Building a runtime state tracing kernel

Ananth Chakravarthy, Vinay G. Vaidya

Symbiosis Deemed University
Senapati Bapat marg,
Pune, India

chakri.ananth@gmail.com
vaidya.vinay@gmail.com

Abstract: A process is run by executing a sequence of instructions by the processor. However it is probable that not all of the instructions are executed as there are hundreds of paths that can be taken by the executable to complete its execution. The path chosen is dependent on a host of factors like the environment, user input, the platform etc. As such, at any given instant of time, the process might be in any of the possible states S^n after traversing states $S^1,S^2,S^3..$ where S^1,S^2,S^3 ....S^n,S^{n+1}, $S^{n+2},..S^M$ depict the total M states that can be taken by the executable. There is no mechanism currently inside the Linux kernel to peek into the state of the process to find out which if these states is it currently in and what are the states it has "traversed" to reach the current state while it is executing. If such an effective tracing can be achieved, it would lead to better operating system security. Other advantages are better logs or even building a verifiable software system. This paper looks at the infrastructure that has been developed to realize such a functionality in the Linux kernel and thereby increase the security of the running process. Of particular mention is the framework that has been developed to peek into the state of a running process as it executes and the various mechanisms that could be used to ascertain the state of the running process.

Tags: Security, State tracing, ELF, Reverse Engineering, Linux, System Calls, dynamorio

1 Introduction

There are a host of mechanisms in which security is compromised in the organizations today. Buffer overflow attacks, viruses, trojans, worms, rootkits, social engineering, spy-ware or hacking tools like key-loggers, password rippers, net-cat kind of utilities are only some of the plethora of options that are available today to break into a secure system. While solutions do exist to counter each of these kind of attacks, most of them do not seem to really counter the attack even before it has actually taken for the first time. For example, an anti-virus cannot detect a new virus until it obtains the signature of the virus. All of the drawbacks for the current security tools point out the need for a security framework using which the working of an executable can be verified against some sort of description of the system or the binary which is getting executed. Such a proactive verification of the working of an executable would go a long way in providing the requisite capability for the operating system to monitor and thwart any malicious attempts before the system is compromised. One of the first and foremost feature of such a security framework is to have the capability to find out what is the process state at any given instant of time. The approach taken by the author to counter these attacks is to provide a description of the binary to the kernel so that while the binary is getting executed, the Linux kernel verifies the state of the running binary. If the state does not match to a valid state as described in the description document, then appropriate logic could be triggered from the kernel. The following paper is an attempt to propose and discuss a mechanism to enable process state capturing and querying while the process is getting executed on the operating system.

2 Problem Description

Problem statement

The problem that the paper is trying to solve is "There are no operating system modules inside the Linux kernel that help in effectively tracing the current state of a running process and the path of its state traversal to its current state".

Enabling such a feature in the operating system would help in taking proactive steps in case of suspicious activity. The first step towards building such a self-protecting operating system model is to make the operating system aware of the various states that the binary can traverse while it is executing. Hence it becomes imperative that the operating system modules possess a mechanism of running an executable on a "state aware" basis.

To build a modified Linux kernel that can provide interfaces about the current state of the process to the external world as well other kernel modules is the subject of this paper.

Before trying to build a state aware Linux kernel, the concept of a "state" needs to be defined. The definition of a state could be as simple as beginning of a function entry to the point of exit of that function. Another approach is to define a state as a sequence of instructions which implement a functionality, for example opening a file and writing something to the opened file. However, the current approach in this paper is to define a "state" as a sequence of machine instructions which do not have a "call" or "jmp" category of instructions.

3 Literature Review

Though no direct references were found in Literature to build a state verification Linux Kernel, there are a large number of references with respect to state tracing. Below are some of the references found with respect to tracing.

Currently applications rely on the logging mechanisms to trace the flow of execution. The developers write code as part of the application log to generate log messages to a predetermined log file. The log file is then later analyzed mostly manually as the messages requires human intelligence to interpret the state flow. Another drawback with this approach is that extensive logging commands need to be written by the developer to trace the full state of the application. Moreover, the application would not be able to detect the changes in the flow of the application state flow. For example, consider the following hypothetical state flow from state S^m to S^n . Suppose the developer has written log messages which ascertain the state traversal for both of these states. If this executable has been infected with a virus which introduces a jump instruction at the end of the state S^m to the malicious code and at the end of the malicious code transfers the control to the state S^n. This may be depicted as follows:

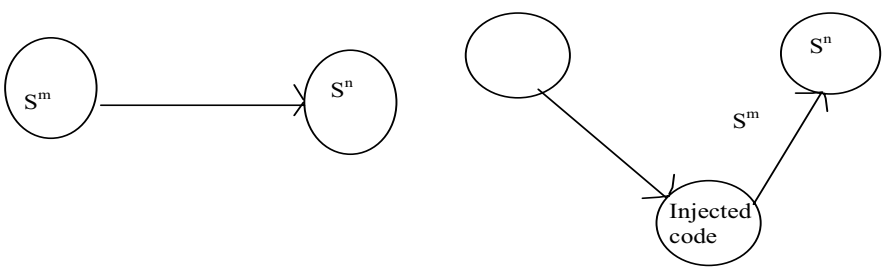


Figure 3.1 showing the normal state flow to the left and the infected state flow to the right

The issue with this approach is that the logs still show what the analyzer is expecting to see and nothing seems to be wrong as per the logs. Another drawback is that the state tracing by logging only works only if we have the source code for the binary for which the analysis is being done.

Linux Trace toolkit (ltt) is an open-source project which involves patching up the standard kernel to introduce changes to keep accounting of various aspects like kernel locks, files open and the network interfaces. Below is a snapshot of the tool.

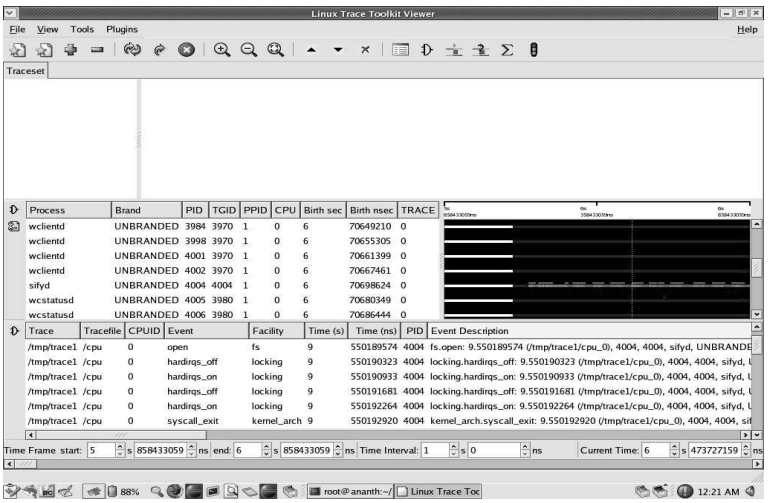


Figure 3.2: snapshot of the LttViewer tool

As the figure depicts, this tool helps in keeping a track of the various resources available in the kernel like the amount spent in the kernel, the number of times the lock was acquired. The downside of this approach is that it ignores the state of the binary in the user space completely. This logic kicks in mostly when the application has issued a system call. However, the tracing mechanism is in the hidden state when the binary is executing in the user space. However, it has a a very novel way of describing facilities, that is declaring new states, that need to be traced. This approach can come in quite handy when we are developing applications from the scratch and want to trace new states as decided by the application developer. The entire module is geared towards logging the state of the kernel and not any particular state identification in the kernel. The other drawback with this approach (for that matter any source code based state tracing approach) is that it still suffers from the redirection problem that is mentioned in Figure 3.1 as given above.

