논문 2018-13-26

제어흐름 에러 탐지를 위한 분리형 시그니처 모니터링 기법

(Separate Signature Monitoring for Control Flow Error Detection)

최 기 호, 박 대 진*, 조 정 훈*
(Kiho Choi, Daejin Park, Jeonghun Cho)

Abstract : Control flow errors are caused by the vulnerability of memory and result in system failure. Signature-based control flow monitoring is a representative method for alleviating the problem. The method commonly consists of two routines; one routine is signature update and the other is signature verification. However, in the existing signature-based control flow monitoring, monitoring target application is tightly combined with the monitoring code, and the operation of monitoring in a single thread is the basic model. This makes the signature-based monitoring method difficult to expect performance improvement that can be taken in multi-thread and multi-core environments. In this paper, we propose a new signature-based control flow monitoring model that separates signature update and signature verification in thread level. The signature update is combined with application thread and signature verification runs on a separate monitor thread. In the proposed model, the application thread and the monitor thread are separated from each other, so that we can expect a performance improvement that can be taken in a multi-core and multi-thread environment.

Keywords : Separate signature-based control flow monitoring, Control flow error detection

Ⅰ. 서 론

제어흐름 에러는 메모리의 취약점으로 인해 발생할 수 있다. 전자기파의 간섭, 물리적 충격, 급격한 온도변화는 메모리상의 비트플립 현상을 일으킬 수 있다. 이러한 비트 플립 현상은 특히, 제어흐름 에러를 발생시킬 수 있고 이는 치명적인 시스템 실패로 이어진다. 제어흐름 에러로 인한 시스템 실패를 소프트웨어적으로 탐지하여 이를 경감시키기 위한 대표적인 제어흐름 모니터링 기법으로, 시그니처 기반 제어흐름 모니터링 기법이 있다.

*Corresponding Authors (boltanut@knu.ac.kr, jcho@knu.ac.kr)
Received: Aug. 15. 2018, Revised: Oct. 1. 2018, Accepted: Oct. 2 2018.
K. Choi, J. Cho, D. Park: School of EE, Kyungpook National University
※ 본 논문은 교육부의 재원으로 한국연구재단의 기초과학연구 프로그램의 지원을 받아 수행된 연구결과임 (No. 2014R1A6A3A04059410, 2016R1D1A1B03934343).

ⓒ IEMEK J. Embed. Sys. Appl. 2018 Oct. 13(5) 225-234
ISSN : 1975-5066
http://dx.doi.org/10.14372/IEMEK.2018.13.5.225

제어흐름이란 소프트웨어가 가지고 있는 결정되어 있는 동작 순서를 의미하고, 제어흐름 에러란 제어흐름을 벗어난 동작 상태를 의미한다. 제어흐름은 그림 1과 같이 단방향 그래프로 나타낼 수 있다. 그래프의 각 노드v_n는 제어흐름의 베이직 블록을 의미하고, 에지 집합e_n는 노드v_n로부터 시작하는 베이직 블록 간 연결 집합을 의미한다. 노드v_p가 노드v_q로 연결되어 있다면, 이를 e_p와 v_q에 대해 $v_q \in e_p$ 관계로 나타낼 수 있고, v_p와 v_q는 $v_p = prior(v_q) = v_{q'}$로 나타낼 수 있다. 이 경우, 제어흐름 에러는 현재 노드 v_x가 $v_x \notin e_{x'}$인 상태를 의미한다.

시그니처 기반 제어흐름 모니터링 기법은 '시그니처'라 불리는 시그니처 변수를 통해 제어흐름을 나타내고, 제어흐름 에러를 검출하는 기법이다. 이 기법은 '시그니처 업데이트'와 '시그니처 검증'이라는 두 가지 루틴이 베이직 블록 단위로 번갈아 가며 진행된다. 시그니처 업데이트를 통해 현재의 위상 (현재 어느 베이직 블록에 있는지)에서 다음 위상으로 이동하게 되고, 시그니처 검증을 통해 이동된 위상이 제어흐름에서 벗어나 있지 않은지 검증

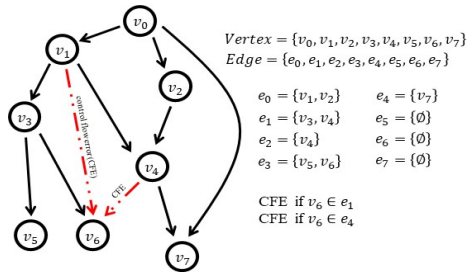

그림 1. 제어흐름 및 제어흐름 에러
Fig. 1 Control flow and control flow error

하게 된다. 즉, 시그니처 기반 제어흐름 모니터링 기법은 시그니처를 통해, 시그니처 업데이트와 시그니처 검증이 각 베이직 블록마다 연속적으로 진행되며 제어흐름 모니터링이 이루어지는 모델을 기본으로 하고 있다.

본 논문에서는 기존에 제시된 시그니처 기반 모니터링 기법을 바탕으로 하되, 시그니처 업데이트와 시그니처 검증을 스레드 레벨에서 서로 분리하여 제어흐름을 모니터링 하는 새로운 분리형 모델을 제시하고자 한다. 시그니처 업데이트는 모니터링 대상 어플리케이션 스레드에서, 시그니처 검증은 별도의 모니터 스레드에서 동작하도록 하는 분리형 시그니처 기반 제어흐름 모니터링 기법을 통해, 멀티스레드 및 멀티코어 환경에서 제어흐름 모니터링으로 인한 오버헤드를 줄일 수 있도록 한다. 또한 제어흐름 모니터링 범위를 함수 내 제어흐름에서 함수 간 제어흐름으로 그 범위를 확장하여 제어흐름 에러 검출률을 향상시킬 수 있도록 한다.

