BeCFI: Detecting Hidden Control Flow with Performance Monitoring Counters

Zhou HongWei
Information Engineering University
Zhengzhou, China
Key Laboratory of Data Engineering and Knowledge Engineering (Renmin University of China), Ministry of Education
Beijing, China
E-mail: hong_weizhou@hotmail.com

Shi WenChang
School of Information, Renmin University of China
Key Laboratory of Data Engineering and Knowledge Engineering (Renmin University of China), Ministry of Education
Beijing, China
E-mail:wenchangshi@gmail.com

Yuan JinHui*
Corresponding author
Information Engineering University
Zhengzhou, China
E-mail: jcyjh@126.com

Li FuLin
Information Engineering University
Zhengzhou, China
E-mail: leefulin@163.com

Abstract: Most of existing control flow integrity efforts target keeping intended control flow in good integrity. However, they fail to expose hidden control flow that may be introduced by the execution of rootkits, ROP gadgets, etc. To overcome the challenge, we propose an innovative approach BeCFI to detect hidden control flow based on cross-view principle. Since modern processors are capable of observing the execution of all branch instructions, BeCFI obtains the hardware view with the support of performance monitoring counters(PMC). To obtain software view, we build a software-based counters by compiler-patching and binary-overwriting, and monitors the execution of branch instruction with software-based counters. If a control transfer only appears in hardware view, BeCFI considers that it is hidden control transfer. We have developed a prototype system on Intel x86 Linux kernel. Our evaluations show BeCFI is capable of detecting the hidden control flow introduced by kernel rootkits and ROP attacks. Furthermore our performance tests demonstrates that BeCFI incurs an acceptable overhead.

Keywords: control flow integrity; operating system; kernel; branch; performance monitoring counters.

Biographical notes: Zhou Hongwei, received his Ph.D. degree in Renmin University of China in 2012. He now works at Information Engineering University of China. He is involved in several research programs, including National 863 project, NSFC, and so on. His main research interests include Trusted Computing and Operating System.

Shi Wenchang, Ph.D., Professor, Head of the Department of Computer Science and Technology, Renmin University of China. He is a senior member of China Computer Federation. He has carried out a series of research programs in the area of Information Security and System Software. His current research interests include Information Security and Trusted Computing.

Yuan Jinhui, received her Ph.D. degree from Renmin University of China in 2012. She is a Lecturer in Information Engineering University of China. Her main research interests include Information Security and Wireless Sensor Network.

Li FuLin works at Information Engineering University of China. His main research interests is information security.
1 Introduction

Control flow integrity (CFI) is considered as a strong security property. It is hard to launch the attacks without breaking control flow integrity. Furthermore, CFI provides a basis for building other protecting mechanisms Zeng et al. (2011). A promising circumvent approach is non-control-data attack Chen et al. (2005) because it maliciously changes the behavior of the softwares by tampering with their non-control data. Thus the software still run along control flow graph (CFG), and its control flow is still in good integrity. However, the attackers have to reach their malicious end with “good” control flow, and this heavily limits the capability of the attackers.

What is hidden control flow? Control flow can be divided into two categories: intended and unintended. Intended control flow is produced by the execution of original instructions, while unintended control flow is introduced by injected new instructions or misused existing instructions. For example, an attacker installs a kernel rootkit, and the rootkit inevitably produces unintended control flow which are out of CFG. As other example, to a malicious end, ROP attacks wave short gadgets together by misusing existing instructions, and produce some sneaky control flow. Unintended control flow are also called hidden control flow for their covert nature. For ease of presentation, we use the term unintended control flow and hidden control flow interchangeably in this paper.

There is a collection of CFI efforts Abadi et al. (2005); Petroni and Hicks (2007); Bletsch et al. (2011); Li et al. (2011); Zhang et al. (2013). However, most of them target to keep known control flow in good integrity, but fail to catch hidden control flow. The cause is that they limit their monitoring objects to intended indirect jump instructions such as indirect jmp/call, ret. As an example, CFI Abadi et al. (2005) identifies some indirect jump instructions in the software, and inserts the label and label-checking instructions around every monitored instruction. At running time, CFI checks the destination of every indirect jump instruction against the inserted label. However, the labels and label-checking instructions are inserted around the intended instructions. This means that CFI only detects the violation of known control flow, and fail to expose hidden control flow.