Ptrace approach is another mechanism which is used to trace the binary by making ptrace system calls. This approach involves an attempt by the developer to make ptrace family of system calls which essentially introduces a single step instruction interrupt handler for the binary which can then inject the code to make further calls to read/write the data from particular memory locations. Core dump analysis is another approach to peek into the state of the binary which generated the core dump just before crashing. This approach is mostly useful take a post-mortem approach for analyzing the state of the system after the binary execution crashed and hence cannot be used for effective protection mechanisms.

The approach taken by Yashushi and Sejichi in their paper [10] involves single stepping every instruction in the kernel is not a practical approach. This is be cause single stepping for each and every instruction will prove to be very costly in terms of because it involves having separate hardware which can hold the trace data information. The amounts of data that is generated when this approach is taken is too much to be handled at runtime. It needs to be noted that the amount of time that it takes to analyze and conclude that the binary is in a particular state needs to be completely optimized for performance.

In the paper Dynamically discovering likely program invariants by Michael [11], [12], the approach involves discovering the program state by using the state variables and their values at runtime. The kernel space is neglected in this approach and primarily concentrates on the state of the running program as expected before hand. For example, it can validate the values of certain variables in the binary by running assert statements.

From the work done in the paper Speeding up of synthesis from program traces[13], points out some mechanisms which could be used to build up a program by tracing the instructions while executing the program. Though the work is not directly related to tracing, it provides some insights into identifying the concepts of states and state transitions.

From the work done in the paper, Object oriented Program Tracing and visualization [14], the author identifies the all important state explosion problem and proposes the mechanisms like Pruning, Merging and Slicing to deal with the problem of tracing object-oriented programs.

From the work done in the paper Classification of anomalous traces of privileged and parallel programs by neural networks[15], the authors present the concept of tracing the process state whenever there is a system call. This approach completely ignores the user space execution.

4 Architecture of the runtime state tracing Linux

As seen earlier, one of the primary requisites for building a kernel which can monitor the execution of the binary requires that the control is transferred to the kernel space at frequent points during execution. The current design of the operating system permits the control to be transferred to the kernel only when there is a system call triggered from the user space. The following flow chart gives the current flow for the currently available Linux kernel.

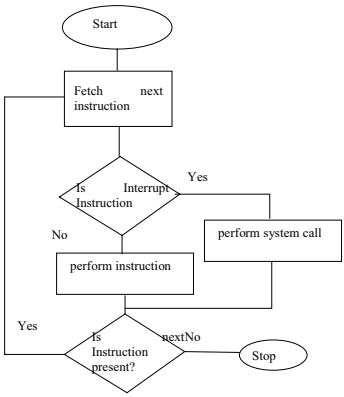


Figure 4.0.a Flow chart giving the current flow of instructions for a binary

The primary requisite for the kernel to track the binary is that it should get the control to determine the current state of the binary. Another requirement is that the mechanism to trigger the context switch to the kernel should be independent of the binary code that is getting loaded as a process. This is to avoid the problem described in Figure 3.1 as there is a risk that the context switch code itself is circumvented by the malicious code if the original binary which has the context switch code is infected with malicious code stubs. Henceforth, the intermediate code that invokes the context switch code is referred to as the interpreter code. Another requirement is that the interpreter that keeps interpreting the binary as it is getting executed needs to define what is a "state" and that definition needs to be congruent as to what the kernel expects when the process is executing in the kernel space. Along with the state definition, tools need to be developed to identify all the states that a binary can be associated with, and this state information if passed to the kernel while the process is about to execute would result in the kernel verifying the state of the process as it executes. Thus a requirement arises that the kernel has a mechanism to peek into the various aspects of the process when the context switch happens to the kernel space as the process keeps executing. Summarizing, here are the changes that need to be done in the kernel and the set of the tools that need to be developed to realize the entire workflow.

- Tool to reverse engineer a binary to identify the complete set of states
- Tool to identify what are the characteristics for each of the states identified in the above step.
- An interpreter which keeps triggering the kernel verification code whenever there is a state transition.
- A modified kernel that accepts calls from the interpreter and verify the state transitions
- A mechanism inside the kernel to verify various aspects of the running process

The following flow chart describes modified flow of control for the proposed Linux kernel and its execution model. The framework involves launching the binary through an interpreter which would fetch blocks of code called as cache, and then fetch the code which hereby is referred to as the injected code in the form of a library. Then both the cache and the code to be injected are executed. The interpreter then saves the context of running binary, executes instructions which form part of the

interpreter itself. It may be noted that the interpreter and the running binary have their own stacks. The interpreter then calls the Linux kernel to pass information like the address of the last instruction that got executed when the context switch was happening from the running process to the interpreter context. The modified Linux kernel then does some verification based on the information passed by the interpreter and also does some book keeping.

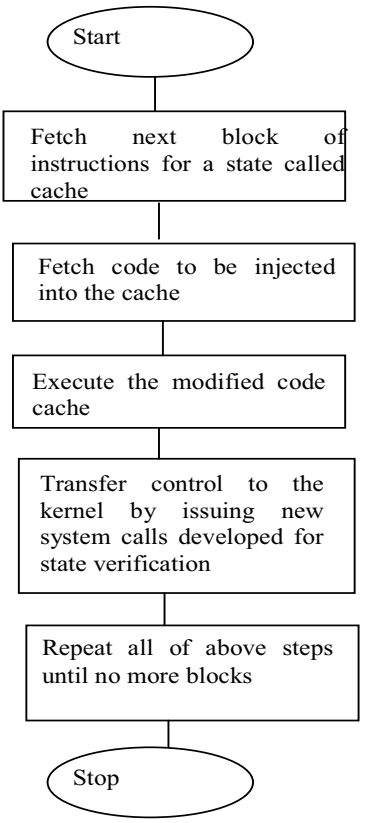


Figure 4.0 giving the flow chart for the current execution and the proposed execution model for the runtime state tracing kernel

The state chart diagram for the approach presented herewith in this paper is shown in figure 4.1 as follows. The normal execution of a single binary is to the left and the execution of the binary using the new approach is presented to the right. (The blocks in the picture are not to scale) . In the picture in the left, the times t_2 and t_4 represent the time spent by the binary in the kernel space (probably executing a system call) and the times t_1,t_3 and t_5 represent the time spent in the user space.