본 논문의 구성은 다음과 같다. Ⅱ장에서는 관련 연구에 대해 살펴보고, Ⅲ장에서는 제안하는 분리형 시그니처 기반 제어흐름 모니터링 기법에 대해 살펴본다. Ⅳ장에서는 가상의 모니터링 시나리오를 통해 제안하는 기법의 타당성을 살펴보고, Ⅴ장에서는 제안하는 분리형 시그니처 기반 제어흐름 모니터링 기법을 자동으로 적용해주는 프레임워크를 소개한다. Ⅵ장에서는 실험을 통해 제안하는 기법의 타당성을 입증하며, 마지막으로 Ⅶ장에서 결론을 맺는다.

Ⅱ. 관련 연구

1. 관련 연구

제어흐름 에러로 인한 시스템 실패를 경감시키기 위해 많은 제어흐름 에러 탐지 기법들이 소개되어 되어왔고, 시그니처 기반 제어흐름 모니터링은

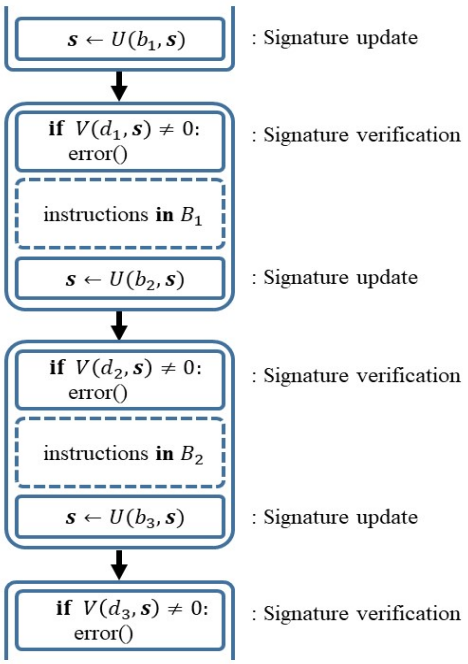

그림 2. 기존 시그니처 기반 제어흐름 모니터링의 기본 모델
Fig. 2 Existing signature monitoring model for control flow error detection

그 중 하나의 대표적인 소프트웨어 기반 모니터링 기법이다. 그림 2는 기존의 시그니처 기반 제어흐름 모니터링 기법의 기본 모델을 나타낸다. 시그니처 기반 제어흐름 모니터링은 시그니처 업데이트 U와 시그니처 검증 V, 두 개의 루틴으로 구성된다.

B_n는 n번째 베이직 블록을 나타내고, b_n는 컴파일 타임에 결정되는 각 베이직 블록의 시그니처 변수를 의미한다. s는 런타임 상에서 결정되는 시그니처 변수이며, 런타임 상에서 각 B_n에서 b_{n+1}과 일련의 연산을 통해 업데이트 $U(b_n, s)$ 된다. d_n은 컴파일 타임에 $d_n = U(b_n, s)$으로 결정되는 시그니처 변수이며, 시그니처 검증에 사용된다.

업데이트 된 런타임 시그니처s 는 다음 베이직 블록 B_{n+1}에서 시그니처 검증과정 $V(s, d_n)$을 거치고 $V(s, d_n) \neq 0$ 을 만족하는 경우, 제어흐름 에러 상태로 판단한다. 간단한 시그니처 기반 제어흐름 모니터링 모델로 시그니처 업데이트와 시그니처 검증 루틴을 $U(b_n, s) = b_n$, $V(s, d_n) = s - d_n = s - b_n$로 선택할 수 있다.

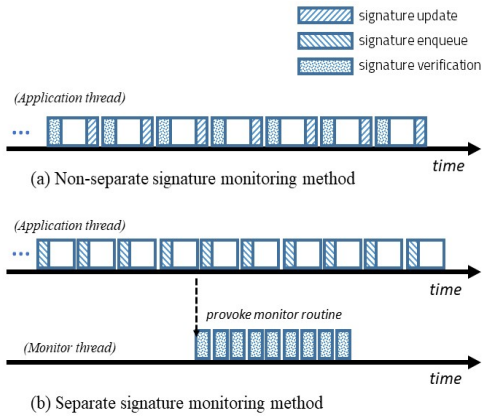

그림 3. 기존의 비분리형 및 분리형 시그니처 모니터링 기법

Fig. 3 Non-separate and separate signature monitoring method

기존의 많은 시그니처 기반 제어흐름 모니터링 연구들은 이러한 시그니처 업데이트 U와 시그니처 검증 V을 중점으로 하여 진행되었다. CFCSS [1], YACCA [2], ECCA [3], RSCFC [4], SEDSR [5], SCFC [6], RASM [7] 등 다수의 시그니처 기반 제어흐름 모니터링 기법들에서 제어흐름 에러 검출률은 증가시키고 모니터링 오버헤드는 낮추기 위해, 사용되는 런타임 시그니처 및 컴파일타임 시그니처의 수, 시그니처 업데이트/검증 루틴의 베이직 블록 내 삽입 위치 및 횟수, 시그니처 업데이트 및 시그니처 검증 알고리즘 등이 제안되어 왔다.

2. 동기 및 주안점

멀티 스레딩 기법 [8]과 멀티코어를 통한 병렬화 기법 [9, 10]은 시스템의 전반적인 성능 향상을 이끌어 내었다. 멀티 스레딩 기법은 주어진 처리기에 대해 최대한의 리소스를 사용할 수 있도록 해주며, 멀티코어를 통한 병렬화 기법은 동작하는 처리기의 수를 증가시켜 병렬처리를 가능하게 하였고, 이는 시스템 성능 향상을 이끌어 내었다.

