as in previous approaches [6, 23, 20]. Second, operations on the opaque predicates are moved out of the critical execution path and could be executed in parallel with normal execution, which creates notable performance advantages. Finally, the control flow transfers are done using normal instructions instead of explicit *jump* instructions, making a static analyzer difficult to detect where the control transfer might happen.

Though information flow tracking is still an active research topic, modern commercial processors (i.e., Itanium I and Itanium II) have already been built with the support for information flow tracking. Specifically, Itanium extends each general purpose register with an additional bit (e.g., NaT bit) to track the deferred exceptions during speculative execution. We thus implemented a working prototype on Itanium processors using GCC to do transformation, instead of using simulation or emulation. Our preliminary results on SPECINT2006 show that BOSH can obfuscate the whole control flow graph, with only a mean of 26.7% (ranging from 4% to 59%) performance overhead. We also analyzed the resilience of the obfuscation and evaluated BOSH using IDA-pro.

In summary, this paper makes the following contributions:

- The use of the propagation of information flow tags as flow-sensitive opaque predicates.
- The BOSH design for binary obfuscation, which leverages the taint-propagation and user-level exception handling features in information flow tracking to obfuscate binaries.
- A working implementation of the above techniques on commercial processors (i.e., Itanium) through the novel use of hardware support for exception token propagation in control speculation. The evaluation results show that BOSH can obfuscate the whole control flow graph, yet comes with very low overhead.

The rest of the paper is organized as follows. The next section reviews previous literatures and provides some background information on control flow obfuscation and information flow tracking. Section 3 describes the design of BOSH that uses the architectural support of information flow tagging as the main building block. Section 4 follows with the implementation issues of BOSH on Itanium processors by leveraging the support for speculative execution. After some descriptions on the setup of experiments, the resilience of BOSH and its incurred performance overhead are evaluated in section 5.1 and section 5.2 accordingly. Finally, section 6 concludes the paper.

2. BACKGROUND AND RELATED WORK

BOSH is related to both control flow obfuscation and information flow tracking. This section reviews previous literatures and gives a brief introduction to control flow obfuscation (and its countermeasures), as well as information flow tracking and its architectural support.

2.1 Binary Obfuscation

According to Collberg et al. [5, 6], the process of software obfuscation is a set of semantic-preserving transformations that converts a program into an unintelligible one, yet with similar observable behavior. There are generally three types of program obfuscations: layout obfuscation, which applies a source-to-source transformation to render a program unreadable by human; intermediate-level obfuscation, which obfuscates a program at intermediate representation (IR) level, which is typically used for interpretation-based language such as Java; and binary obfuscation, that obfuscates the layout and control flow of binary code. This paper focuses only on binary obfuscation.

Generally, there are two necessary steps to reverse engineer binaries: disassembly [14], which transfers object code into assembly code; decompilation [4], which recovers the high level semantics from assembly code. Therefore, the general binary obfuscation approach focuses on two aspects: confusing the disassembly [18, 24], which is usually implemented on CISC machines (e.g., x86), by utilizing the variable length of instructions; disrupting the decompilation, usually by obfuscating the control and data flow, thus makes it difficult to uncover program semantics. Though the idea of BOSH can be applied to confuse both processes, we currently only focus on confusing the process of decompilation for the sake of portability.

Many current obfuscation schemes use opaque predicates [6, 20] to fake infeasible control flow, and then insert bogus code that further obfuscates the control and data flow. Opaque predicate is a

[2] BOSH is short for binary obfuscation using speculative hardware, as BOSH is prototyped on an Itanium machine with speculative hardware support.

conditional code that is always true or false (but not both), but the value is difficult to deduce by a deobfuscator.

```
if (x*x >= 0) {
       s1;
}
else {
       s2; // bogus code
}
```

The above code fragment shows a naive example of opaque predicate. Since "$x * x >= 0$" will always be evaluated to be true, statement "S1" can be obfuscated as an "if-then-else" statement and the additional bogus code "S2" can be inserted. Nevertheless, the shown example is rather simple and real obfuscation schemes usually use more sophisticated approaches such as aliased opaque predicates that are based on the difficulty of alias analysis.

Wang et al. [28] proposed a set of obfuscating transformations such as control-flow flattening and function pointers to obfuscate both the control and data flow of a program. Some of these techniques have already been used in commercial obfuscation software [3]. However, these techniques are implemented only at intraprocedural level, limiting their resilience. To improve the resilience of obfuscation, researchers propose implementing obfuscation based on the difficulty of interprocedural analysis, such as alias analysis and function pointer analysis [22], which is shown to be NP-hard. To defend against possible program slicing [29] attacks, Majumdar et al. [19] correlate each program slice and create data dependency between slices, thus forming a large program slice. Ge et al. [12] use a control-flow based obfuscation that separates one application into client-server like communicating processes. The client process relies on the server process to make decision on the target of the control flow. However, the communication cost can be prohibitively high. Most of these systems usually incur notable performance overhead when applied to the whole program level.

It should be noted that all of the above techniques obfuscate the normal control and data flow of a program using normal control flow transfer such as branches. For example, to fake an infeasible control flow, an obfuscator usually uses a predicate that is always true (or false) as the branch condition. It might be subject to constraint analysis of conditional code to filter out of unsatisfiable conditions [26]. To increase the stealthiness of binary obfuscation, Popov et al. [24] propose a signal based scheme that relies on exceptions to obfuscate the normal control and data flow to defend against static disassembly. Since exception is a natural behavior of regular instructions, a static deobfuscator can hardly determine its existence. However, since the cost of a signal is usually expensive (more than 1000 CPU cycles), frequent exceptions could significantly degrade the program performance.

2.2 Information Flow Tracking

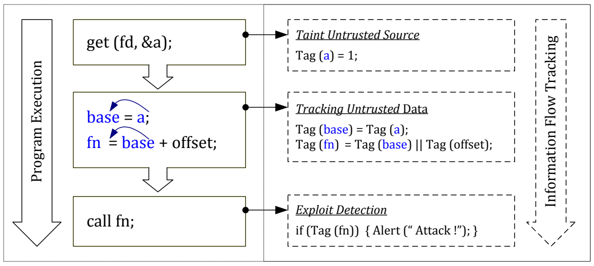


Figure 1: The general work flow of information flow tracking.