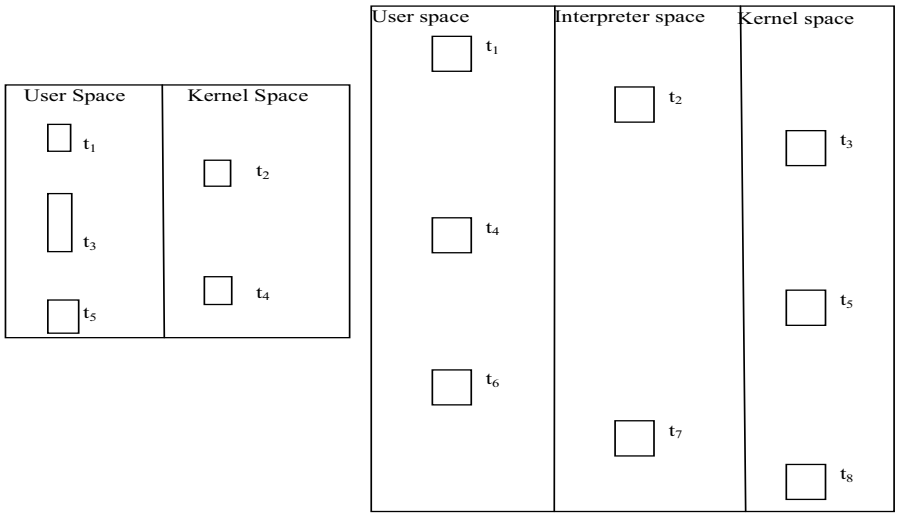


Figure 4.1 Giving the state flow diagram for building the runtime state tracing Linux kernel

In the state flow diagram above, t_i occurs t_j if $i < j$.

The following observations need to be made from the state transition diagram from the two approaches as given above

- The amount of total time taken to execute the binary is definitely going to increase.
- The Interpreter acts as a sandbox under which the binary to be executed is to be run.
- There is some code as part of the interpreter which is executed intermixed with the code of the binary
- The number of system calls may increase proportionally to the number of states.

5 Building the state set

Building the state set involves in reverse engineering the binary. A state may be defined as the collection of sequential instructions that do not branch off due to a jump (conditional/non-conditional) or call instructions. A jump or call instructions is called a transition. A transition path may be taken or not taken based on the execution context. This context is generally decided by a number of factors like user input, state of the data read from external files and a host of other parameters. There are currently more than one executable formats for the Linux platform (ELF, a.out formats). The current effort involves revere engineering the ELF format executables. Following is a brief primer on the ELF executable structure and how it can be reverse engineered to obtain the total set of states.

Another characteristic of the states that are generated is that these states can be assigned a signature if required. It may not be possible that all of the states that are generated can be assigned a signature. It may be noted that the state set is generated on a binary and the signatures that are generated for the states is based only on the static analysis but not on dynamic analysis. The very reason that the signature cannot be extracted for some of the states that the state traversal is based on the fact there are a lot of factors that are decided based on the context like the state of the registers and the value of the memory locations.

This section describes in detail the approach taken by the tool developed to generate the state set. It may be noted that the states and state signature cannot be developed from source code if the developer chooses to mark a state signature because a single line of code translates into a number of states. Before looking into the state generation, one needs to understand the layout of a binary.

An ELF executable/binary is made of sections and segments. Here is the execution view of the ELF format binary.

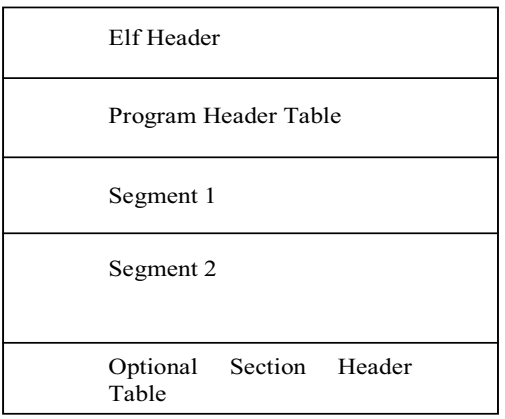


Figure5.1. A block diagram giving the organization of an ELF binary

Segments are loaded into the memory allocated for the execution of the binary. A segment for analysis purposes can be seen as a collection of sections. Each section is made up for a particular kind of information for the execution of the binary. For example, text section contains the instructions that are executed when the control reaches the user space of the binary, the PLT section contains the code to connect the text section based library calls to the actual text section of the library function that is being called. Other sections like the dynamic section contains the dynamic symbols information that are required by the loader to build the image of the binary and the required libraries properly. For more details, the reader is encouraged to the ELF binary specification which gives the meaning of the sections that can be associated with a ELF binary on the Linux Intel platform.

For finding out the sequences of instructions that do not involve a jump/call instructions, the text section needs to be disassembled to get the collection of instructions in the text segment. The PLT section also needs to be disassembled for more advanced analysis of the text section; for example finding out the library function being called. An XML version of the disassembled code complete with the library function call references as generated by the tool developed is given hereunder

```
        08056E01        8B 85 F4 EF FF FF       mov     eax
        [ebp-4108]

        08056E07        05 00 10 00 00  add     eax     0x00001000
        08056E0C        3B 45 14 cmp    eax     [ebp+20]
        08056E0F        0F 83 8E 00 00 00 jnc   0x08056EA3
        08056E15        83 EC 04        sub     esp     0x04
        08056E18        68 00 10 00 00  push    0x00001000
        08056E1D        8D 85 F8 EF FF FF       lea     eax
        [ebp-4104]

        08056E23        50      push    eax
        08056E24        FF 75 0C        push    [ebp+12]
        08056E27        E8 D8 33 FF FF  call    0x0804A204
                <FunctionHint>read</FunctionHint>

        08056E2C        83 C4 10 add    esp     0x10
08056E2F        3D 00 10 00 00  cmp     eax     0x00001000
        08056E34        74 1F   jz      0x08056E55
        08056E36        83 EC 0C        sub     esp     0x0C
        08056E39        68 14 15 09 08  push    0x08091514
        08056E3E        E8 C1 2E FF FF  call    0x08049D04
         <FunctionHint>perror</FunctionHint>            </InstructionList>
</FunctionCodeChunk>
```

Figure 5.2 Disassembled code for a sample text section

On analyzing the figure 5.2, it may be observed that there are four transitions that can take place at the virtual addresses 0x08056E0F, 0x08056E27, 0x08056E34, 0x08056E3E. As per the definition of state as given earlier, the above disassembly would yield four states and four transitions for each of the states identified. Hence the states are as follows: 08056DEE – 08056E0C , 08056E15 – 08056E24, 08056E2C - 08056E2F, 08056E36 – 08056E39.

The tool that has been developed would help in identifying all the possible states for a given binary using the above approach. Here is a chart that gives an estimate of the number of states that could be generated for some of the common binaries on the Linux Intel platform.