본 논문에서는 멀티스레딩 기법과 멀티코어를 통한 병렬화 기법이 시스템의 성능 향상에 기여할 수 있다는 것에 착안하여, 기존의 시그니처 기반 제어흐름 모니터링 기법을 바탕으로 하되 시그니처 기반 모니터링 기법이 멀티 스레드 및 멀티코어 환경에 적용될 수 있도록 시그니처 업데이트와 시그니처 검증 과정을 분리하는 새로운 분리형 모델을 제시하고자 한다. 제안하는 분리형 시그니처 기반 제어흐름 모니터링 기법과 기존의 시그니처 기반 제어흐름 모니터링 기법의 구별과 용어의 편의를 위해 이하 제안하는 기법을 '분리형 시그니처 모니터링 기법 (separate signature monitoring method)', 기존의 모니터링 기법을 '비분리형 시그니처 모니터링 기법 (non-separate signature monitoring method)'이라 칭하도록 하겠다.

본 논문은 다음을 중점으로 하여 기술 되었다. 첫째, 기존의 시그니처 기반 제어흐름 모니터링 기법을 확장하여 멀티 스레드 환경에서 성능향상을 꾀하는 분리형 시그니처 기반 제어흐름 모니터링 기법을 제안하고, 제안하는 기법의 타당성을 보이고 있다. 둘째, 본 논문에서 제안하는 분리형 모니터링을 적용할 수 있는 분리형 시그니처 기반 모니터링 자동생성 프레임워크를 구현하여 제시하였다.

Ⅲ. 분리형 시그니처 기반 제어흐름 모니터링 기법

1. 시그니처 업데이트/검증 루틴의 분리

시그니처 기반 모니터링 기법을 멀티스레드 및 멀티코어 환경에 적용하기 위해선 기존의 시그니처 기반 모니터링 기법을 스레드 레벨에서 나눌 필요가 있다. 본 논문에서는 시그니처 업데이트와 시그니처 검증과정을 선택하였다.

그림 3 (a)와 그림 3 (b)는 기존의 비분리형 시그니처 모니터링 기법과 본 논문에서 제안하는 분리형 시그니처 모니터링 기법을 나타낸다. 비분리형 시그니처 모니터링 기법에서는 시그니처 업데이트와 검증 루틴이 각 베이직 블록 단위에서 순차적으로 이루어지지만, 제안하는 분리형 시그니처 모니터링 기법에서는 두 루틴이 스레드 레벨에서 서로 분리되어 동작한다. 시그니처 업데이트 (시그니처 인큐)는 모니터 대상 어플리케이션 스레드에서 이루어지고, 시그니처 검증은 별도로 생성된 시그니처 검증 스레드에서 이루어진다. 이하 시그니처 검증을 위한 별도의 스레드를 '모니터 스레드'라 칭하도록 하겠다.

2. 시그니처 큐

기존의 비분리형 시그니처 모니터링 기법에서는 하나 또는 두세 개의 시그니처를 사용하여 시그니처 업데이트와 시그니처 검증을 진행한다. 이 때, 각 베이직 블록의 컴파일타임 시그니처는 각 베이직 블록별로 고유한 값을 지닌 변수이지만, 런타임

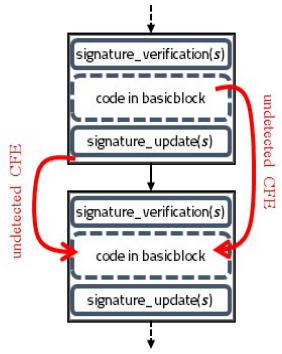

그림 4. 런타임 시그니처 공유로 인해 탐지되지 않는 제어흐름 에러
Fig. 4 Undetected control flow error caused by the shared runtime signature

시그니처는 각 베이직 블록에서 공유되며 업데이트 되는 변수이다.

제안하는 분리형 시그니처 모니터링 기법에서는 시그니처 검증이 모니터 스레드에서 일괄적으로 이루어지도록 '시그니처 큐'의 개념을 도입하였다. 시그니처 큐는 다음에서 설명할 시그니처 인큐(Signature Enqueue) 루틴을 통해 각 베이직 블록의 컴파일타임 시그니처로 채워진다. 시그니처 큐는 각 베이직 블록에서 업데이트 되지만 베이직 블록 간 공유되지 않으므로, 비공유성을 지닌 런타임 시그니처라 볼 수 있다. 이러한 런타임 시그니처의 비공유 성질은 기존의 시그니처 모니터링 기법에서 런타임 시그니처 공유로 인해 발생하는 제어흐름 에러 검출률 저하 문제를 해결하는 부가적인 효과로 이어질 수 있다.

그림 4는 기존의 비분리형 시그니처 모니터링 기법에서 발생할 수 있는 제어흐름 에러 미검출 예를 나타낸다. 런타임 시그니처 변수s를 각 베이직 블록에서 공유하므로, 그림 4와 같은 제어흐름 에러가 발생하는 경우 제어흐름 에러 미검출로 이어진다. 이는 런타임 시그니처가 각 베이직 블록에서 공유되므로 덮어써지기 때문이다. RASM [7]에서는 두 단계 시그니처 업데이트를 각 베이직에 두어 해당 문제를 경감시키고 있지만 이는 런타임 시그니처 공유로 인한 근본적인 문제를 해결할 수 없다.