Figure 1 gives an overview of the work flow of information flow tracking (IFT). As shown in the figure, IFT has three stages: (1) tagging sources, by marking data from specific sources or execution results under some special conditions; (2) tracking tags in program execution. IFT assigns a tag to each memory location and the tag is propagated during program execution according to the program dependency (control or data dependency). A memory location is tagged if it is derived from tagged sources; (3) detecting potential tag violation. The tag marked or propagated in the previous two stages is checked before program data is used in an unexpected way. If an unexpected usage is detected, a tag violation alert is raised to report possible exceptions. Specifically, in taint tracking, a tag violation indicates a possible security attack is raised, while in control speculation, a tag violation means a piece of speculatively executed code should be reexecuted.

One typical use of information flow tracking is the use of taint tracking [9, 1] to defeat various attacks. Besides its application in security, IFT has also been used in understanding the life cycle of sensitive data [2] , visualizing the data flow of a full system [21] and improving the coverage of software testing [16].

Implicit vs. Explicit User-level Exception Handling: Recently, due to the importance of using information flow tracking for filtering out false alarms, intrusion analysis and debugging, information flow tracking systems are also extended with the support for user-level security exception handling [9]. Handling tag exception at user-level allows application software to directly gain control of tag violations, without the kernel intervention. Hence, the time spent on handling security violations (e.g., filtering false alarms) can be largely reduced. There are generally two kinds of user-level exception handling: implicit one, all instances for one type of exceptions jump to one centralized pre-registered handler [9], which resembles the signal handling mechanism; and explicit one, each instance of an exception has its own handler and explicit instructions are provided to jump to the handler [13].

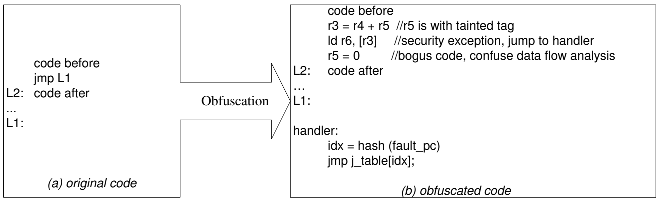
3. CONTROL FLOW OBFUSCATION WITH INFORMATION FLOW TRACKING

In this section, we describe how to explore the microprocessor support for information flow tracking to build a low-overhead, resilient and stealthy binary obfuscator against static analysis. We assume that the underlying processor at least has the tag propagation mechanism as well as user-level tag exception handling. One can also design new hardware for control flow obfuscation. However, we believe the essential parts should be similar.

3.1 Overview: From IFT to Obfuscation

To confuse a static deobfuscator, BOSH is designed to obfuscate the normal control and data flow of a program. To achieve this, BOSH exploits the fact that the value of the tag for a variable is flow-sensitive and hard to deduce for a static analyzer. Based on this fact, BOSH reuses the tags as *opaque predicates*. The value of a tag for a variable is used to determine the transfer of control flow, either normal or exceptional.

A simple and intuitive example using implicit exceptional handling is shown in Figure 2, which also depicts the main idea of BOSH. The left side of Figure 2 is the original code, which has one unconditional control flow to a code label *L1*. In the right side, the obfuscated code instead appears as a sequentially executed instruction sequences and contains no explicit control flow. The tagged data is obtained from the *r5*, which contains a tag. Since a register with a tag cannot be used as a load address (e.g., in taint tracking), loading from *r3* will trigger a tag violation and transfer the control



```
        code before
        jmp L1
L2:     code after
...
L1:
```

(a) original code

```
        code before
        r3 = r4 + r5  //r5 is with tainted tag
        ld r6, [r3]     //security exception, jump to handler
        r5 = 0          //bogus code, confuse data flow analysis
L2:     code after
...
L1:

handler:
        idx = hash (fault_pc)
        jmp j_table[idx];
```

(b) obfuscated code

Figure 2: An example of obfuscating control flow in BOSH using implicit user-level tag exception handling: an explicit *jmp* instruction is converted to several sequentially executed instructions; a tainted tag in r3 as the loading address to *ld* instruction will cause control tranfer to the violation handler, where the real control transfer happens.

to a user-level handler, which relies on a jump table to complete the real control transfer.

This simple example shows the key techniques that make BOSH satisfy the desired criteria for obfuscation:

- *Low-overhead:* BOSH incurs very low overhead for two reasons. First, the tags can be propagated in parallel with normal program execution, making maintaining the opaque predicates at virtually no cost. Second, the control transfer is done via the user-level tag exception handling, instead of expensive OS-trap, which indicates significant performance advantages.

- *Resilience:* The opaque predicates (i.e., tags) are summarized and stored in hardware, instead of relying on code property such as "$x * x >= 0$". Therefore, it is less vulnerable to those attacks that use a constraint solver to filter out infeasible conditions [26]. Moreover, to statically determine whether some data is tagged or not, an attacker needs to obtain the global control and data flow information, which could be further obfuscated by adding bogus code. BOSH can insert bogus code that confuses the tag propagation (e.g., clearing the tags), fakes infeasible control flow edges and nodes and even falsifies call sites and targets.

- *Stealthiness:* As a number of instructions may trigger tag exceptions, it is hard to statically determine where the actual control flow transfer happens. For example, according to the pointer taintness policy [25, 9] in taint tracking, *load* from a tainted address should trigger an exception. Using tainted data as the conditional code or branch targets may also trigger exceptions. For a CISC architecture such as X86, nearly every instruction may operate on memory. This makes the obfuscation even stealthier.

3.2 Binary Obfuscation Using IFT

To obfuscate a program, BOSH mainly relies on the following steps: (1) stealthily generating the source of tags using benign tag exceptions; (2) statically creating an interprocedural data flow to propagate the tags and changing the normal control flow according to the value of tags; (3) Inserting bogus code which has no visible side-effects to obfuscate the normal control and data flow to static analysis.

3.2.1 Manufacturing Stealthy Tag Source Using Benign Tag Exceptions

The first step to create a flow of tags is obtaining the tag source. Knowing the tag source will not allow a static analyzer to easily derive the taint flow due to the sophisticated control and data transformations (section 3.2.2 and section 3.2.3). However, to further confuse a static analyzer, BOSH makes novel uses of *tag exceptions* to generate tag source, but in a benign way.

```
//attstr can be dynamically forged using string composition.
//attstr is marked as tainted.
char * attstr="<Stackpop><<addr1><addr2><addr3><addr4>>
               <%N1u%n%N2u%n%N3u%n%N4u%n>";
...;
char    buffer[512];
snprintf (buffer, sizeof (buffer), attstr);
buffer[sizeof (buffer) - 1] = '\0';
```

Figure 3: Example attack string to generate taint source.

