Architecture- and OS-Independent Binary-Level Dynamic Test Generation

Gen Li, Kai Lu, Ying Zhang, Xicheng Lu, and Wei Zhang

School of Computer, National University of Defence Technology, ChangSha, China
superligen@gmail.com, {kailu,zhangying,xichenglu,wz}@nudt.edu.cn

Abstract. Dynamic test generation approach consists of executing a program while gathering symbolic constraints on inputs from predicates encountered in branch statements, and of using a constraint solver to infer new program inputs from previous constraints in order to steer next executions towards new program paths. Variants of this technique have recently been adopted in finding security vulnerabilities in binary level software. However, such existing approaches and tools are not retargetable: on the one hand, they can only find vulnerabilities in the binaries for a specific ISA; on the other hand, they can only find vulnerabilities over a specific OS because the execution trace is totally OS-dependently recorded in these tools. This paper presents a new dynamic test generation technique and a tool, ReTBLDTG, short for *ReTargetable Binary-Level Dynamic Test Generation*, that implements this technique. Unlike other such techniques, ReTBLDTG can deal with binaries for any ISAs over any OSes. ReTBLDTG is based on the whole system virtual machine that provides OS-independent and fast *concrete execution* of the target program. And which thread the executing instruction belongs to is OS-independently identified by analyzing the registers' value and hardware events over the virtual machine. Thus, the execution trace is recorded, without knowing the internal structure of the guest OS. At the same time, ReTBLDTG defines a *Meta Instruction Set Architecture* (MetaISA); ReTBLDTG maps the execution information, which is collected during the binary source code execution, to MetaISA; and symbolic execution, constraint collection and constraint solver operates on MetaISA, thus making these tasks ISA-independent. We have implemented our ReTBLDTG, retargeted it to 32-bit x86, PowerPC and Sparc ISAs, and used it to automatically find the six known bugs in the six benchmarks over Linux and Windows. Our results indicate that our ReTBLDTG can be easily retargeted to any ISA with only a few overheads; and ReTBLDTG can effectively find bugs located deep within large applications over any OS.

1 Introduction

Dynamic test generation approach, like DART [6], EXE [3] and SAGE [7], is becoming increasingly popular to find security vulnerabilities in software. Starting with a fixed input, the approach symbolically executes the program, gathering input constraints from conditional statements encountered along the way.

S. Qing, C.J. Mitchell, and G. Wang (Eds.): ICICS 2009, LNCS 5927, pp. 198–210, 2009.
© Springer-Verlag Berlin Heidelberg 2009

The collected constraints are then systematically negated and solved with a constraint solver, yielding new inputs that exercise different execution paths in the program. For example, symbolic execution of the conditional statement "if (x==10) then" on the input x = 0 generates the constraint x != 10. Once this constraint is negated and solved, it yields x = 10, which gives us a new input that causes the program to follow the then branch of the given conditional statement. This allows us to exercise and test additional code for security bugs, even without specific knowledge of the input format. Furthermore, this approach automatically discovers and tests corner cases where programmers may fail to properly allocate memory or manipulate buffers, leading to security vulnerabilities.

More and more research institutes and groups use this approach to find security vulnerabilities in the pre-release software, which is usually shipped in binary code, after it has been heavily tested using a combination of code review, manual and random testing, dynamic tools and static analysis, because it finds security vulnerabilities without generating false alarms and requires no domain knowledge.

However, existing such tools are not retargetable. On the one hand, they can only find vulnerabilities in the binaries for a specific ISA, due to specific architecture details of different *Instruction Set Architectures* (ISAs).

On the other hand, they can only find vulnerabilities over a specific OS because the execution trace is totally OS-independently recorded in these tools. The execution trace, made up of the instruction flow, of the target program run with the initial input, should be recorded for the following constraint generation and solver. Currently, the execution trace is recorded as the program is executed either by statically injected instrumentation code or with the help of binary instrumentation tools such as Nirvana [2] or Valgrind [12]. However, these instrumentation tools strongly depend on the OS (Operation System), thus make existing dynamic test generation un-retargetable. These instrumentation tools run over the guest OS and call OS-dependent *Application Programming Interfaces* (APIs) to identify the process and its threads of the target program, and monitor its thread switch. Because the process and thread management are totally defined by the OS, the instrumentation tools must know the internal structure of guest OS.