3. 시그니처 인큐 (Signature Enqueue)

시그니처 인큐는 기존의 시그니처 업데이트와 유사한 역할을 수행한다. 다만, 기존의 시그니처 업데이트가 다음 베이직 블록에서의 시그니처 검증을 위해 런타임 시그니처를 업데이트 하였다면, 시그니처 인큐의 경우에는 각 베이직 블록의 컴파일 타임 시그니처를 시그니처 큐에 넣어준다. 그림 5는 시그니처 인큐 (Signature Enqueue) 알고리즘을 나타낸다. 시그니처 큐가 가득 찰 때까지 시그니처 큐에 컴파일 타임 시그니처를 넣으며, 시그니처 큐가 가득찬 경우 모니터 스레드를 생성하고, 모니터 루틴(Monitor Routine)을 동작시킨다.

SignatureEnqueue($Q_s, \widehat{Q_s}, \widehat{Q_s^{max}}, G$)

Q_s is signature queue
$\widehat{Q_s}$ is the current size of Q_s
$\widehat{Q_s^{max}}$ is the maximum size of Q_s
s is the input signature

$Q_s[\widehat{Q_s}] \leftarrow s$
if $\widehat{Q_s} = \widehat{Q_s^{max}}$ **do**
    generate_thread(MonitorRoutine, Q_s)
    $\widehat{Q_s} \leftarrow 0$
    $Q_s \leftarrow generateNew(\widehat{Q_s^{max}})$
else do
    $\widehat{Q_s} \leftarrow \widehat{Q_s} + 1$
end if

그림 5. 시그니처 인큐
Fig. 5 Signature enqueue

4. 모니터 루틴 (Monitor Routine)

어플리케이션 스레드에서 시그니처 큐가 가득찬 경우, 모니터 스레드를 생성하고 모니터 루틴(Monitor Routine)을 동작시킨다. 그림 6은 모니터 루틴 알고리즘을 나타낸다. 시그니처 큐 Q_s가 빌 때까지 시그니처 검증 루틴 (Signature Verification)을 수행한다.

5. 시그니처 검증 (Signature Verification)

그림 7은 시그니처 검증 과정을 나타낸다. 시그니처 검증은 시그니처 큐에 들어 있는 런타임 시그니처와 모니터링 대상 어플리케이션 분석을 통해 얻은 함수 간 제어흐름 그래프간 비교를 통해 이루어진다. 시그니처 큐의 런타임 시그니처가 함수 간 제어흐름 그래프를 벗어난 경우, 제어흐름 에러를 검출한다.

6. 함수 간 제어흐름 그래프

본 논문에서는 제어흐름의 모니터링 범위를 함수 내의 제어흐름 뿐만 아니라, 함수 간 제어흐름까지 포함하고 있다. 기존의 시그니처 기반 제어흐름

MonitorRoutine($\widehat{Q_S}, Q_S$, G, $\dot{v}_s$)

Q_s is signature queue
$\widehat{Q_s}$ is a current size of Q_S
G is the inter-procedural CFG
v_s *is the node represented by signature* s *in* G
$\dot{v}_s$ is the current traversing node in G

while $\widehat{Q_S} = 0$ **do**
    SignatureVerification($\widehat{Q_S}, Q_S$, G, $\dot{v}_s$)
end while

그림 6. 모니터 루틴
Fig. 6 Monitor routine

모니터링 기법들은 대부분 함수 내 제어흐름을 모니터링 범위로 하고 있다 [1-7]. 따라서 함수 호출을 가진 베이직 블록 또는 함수 리턴을 가진 베이직 블록 내에서 제어흐름 에러가 발생 할 경우, 제어흐름 검출에 어려움이 있고 이는 제어흐름 검출률 저하의 요인이 될 수 있다.

그림 8은 본 논문에서 다루는 함수 간 제어흐름 그래프 (inter-procedural control flow graph)의 한 예이며, 두 함수 f_1과 f_2의 제어흐름 그래프가 연결되었다. 함수 간 제어흐름 그래프 생성은 다음의 루틴을 따른다. 1) 각 함수의 제어흐름 그래프의 노드는 베이직 블록을 의미한다. 2) 각 노드는 Branch, Call, Entry, Return 타입으로 나뉜다. 3) Call 타입 노드에는 하나의 함수 호출만이 존재한다. 즉, 베이직 블록은 함수호출 단위로 분리된다. 4) Call 타입 노드는 호출되는 함수의 Entry 타입 노드와 연결된다. 5) Return 타입 노드는 함수간 호출 그래프에서 따로 연결하지 않는다.

함수 호출의 경우에는 연결되는 베이직 블록이 현재의 위상에 따라 하나로 유일하게 결정되지만, 함수 리턴의 경우 복수개의 함수에서 해당 함수를 호출 할 수 있으므로, 이를 모두 제어흐름 그래프에 포함시키면 제어 흐름 에러 검출률 저하를 일으킬 수 있다. 따라서 본 논문에서는 함수 리턴 시 해당 시점에서의 정확한 제어흐름 검증을 위해 함수 리턴 스택을 도입하였다.