Figure 3 shows an example to manufacture stealthy taint source using memory errors in taint tracking. There are various types of memory errors such as *buffer overflow*, *heap overflow* and *format string* that are hard to detect by a static analyzer. Most existing static analyzers require accesses to the source code, and with some program annotations [15], thus are hard to be applied to binary code with sophisticated obfuscating transformations. If a memory error cannot be detected, it can virtually modify any parts of the program's code and data.

The piece of code shown in Figure 3 is simplified from wu-ftpd. The taint propagation policy is used as follows: storing a tainted value to an address will taint the address. In this example, a tainted input format string *attstr* is sent to the *snprintf* function. The *Stackpop* in attstr is a macro and used to increase the stack pointer to point to $addr_1$. Afterwards, each time the *snprintf* encounters %n, N_i will be written into $addr_i$ (i=1, 2, 3, 4), thus tainting $addr_i$. Hence, by forging N_i and $addr_i$, any memory locations can be tainted in this way. Note that the *attstr* can be dynamically forged using string composition to make it difficult to guess the address. The calls to *snprintf* can also be converted to indirect calls using function pointers.

3.2.2 Control Flow Transformation Using the Tag Flow

The goal of this step is to confuse the static analyzer from obtaining a correct *control flow graph* (CFG). Determining the CFG in a normal program is not difficult since the branch and control transfer instructions are usually easy to identify. Thus, BOSH aims to hide both the branch and control transfer instructions using security exceptions. The basic mechanisms were already shown in Figure 2.

BOSH first predetermines a static flow of tags at compile time. The tags can be propagated in normal program execution but cannot be used in inappropriate ways (such as call targets). Second, BOSH collects all control flow transfer instructions in a program. It then randomly converts a certain percentage of such instructions by triggering tag exceptions and using exceptional control flow to do the transfer. Since the exceptional control flow is essentially unconditional and thus cannot handle a conditional control transfer, BOSH converts the conditional control transfer to unconditional control transfer before using the exceptional flow.

All exceptional control transfer will go through the tag exception handling mechanism, either an implicit or explicit one. For an implicit exception handling mechanism, BOSH uses a jump table to do the control transfer. To get the correct jump target for a security exception, BOSH needs to maintain the mapping between the *PC* of the original control transfer instruction and its jump target. To avoid exposing the mapping table, a perfect hash function can be used to map the *PC* to the index of the jump table and use it to access the real jump target. Some bogus jump targets can be forged into the jump table. This can further confuse the static analyzer from guessing the start of a basic block using the entries in the table. The implicit exception handling approach might incur some performance overhead due to the computation of hash and indirect references to the jump table, but is very effective in hiding the normal control flow.

For explicit exception handling, BOSH inserts an instruction to check the existence of security exceptions and redirect the control flow directly to the real target if necessary, instead of going through a centralized handler. This can be more efficient, though at the expensive of stealthiness.

3.2.3 Data Flow Transformation to Hide Tag Flow

The flow of tag propagation is still not resilient enough since it only involves normal use of tagged data. BOSH thus inserts more bogus code and data that involves potential abnormal uses of tags to confuse the static analyzer. For example, BOSH creates aliased pointers to the tagged locations and uses pointers to modify the values. BOSH also creates artificial basic blocks and control flow transfers to confuse a flow-sensitive analysis. To obfuscate the call graph, function calls are selectively replicated in the bogus code to mess the call graph.

3.3 Discussion

3.3.1 Obfuscation Strength

Attackers need to derive the tag flow by distinguishing the control flow and identifying infeasible data flow. Theoretically, BOSH uses aliased pointers to necessitate an interprocedural, context-sensitive, may-alias pointer analysis. Attackers are required to understand precisely the may-alias set from an obfuscated control/data flow graph.

3.3.2 Attacking Scenarios

The necessary steps to attack obfuscated binary code involve disassembling the binary code, gaining the control flow graphs and then performing program analysis and transformations to undo the effects of obfuscation.

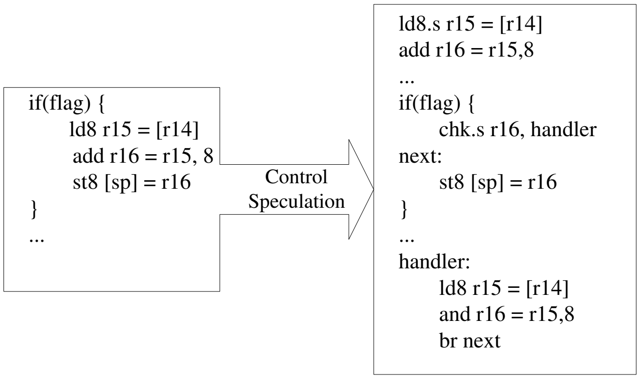
However, to obtain the real control flow and data flow graphs, a determined static analyzer (i.e., being aware of the approach taken by BOSH) must first derive the whole tag flow. As the tags are stored in hardware instead of using code property, it is less vulnerable to symbolic execution or constraint solving. Further, as there are no explicit code patterns for real control transfer, using statistical analysis can derive little information about the probability of control transfer for an instruction.

Besides, BOSH brings no convenience to attackers using dynamic analysis approaches. Since the flow-sensitivity nature of BOSH, it is less vulnerable to single-point of leakage. That is, knowing a single value of a tag does not lead to the leakage of other tags in a code region. This is because, the value of a tag is sensitive to the flow of execution, instead of being always the same value as in other approaches [6].

Finally, BOSH currently does not aim to protect against dynamic program analysis. One can also incorporate other approaches such as anti-debugging [11] and anti-analysis techniques to further defend against dynamic analysis.