Name of the binary	#of states
/bin/ls	2883
/bin/cat	495
/opt/ibm/java2-i386-50/bin/java	2030
/usr/bin/gcc	360
/usr/bin/find	2042

Figure 5.3 Table giving the number of states for some of the binaries available on the RedHat Linux kernel 2.6.19

On generating the set of states that can be reached for a binary, the next step is to generate the state signatures. A state signature forms the building block for verifying the state and hence the state transition when the context switch happens in the kernel space. The signature of a state can be arrived at by using a number of factors or a combination of these factors. For example, the state signature can be specified manually by specifying the signature of a state by using host of factors of the current execution context like number of open files, state of the network interface, number of system calls made till that state is reached, and the time that could be taken to arrive at that state. Additionally, a static analysis of the binary could be performed and then add this set to the list of state signatures.

From a static analysis point of view, some of the factors that could be used to build a state characteristic are as follows:

- Memory state of the registers
- Memory state of some of the global variables
- Memory state of the function variables.

While manually specifying the state signature might be easy(for example, stating that when the control reaches a particular state, the process should have this many files open and this many child processes), it might be too vague because of the following reasons.

Programs are mostly written in a high level programming language and a single line of code might transform into a number of states in the binary format. Hence associating a state signature using manually identifiable characterstics is difficult to arrive at.

It is possible that such a signature can be associated with multiple states. It is possible that the all of these state transitions occur in a serial fashion and hence indistinguishable.

2 Identifying state signature using register states

Identifying the state by peeking into the state of the registers by doing a static analysis assumes that after each transition, the state of some of the registers is going to change or not change at all. (Even a no change can be used to obtain the signature of a state). It is also possible that the contents of the register can go into an indeterminate state. Hence it is possible that not all of the states can be associated with a state signature.

Following is the algorithm that could be used to extract the state signature of a state basing on the register states if it exists.

```
Initialize the EAXVAL, EBXVAL, ECXVAL, EDXVAL, ESPVAL, ESIVAL,
EDIVAL to UNDETERMINATE state.

Initialize the flags to track whether a register has changed; //  To  set  no change in
// register as another signature

For Each Instruction in the total instructions of the state  {

        instr = current Instruction in the instruction list;

        if (instr == PUSH ) decrease the ESPVAL by 4; // Assuming 32 bit
        code

        if (instr == POP ) increase the ESPVAL by 4;

        Then For Each Register EAX, EBX, ECX, EDX, ESI , EDI  Repeat

        if (instr contains current Register) {

                if instr contains “mov” then set current Register as value of
                the second operand
```

if (instr contains “add” && second operand can be resolved as a number) mark that the register value has increased by the value of second operand;

Perform equivalent operations for subtraction, multiplication and division

if instr changes the register to something which cant be resolved statically set register as indeterminate.

If the register goes into an undeterminate state then remove all inferences arrived at till this point

Figure 6.1 Algorithm for extracting the register state for instructions that constitute a state by static analysis.

Following table lists the observations for some of the common executables available on the Linux Intel platform.

Name of the binary	# of states for which signature can be extracted excluding stack changes but including relative register changes(A)	# of states for which signature can be extracted including stack changes (B)	# of states for which signature could be extracted versus the total # of states (A+B)/Total States
/bin/ls	296	682	682/2883 = 23.65 %, (10.26 % excluding stack)
/bin/cat	45	142	142/495 = 28.68 % (9.09 % excluding stack)
/opt/ibm/java2-i386-50/bin/java	152	991	991/2030 = 48.81 % (7.48% excluding stack)
/usr/bin/gcc	32	113	113/360 = 31.38 % (8.88 % excluding stack)
/usr/bin/find	175	492	492/2042 = 24.09 % (8.5 % excluding stack)

Figure 6.2 Table giving the number of states that can be verified versus the total number of states

As can be seen not all of the states can be verified by using the mechanism of register state changes. The above calculation distinguishes between stack based changes versus non stack based changes as stack based changes are relative to previous state of the stack.

2 Identifying state signature using memory state of variables

State signature can also be calculated using the state of some of the global variables whose state changes following transition triggers. For example, in one of the states, the value of a global integer variable might be changed from 4 to 5. To track the changes , the analysis tool needs to analyze instructions that set the value of the global variables that are described in the symbol table which is present as ".symtab" section inside the binary(assuming the binary is not stripped of symbol information). If the file is stripped of symbol information, then the tool needs to analyze the ".text" section for all the instructions that refer the ".data" and the ".bss" sections. Both the ".data" and the ".bss" sections constitute to the data segment in the process image. As per the ELF binary specification, ".bss" section contains the uninitialized data that contributes to the process image. This section does not contribute to any space in the file but occupies memory once the ".bss" section is loaded into the memory. On the other hand, ".data" section contains the data variables that are initialized to some value. Below is the algorithm that is used to obtain the state signature basing the change values to the global data variables.

```
If symboltable is present in the binary then do the following else go to
RAWPARSE:

For each symbol in the symbol table

  Get symbol with the following characteristics from the symbol table

  Symbol type = Object type symbol.

  Applicable Section Header index == Section index of ".bss" section or the
".data" section

  If a match is found get the Symbol size.
  End for.

For each state perform

  For each instruction in the state perform the following

        If the instruction modifies any of the symbols obtained in the previous
block which can be identified explicitly, mark the state change

  End Inner For

End Outer For; goto END

RAWPARSE:

For each state perform

  Obtain the start and end virtual address of the ".bss" and ".data" sections

  For each instruction in the state perform the following

        If the instruction refers an address with the range of the virtual
addresses obtained in step one then

        Obtain the size of the symbol based on the opcode of the instruction
(for further analysis if required)

        See if the instruction helps in generating the state signature mark the
state as recognizable

  End Inner For

End Outer For

END:

  Stop.
```

Figure 6.3 giving the algorithm for state signature extraction based on variables

Following are the observations regarding the state signature extraction based on the value of global variables.

Name of the binary	# of states for which signature can be extracted using global variable change (A)	# of states for which signature could be extracted versus the total # of states (A)/Total States
/bin/ls	1	1/2883 = 0.03 %
/bin/cat	1	1/495 = 0.2%
/opt/ibm/java2-i386-50/bin/java	16	16/2030 = 0.7 %
/usr/bin/gcc	9	9/360 = 2.5 %
/usr/bin/find	60	60/2042 = 2.9 %

Figure 6.4 Table giving the results for state signature using global variables

From the above table it is clear that the number of states that could be detected using the global variables state is very little. Also it maybe noted that these states do not take into account that there are some states whose signature can be calculated using the state of some of the registers during the previous state.