7. 함수 리턴 스택

그림 9는 함수 간 제어 흐름 그래프에서 함수 리턴 스택을 이용하여 함수 호출 및 리턴을 처리하

SignatureVerification($\widehat{Q_S}, Q_S$, G, $\dot{v}_s$)

Q_s is the signature queue
$\widehat{Q_s}$ is the current size of Q_S
$\dot{s}$ is a current signature dequeued from Q_S
G is the inter-procedural CFG
v_s *is the node represented by signature* s *in* G
$\dot{v}_s$ is the current traversing node in G
$\dot{v}_s'$ is the parent node of $\dot{v}_s$
S_{NEXT} is the set of signatures that represent the nodes which are directed by $\dot{v}_s'$

$\hat{s} \leftarrow$ dequeue(Q_S)
$S_{NEXT} \leftarrow$ getNextNode(G, $\dot{v}_s$)
if not $\dot{s} \in S_{NEXT}$ **do**
    error()
end if
$\widehat{Q_S} \leftarrow \widehat{Q_S} - 1$

그림 7. 시그니처 검증
Fig. 7 Signature verification

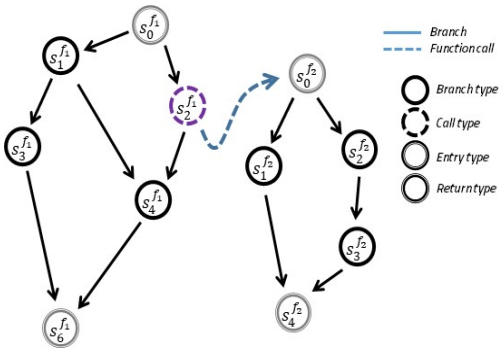

그림 8. 함수 간 제어 흐름 그래프
Fig. 8 Inter-procedural control flow graph

는 것을 보여준다. 함수 리턴 스택은 함수 호출 시, 현재 베이직 블록의 다음 베이직 블록을 푸시(push)하고, 함수 리턴 시 팝 (pop)을 수행한다.

8. 모니터 초기화 루틴

모니터 초기화 루틴 (Monitor Init)은 어플리케이션의 가장 첫 번째 베이직 블록의 가장 앞쪽에 삽입되며, 어플리케이션의 제어흐름 분석정보를 이용하여 함수 간 제어흐름 그래프를 생성한다. 이 때 생성된 함수 간 제어흐름 그래프는 앞서 서술된 것과 같이 모니터 스레드에서 시그니처 검증 루틴에서 사용된다.

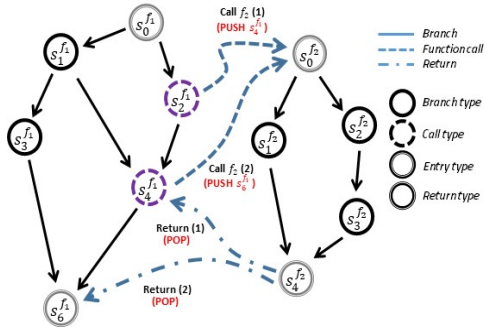

그림 9. 함수 리턴 스택
Fig. 9 Function return stack

Ⅳ. 분리형 시그니처 기반 제어흐름 모니터링 시나리오

본 장에서는 제안하는 분리형 시그니처 기반 제어 흐름 에러 검증 기법이 기존의 시그니처 기반 제어 흐름 에러 검증 기법과 비교하여, 어떠한 성능적 이점이 있는지 네 가지 가상 환경에서 동작 시나리오를 통해 직관적으로 보여주고자 한다.

시나리오 환경은 멀티코어의 수C, 시그니처 큐의 크기Q, 어플리케이션의 blocking I/O의 유무I로 결정된다.

1. 멀티코어 환경에서 오버헤드 감소

그림 3 (a)와 그림 10 (a)의 비분리형 모니터링 기법과 분리형 모니터링 기법을 비교해보면, 분리형 모니터링 기법이 비분리형 모니터링 기법에 비해 더 적은 실행시간 오버헤드를 가질 수 있다는 것을 직관적으로 확인할 수 있다. 모니터링 대상 어플리케이션 스레드에서 시그니처 검증 루틴을 각 베이직 블록 마다 가질 필요가 없으며, 검증 루틴은 모니터 스레드에서 분리되어 실행되기 때문이다. 물론, 이 경우는 코어의 수C가 $C \geq 2$인 경우로 멀티코어 환경에서 가질 수 있는 이득을 의미한다. 그러나 기존의 비분리형 모니터링 기법에서는 멀티코어 환경에서 이러한 성능향상을 가질 수 없다.

2. 블로킹 I/O 및 멀티코어 환경에서 오버헤드 감소

그림 10 (b)는 블로킹 I/O를 가지는 어플리케이션에 분리형 모니터링 기법이 적용된 경우를 나타낸다. 비분리형 모니터링 기법이 적용될 경우 어플리케이션이 블록 되어 있는 동안 제어흐름 모니터링 또한 블록 되지만, 본 논문에서 제안하는 분리형 모니터링 기법이 적용된 경우 어플리케이션이 블록 상태에 있더라도 모니터 스레드가 동작할 수 있다.