3.3.3 Impact on Other Uses of Tags

Using tags as flow sensitive opaque predicates might prohibit the original use of the tags, such as taint tracking or speculative execution. However, the tag flow used for control flow obfuscation is determined at compile time. Thus, other flows not used for obfuscation can still be used for the intended uses, which might further confuse a static analyzer. Fortunately, recent research proposes multiple bits of tags and allows flexible uses of tags [9, 27], which can further allow the coexistence of control flow obfuscation with other uses of tags.



```
if(flag) {
      ld8 r15 = [r14]
      add r16 = r15, 8
      st8 [sp] = r16
}
...
```

```
ld8.s r15 = [r14]
add r16 = r15,8
...
if(flag) {
      chk.s r16, handler
next:
      st8 [sp] = r16
}
...
handler:
      ld8 r15 = [r14]
      and r16 = r15,8
      br next
```

Figure 4: An example of control speculation: the ld instruction is speculatively moved up to hide its latency. An instruction checking exceptions (chk.s) is inserted in its initial location to catch speculation failure and jump to the handler.

4. IMPLEMENTING OBFUSCATION ON ITANIUM

Though information flow tracking is still an active research area, Itanium has already been built with the support for information flow tracking in the form of deferred exception tracking. We thus choose to implement BOSH on Itanium instead of relying on simulation or emulation. We have implemented a working prototype using GCC4.2.3 [3] to do obfuscating transformations.

This section first describes the hardware features that are useful to support binary obfuscation. Then, it presents how BOSH is implemented on Itanium based on control speculation.

4.1 An Overview of Architectural Support on Itanium

To hide the high latency of memory operations, Itanium is built with the support for control speculation, which allows instructions to be speculatively executed before knowing whether they need to be executed or not. However, executing such speculative instructions might cause exceptions, which might not belong to the normal behavior of the program. To handle this case, Itanium implements *deferred exceptions* for speculatively executed instructions, by which an exception is deferred instead of being thrown out immediately. For example, Itanium provides speculative load (e.g., ld.s) instructions, which are with similar semantics with regular load instructions but the exceptions are deferred for later handling.

To support deferred exceptions, each general purpose register is extended with an additional deferred exception token (e.g., NaT, Not a Thing) to keep track of exceptions. The token is propagated along with program execution. Itanium also provides instructions to check the existence of exceptions such as *tnat*, which tests if a

[3] While BOSH can be similarly implemented on other aggressively optimized compilers such as IMPACT and ICC, we choose to implement it on GCC because GCC is a standard distribution along with Linux.

register contains an exception token (i.e., NaT bit) and *chk.s*, which jumps to some recovery code if the register is with an exception token. Figure 4 gives an example of control speculation: the *ld* instruction is speculatively moved up to hide its latency. If an exception occurs in *"ld8 r15 = [r14]"*, an exception token will be recorded in *r15*. The token is then propagated to *r16* and tested by the *chk.s* instruction, which examines the exception token in *r16* and transfer the control to the handler code.

4.2 Implementing Obfuscation on Itanium

From the previous section, we can see that the microarchitectural support for control speculation can be viewed as another form of IFT: both of them extend the general purpose registers with additional tokens and propagate the tokens along program execution; both of them provide mechanisms to handle user-level exceptions. However, there are still some differences between them, which pose challenges to the implementation of BOSH.

First, the exception tokens on Itanium are not allowed to flow to memory systems, while using explicit instructions to save the tokens in memory will provide clues to attackers. For example, a special store instruction (i.e., st.spill) should be used to save the NaT token in a register to memory, which is a hint to the existence of a NaT token. Propagating tokens only in registers might limit the stealthiness. Second, the user-level exception handling is explicit instead of implicit, thus requiring instructions to trigger them and do the control transfer. For example, to trigger control transfer according to the NaT token, either *tnat* or *chk.s* should be used to test a register. This might sacrifice the stealthiness of control transfer.

To overcome these problems, we make several design tradeoffs in implementing BOSH on Itanium, to preserve the resilience and performance of BOSH. To preserve resilience, we partition the flow of tags into the interprocedural level and intraprocedural level. For interprocedural level tags, we simulate them based on the data-flow problem that, determining whether a pointer is aligned or not in the presence of pointer alias requires interprocedural, context-sensitive, may-alias pointer analysis, which requires precisely understanding the may-alias set, thus is very difficult. Specifically, we simulate the tags by using the property whether a pointer is aligned to a specific value (e.g., 1, 2, 4 or 8). In Itanium, loading from an unaligned address will trigger an exception, which will be deferred and recorded in the NaT token if the load instruction is speculatively executed. Thus, the tag propagation is maintained at the interprocedural level by modifying the values of pointers. The alignment of a pointer is flow-sensitive and not constantly the same. Aliased pointers are used to modify the alignment of a pointer during the control flow.

For intraprocedural propagation of tags, we make use of the fact that Itanium has a rich set of registers and propagate them mainly in registers. The source of the NaT token is obtained by loading from the interprocedural pointers.

To save the performance loss, we use the explicit way of triggering control transfer according to NaT tokens. However, to make it difficult for an attacker to guess the real control transfer by identifying the checking instruction (e.g., tnat and chk.s), we convert all explicit control transfer instructions to the form that uses the NaT tokens as oracles to direct the control flow, as well as inserting faked control transfer using NaT tokens in bogus code.

4.3 Implementation Details

We have implemented BOSH by modifying GCC4.2.3 for the Itanium processor. The changes to GCC consist of several passes added to the compilation process. The following details the implementation of BOSH on Itanium using GCC to do transformations.

4.3.1 NaT Token Generation

For the intraprocedural tag propagation, we insert a number of pointers as the tag sources. The initial alignment of the pointers is crafted using our faked *format string attacks*, as described in section 3.2.1.

To prevent a static analyzer from distinguishing between loads and speculative loads that generate NaT tokens, all load instructions are converted to speculative loads. To avoid the visible side effects from the conversion, we adjust several hardware bits controlling the NaT bit generation and make the speculative load behaves similarly to normal load, except that loading from an unaligned address will generate a deferred exception.

4.3.2 Steps in Binary Obfuscation

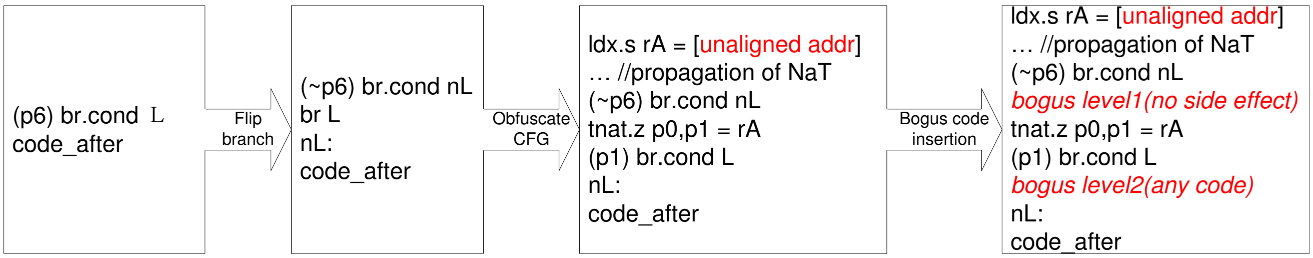
The following paragraphs describe the steps to do obfuscation transformation, code example of obfuscating a branch instruction shown in Figure 5.