There is another observation drives us to detect hidden control flow: above solutions can not completely protect intended control flow. Though these work heavily limit the attacker’s capability to subvert computer system without being detected, they still leave a narrow gap open. Ideally, if every intended control transfer is legal, the attackers will never have the chances to launch malicious execution. Thus there are no hidden control flow in computer system. However, due to the dynamic nature, it is impossible to identify all destinations of indirect control transfer. It is a hard challenge, and no solution completely overcome it. Some of existing CFI efforts have to conservatively assume the fully coverage. Therefore, it is possible to circumvent the current protections and introduce hidden control flow. Since hidden control flow are the side-effect of the attacks, detecting hidden control flow is a fundamental approach to catch them. We believe that the approaches of detecting hidden control flow are complementary to above solutions.

The modern processors provide a hard ground to expose hidden control flow. An attacker may cheat computer system administrator with hidden control flow, but it is impossible to fool the processors. Since every instruction must be issued to the processors, the execution inevitably changes the context of processors. Thus it leaves a key to find hidden control flow. Recently some hardware-based approaches, which are capable of detecting hidden control flow, are presented. CFIMon Zhang et al. (2013) utilizes a processor’s feature, called branch trace store (BTS) Intel (2009), to capture all interesting control flow in computer system. However, it detects the violation of control flow by checking the targets of control transfer, and incurs the above incomplete coverage. With Last Branch Record (LBR) Intel (2009), R0Pecker Cheng et al. (2014) is able to detect the hidden control flow introduced by ROP attacks. However LBR only stores serval branch records and can not fully support monitoring a large body of instructions.

In this paper, with the support of Performance Monitoring Counter (PMC) Intel (2009), we propose an innovative approach, which is called BeCFI (Branch Length Control Flow Integrity), to detect hidden control flow. Originally, PMC is designed to count some interesting performance events including the execution of special branch instructions. By utilizing PMC, we are able to know the sum of branch instructions issued to the processors including intended instructions and unintended instructions. In other words, with the support of PMC, we can obtain a hardware view that contains intended control flow and hidden control flow. This provides a chance to catch hidden control flow.

BeCFI is a cross-view approach to detect hidden control flow. As mentioned earlier, BeCFI builds a hardware view with the support of PMC. Compared to it, we obtain a software view by monitoring the execution of intended instructions. Our solution is to build a software-counter by compiler-patching and binary-overwriting. More specifically, we insert a snippet of instructions before monitored branch instruction, and the instructions increment the software-counter when the monitored instruction is taken. If a control transfer only appears in the hardware view, BeCFI concludes that it is hidden control transfer. On the difference between two views, BeCFI detects the existence of hidden control flow.

We have developed a prototype system of BeCFI on Intel x86 Linux kernel. BeCFI is implemented as a configurable-target system. To reach the goal, BeCFI is accompanied with the configurable map. Once BeCFI is loaded in the kernel, it selectively actives the
increment triggers that scatter in the whole kernel with the configurable map. Our evaluations show BeCFI is capable of detecting the hidden control flow introduced by kernel rootkits and ROP attacks. Furthermore, our performance tests demonstrates that BeCFI incurs an acceptable performance overhead.

The rest of the paper is organized as follows. Section 2 and section 3 present our design and implementation of BeCFI respectively. Section 4 is the evaluation of BeCFI. The discussion of our solution is detailed in section 5. Section 6 surveys related work. At last, we concludes this paper.

2 Design

As shown in the figure 1, BeCFI is a cross-view solution including software view and hardware view. We obtain hardware view with the support of PMC, and hardware view records the trace of intended control flow and unintended control flow. We build software view on our developed software-based counter that is called SCounter in this paper. Noted that SCounter only records intended control flow. Thus, BeCFI is capable of detecting hidden control flow on the difference between hardware view and software view.

Figure 2 shows an example that BeCFI detects hidden control flow. As shown in the figure 2, for a given binary snippet in a clean running, there are 6 branch items. However, an attacker may subvert the software, and redirect the control flow to the malicious binary. The malicious execution introduces 2 hidden control transfers that are shown as the shadow rectangle in the figure 2. These hidden control transfers are redundant for a normal execution. Since PMC is capable of counting every control transfer, we are able to catch the redundant control transfers with its support. Furthermore, BeCFI detects hidden control flow on the redundant control transfer.

2.1 Obtaining software view

To prepare data for building software view, we develop a software counter which is called SCounter. Existing software architecture fails to count taken branch instructions. To overcome it, we build SCounter by control flow redirection. SCounter is accompanying with thousands of increment-triggers and one increment-body. Increment-trigger is a five-byte instruction which is inserted just before each monitored branch instruction.