Identifying the state signature using the function variables

The state of the function variables could be used to calculate the state signatures. For example, considering the following piece of dummy code.

```
static int staticfuncWithIntReturn(int one, int two, char three) {
  int localVarOne =6;
  printf("Inside the static Function with int return %d %d %c\n",one,two,three);
  scanf("%d",&localVarOne);
  if (localVarOne > 12) {
        localVarOne +=7;
  }
  else localVarOne +=2;
  printf("The value of the local variable is %d\n",localVarOne);
}
```

Figure 6.5 giving the a piece of code to describe function variables access

It is evident that there is a local variable that is being manipulated at various points in this function. It can also be concluded that this function can be associated with multiple states. The disassembled code for the above function from the tool developed is as follows:

The following observations could be made from Figure 6.6 and Figure 6.5. The function is passed three variables and has got one local variable "localVarOne" which is of type int. Looking at the Intel instruction set volume I, following is the structure of the stack when a function call is being made.

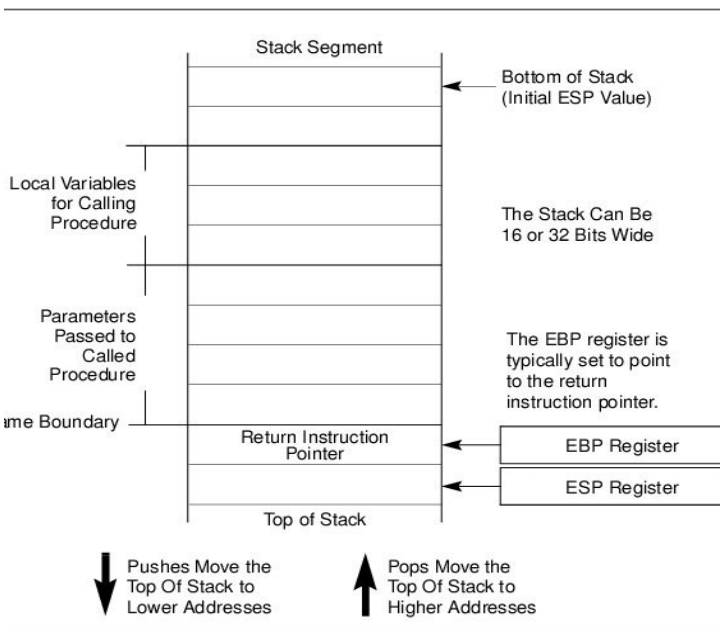


Figure 6.7 Stack layout using the stack call layout (Source Intel Instruction Set Volume I)

Correlating figures 6.5, 6.6 and 6.7, it can be deduced that the local variable "localVarOne" is being referred by the memory location [ebp+248]. However it may be observed that in some locations, the reference is passed to another register and the register is manipulated by using a reference mechanism. Hence the challenge lies in locating such a pattern and identifying the state changes. From the above section of disassembled code in Figure 6.6, it can be observed that the address ranges 0804839C-080483BF , 080483C4- 080483D3, 080483D8- 080483DF, 080483E1-080483E7, 080483E9- 080483FA, 080483FF- 08048403 constitute the six states S^1,S^2,S^3,S^4 ,S^5 and S^6 of the function. The state signature for the state S^1can be verified by checking whether the memory value at the location given by [ebp+248] to be the value 0x00000006. It can also be concluded that the traversal path can be deduced by checking the value of the memory location [ebp+248]. If it has increased by 0x7 then the state S^4 has been traversed whereas if the memory location has increased by 2 then the state S^5 has been traversed. This checking can be initiated in the state S^6 which comes later after traversing one of the states S^4 or S^5.

The following table summarizes the number of states whose signature could be calculated using the state of the function variables by using the approach mentioned above.

The following observations could be made from Figure 6.6 and Figure 6.5. The function is passed three variables and has got one local variable "localVarOne" which is of type int. Looking at the Intel instruction set volume I, following is the structure of the stack when a function call is being made.

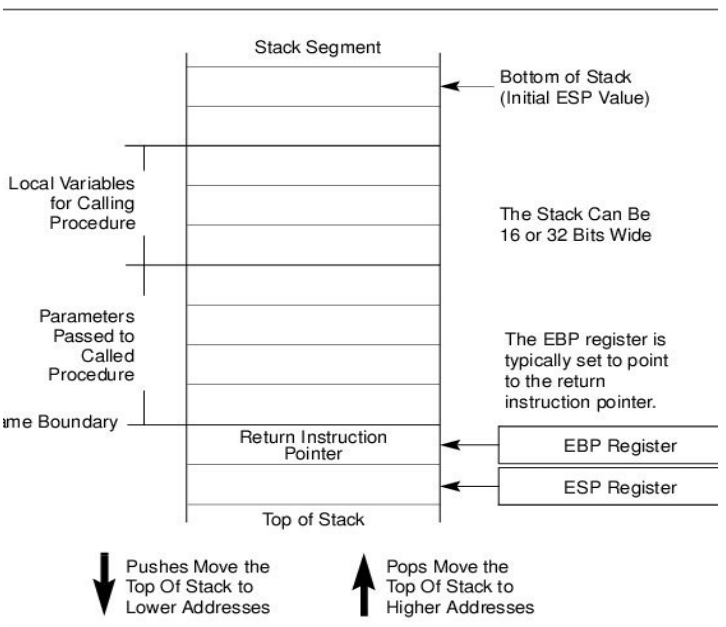


Figure 6.7 Stack layout using the stack call layout (Source Intel Instruction Set Volume I)

Correlating figures 6.5, 6.6 and 6.7, it can be deduced that the local variable "localVarOne" is being referred by the memory location [ebp+248]. However it may be observed that in some locations, the reference is passed to another register and the register is manipulated by using a reference mechanism. Hence the challenge lies in locating such a pattern and identifying the state changes. From the above section of disassembled code in Figure 6.6, it can be observed that the address ranges 0804839C-080483BF , 080483C4- 080483D3, 080483D8- 080483DF, 080483E1-080483E7, 080483E9- 080483FA, 080483FF- 08048403 constitute the six states S^1,S^2,S^3,S^4 ,S^5 and S^6 of the function. The state signature for the state S^1can be verified by checking whether the memory value at the location given by [ebp+248] to be the value 0x00000006. It can also be concluded that the traversal path can be deduced by checking the value of the memory location [ebp+248]. If it has increased by 0x7 then the state S^4 has been traversed whereas if the memory location has increased by 2 then the state S^5 has been traversed. This checking can be initiated in the state S^6 which comes later after traversing one of the states S^4 or S^5.

The following table summarizes the number of states whose signature could be calculated using the state of the function variables by using the approach mentioned above.

Name of the binary	# of states for which signature can be extracted using function variable change (A)	# of states for which signature could be extracted versus the total # of states (A)/Total States
/bin/ls	146	146/2883 = 5.06 %
/bin/cat	32	32/495 = 6.46%
/opt/ibm/java2-i386-50/bin/java	63	63/2030 = 3.1%
/usr/bin/gcc	0	0/360 = 0 %
/usr/bin/find	113	113/2042 = 5.53 %

Figure 6.8 Table giving the number of states whose signature can be calculated using the function variables state .