- *Flipping conditional branch:* Control flow transfer using NaT tokens is only applicable to unconditional branches. According to our measurements, a large proportion of the branches are conditional. To increase the candidates, BOSH first converts all conditional branches to unconditional branches. This is done by first reversing the predicate register (i.e., *∼p6*) and creating a conditional jump to the fall-through code label (i.e., *nL*), then an explicit unconditional jump (i.e., *br*) is inserted after it.

- *Obfuscating control transfer:* Two candidates are available to obfuscate control transfer in Itanium: *tnat* and *chk.s*. However, *chk.s* requires 18 cycles to do a control transfer if the register is with a NaT token, while tnat requires only 1 cycle. Thus, BOSH chooses *tnat*. In this step, the jump instruction is replaced using *tnat* to test the existence of NaT in this step. In the code example, the instruction "tnat.z p0,p1 = rA" tests the existence of NaT token and sets the corresponding predicate registers. The actual control transfer then depends on the value of the predicate register (i.e., *p1*).

- *Inserting bogus code:* Two levels of bogus code can be inserted in this step, which provides opportunities to implement control and data flow obfuscation described in section 3.2.3. The first level bogus code is on the real execution path and should have no visible side effect on a program. To prevent unexpected control flow transfer in the first level bogus code, BOSH relies on the data-flow analysis in the compiler to guarantee that a NaTed register will not be used in an unsafe way. One interesting feature on Itanium is that using a register with NaT token as comparing target will set both predicate registers to zero, preventing both branches from being taken. This feature essentially breaks the assumptions by most de-obfuscators, which assume that *"conditional jumps can be either taken or not taken"* [14]. Hence, an informed attacker now needs to assume three possible targets for a branch. As shown in figure 6. BOSH explores this feature to create faked control flows and bogus code, which makes static analyzer much harder to analyze the control flow graph.

  The second level bogus code will not be executed and thus can be arbitrary code. Here, BOSH inserts code mainly aiming to introduce non-trivial aliases to confuse the analysis of tag propagation. Specifically, BOSH inserts arbitrary number of pointer variables and creates indirect references to normal variables using the pointers. These pointers appear to



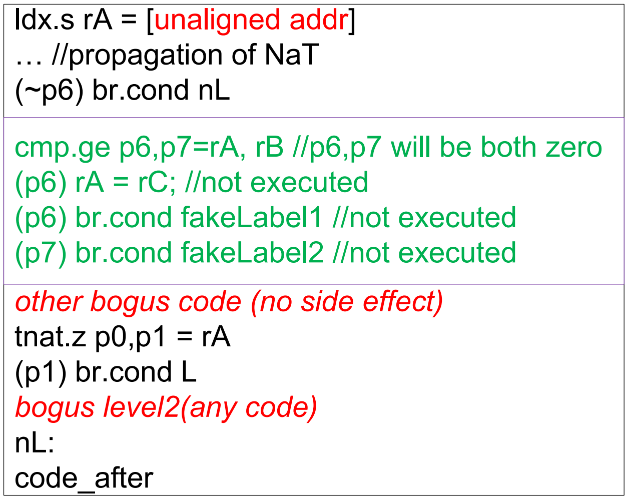
```
(p6) br.cond L
code_after
```

```
(~p6) br.cond nL
br L
nL:
code_after
```

```
ldx.s rA = [unaligned addr]
… //propagation of NaT
(~p6) br.cond nL
tnat.z p0,p1 = rA
(p1) br.cond L
nL:
code_after
```

```
ldx.s rA = [unaligned addr]
… //propagation of NaT
(~p6) br.cond nL
bogus level1(no side effect)
tnat.z p0,p1 = rA
(p1) br.cond L
bogus level2(any code)
nL:
code_after
```

Figure 5: Example of the steps to obfuscate a conditional control transfer. The conditional branch is executed if the predicate register p6 is true.



```
ldx.s rA = [unaligned addr]
… //propagation of NaT
(~p6) br.cond nL
cmp.ge p6,p7=rA, rB //p6,p7 will be both zero
(p6) rA = rC; //not executed
(p6) br.cond fakeLabel1 //not executed
(p7) br.cond fakeLabel2 //not executed
other bogus code (no side effect)
tnat.z p0,p1 = rA
(p1) br.cond L
bogus level2(any code)
nL:
code_after
```

Figure 6: Insertion of bogus code that breaks the assumption that conditional jumps can be either taken or not taken.

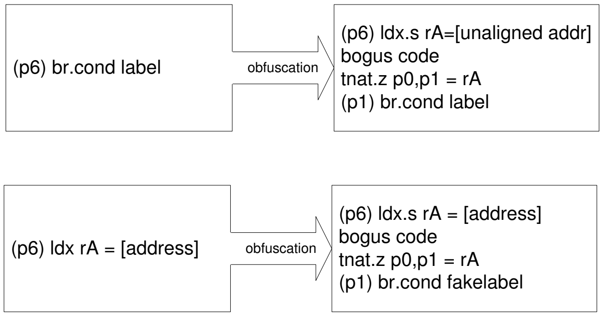
have dependency and alias with the NaTed registers and unaligned memory locations. Further, BOSH selectively replicates control transfer code from existing code to the bogus code, to shuffle the control flow. BOSH also replicates some calls to existing functions to mess the call graph. Since the bogus code in the second level will not be actually executed. We can virtually add arbitrary code with very little overhead.

- *Converting ld and inserting bogus code:* this step converts all normal ld into the speculative versions, and randomly adds bogus code after the conversion. Figure 7 shows the example code to obfuscate the *ld* and branch (i.e., *br.cond*) instruction. It can be found that the code used to obfuscate *ld* and *branch* is almost the same. The only difference lies in the alignment of the *address*, which can hardly be deduced by a static analyzer.