Thus, before branch instruction is taken, increment-trigger is waken to call increment-body. Thus, we redirect the control flow to our code. The principal task of increment-body is to increment SCounter. However, the increment disturbs original context of the processors. To address it, before incrementing SCounter, increment-body performs context-switch. More specifically, BeCFI pushes the status of a processor onto the stack once it is triggered. Furthermore, before recovering original execution, BeCFI pops up the contents of the stack to restore the context of processors. Thus BeCFI increments SCounter without breaking original context.

There is a requirement of monitoring a specially kind of branch instruction in a given execution path. For example, to check ROP attacks Shacham (2007) which is launched on the gadgets ending with ret instruction, BeCFI is able to detect ROP attacks by only monitoring ret instruction. Therefore, we only insert an increment-trigger before each ret instruction, and let other branch instructions free. There may be an extreme scenario that all branch instructions are monitored. Furthermore, in most of time, only some snippets of instruction sequences can attract the attention of people. To balance the performance overhead and coverage of detection, people usually focus on some critical execution paths.

To meet the dynamic requirement, a configurable checking map is developed that holds all increment-points for current detection. Increment-point is the location for inserting increment-trigger. At the beginning of the detection, thousands of increment-triggers are inserted into the software binary according to increment-points in the checking map. Since checking map is configurable, a user can build a checking map for monitoring one kind of branch instruction in some given execution paths. Thus BeCFI only inserts the triggers before the interesting branch instructions on the map. Moreover a user may have some different checking maps for different detections.

Now, we summary our approach to obtain software view with the figure 3. We build an checking map to hold all increment-points for covering the interesting branch instructions. As shown in the figure 3, indirect call instructions are our interesting instruction, and an increment-trigger is inserted just before each one. In the checking time, there are thousands of increment-triggers that are inserted into software binary with the configurable checking map, and these triggers share
an increment-body for incrementing SCounter once the monitored instruction is taken. The solid lines in the figure 3 show the control flow redirection after the triggers are inserted into the binary, while the dashed lines demonstrate the data flow. Furthermore, to avoid breaking context of original execution, BeCFI perform context-switch in time as shown in the figure 3.

2.2 Obtaining hardware view

We obtain hardware view with the support of PMC Intel (2009). In our opinion, PMC can be considered as a collection of some data MSRs and control MSRs Intel (2009). Each data MSR is accompanying with a control MSR. A control MSR sets the interesting performance event that we want to catch, while accompanying data MSR counts the event. For ease of presentation, in this paper, data MSR is called hardware counter. To prepare the data for building hardware view, we have to read the same hardware counter twice. More specifically, we read the hardware counter to get one beginning-reading at a time. After some time, we read the same hardware counter again, and get one ending-reading. With two readings, we have known the times that the processor incurs the performance event in the monitoring time.

To catch the monitored performance event, we have to set the control MSR, and some performance events can be used for our detection. PMC is able to count hundreds of performance events, and part of them are our candidates. However not every kind of branch instruction is accompanied with an appropriate event. Fortunately, there are some approximate events as the alternatives. In our opinion, although there is a manifest distinction between them, it does not weaken checking capability with a prompt detection. Some observations support our conclusion. We have monitored the benign executions of ref instruction with alternative event in the kernel, we got 0.53% more that software counter. This means that a little noise is introduced. However, with a prompt detection, the noise can be discriminated easily. Therefore, in this paper, we conservatively assume that PMC provides the fitting events.

Noted that SCounter is used as the storage in building hardware view. In other words, at the time that SCounter is also incremented by increment-body, SCounter is updated with the readings of the hardware counter. The principal goal for our design is to limit the security sensitive object to one counter. Thus we can focus on the protection for SCounter. In our design, SCounter can hold an unlimited number of readings. To the end, SCounter is updated with a special rule. Once BeCFI collects a reading from the hardware counter, BeCFI updates the reading to SCounter. The pseudo code for the update is: value_new = value_old - reading.

When do we read hardware counter? We propose a SCounter-based approach. More specifically, we define the overflow condition of SCounter, and check the status of SCounter overflow once SCounter is incremented. Once SCounter overflow condition has occurred, we read hardware counter to obtain the prepared data. In this paper, we call the overflow condition as checking-step. In fact, we detect hidden control flow at the time of reading hardware counter. It is evident that the frequency of checking depends on checking-step.