The coupling of binary-level dynamic test generation with specific architecture or OS details creates an interoperability problem that hinders the wide adoption of binary-level dynamic test generation. To adopt this approach to find security vulnerabilities for any other ISAs or over any other OSes, one has to develop another separate tool for the specific ISA or OS.

This paper presents a new binary-level dynamic test generation technique and a tool, ReTBLDTG, short for *ReTargetable Binary-Level Dynamic Test Generation*, that implements this technique. Unlike other dynamic test generation techniques that operate only on binaries for a specific ISA, ReTBLDTG can process binaries for any ISAs over any OSes and dynamically generates new inputs that exercise different control paths in the program, which may lead to security vulnerabilities.

To mask the difference of the CPU ISAs, ReTBLDTG defines a *Meta Instruction Set Architecture* (MetaISA). When working, ReTBLDTG maps the execution information, collected during the execution of binary source code, to MetaISA. And symbolic execution, constraint collection and constraint solver operate on the code in our MetaISA, thus making the three processes ISA-independent. As shown in Section 2, ReTBLDTG consists of 208KLOC and these three processes represent over 94% of our code base. Thanks to MetaISA, ReTBLDTG is retargetable with only a few overheads. To port ReTBLDTG to a new CPU platform, we only need to implement a new decoder and an ISA mapper for it.

Because symbolic execution, constraint collection and constraint solver operate on the code in our MetaISA, the following key issues must be considered when MetaISA is designed:

- The MetaISA should be as simple and uniform as possible in order to facilitate the following three processes. This requires all meta instructions should be arithmetic instructions. The conditional instructions, such as `cmp`, should be transformed to the change to flag bit, including `ZF, OF, and CF`, and so on; the branch instructions, such as `Jnz`, should be transformed to the selection of PC based on some registers' value; the complex instructions, such as `bsf` (Bit Scan Forward) that searches the source operand for the least significant set bit, and `rep movsd` that copies data from source to destination until $ecx == 0$ from x-86 ISA, should be expressed by the combination of simple arithmetic instructions. Thus, the constraints expressed in simple meta arithmetic instructions can be mapped to the SMT solver smoothly. Additionally, this can also simplify the symbolic execution.
- Each instruction operation of the MetaISA should be bit-precision. This requires how each bit of each variable of the left-hand side (LHS) of a meta instruction is computed from every bit of the right-hand side (RHS) must be precisely expressed. This is because the SMT solver, used to generate a new input excising to a different control path based on the gathering constraints, adopts *bit-vector* theory that demands all constraints expressed as bit-precision .
- The design for MetaISA should consider its effect on performance and memory consuming of ReTBLDTG. In 32-bit x86 ISA, for example, most instructions operate on 32-bit data and only a few, such as `mul, div` and `mod`, generated 64-bit medium data. When 32-bit x86 ISA is mapped to MetaISA, it is easy to map all operands to 64-bit registers. However, when ReTBLDTG deals with very large real applications with millions of instructions, the total memory requirement of symbolic execution, constraint collection and constraint solver would be huge. However, if all 32-bit x86 instructions are mapped to meta instructions operating on 32-bit registers, the total memory requirement can be reduced to nearly half, and improve the efficiency of the SMT solver.

To mask the difference of the OSes, the execution trace of the target program should be under the OSes or over the naked CPU. Otherwise, the identification of

the process and its thread must call the OS API. Fortunately, running the target program on the whole system simulator is a good choice to address the above problem: on the one hand, the registers and hardware events can be monitored over the whole system virtual machine, without any help of OSes; on the other hand, the whole system virtual machine can execute the target program over the guest OS with quite fast and acceptable speed compared with the time-consuming constraint collection and solver.