Note that we mainly do most of the transformations of BOSH on the low-level IR (intermediate representation, RTL in GCC). These transformations thus interact with the normal compiler optimization such as register allocation and instruction scheduling. These compiler optimizations will bring more obfuscation effects on the final code and make the code more natural.

5. EXPERIMENTAL SETUP

All tests were performed on an HP Integrity rx1620 server equipped with two 1.6GHz Itanium processors and 4GB of memory running



```
(p6) br.cond label
```

```
(p6) ldx.s rA=[unaligned addr]
bogus code
tnat.z p0,p1 = rA
(p1) br.cond label
```

```
(p6) ldx rA = [address]
```

```
(p6) ldx.s rA = [address]
bogus code
tnat.z p0,p1 = rA
(p1) br.cond fakelabel
```

Figure 7: The obfuscation effects for ld instruction and branch instruction, which are almost the same. The difference lies in whether the *addr* is aligned or not.

Redhat Linux Enterprise 4. We test all C benchmarks except perlbench[4] of SPECINT2006 executed with all reference inputs. To evaluate the efficiency of BOSH, we measure both the effects and resilience of obfuscate code, as well the incurred overhead on both execution time and code size.

For resilience tests, we use the *control flow error* as the metric and evaluate the relative control flow errors by comparing the obfuscated code with the original code. For performance tests, we compare the performance of the benchmarks compiled using the original GCC4.2.3 and our obfuscating compiler at the -O2 optimization level.

5.1 Obfuscation Strength

It is generally hard to quantify the strength of an obfuscation scheme. In this section, we attempt to evaluate the effectiveness of BOSH to count the control flow errors. The results are obtained by using IDA-pro [10], which is a state-of-the-art reverse engineering tool, to analyze both the control flow graph of the original and obfuscated programs. We also show the stealthiness of BOSH by comparing the instruction distribution of an obfuscated program with that of unmodified one. Finally, we use IDA-pro to visualize the effect of obfuscation on a simple bubble sort program.

5.1.1 Control Flow Errors

Generally, there are two sources of errors in identifying the control flow graph of an obfuscated program for a static analyzer. First, the control flow edges appeared in the original program may not appear in the obfuscated program , since BOSH may have converted the original control flow into control transfer using NaT tokens. Second, there are several sources of bogus code control transfer added in the obfuscated program, such as converting from *load* to

[4] Perlbench failed to compile on IA64 with gcc4.2.3.

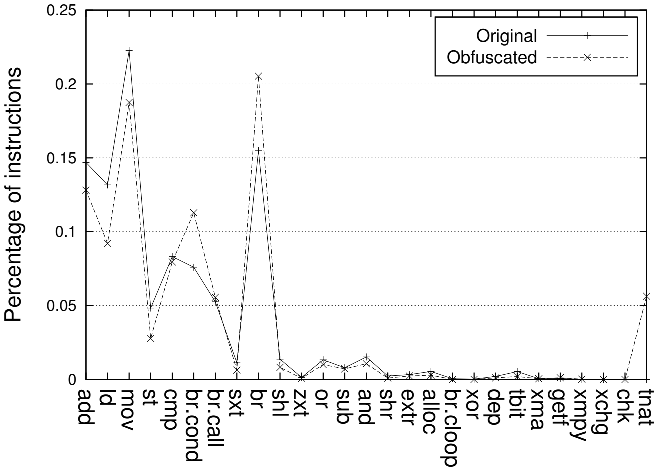


Figure 8: The obfuscation stealthiness: Distribution of Individual Opcodes of gcc in SPEC2006

speculative load (Fig. 7), bogus control transfer when using NaTed register as comparing targets, as well as the arbitrary control transfer code in the second level bogus code in obfuscating a conditional branch.

Table 1 shows the obfuscating effects on the whole control flow graph of eight applications in the SPECINT2006 benchmark suite. We measure the effect of obfuscating transformations in three metrics: changes to the call graph, control flow edges and the basic blocks. As shown in the table, on average, there is a factor of 3.62 of call edges and a factor of 5.68 of control flow edges, which means BOSH inserts more than 4.68X bogus edges including 2.62X bogus call, which IDA-pro failed to distinguish. Also, for the basic blocks, we can see a mean factor of 5.02.

The significant changes to control flow graph, cooperated with the bogus code that changes the data flow, can make a static analyzer (IDA-pro in our case) hard to obtain the true control flow graph.

5.1.2 *Obfuscation Stealthiness*

Figure 8 shows the distribution of major instruction opcodes. We choose gcc to illustrate results as gcc has more types of opcodes compared to others such as bzip2. As shown in the figure, the percentage of each opcode in the obfuscated version is still close to that in the original version, except for branches and tnat. This is expected because BOSH adds faked control flow graphs and call graphs, causing the increases of branches. Besides, a number of tnat instructions are introduced to obfuscate control transfer.

5.1.3 *Visualizing the Effects of Obfuscation*

To give an intuitive view on the effect of obfuscation, we use IDA pro to extract the control flow graph of both the original program and obfuscated one. To make the graph still distinguishable, we use a small program that implements the bubble sort algorithm.

Figure 9 shows the original control flow graph and the obfuscated one. The green, red and blue edges represent the backward edges, non-fallthrough edges of conditional branches and fall-through (also including the call and unconditional branch) edges, respectively. It can be seen that IDA pro cannot distinguish most of the faked infeasible control flow edges. Hence, we can see a dramatically change to both the control flow edges and nodes.

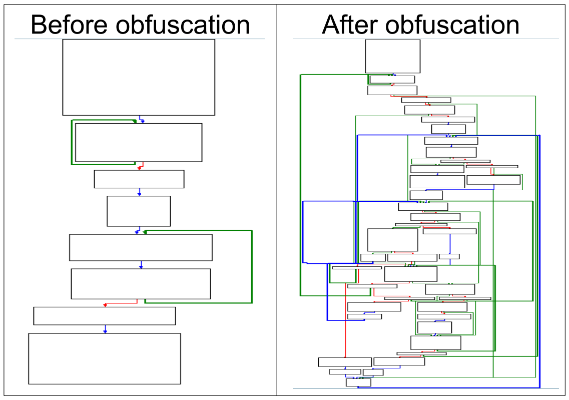


Figure 9: The obfuscation effects on bubble sort. The green, red and blue edges represent the backward edges, non-fallthrough edges of conditional branches and fall-through (also including calls and unconditional branches) edges.