The remaining challenge is the method to monitor a given execution path. To address it, we improve some increment-points as by adding a working token or sleeping token. When BeCFI incurs a increment-point with working token, it begins to check hidden control flow and read hardware counter. Similarly, BeCFI stops the detection when it incurs sleeping token. To monitor an execution path, we first identify the beginning increment-point and ending increment-point, then add a working token or sleeping token for them. Noted that one increment-point may has two tokens, and this means the monitored path is a circle.

2.3 Detecting algorithm

Our detecting algorithm is SCounter-central. At the first checking time, SCounter is set to zero that is a checking baseline. Then SCounter is updated with the reading of hardware counter, and updating rule is described as value_new = value_old - reading. At the next checking time, SCounter is updated with the new reading of hardware counter again. Between the checking times, SCounter is incremented once monitored branch instruction is taken. Ideally, at the checking time, SCounter should be zero. This means the processors do not execute redundant monitored branch instructions, and no existence of hidden control flow.

Although detecting algorithm is not complex, there are something to be discussed in detail. First, our algorithm is n-steps algorithm, and n is a vital element impacting on runtime overhead. In our design, n is checking-step which is the moment that SCounter overflows. In other words, if SCounter has performed n increment, BeCFI reads hardware counter and checks hidden control flow. It is evident that n is close related to the performance overhead of BeCFI. For example, 1-step detection means that BeCFI performs the detection once SCounter is incremented. 1-step detection may frequently read and write the counters, and it may incur heavily performance overhead. However, if n is too big, BeCFI may incur Time-Of-Check-to-Time-Of-Use(TOCTOU).

Second, we have to handle the overflow of hardware counter. In essence, a hardware counter is a 32-bit
register, and it is destined to overflow after a long
time running. There is an interesting problem: how
long a hardware counter overflows? In our opinion, the
overflow circle is closely related to two factors. One is
the executed instruction sequence. If there is a large
body to be monitored, it maybe overflow quickly. The
other is the type of monitored instruction. For example,
we want to monitor ret instructions or all branch
instructions. It is evident that the latter would make
the overflow more faster. To avoid hardware counter
overflow, a complete solution is to reset hardware counter
in time. An alternative is to query overflow condition
of hardware counter when the counter overflows. To
decrease performance overhead, we refer to the latter.

Last is the noise introduced by the execution of
BeCFI. BeCFI has to take some branch instructions
for performing the detection. For example, increment-trigger is a five-byte call instruction, and it would
produce one branch item. These new introduced branch items are caught by PMC, and only occur in hardware
view. The cause is we do not insert increment-trigger for BeCFI. Moreover, the noise is able to be cleared
because BeCFI introduces a known number of branch
items. Thus, we can clear the noise by adding them into
SCounter.

Now, we conclude our detecting algorithm with
algorithm 1. At the beginning time, we set SCounter
to zero as the baseline. Then we update SCounter
with the reading of hardware counter. Before reaching
the next checking point, SCounter is incremented once
a monitored instruction is taken. To clear the noise introduced by the execution of BeCFI, we increase
SCounter by the noise since hardware counter also
do that. Once SCounter overflows, BeCFI updates
SCounter with the current reading of hardware counter
again. After that, if SCounter is zero, BeCFI considers
that current system is normal and no existence of
hidden control flow. Otherwise, there have to a further
detection. On the above discussion, there are two factors contribute to the abnormality. One is the overflow
of hardware counter, the other is hidden control flow.
Therefore, BeCFI queries the condition of hardware
counter to make the situation clear. If hardware counter
does not overflow, BeCFI report hidden control flow.

Algorithm 1:

Require: \(SCounter = 0, noise = k, step = n\),
readings of hardware counter is \(v_1, v_2, ...\),

\[SCounter = SCounter - v_1;\]

while (monitored instruction is taken) and (BeCFI
has a working token) do

\[SCounter++;\]

\[SCounter = SCounter + k;\]

if \((SCounter \geq n \times (k + 1))\) then

\[SCounter = SCounter - v_2;\]

if \(SCounter = 0\) then

no existence of hidden control flow.
else

if (hardware counter does not overflow) then

DETECT HIDDEN CONTROL FLOW.

end if

end if

end while

3 Implementation

We have implemented a prototype of BeCFI on Intel
x86/Linux. BeCFI is implemented as a loadable module,
and targeting OS is fedora 5 with a 2.6.15-1 kernel.
When BeCFI module is loaded in the kernel, it inserts
thousands of increment-triggers into the kernel with the
configurable checking map which is determined ahead
of time. In the running time, BeCFI monitors the
given execution paths for obtaining hardware view and
software view. On cross-view principal, BeCFI detects
hidden control flow in the kernel.