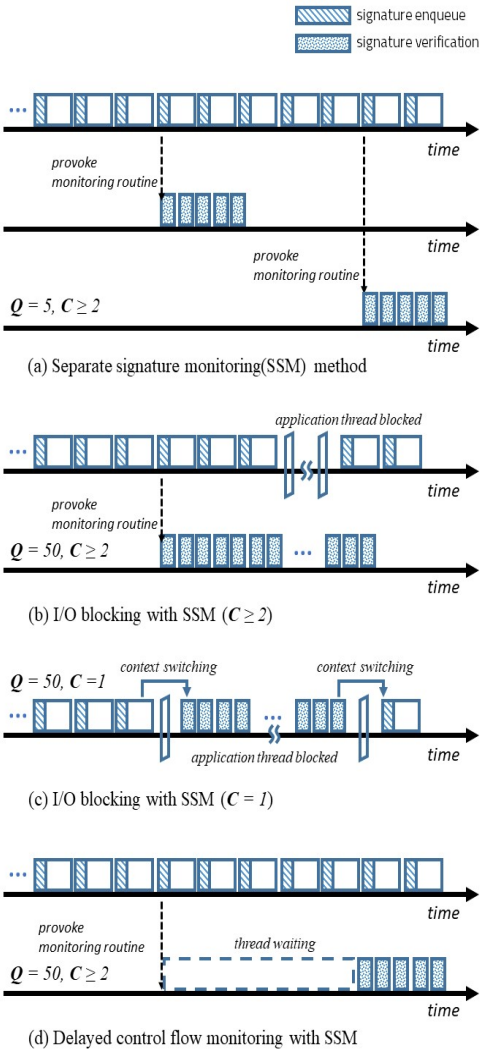

그림 10. 분리형 모니터링 시나리오
Fig. 10 Separate monitoring scenario

3. 블로킹 I/O 및 멀티쓰레딩 기법을 통한 오버헤드 감소

그림 10 (c)는 단일코어 또는 점유가능한 코어의 수가 하나뿐인 환경에서 블로킹 I/O를 가지는 어플리케이션에 분리형 모니터링 기법이 적용된 경우를 나타낸다. I/O로 인해 어플리케이션 스레드가 블록 되더라도 모니터 스레드로 컨텍스트 스위칭 되어 모니터 스레드가 동작할 수 있다.

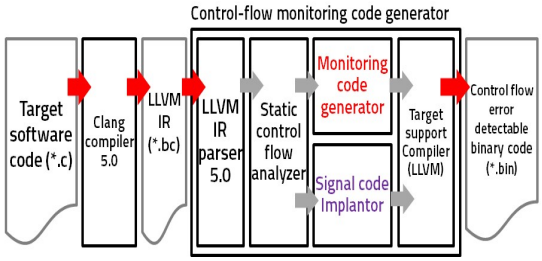

그림 11. 분리형 시그니처 기반 모니터링을 위한 코드생성 프레임워크의 전반적인 구조

Fig. 11 Overall structure of code generation framework for SSM

4. 지연된 모니터링

그림 10 (d)는 제안하는 비분리형 모니터링 기법이 적용된 경우에 지연된 모니터 스레드 동작을 나타낸다. 멀티코어 환경에서 여러 스레드들이 동작할 경우 어떠한 시점에는 많은 스레드들이 동작하여 코어가 경쟁적으로 점유될 수 있고, 어떤 시점에서는 코어의 점유율이 낮을 수 있다. 분리형 모니터링 기법에서는 모니터 스레드의 우선순위를 낮추어 코어의 점유율이 낮을 때 모니터링이 이루어지도록 할 수 있다.

Ⅴ. 분리형 시그니처 기반 모니터링 기법을 위한 모니터링 코드 자동생성 프레임 워크

본 장에서는 제안하는 분리형 시그니처 모니터링 기법을 자동으로 적용시켜주기 위한 분리형 시그니처 기반 모니터링을 위한 코드생성 프레임워크의 전반적인 구조에 대해 설명하고 및 각 모듈이 어떻게 동작하는지 알아보고자 한다.

그림 11은 분리형 시그니처 기반 모니터링을 위한 프레임워크의 전반적인 구조를 나타낸다. 제안하는 프레임워크는 모니터링 대상 어플리케이션의 소스코드를 입력으로 받고, 분리형 시그니처 기반 모니터링이 적용된 어플리케이션을 출력으로 가진다. 전반적인 구조는 Clang 및 LLVM IR 파서, 제어흐름 정적 분석기 (control flow static analyzer), 모니터링 코드 생성기 (monitoring code generator), 시그니처 코드 삽입기 (signature code implanter), 타겟 바이너리 코드 생성기 (target support compiler)로 구성되어 있다.

표 1. 벤치마크

Table 1. Benchmarks

No.	name	TNF	TNBB	TNFC	NIPBB	TENB (k)
1	WS	4	77	20	8.08	1,103
2	DS	12	91	28	6.66	84
3	FFT	7	107	36	6.69	4,669
4	DJ	6	59	16	5.50	14,447
5	PT	6	212	68	5.40	348
6	SHA	8	66	23	9.46	2,558
7	BM	5	110	35	5.98	11,962
8	SS	10	166	22	5.10	10,837
9	CRC	4	29	11	5.89	41,066

TNF: total number of functions
TNBB: total number of basic blocks
TNFC: total number of function call
NIPBB: number of instructions per basic block
TENB: total number of executed basic blocks

1. Clang 및 LLVM 파서

본 논문에서는 모니터링 대상 어플리케이션에 소스 코드에 대한 베이직 블록 레벨의 분석을 위해 LLVM Intermediate Representation (IR)을 사용하였다. Clang은 LLVM infrastructure [11]에서 front-end를 담당하며, C/C++ 코드를 중간코드인 LLVM IR로 변환한다. LLVM 파서는 베이직 블록 레벨 분석이 용이하도록 데이터 구조를 제공해 준다. 본 논문에서 제안하는 프레임워크의 구현에는 LLVM 5.0이 사용되었다.

2. 제어흐름 정적 분석기

제어흐름 정적 분석기는 파싱된 LLVM IR 코드를 분석하여, 모니터링 코드 생성기와 시그니처 코드 삽입기에서 사용되는 함수 콜 그래프, 각 함수별 제어흐름 그래프, 함수 간 제어흐름 그래프 및 컴파일타임 시그니처와 같은 메타 데이터를 생성한다.

3. 모니터링 코드 생성기

모니터링 코드란 모니터 스레드에서 시그니처 검증 루틴을 수행하는 코드를 의미한다. 모니터링 코드의 역할은 두 가지이다. 하나는 함수 간 콜 그래프를 초기에 생성하는 것이고, 다른 하나는 모니터 스레드에서 함수 간 제어흐름 그래프와 시그니처 큐를 비교하여 시그니처 검증 루틴을 수행하는 것이다.