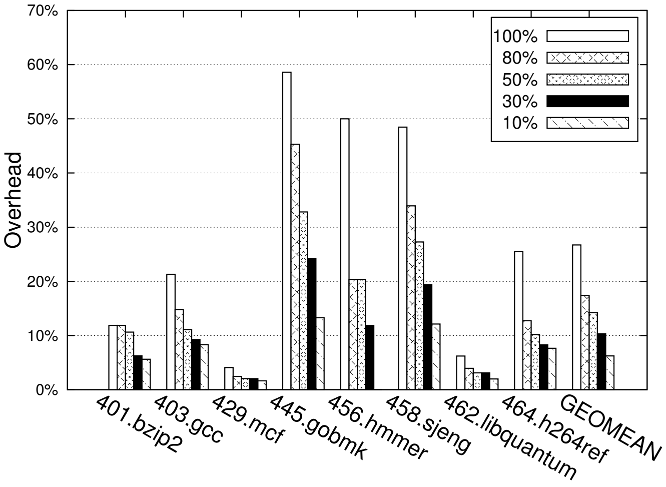


Figure 10: The performance overhead of BOSH for SPECINT2006 when randomly obfuscating 100%, 80%, 50%, 30% and 10% of the control flow transfer.

5.2 Performance Evaluation

A practical software obfuscator should not incur notable performance slowdown, or users are unwilling to trade performance for obscurity. In this section, we quantitatively evaluate the impact of the obfuscating transformations on the performance, code size and compilation time.

5.2.1 *Execution Time*

We measured the performance overhead of eight applications in SPECINT2006, as well as the sensitivity of the performance on the proportion of obfuscated control transfer. The results are shown in Figure 10. For full program obfuscation, the performance slowdown ranges from 59% (*gobmk*) to 4% (*mcf*), with a geometric mean of 26.7%. When the proportion of the obfuscated branch decreases, the mean overhead also degrades: the mean overhead for obfuscating 80%, 50%, 30% and 10% of the control flow are 17%, 14%, 10% and 6%, respectively. The performance overhead for *hammer* behaves a bit abnormally which is the same for obfuscating 80% and 50% and no overhead when obfuscating 10%, probably due to the unexpected interaction between the compiler opti-

Application	Call Graph			Overall Control Flow Edges			Basic Blocks		
	Ec_{ori}	Ec_{obf}	Ec_{ori}/Ec_{obf}	Ea_{ori}	Ea_{obf}	Ea_{ori}/Ea_{obf}	N_{ori}	N_{obf}	N_{ori}/N_{obf}
bzip2	686	6835	9.96	2902	28783	9.92	2288	22323	9.76
gcc	71132	206076	2.90	191586	935657	4.88	175866	733303	4.17
mcf	197	531	2.70	891	5224	5.86	672	3485	5.19
gobmk	16723	46883	2.80	46915	215614	4.60	42529	168944	3.97
hmmer	6797	21175	3.12	18103	92474	5.11	16594	73117	4.41
sjeng	2171	7690	3.54	7701	38343	4.98	6389	29280	4.58
libquantum	915	2702	2.95	1826	8631	4.73	1904	7627	4.01
h264ref	6291	25967	4.13	21650	149233	6.89	18307	108169	5.91
GEOMEAN	3287	11894	3.62	10314	58572	5.68	8985	45069	5.02

Table 1: The effects of obfuscating transformations on control flow graph. Ec_{ori} and Ec_{obf} represent the number of original edges, obfuscated edges in original and obfuscated call graphs respectively, while Ea_{ori} and Ea_{obf} represent the number of all original and obfuscated edges in control flow graph. N_{ori} and N_{obf} are the numbers of basic blocks in the original and obfuscated programs, respectively.

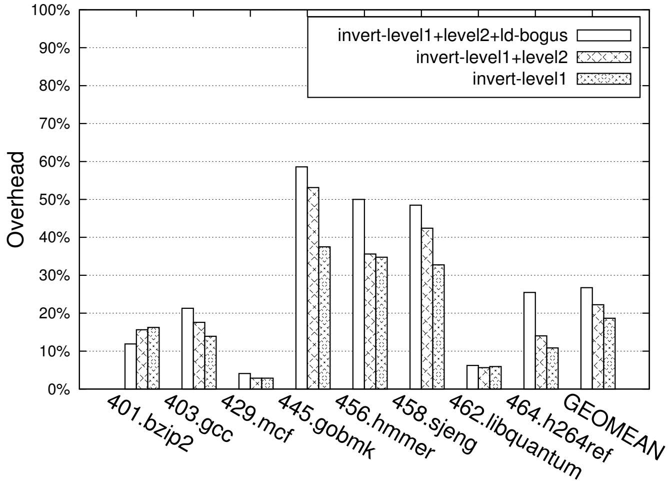


Figure 11: A decomposition of performance overhead of BOSH on SPECINT2006 under full program obfuscation.

mizations such as instruction scheduling and register allocation, as well as the variety in the proportion of randomly selected branches from the hot trace.

Note that there is some associated overhead due to the insertion and maintenance of aliased pointer variables, as well as the impact of obfuscation on compiler optimizations. Further, the added bogus code also affects the branch predicator. Thus, even obfuscating a small percentage (i.e., 10%) of control transfer still brings about 6% performance slowdown. The profiling results using oprofile show that, there is a 39% increase (from 11,430,369 to 15,901,799) in the number of branch mis-predications for *bzip2* even only 10% of the control flow transfer is converted using NaT tokens.

To further understand the source of the overhead, we present a decomposed overhead for the obfuscation transformations: inverting conditional branches and inserting the level-1 bogus code [5] (*invert-level1*), inserting level-2 bogus code (*level2*) and converting normal ld into speculative one and adding bogus code (*ld-bogus*). As shown in Figure 11, the step *invert-level1* brings the most performance overhead (a mean of 18.7%), due to the fact that the level1 bogus code is still in the code execution path. Inserting the second level of bogus code does not add much overhead (a mean of 3.5%) since the code are actually not executed. Converting *ld* to *ld.s* does not bring performance overhead but adding the bogus code after *ld.s* causes a 4.7% increase in performance overhead. Nevertheless, the overall performance overhead is still small and thus the obfuscation transformations are still cheap enough to be applied for production runs.