In our implementation, the main challenge is to insert
the trigger instructions into the kernel. To overcome
it, we present the solution combining compiler-patching
with binary-overwriting. We first patch \texttt{gcc} that is
an open source compiler, and recompile the kernel
with patched \texttt{gcc} for build a patched kernel. In this
step, our goal is to insert some placeholders just
before each monitored instruction. In this paper, the
placeholder is a five-NOP instruction which will be
overwritten by increment-triggers. At the time of BeCFI
module is loaded, BeCFI module queries the map to
obtain the addresses of placeholders, and replaces these
placeholders with increment-triggers. After that, BeCFI
is ready to detect hidden control flow.

The detail of compiler-patching is first to be
discussed. The compiler is \texttt{gcc} of version 3.4.0. It is well
known that \texttt{gcc} makes three transformations, and
the last is from RTL instruction list to assembly code, and
our approach targets the last transformation. During
the last transformation, \texttt{gcc} generates assembly code
with a special machine description. Instruction patterns
is a part of machine description, and we modify them
to patch \texttt{gcc}. More specifically, we modify \texttt{i386.md}
file for our processor, and ask \texttt{gcc} to insert a five-NOP
instruction just before the branch instruction when the
kernel source code is recompiled. Consequently, the
kernel has a five-byte space as the placeholder to hold
trigger instruction. However, in our implementation,
patched \texttt{gcc} fails to manage the inline assembly code in
a Linux kernel. To cover them, we have to modify kernel
source code to hold trigger instructions.

Binary-overwriting is finished by BeCFI module. At
the module is installed in the kernel, it overwrites the
five-NOP instruction with increment-trigger instruction.
As mentioned earlier, checking map holds the addresses
of these placeholders that will be overwritten. Therefore,
we have to know every address of them. To do that, we
utilize a tool, called \texttt{objdump}, to recover the assembly
code of patched kernel, and record the address of
every five-NOP instruction. With checking map, BeCFI
smoothly inserts increment-triggers into the kernel. The user of BeCFI can generate his/her checking map by choosing some from the sum of all addresses. The selected ones means that these five-NOP instructions will be overwritten.

4 Evaluation

To evaluate its effectiveness, we have tested BeCFI with two ported real-world rootkits and one ROP attack. Test results indicate that BeCFI is capable of detecting hidden control flow. To test the performance overhead, we have tested it with UnixBench, and performance evaluations show BeCFI incurs a acceptable performance overhead.

4.1 Effectiveness

To evaluate its effectiveness, we had tested BeCFI with 4 ported real-world rootkits and one ROP attack. With the table 1, we presented the causes that we tested BeCFI with them. There are two steps to produce hidden control flow. First step is breaking control flow. To do that, the attacker tamper legitimate code or control data. In the kernel, there are two main categories of control data: function pointer and return address. As shown in the table 1, the attacks are launched by tampering the above kinds object. Second step is the execution on the malicious code. Most of attacks depend on the injected malicious code, but some tricky attacks on the existing code such as ROP attack. So we tested BeCFI with a ROP attack. We believe our checking attacks were very representative. First, they covered main methods to produce hidden control flow. Moreover, they were launched by two main methods including tampering code or control data.

We had implemented the tests as follows. We first loaded BeCFI module into patched kernel. Then the rootkits or ROP attack were launched in the kernel. At last, BeCFI detected the hidden control flow introduced by the malicious execution. To ready for the tests, we configured a ROP module in the kernel following the method discussion in ref Shacham (2007). The module provided the gadgets ending with ret instruction and had a stack overflow exploit. Our designed ROP module may be considered as a virtuous attack for it only increased one word in the kernel. However, if it targeted one critical kernel non-control-data, it was able to subvert the kernel Chen et al. (2005).

We explained further our tests with the table 2. We tested BeCFI with three checking-steps and two monitored performance events. Our evaluation indicated BeCFI was able to report hidden control flow with 1-checking-step. However, in some checking scenarios, BeCFI failed to detect hidden control flow with 100-checking-step. Moreover, we had detected hidden control flow introduced by ROP attack with 100-checking-step by monitoring executed branch instructions, and BeCFI found hidden control flow in ten tests. However BeCFI failed to do that when it monitored ret instructions with the same checking-step.