It maybe noted that there could be some indeterminable states based on the algorithm because of the states use indirect manipulation by first copying the reference of the memory location of the function variable into a different register and then perform operations using references.

Combining the mechanisms to extract the state signature using the various mechanisms mentioned in the previous three sections, the following table summarizes the total number of states versus the total number of states for a binary which can be interpreted at runtime.

Name of the binary	# of states for which signature can be extracted using registers, global variables and function variables	# of states for which signature could be extracted versus the total # of states /Total States
/bin/ls	682+1+146	829/2883 = 28.75 %
/bin/cat	142+1+32	175/495 = 35.35 %
/opt/ibm/java2-i386-50/bin/java	991+16+63	1070/2030 = 52.70 %
/usr/bin/gcc	113+9+0	113/360 = 33.38 %
/usr/bin/find	492+ 60+113	665/2042 = 32.56 %

Figure 6.9 Table giving the number of state signatures that can be verified versus the total number of states

Having seen how the state signatures can be extracted from a binary, the next step lies in building the interpreter which would interpret the execution of the binary and possibly verify the state signature and pass control to the kernel to further verify in case it mandates the kernel verify the state of the running process. The following section explains the interpreter that has been developed and extended using the dynamorio framework.

The Interpreter

It may be noted that the interpreter could be either the loader that is loading the process or a separate process that could get control by interpreting each instruction of the running process just like a normal interpreter. The interpreter needs to have the following characteristics:

It should be able to interrupt the process at the end of each state (It may not necessarily do so for each and every state to increase performance)

It should have the mechanism to inject code into the running process as required.

It should be able to peek into the state of the running process to extract values like register contents and the memory locations

It should be able to make system calls to the underlying kernel so that extra validation could be done inside the kernel. Some of the common verification mechanisms are currently open file handles, network sockets etc.

Dynamorio is a framework that meets all of the above requirements. It is a code caching framework that fetches blocks of code and executes the cached code and then gains control back after executing the block. Before executing the basic block, dynamorio allows the programmer to inject instructions into the code cache and there by mix code from both the verifier module that is verifying the state of the binary and the code from the binary itself. Dynamorio was chosen to showcase that the whole approach of building a verifiable system by using some of the already existing frameworks and not building from the scratch. The overhead of having a code caching framework over a dynamic interpreter has been overlooked to verify the idea of runtime quickly.

Dynamorio is the primary process that is launched by the user when executing the binary. Dynamorio , based on the options passed through environment variables, lets additional libraries to be loaded before the binary is launched. This way, custom code could be injected into the execution path of the binary. For the runtime state tracing framework to work as designed, a library has been developed that has the following characteristics

An xml parser sub-module that can parse the output generated from the tool that generated the state set and their verification signatures. It may be noted that the output of the state generation tool is in XML format and hence the need for the xml parser sub-module.

A mechanism to track state traversal based on the EIP values i.e. the current point of execution of the binary. (The value might be a valid state traversal or invalid the binary is injected with malicious code.) This part of the module just ensures that the control has gone from a state with an address A to another state with address B. It might be verified through other part of the modules of the library that address B is a valid or not valid state traversal.

A sub-module that has mechanisms to peek into the state of the running process. This should have the capability to read the state of the registers and memory locations that are currently allocated to the binary.

A mechanism to talk to the kernel to mark the beginning of the execution of the binary, the transfer of control to various states and the mark the stopping of the execution of the binary.

Since dynamorio takes control during various points of execution, this means that dynamorio saves the context of the running process on the stack designated for dynamorio and this stack is different from the stack that is used by the binary. Hence care needs to be taken in the library that was developed, so that while peeking the code or state of the running binary, it should not be the dynamorio stack but the stack and context of the running binary that should be examined.

The library with the above characteristics is loaded by dynamorio and then dynamorio makes system calls to the underlying modified Linux kernel . The changes made to the Linux kernel are explained in the next section.

Changes to the Linux kernel

After having the interpreter in place, the Linux kernel also needs to be modified because mechanisms need to be in place for

Tracking additional information for the life of the process like the states traversed from the beginning of execution of the binary to any point later in time, the total number of states for the running process and the valid state model for the binary.

Extra API/system call mechanisms to mark the beginning, execution and end of the running process through the intermediary, the dynamorio library, that was developed as described in the previous section.

Track the usage of various resources like the open file handles, the number of sockets open, time spent in the user space and the kernel space.

The following paragraph describes the changes made to the kernel to implement the above mentioned functionality.

"struct task_struct", the data structure that holds the process specific information is modified as follows.

```
struct task_struct {

  volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */

  struct thread_info *thread_info;
  struct reg_vals * reg_vals_sem;

  struct process_runtime_semantic_desc_node * runtime_history;

  struct process_runtime_semantic_desc_node * current_point_of_execution;

  struct process_static_semantic_state_node * static_model;

  struct process_static_semantic_state_node * current_point_in_execution_model;

  struct process_info * process_metadata_info;

  long int num_of_places;
```

Figure 8.1 The primary data structure for process tracking in the Linux Kernel

As can be seen, the structure in the modified kernel holds the following additional information.

The variable “reg_vals_sem” represents the current value of the running process when the context switch happened from the user space to the kernel space. “runtime_history” represents the states traversed from the beginning of execution till the current point in time. “current_point_of_execution” represents the current node that has been identified using the signature. “static_model” represents the model that has been generated for the binary and the variable “current_point_in_execution_model” represents the state signature that the variable “current_point_execution” should be verified with. The variable “process_metadata_info” represents the information about the binary like the name of the binary, the value of the starting virtual address and other information about the process program header table. The variable “num_of_places” represents the total number of states that can be associated with the binary.

Apart from modifying the task_struct data structure, the kernel needs to have new system calls so that control is transferred to the kernel via the dynamorio interpreter library. Some of the system calls that are implemented are as follows:

- asmlinkage **long sys_semlog(long int** model_desc,**long int** instr_addr, **long int** target_addr, **long int** reg_state)
- asmlinkage **long sys_seminit(char** * name_of_exec ,**long int** ptrToModel,**long int** numberOfPlaces, **long int** ptrToProcInfo)
- asmlinkage **long sys_semdestroy(long int** firstparam ,**long int** entryname,**long int** thirdparam)

The first system call semlog is the mostly used sytem call by the interpreter dynamorio library to mark the state traversal. Some of the parameters that are passed to the kernel through this system call are the current instruction virtual address, the target virtual address which the control would go to in the case of the state transition occurs when the context switch happens to the user space back.

The second system call seminit is used to mark the beginning of execution of the process. Some of the parameters that are passed to the kernel are the name of the process that is being launched, a pointer to a memory location in the user space so that the kernel can copy the model to the dynamorio accessible memory location to implement state signature verification if required. Another parameter “ptrToProcInfo” is used to pass the proc_metadata_info that is used in the kernel task_struct data structure to keep track of the process program header table information.

The last system call is used to free any kernel memory acquired by calling kmalloc kind of kernel memory allocation calls. It may be noted that the primary task is done in the semlog system call which is used to perform various verifications.