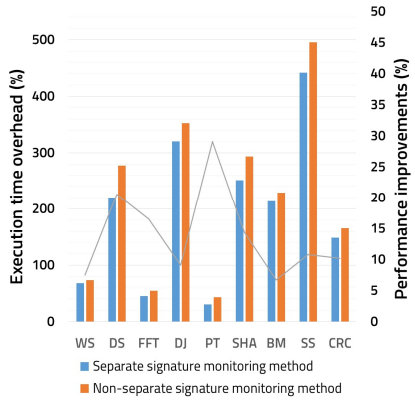

그림 12. 분리형/비분리형 시그니처 기반 모니터링 실행시간 오버헤드 비교
Fig. 12 Execution time overhead of SSM and Non-SSM

4. 시그니처 코드 삽입기

시그니처 코드 삽입기는 모니터링 대상인 어플리케이션 스레드에서 동작하는 두 종류의 시그니처 코드를 삽입한다. 하나는 어플리케이션의 최초 베이직 블록의 가장 처음 명령어 앞에 삽입되는 모니터 초기화 루틴 (Monitor Init)이고, 다른 하나는 그림 3 (b)의 분리형 모니터링 기법에서 나타낸 것과 같이 각 베이직 블록마다 베이직 블록의 가장 처음 명령어 앞에 삽입되는 시그니처 인큐 루틴 (Signature Enqueue)이다.

5. 타겟 바이너리 코드 생성기

타겟 바이너리 코드 생성기는 어플리케이션이 동작할 시스템의 아키텍처에 맞게 바이너리 코드를 생성한다. 본 모듈은 LLVM llc 모듈로 구성되었다.

Ⅵ. 성능 평가

본 장에서는 제안하는 분리형 시그니처 모니터링 기법이 기존의 비분리형 시그니처 모니터링 기법에 비해 멀티 스레드 및 멀티 코어 환경에서 가질 수 있는 실행시간 오버헤드 성능향상을 실험을 통해 타당성을 제시하고자 한다.

1. 성능평가 환경 및 벤치마크

제안하는 분리형 모니터링 기법의 타당성을 검증하기 위한 성능평가는 Ubuntu 16.04.5 Linux, x86-64 architecture, 4cores, 4GB 환경에서 진행되었다. 표 1은 성능평가에 사용된 벤치마크 및 각 벤치마크의 특성에 대해 나타내고 있다. 성능평가를 위한 벤치마크로는 whetstone (WS) 1.1 [12], drystone (DS) 2.1 [13]과 Mibench 1.1 [14]에서 제공하는 Fast Fourier Transform (FFT), DiJkstra (DJ), Secure Hash Algorithm (SHA), BasicMath (BM), StringSearch (SS), Cyclic Redundancy Check (CRC)를 선택하였다. 표 1에서 각 벤치마크에 대한 총 함수 개수, 베이직 블록 개수, 총 함수 호출 개수, 각 베이직 블록 당 명령어 개수 및 런타임 상에서 실행 된 베이직 블록의 수를 나타내고 있다. 일반적으로 실행시간 오버헤드는 각 베이직 블록 당 명령어 개수가 많을수록 (NIPBB), 실행시간에 대비 런타임 상에서 실행 된 베이직 블록의 수 (TENB)가 적을수록 적다.

2. 성능평가 방법

제안하는 분리형 모니터링 기법과 비교대상으로, 기존의 비분리형 시그니처 기반 모니터링 모델을 기반으로 하되 $U(b_n, s) = b_n, V(s, d_n) = s - b_n$ 로 선택하는 매우 단순한 모델을 구성하였다. 해당 비교 대상은 SEDSR [5]와 의미적으로 동일하다고 할 수 있다. 해당 비교 대상 기법과 제안하는 분리형 모니터링 기법의 실행시간 오버헤드를 비교하여, 제안하는 기법이 멀티 코어 및 멀티 스레드 환경에서 가지는 성능향상을 확인하도록 한다.

3. 성능평가 결과

그림 12는 본 논문에서 제안하는 분리형 시그니처 기반 모니터링 기법과 기존의 비분리형 시그니처 기반 모니터링 기법을 각 벤치마크에 적용한 경우 발생하는 실행시간 오버헤드 및 기존 기법 대비 제안 기법의 실행시간 오버헤드 감소율을 나타낸다. 제안하는 분리형 시그니처 기반 모니터링 기법이 비분리형 시그니처 기반 모니터링 기법 대비 최대 29.1%, 평균 13.89% 실행시간 오버헤드 감소를 확인할 수 있었다. PT의 경우 기존 기법 대비 최대 오버헤드 감소율인 29.1%을 보이고 있고, 각 벤치마크 WS, DS, FFT, DJ, SHA, BM, SS, CRC에 대해 7.53%, 20.58%, 16.52%, 9.12%, 14.41%, 6.72%, 10.83%, 10.20%를 나타내었다.

4. 성능평가 결과 분석

상기의 성능평가에서 평균적으로 제안하는 분리형 시그니처 기반 모니터링 기법이 기존의 기법대

비 약 10%대 정도의 실행시간 오버헤드 감소율의 성능향상을 보이고 있다. 이는 기존의 시그니처 기반 모니터링 기법들이 멀티코어 및 멀티스레드 환경에서 어떠한 이득을 얻지 못한 것에 반해, 본 논문에서 제안하는 분리형 시그니처 기반 모니터링 기법은 모니터링에 소요되는 오버헤드를 별도의 모니터 스레드로 넘겨줌으로써 어플리케이션에서 그에 해당하는 만큼 오버헤드를 짊어지지 않기 때문이다.