<table>
<thead>
<tr>
<th>Attack</th>
<th>checking-step</th>
<th>monitored event</th>
</tr>
</thead>
<tbody>
<tr>
<td>adore-ng 0.56</td>
<td>1/50/100</td>
<td>branch instruction</td>
</tr>
<tr>
<td>enyelkm 1.1</td>
<td>1/50/100</td>
<td>branch/ret instruction</td>
</tr>
<tr>
<td>ROP</td>
<td>1/50/100</td>
<td>branch/ret instruction</td>
</tr>
<tr>
<td>wnp5 0.26</td>
<td>1/50/100</td>
<td>ret instruction</td>
</tr>
<tr>
<td>override</td>
<td>1/50/100</td>
<td>branch instruction</td>
</tr>
</tbody>
</table>

Table 2 The detail of our tests.

What contributed to the above problem? Ideally, BeCFI was able to expose any hidden control flow if every benign execution paths were monitored. However, BeCFI did not cover all execution paths in the kernel. More specifically, there are two causes contributed to above problem. First, our implementation failed to manage inline assembly code in the kernel. However, we believe that it did not weaken BeCFI. To clear it, we had performed a test as following. We recovered the assembly code of the kernel with objdump, and checked every ret instruction. We found that only about 0.5% ret instructions were not accompanied with our increment-trigger. We believed that it did not disturb the detection. Second, we failed to insert increment-trigger into some kernel modules. We were not able to obtain the source code of all modules. Therefore we failed to recompiled them with our patched gcc to insert the triggers into them. In fact, we had performed some tests to identify them. When BeCFI monitored branch instructions with 100-checking-step, BeCFI was capable of detecting hidden control flow in 50 tests. However, when BeCFI monitored ret instruction with same checking-step, BeCFI usually reported hidden control flow before ROP attack was launched. We found the cause of failure was that some unrecompiled modules were launched, and BeCFI considered that they produce hidden control flow because we failed to insert increment-triggers into them. If all code were recompiled by our patched gcc, BeCFI was definitely capable of exposing any hidden control flow. However, we advise that BeCFI should monitor a given execution path, but fully software.

As shown in the table 3, the above tests showed that BeCFI was capable of detecting any hidden control flow, and exposing the rootkits and ROP attacks that produced hidden control flow. The cause was that BeCFI performed the detection on the nature of malicious
execution that some instructions had to issued to processors for the malicious end. Thus we were able to catch them with the support of PMC. There are two different detection methods related to our solution. First is rootkit checker, and second is ROP attack detector. Compared to them, BeCFI detected malicious executions on the nature of malicious execution. Most of existing rootkit checkers, like BeCFI Petroni and Hicks (2007), defeated the rootkits on the characteristics of the rootkits, and failed to caught hidden control flow or ROP attacks. CFIMon Zhang et al. (2013) had the potentiality to catch hidden control flow because it recorded all issued branch instructions. However, it detected the abnormality by checking the target of every control transfer, and failed to catch hidden control flow. Some ROP defenders, like DynIMA Davi et al. (2009), were capable of exposing hidden control flow produced by ROP attack. To the end, these solutions checked the frequency of issued ret instruction or the length of instructions between issued ret instructions. However, they were not able to find the rootkits. Moreover, once some novel ROP attacks are presented, they would fail to detect the new ROP attacks.

<table>
<thead>
<tr>
<th>solution</th>
<th>hidden control flow</th>
<th>rootkit</th>
<th>ROP</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeCFI</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>SBCFI etc</td>
<td>Fail</td>
<td>OK</td>
<td>Fail</td>
</tr>
<tr>
<td>CFIMon</td>
<td>?</td>
<td>OK</td>
<td>Fail</td>
</tr>
<tr>
<td>DynIMA etc</td>
<td>OK</td>
<td>Fail</td>
<td>OK</td>
</tr>
</tbody>
</table>

Table 3. The comparison of effectiveness.

4.2 Performance

To evaluate the performance overhead of BeCFI, we tested it with with UnixBench of version 4.1.0 with the default configuration UnixBench (2012). The testing hardware environment was a Thinkpad x200 with an Intel 8600 CPU, and the targeting OS is fedora 5 with a Linux 2.6.15-1 kernel. To make result precise, we took the average of ten performance experiments as the final result. The tests were implemented as follows. We first recorded the final score of UnixBench for a clear kernel running without any increment trigger inserted. Then we recorded the score when some triggers are inserted into the kernel. At last, we compared the final scores of two scenarios, and obtained the performance overhead of BeCFI.