ReTBLDTG is based on the whole system virtual machine Simics but is not dependent on it. The whole system virtual machine provides OS-independent and fast *concrete execution* of the target program. And which thread the executing instruction belongs to is OS-independently identified by analyzing the registers' value and hardware events over the virtual machine. Thus, the execution trace is recorded, without knowing the internal structure of guest OS.

The main contributions of this paper are as follows.

- We design a meta ISA;
- We present a method to online identify the process;
- We present, for the fist time to the best of our knowledge, a method to online identify the thread;
- We build, for the first time, a new binary-level dynamic test generation technique and a tool, ReTBLDTG, that can find bugs from binaries for any ISA over any OSes, based on a whole system virtual machine;
- We have retargeted ReTBLDTG to 32-bit x86, PowerPC and Sparc ISAs by now;
- We have retargeted ReTBLDTG for the Linux and Windows binaries by now;
- Our ReTBLDTG efficiently found the bugs from the Linux and Windows the binaries for 32-bit x86, PowerPC and Sparc ISAs.

The rest of this paper is organized as follows. The ReTBLDTG system architecture is given in Section 2. Section 3 identifies the process and its thread of the running target program based on the execution of virtual machine. In Section 4, we show how MetaISA is designed in order to make ReTBLDTG architecture-independent. Our experiments and performance evaluation appear in Section 5. The related work is discussed in Section 6 and we conclude in Section 7.

2 The ReTBLDTG System Architecture

As shown in Figure 1, ReTBLDTG is built around four levels of abstraction to make it architecture- and OS-independent. Presently, ReTBLDTG consists of 208KLOC with 0.6% in the Virtual Execution Layer (VEL), 5.0% in the Process/Thread Identification Layer (PTIL), 1% in the MetaISA Layer (ML) and 94% in the Constraint Analysis Layer (CAL). We describe each layer only briefly in the rest of this section and focus mostly on introducing how PTIL as well as VEL make CAL OS-independent in Section 3, and how ML makes CAL ISA-independent in Section 4.

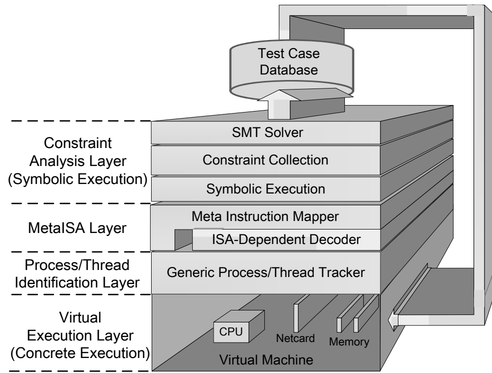

Fig. 1. ReTBLDTG system architecture

2.1 Virtual Execution Layer

VEL is essentially a virtual machine (VM) or a whole system emulator like VMware, Simics or QEMU. Presently, ReTBLDTG is based on Simics but is not dependent on it. VEL provides OS-independent and fast *concrete execution* of an application.

As a VM, ReTBLDTG is equipped with the capability of analyzing applications that are time-sensitive or protected by anti-debugging techniques. ReTBLDTG can freeze the entire system, including the clock when performing time-consuming tasks like points-to analysis at a higher layer. As a result, the OS and the application are not even aware of the elapse of the time. Furthermore, all test cases can be restarted from exactly the same system state.

2.2 Process/Thread Identification Layer

PTIL distinguishes different processes and segments the instructions flowing through the CPU in the same process into different instruction sequences belonging to different threads. This is necessary since different threads in a process should have their own sets of registers associated with them.

The principle behind PTIL is based on some hardware events, independent of the OS. For example, in x86 ISA, process switching is recognized by listening to the CR3-Changed-Event as in [8]. Thread identification appears to be novel (to the best of our knowledge). Thread starting or stopping is recognized by identifying a thread using its stack pointer (ESP) and by monitoring the CPU privilege level transitions between Ring0 and Ring3. These ideas generalize to other architectures.