Ⅶ. 결 론

본 논문에서는 기존의 시그니처 기반 제어흐름 모니티링 기법을 바탕으로 멀티스레드 및 멀티코어 환경에서 성능향상이 이루어질 수 있도록, 시그니처 업데이트 루틴과 시그니처 검증 루틴을 분리하는 분리형 시그니처 기반 제어흐름 모니터링 기법을 제안하고, 제안하는 기법의 타당성을 제시하였다. 또한, 제안하는 분리형 모니터링을 적용할 수 있도록 분리형 시그니처 기반 모니터링 자동생성 프레임워크를 구현하여 제시하였다. 본 논문에서 제안하는 분리형 시그니처 기반 모니터링 기법을 사용할 경우, 기존의 비분리형 시그니처 기반 모니터링 기법 대비 최대 29.1%, 평균 13.89% 실행시간 오버헤드 감소를 확인하였다.

References

[1] N. Oh, P.P. Shirvani, E.J. McCluskey, "Control-flow Checking by Software Signatures," Proceedings of IEEE Transactions on Reliability, Vol. 51, No. 1, pp.111-122, 2002.

[2] O. Goloubeva, M. Rebaudengo, M.S. Reorda, M. Violante, "Softerror Detection Using Control Flow Assertions," Proceedings of 18th IEEE Int. Symp. Defect Fault Tolerance VLSI Syst, pp. 581-588, 2003.

[3] Z. Alkhalifa, V.S. Nair, N. Krishnamurthy, J.A. Abraham, "Design and Evaluation of System-level Checks for On-line Control Flow Error Detection," IEEE Trans. Parallel Distrib. Syst., Vol. 10, No. 6, pp. 627-641, 1999.

[4] A. Li, B. Hong, "Software Implemented Transient Fault Detectionin Space Computer," Aerosp. Sci. Technol., Vol. 11, No. 2, pp. 245 - 252, 2007.

[5] S.A. Asghari, A. Abdi, H. Taheri, H. Pedram, S. Pourmozaffari, "SEDSR: Soft Error Detection Using Software Redundancy," Softw. Eng. Appl., Vol. 5, No. 9, pp. 664-670, 2012.

[6] S.A. Asghari, H. Taheri, H. Pedram, O. Kaynak, "Software-based Control Flow Checking Against Transient Faults in Industrial Environments," Proceedings of IEEE Trans. Ind. Informat., Vol. 10, No. 1, pp. 481-490, 2014.

[7] J. Vankeirsbilck, N. Penneman, H. Hallez, J. Boydens, "Random Additive Signature Monitoring for Control Flow Error Detection," in IEEE Transactions on Reliability, Vol. 66, No. 4, pp.1178-1192, 2017.

[8] G, Blake, R.G. Dreslinski, T. Mudge, "A Survey of Multicore Processors," IEEE Signal Processing Magazine, Vol. 26, No. 6 pp. 26-37, 2009.

[9] D.M. Tullsen, S.J. Eggers, H.M. Levy, " Simultaneous Multithreading: Maximizing On-chip Parallelism," Proceedings of 22nd Annual International Symposium on Computer Architecture, pp. 392-403, 1995.

[10] Y.J. Kong, S.K. Woo, "Real-Time Implementat ion of Doppler Beam Sharpening in a SMP Multi-Core Kernel", IEMEK J. Embed. Sys. Appl, Vol. 11, No. 4, pp. 251-257, 2016 (in Korean).

[11] C. Lattner, V. Adve, "LLVM: a Compilation Framework for Lifelong Program Analysis & Transformation," Proceedings of CGO '04, pp. 75-86, 2004.

[12] R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," Communications of the ACM, Vol. 27, No. 10, pp. 1013-1030, 1984.

[13] H. J. Curnow, B.A. Wichman, "A Synthetic Benchmark," The Computer Journal, Vol. 19, No. 1, pp.43-49, 1976.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: a Free, Commercially Representative Embedded Benchmark Suite," Proceedings of 4th Annual IEEE International Workshop Workload Characterization, pp.3-14, 2001.

Kiho Choi (최 기 호)

He received the B.S. degree in electronics engineering from Kyungpook National University, Daegu, Korea in 2017. He is currently a M.S student in department of electronics engineering at Kyungpook National university, Daegu, Korea. His research interests include embedded software analysis algorithm for software safety and software testing code generations.
Email: posjkh22@gmail.com

Daejin Park (박 대 진)

He received the B.S. degree in electronics engineering from Kyung- pook National Univer- sity, Daegu, Korea in 2001, the M.S. degree and Ph.D. degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 2003, and 2014, respectively. He was a Research Engineer at Major Semicond- uctor Companies such as SK Hynix Semiconductor, Samsung Electronics over 12 years from 2003 to 2014, respectively and have worked on processor architecture design and low-power ASIC implementation with custom designed software algorithm optimization. Dr. Park is now with School of Electronics Engineering as full-time assistant processor in Kyungpook National University, Daegu, Korea and presidential research fellow.
Email: boltanut@knu.ac.kr

Jeonghun Cho (조 정 훈)

He received the B.S. degree in EE, the M.S. and the Ph. D degree in EECS from the Korea Advanced Institute of Science and Technology (KAIST), Deajeon, Korea in 1996, 1998, and 2003, respectively. He was a senior engineer at Hynix Semiconductor from 2003 to 2005. Main role was development of a C compiler for 8-bit microcontrollers. He is currently a professor with the School of EE of Kyungpook National University, Daegu, South Korea since 2005. His research interest includes a binary translation, software safety and security for automotive, AUTOSAR, run-time monitoring and logging for embedded system. Dr. Cho is a member of IEEE.
Email: jcho@knu.ac.kr