We had performed the tests to show performance overhead of full detection and partial detection. Full detection was an extreme case that we inserted all increment-triggers into the kernel. Noted that some modules are free of BeCFI, and no increment-triggers were inserted in them. Unlike full detection, partial detection only monitored some critical execution paths. Since system call is vital for the security of the kernel, we performed the partial detection by monitoring the kernel execution path of system call. We believed that monitoring the critical execution path was a good choice.

Figure 4 indicates the performance evaluation of BeCFI when it performed full detection or partial detection. In the tests, ret was the monitored branch instruction, and 12137 increment-triggers were inserted into the kernel. Moreover BeCFI performed a 50-step detection to expose hidden control flow in time. As shown in the figure 4, when BeCFI was installed in the kernel, the different tasks of UnixBench had a different performance slowdown. Some tasks incurred the most performance slowdown, while some ones seldom incurred slowdown. The final score showed that BeCFI incurred a 32.1% performance slowdown when it performed a full detection. Compared to full detection, BeCFI, which performed a partial detection, only introduced a 15.8% performance overhead. The performance overhead of BeCFI is acceptable.

To identify the main contributor of performance overhead, we had performed the following tests. BeCFI checked the kernel with the different checking-steps, and performed a full detection. To make clear of the result, we considered the score of clear run as the baseline. As shown in the figure 5, if BeCFI checked the kernel with 500-checking-step, it introduced 27.7% slowdown. Meanwhile, 1-checking-step incurred 70.6% slowdown. In our opinion, reading hardware counters may be a main contributor of performance overhead. On the test, it was possible that BeCFI only introduced about 27% performance overhead if we can release “noise” further.
Compared to CFIMon Zhang et al. (2013) and DynIMA Davi et al. (2009), BeCFI incurred less performance overhead. CFIMon Zhang et al. (2013) was developed with the support of BTS. However, BTS asked the processors to store the information of control transfer into the memory, and it introduced heavy performance overhead. DynIMA Davi et al. (2009) was configured on the binary instrumentation framework, and every instruction was checked before issued to the processors. Therefore DynIMA also incurred heavy slowdown.

5 Discussion

SCounter may be an attack hot. If SCounter is modified by the attacker, the attacker can circumvent our detection. In this paper, we conservatively assume it is in the integrity. We do not discuss how to protect SCounter, and consider that it is out of the scope of this paper. Although SCounter is a dynamic kernel data that the attackers usually modify, we limit the protected targets to a data. This make it more easy to be protected than before. Moreover, there are some promising approaches to the end. For example, in our opinion, SIM Sharif et al. (2009) provides a ground for restricted access to SCounter.

There is a significant difference between BeCFI and most of existing control flow enforcements. BeCFI detects hidden control flow by counting taken branch instruction, while existing solution check the targets of control transfers. Thus BeCFI avoids the challenge to identify all valid targets of dynamic control transfer. Moreover it is impossible to identify their all targets due to the dynamic nature. Unlike these solutions, BeCFI identifies all intended branch instruction. In other words, BeCFI identifies the sources of branch instruction, but targets. Noted that it is feasible to identify all branch instructions. Therefore, we circumvent the hard nut.

In the future, we have to improve BeCFI for overcoming some disadvantages. In this paper, we do not discuss the method to remove the noise introduced by the alternative events. However, to avoid the false positive, we have to overcome it. Moreover, we conservatively assume that PMC provides the fitting events. But we fail to find a fitting event for a special processors in some scenarios. In fact, we fail to identify an event to monitor indirect jmp instruction for our tests processor. As a future work, we plan to identify them for other processors. In our prototype system, we believe some paths are possible to be out of the monitoring. Therefore our solution may incur a false positive. We have to improve the coverage, and a promising solution is the change of our compiler-patching method. At last, it is necessary to further decrease the overhead of BeCFI. In current implementation, tens of instructions are executed for obtaining one hardware counter reading. We believe some instructions may not be necessary with a major redesign.

6 Related work

6.1 Protecting Intended Control flow

In strictly, Control Flow Integrity(CFI) is first presented by M. Abadi Abadi et al. (2005). Since code integrity provides a strong ground for protecting static control flow, CFI focuses on dynamic control flow and takes the indirect jump instructions as its monitoring objects. To check every dynamic control transfer, it inserts the labels and label-checking instructions around the monitored instructions, and checks their targets with the information on the inserted labels. Similarly, Control-Flow Locking Bletsch et al. (2011) also inserts some locking instructions and unlocking instructions in the monitored execution path. Like them, we build software counters by inserting increment-trigger instruction before each monitored instruction.