5.2.2 Code Size Expansion

Application	Orig.(Kb)	Obfusc.(Kb)	Ratio
bzip2	168.76	889.55	5.27
gcc	8636.50	23198.89	2.69
mcf	54.87	123.04	2.24
gobmk	7014.04	10692.10	1.52
hmmer	984.66	2731.95	2.77
sjeng	2882.20	3540.64	1.23
libquantum	111.93	289.92	2.59
h264ref	1992.95	5322.87	2.67
GEOMEAN	878.62	2118.21	2.41

Table 2: Impact of obfuscation on the increase of code size in kilobyte.

Application	$T_{orig}(s)$	$T_{obf}(s)$	Ratio
bzip2	14.08	240.31	17.07
gcc	634.24	1549.36	2.44
mcf	3.05	3.98	1.30
gobmk	154.25	256.06	1.66
hmmer	61.85	92.11	1.49
sjeng	28.06	44.70	1.59
libquantum	6.78	8.97	1.32
h264ref	139.08	298.30	2.14
GEOMEAN	40.24	89.67	2.23

Table 3: Impact of obfuscation on the compilation time in second.

The impact of full program obfuscation on the size of binary code is shown in Table 2. We can see that the code size factor ranges from 1.23 (sjeng) to 5.27 (bzip2), with a mean factor of 2.41. The increase of code size mainly comes from the added bogus code, as well as a little increase from converting a conditional branch to an unconditional one.

[5] We cannot measure the cost of branch inversion standalone since gcc will convert it back if there is no further transformation.

5.2.3 Compilation Time

We also measured the impact of obfuscating transformations on the compilation time for full program obfuscation. As shown in Table 3, the compilation time factor ranges from 1.30 (mcf) to 17.07 (bzip2), with a mean of 2.23. The increase of compilation time is dominated by the proportion of branches. The *bzip2* has a much higher increase than others due to the extremely long time (e.g., from 3.57s to 45.03s in compiling compress.c) in obfuscating functions which are full of branches.

6. CONCLUSION

In this paper, we proposed a flow-sensitive control obfuscation scheme that exploits the microprocessor support for information flow tracking. The proposed system, namely BOSH, makes novel use of tags as opaque predicates to obfuscate control flow graph and insert bogus code, instead of traditional uses as oracles to detect security exploits or speculation failures. We have implemented a working prototype based on Itanium processors, by exploiting the hardware support for deferred exception propagation and user-level exception handling. Experimental results show that BOSH can obfuscate the whole control flow graph, and is resilient to a state-of-the-art reverse engineering tool (i.e., IDA pro). Performance results show that BOSH incurs only small performance degradation even for full program obfuscation on SPECINT2006. The incurred code size expansion and compilation time increase are also modest.

7. REFERENCES

[1] H. Chen, X. Wu, L. Yuan, B. Zang, P. Yew, and F. Chong. From Speculation to Security: Practical and Efficient Information Flow Tracking Using Speculative Hardware. In *Proc. ISCA*, pages 401–412, 2008.

[2] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proc. USENIX Security Symposium*, pages 321–336, 2004.

[3] S. Chow, Y. Gu, H. Johnson, and V. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *Proc. Information Security Conference*, pages 144–155, 2001.

[4] C. Cifuentes and K. Gough. Decompilation of Binary Programs. *Software - Practice and Experience*, 25(7):811–829, 1995.

[5] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. *University of Auckland Technical Report*, 170, 1997.

[6] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proc. POPL*, pages 184–196, 1998.

[7] J. Crandall and F. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proc. Micro*, pages 221–232, 2004.

[8] W. Cui, J. Kannan, and H. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *16th Usenix Security Symposium*, 2007.

[9] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proc. ISCA*, pages 482–493, 2007.

[10] datarescue. IDA Pro. http://www.datarescue.com/idabase/.

[11] M. Gagnon, S. Taylor, and A. Ghosh. Software Protection through Anti-Debugging. *IEEE SECURITY & PRIVACY*, pages 82–84, 2007.

[12] J. Ge, S. Chaudhuri, and A. Tyagi. Control flow based obfuscation. In *Proc. DRM*, pages 83–92, 2005.

[13] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the IA-64 architecture. *IEEE Micro*, 20(5):12–23, 2000.

[14] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static Disassembly of Obfuscated Binaries. In *Proc. Usenix Security Symposium*, 2004.

[15] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proc. USENIX Security Symposium*, 2001.

[16] T. Leek, G. Baker, R. Brown, M. Zhivich, and R. Lippmann. Coverage Maximization Using Dynamic Taint Tracing. Technical Report 112, MIT Lincoln Laboratory, 2007.

[17] R. Lemos. Tracking code red. http://news.cnet.com/2009-1001-270471.html, visited May, 2009, 2001.

[18] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. CCS*, pages 290–299, 2003.

[19] A. Majumdar, S. Drape, and C. Thomborson. Slicing obfuscations: design, correctness, and evaluation. In *Proc. DRM*, pages 70–81, 2007.

[20] G. Myles and C. Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Research*, 6(2):155–171, 2006.

[21] S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood. Understanding and visualizing full systems with data flow tomography. In *Proc. ASPLOS*, pages 211–221, 2008.

[22] T. OGISO, Y. SAKABE, M. SOSHI, and A. MIYAJI. Software Obfuscation on a Theoretical Basis and Its Implementation. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 86(1):176–186, 2003.

[23] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang. Experience with software watermarking. In *Proc. ACSAC*, pages 308–316, 2000.

[24] I. Popov, S. Debray, and G. Andrews. Binary Obfuscation Using Signals. In *Proc. Usenix Security Symposium*, 2007.

[25] G. Suh, J. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proc. ASPLOS*, pages 85–96, 2004.

[26] S. Udupa, S. Debray, and M. Madou. Deobfuscation: Reverse Engineering Obfuscated Code. In *Proc. Working Conference on Reverse Engineering*, pages 45–54, 2005.

[27] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: Programmable Architectural Support for Efficient Dynamic Taint Propagation. In *Proc. HPCA*, 2008.

[28] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *Proc. DSN*, 2001.

[29] M. Weiser. Program slicing. In *Proc. ICSE*, pages 439–449, 1981.

[30] C. Zou, W. Gong, and D. Towsley. Code red worm propagation modeling and analysis. In *Proc. CCS*, pages 138–147, 2002.