Below are some of the system characteristics that could be used to verify the state of the running process. It may be noted that these characteristics does not require a state model to be verified.

The time spent by the process in the kernel and the time spent in the user space (t)

The number of open handles to the file system

The network interfaces open

The number of child processes.

The number of system calls made till that point of execution

The above factors are calculated using the standard kernel data structures in the modified Linux kernel.

Observations

It is evident from the architecture that there is a hit on the performance of the binary because of code caching technique and increased number of system calls. The following observations were made while running some of the common binaries which are available on the Linux platform. The table below summarizes the process accounting information in two scenarios. 1. The execution of the binary under the dynamorio without having the custom library inject any code and 2. The execution of the binary with the code injection also in place.

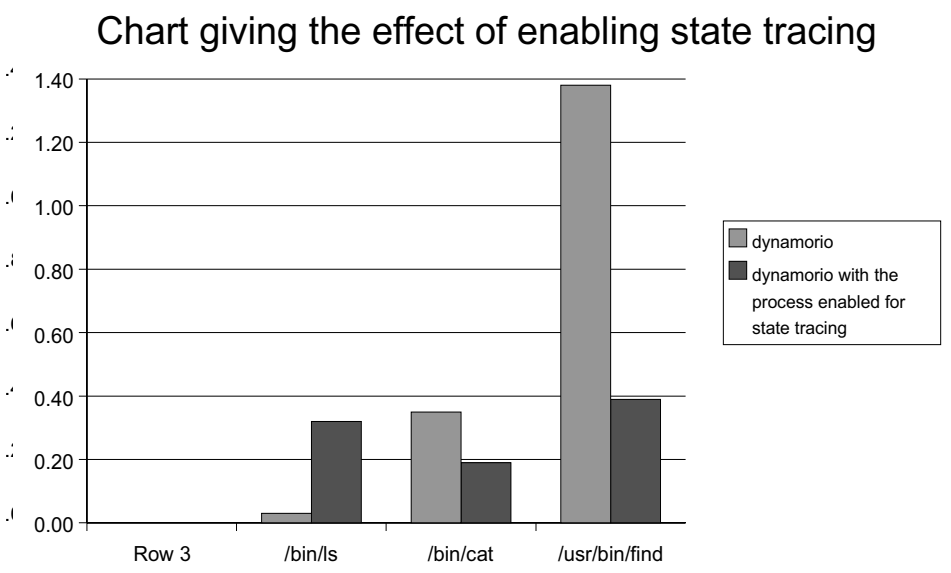


Figure 9.1 Chart giving the performance analysis of the new approach . Time is on y-axis in seconds

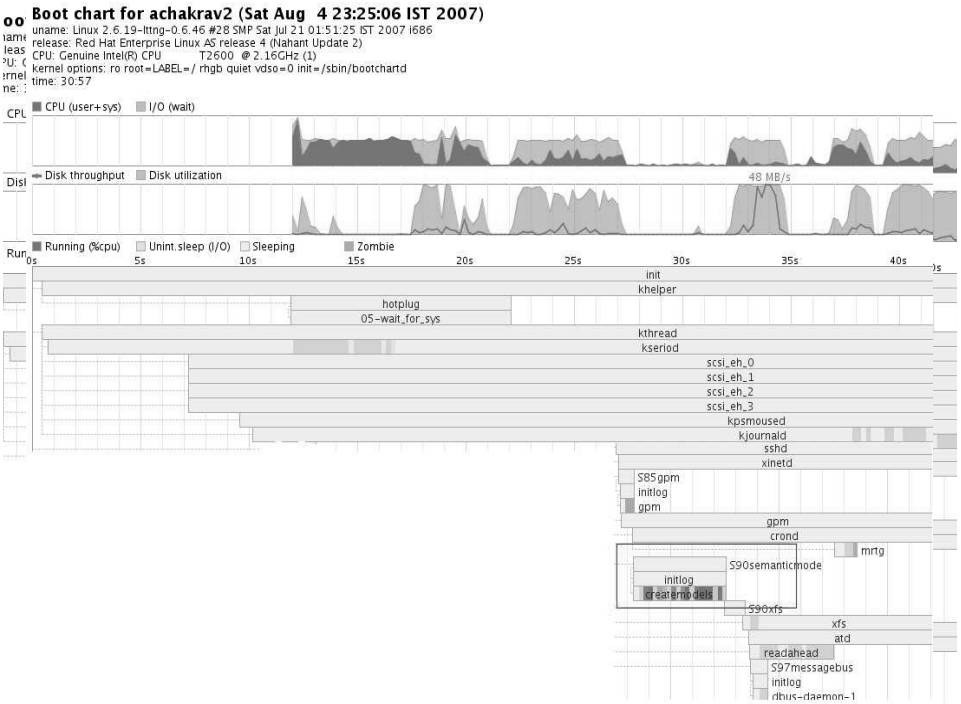
From the chart it is evident that the performance has increased after injecting code. It is because dynamorio enhances the performance of the code caching mechanism once it recognizes the common pattern while executing.

It may be noted that a better performance can be increased by doing most of the verification in the interpreter space rather than the kernel space. This would result in increase of the performance because the number of system calls related to the verification mechanism is going to decrease in proportional to the states whose signature can be verified using the register states. Since this mechanism is applicable on the functionality of the binary but not every binary in general, the following performance improvements have been considered.

Delegate the process of loading the state description files, i.e. the XML files at the time of booting the kernel rather when the process is getting launched for execution.

Delegate the process of verifying the state of the process in the interpreter space and switch to the kernel verification mechanism only when required, for example only when verifying the open file handles, the number of child processes.

The following chart gives the effect of implementing the concept of loading the models at boot time.



The effect of loading the state description files which are in XML format while the kernel is booting rather than when the process is getting launched is seen in the following chart. The chart is shown for a RedHat Linux kernel. The time taken to load the models amounting to a size of 15.2 MB is shown in the chart above. The downside of this approach is that it might a little more time to boot the kernel . From the reading on the chart, a set of models for some of the common binaries with a file size of 15.2 MB amount a boot time delay of 4 seconds.

Regarding the second aspect of improvement, that is by delegating the state verification to the interpreter space rather than the kernel wherever possible will definitely see an increase in the performance. For example if the state signature is to be verified based on the values of the registers than on the state of the kernel resources , then verification could be done in the interpreter space itself.

Conclusions and Future work

As can be seen from the above sections, a state tracing mechanism could be implemented using the approach as discussed in this paper. The first step involves in generating the model by doing a static analysis of the binary and then generating state signatures. These state signatures can then later be verified using an interpreter framework using dynamorio and a modified Linux kernel.

Future work involves in creating an optimized state model for a given binary and a tool which can describe the state model by using a semantic dictionary. By having a model which can be both verified by the kernel while executing and at the same time verifiable by a human can go a long way in improving the security of binaries on operating system.