SBCFI Petroni and Hicks (2007) and Indexed hook Li et al. (2011) present two interesting approaches to protect kernel control flow. Since the attackers usually persistently modify kernel control data to prolong the control of the kernel, SBCFI Petroni and Hicks (2007) checks the kernel control data in interval for the less performance overhead. Moreover SBCFI considers that a control data is in good integrity if the destination is known kernel code. Indexed Hook Li et al. (2011) replaces the kernel control data with the presented index, and the original data are stored in the protected tables. In the patched kernel, index is first fetched to acquire the real destination.

CCFIR Zhang et al. (2013) and FPGate Zhang et al. (2013) all target to protect the control flow of user space applications. CCFIR Zhang et al. (2013) collects the targets of indirect jump instructions together, and puts them into a springboard section. Thus, CCFIR has a faster detecting algorithm than traditional solutions. FPGate Zhang et al. (2013) utilizes relocation tables to identify the indirect transfer instructions and their jump targets. Moreover, FPGate encodes the function pointer to address compatibility issues. However, they are developed on the x86/Windows, and it means can not apply to other mainstream system, like Linux.

There are other CFI solutions related to our work. MoCFI Davi et al. (2012) and CFR Pewny and Holz (2013) present two CFI enforcements for ARM/iOS. MoCFIDavi et al. (2012) extracts the CFG from binaries, and utilizes the trampolines to perform checking in runtime. Compared to MoCFI, CFR Pewny and Holz (2013) is a compiler-based solution that maybe overcome the limitation of iOS. Without the requirement of compiler support and debug information, binCFI Zhang and Sekar (2013) applies CFI to the binaries on x86/Linux. MIP Pewny and Holz (2012) is a coarse-grained CFI that restricts the target of indirect jump instruction to the variable length trunk.

In our opinion, though the above CFI enforcements greatly protect intended control flow, they fail to detect hidden control flow because they only monitor
intended control flow instructions. Moreover these CFI enforcements can not fully defeat control flow hijacking because it is impossible to identify all legal targets of dynamic control transfers. Some attackers may circumvent these solutions. It means the existence of hidden control flow. Our solution is considered as the complementary to the above approaches.

6.2 Detecting Hidden Control flow

CFIMon Zhang et al. (2013) detects the violation of control flow integrity with the support of BTS Intel (2009). With the data provided by BTS, CFIMon checks the targets of the recorded control transfers. Since BTS records all issued branch instructions, CFIMon can expose hidden control flow with an improvement because it fail to discrete between intended instructions and unintended ones. Moreover, CFIMon still checks the targets of each control transfer, and incurs the mentioned uncomplete coverage of dynamic control transfer. BeCFI avoids the problem by counting the taken branch instructions without checking the targets.

Utilizing LBR, ROPecker Cheng et al. (2014) is capable of detecting hidden control flow introduced by ROP attacks. Due to the less capability of LBR, ROPecker presents a slider window mechanism for the runtime checking. In one checking, only several code pages are granted executing privilege. In our opinion, LBR can not be used to check a large body of instruction, and PMC is a promising hardware feature for the similar goal. So we propose HDROP Zhou et al. (2014) which is the prior work of this paper. HDROP detects ROP attacks on the abnormal increase of ret misprediction with the support of PMC.

In our opinion, hidden control flow may circumvent above intended CFI enforcements, but they can be caught by the processors since any instruction has to be issued to the processors. On the observation, some solutions are proposed with the support of processors. In this paper, we present a novel solution detecting hidden control flow with the support of PMC.

7 Conclusion

In this paper, we present a novel solution to detect hidden control flow. Most of existing CFI efforts only keep intended control flow in good integrity. These work greatly limit the attacker’s capability to subvert computer system. However, due to the dynamic nature of indirect jump, these approaches can not completely defeat intended control flow. Therefore, it is possible to produce hidden control flow by circumventing these work. With the support of PMC, BeCFI detects hidden control flow by counting taken branch instructions on cross-view principal. We have developed a prototype on Intel x86/Linux to test its effectiveness and performance overhead. The evaluations indicate it is capable of detecting hidden control flow and performance overhead is acceptable.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. The work is supported in part by the National Natural Science Foundation of China (61303074, 61472429, 61070192, 91018008, 61170240), Natural Science Foundation of Beijing (4122041) and National 863 High-Tech Research Development Program of China (2007AA01Z414).

References


Xia, Y., Liu, Y., Chen, H. and Zang, B. 'CFIMon: Detecting Violation of Control Flow Integrity using Performance Counters', Dependable Systems and Networks (DSN), 2012.