Bibliography

1. [BGA 03] An infrastructure for adaptive dynamic optimization
   Bruening , D.; Garnett, T.; Amarasinghe, S.;
   Code Generation and Optimization, 2003. CGO 2003. International Symposium on
   23-26 March 2003 Page(s):265 - 275
   Digital Object Identifier 10.1109/CGO.2003.1191551
2. [BKGB 06] Thread-shared software code caches
   Bruening, D.; Kiriansky, V.; Garnett, T.; Banerji, S.;
   Code Generation and Optimization, 2006. CGO 2006. International Symposium on
   26-29 March 2006 Page(s):11 pp.
   Digital Object Identifier 10.1109/CGO.2006.36

3. [BA 05] Maintaining consistency and bounding capacity of software code caches
   Bruening, D.; Amarasinghe, S.;
   Code Generation and Optimization, 2005. CGO 2005. International Symposium on
   20-23 March 2005 Page(s):74 - 85
   Digital Object Identifier 10.1109/CGO.2005.19

4. [] Online impact analysis via dynamic compilation technology
   Breech, B.; Danalis, A.; Shindo, S.; Pollock, L.;
   Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on
   11-14 Sept. 2004 Page(s):453 – 457 Digital Object Identifier 10.1109/ICSM.2004.1357834

5. Dynamorio Home page http://www.cag.lcs.mit.edu/dynamorio/

6. Derek Bruening's Ph.d thesis http://www.burningcutlery.com/derek/phd.html

7. Some notes on adding system calls in Linux http://fossil.wpi.edu/docs/howto_add_systemcall.html

8. ELF Specification is available at http://x86.ddj.com/ftp/manuals/tools/elf.pdf

9. ELF specification explained in detail http://www.cs.ucdavis.edu/~haungs/paper/node10.html

10. [YI 01] A super tracer and an analyzer for analyzing detailed behavior of a Linux on a Pentium family processor (STDB)
    Y. Sugimura; S. Ido;
    Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the
    2001 Page(s):298 - 305
    Digital Object Identifier 10.1109/ECBS.2001.922436

11. [E 01] Summary of dynamically discovering likely program invariants
    Ernst, M.D.;
    Software Maintenance, 2001. Proceedings. IEEE International Conference on
    7-9 Nov. 2001 Page(s):540 - 544
    Digital Object Identifier 10.1109/ICSM.2001.972767

12. [ECGN 00] Quickly detecting relevant program invariants
Ernst, M.D.; Czeisler, A.; Griswold, W.G.; Notkin, D.;
Software Engineering, 2000. Proceedings of the 2000 International Conference on;
4-11 June 2000 Page(s):449 - 458
Digital Object Identifier 10.1109/ICSE.2000.870435

13. [BBP 75] Speeding up the Synthesis of Programs from Traces
Biermann, A.W.; Baum, R.I.; Petry, F.E.;
Computers, IEEE Transactions on
Volume C-24, Issue 2, Feb. 1975 Page(s):122 - 136

14. [LN 97] Object-oriented program tracing and visualization
Lange, D.B.; Nakamura, Y.;
Computer
Volume 30, Issue 5, May 1997 Page(s):63 - 70
Digital Object Identifier 10.1109/2.589912
15. [ZBV 03] Classification of anomalous traces of privileged and parallel programs by neural networks
Zhen Liu; Bridges, S.M.; Vaughn, R.B.;
Fuzzy Systems, 2003. FUZZ '03. The 12th IEEE International Conference on
Volume 2, 25-28 May 2003 Page(s):1225 - 1230 vol.2

16. [HS 02] An instruction set and microarchitecture for instruction level distributed processing Ho-Seop Kim; Smith, J.E.; Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on 25-29 May 2002 Page(s):71 - 81 Digital Object Identifier 10.1109/ISCA.2002.1003563

17. [MS 93] Applying algorithm animation techniques for program tracing, debugging, and understanding Mukherjea, S.; Stasko, J.T.; Software Engineering, 1993. Proceedings., 15th International Conference on 17-21 May 1993 Page(s):456 - 465 Digital Object Identifier 10.1109/ICSE.1993.346020
18. [BK 04] Aspect mining using event traces Breu, S.; Krinke, J.; Automated Software Engineering, 2004. Proceedings. 19th International Conference on 2004 Page(s):310 - 315 Digital Object Identifier 10.1109/ASE.2004.1342754

19. [ED 87] Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance Emma, P.G.; Davidson, E.S.;
Computers, IEEE Transactions on Volume C-36, Issue 7, Jul 1987 Page(s):859 - 875

20. [BM 03]Compressing extended program traces using value predictors Burtscher, M.; Metha Jeeradit; Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on 27 Sept.-1 Oct. 2003 Page(s):159 - 169 Digital Object Identifier 10.1109/PACT.2003.1238012

21. [KN 05] Construction and compression of complete call graphs for post-mortem program trace analysis Knupfer, A.; Nagel, W.E.;
Parallel Processing, 2005. ICPP 2005. International Conference on 14-17 June 2005 Page(s):165 - 172 Digital Object Identifier 10.1109/ICPP.2005.28

22. [RR 05] Demonstration of JIVE and JOVE: Java as it happens Reiss, S.P.; Renieris, M.; Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on 15-21 May 2005 Page(s):662 - 663 Digital Object Identifier 10.1109/ICSE.2005.1553642

23. [HMJ 91] Detecting data races from sequential traces Helmbold, D.P.; McDowell, C.E.; Jian-Zhong Wang; System Sciences, 1991. Proceedings of the Twenty-Fourth Annual Hawaii International Conference on Volume ii, 8-11 Jan. 1991 Page(s):408 - 417 vol.2 Digital Object Identifier 10.1109/HICSS.1991.184003

24. [IG 06] Efficient Incremental Optimal Chain Partition of Distributed Program Traces Ikiz, S.; Garg, V.K.; Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on 04-07 July 2006 Page(s):18 - 18 Digital Object Identifier 10.1109/ICDCS.2006.34

25. [L 93] Efficient program tracing Larus, J.R.; Computer Volume 26, Issue 5, May 1993 Page(s):52 - 61 Digital Object Identifier 10.1109/2.211900
26. [PK 93] Multilevel Simulation Of Distributed-memory Program Traces Prost, J.-P.; Kipnis, S.;Simulation Symposium, 1993. Proceedings. 26th Annual March 29- April 1,1993 Page(s):60 – 67

29. [NP 96] The effect of program behavior on fault observability Bowen, N.S.; Pradhan, D.K.; Computers, IEEE Transactions on
Volume 45, Issue 8, Aug. 1996 Page(s):868 - 880 Digital Object Identifier 10.1109/12.536230

30. The open source IDE used in developing the code and the modified Linux kernel http://www.eclipse.org/

31. The Linux Process Manager: The Internals of Scheduling, Interrupts and Signals by John O'Gorman John Wiley & Sons © 2003 For explanation of some of the most commonly used Linux Kernel data structures

32. The eclipse C/C++ plugin site , the IDE plugins used to develop the tool http://www.eclipse.org/cdt/