2.3 The Meta Instruction Set Architecture

One key motivation for introducing a MetaISA is to make CAL ISA-independent. Another is to facilitate our symbolic execution, constraint collection and constraint solver on a simple and uniform MetaISA. The entire CAL layer represents

over 94% of our code base. Thanks to MetaISA, CAL is now retargetable. To port ReTBLDTG to a new CPU platform, we only need to implement a new decoder and an ISA mapper for it.

2.4 Constraint Analysis Layer

As shown in Figure 1, CAL is responsible for performing symbolic execution, constraint collection and constraint solver. This layer focuses on a process being analyzed and ignores the instruction sequences from the other processes and the OS kernel, which is made possible by the PTIL layer below. This layer represents over 94% of our code base and performs the most time-consuming tasks of our ReTBLDTG as evaluated in Section 5. Our CAL is like the of SAGE. Refer to [7] for more details.

3 Making CAL OS-Indepedent

The VEL and PITL work together to make CAL OS-independent. The VEL in ReTBLDTG has two tasks that: 1. VEL fast executes the instruction flow, mixed by guest operating system and destination application; 2. it also submits the specific hardware events like *Page Table Switch* (In x86, it is CR3-Changed-Event) and *CPU Privilege Switch* events and allows PTIL to listen to. Based on these hardware events, PTIL splits the single instruction sequence, flowing CPU, into instruction sequences belonging to different threads of the target process.

It is a meticulous consideration that ReTBLDTG adopts a whole system virtual machine based online approach, instead of an instrumentation as existing dynamic test generation systems. Thus, ReTBLDTG has more flexibility, independent on any specific OS. As discussed before, the instrumentation tools, like Valgrind and iDNA, strongly depend on the OS API. ReTBLDTG is based on the virtual machine and can transparently monitor the registers and hardware events of the virtual machine. Thus, ReTBLDTG records the execution trace by analyzing the CPU behavior, without knowing the internal structure of guest OS. Recording the execution trace will not be affected by the operating system protection or application self-protection. The instrumentation tools, like iDNA [2] and Valgrind, will be disturbed by the software protections, such as Anti-Debug. These protections should not be removed during systemic test because these protections are also part of the software under test. VEL watches the guest OS execution in the view of CPU. Thus, the target running program cannot feel the existence of ReTBLDTG. Therefore, VEL can not be disturbed by software self-protection. ReTBLDTG can effectively process the time-sensitive applications, particularly network applications. For time-sensitive applications, like network applications, the time-pause caused by CAL may lead a timeout for receiving/sending a packet. When ReTBLDTG does time-consuming task, such as constraint collection and solver, ReTBLDTG freeze the whole system clock, without the guest OS and target program aware of it. But the existing dynamic test generation tools, based on instrumentation tools, can not correctly process time-sensitive applications.

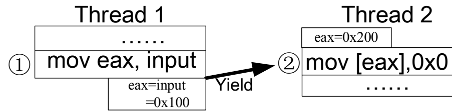

Fig. 2. A case showing that the thread switch interferes with the symbolic execution

The PTIL, as shown in Figure 1, precisely OS-independently tracks the target process and its threads. It dynamically filters the instructions flowing through CPU and segments them into different instruction sequences belonging to different threads of processes, and ReTBLDTG binds each instance of *thread trackers* to track the instruction sequence for each thread.

We must accurately identify each thread of the target process because each thread has its own register space. As shown in Figure 2, when thread 1 is executing instruction 1, `eax=input=0x100`. Assuming that thread 1 switches to thread 2 before executing its next instruction. The concrete execution of the guest OS has switched to the register context of thread 2 , with `eax` updated to `0x200`, before thread 2 runs. Because the symbolic execution of CAL is independent on the concrete execution, the real register status, maintained by virtual machine, must be kept for the following CAL analysis. Otherwise, when the instruction $mem\,[\text{eax}] = 0$ is executed in thread 2, the symbolic execution will be diverged from the concrete execution.

Identifying target process/thread. Segmenting the single instruction sequence flowing through CPU into different instruction sequences, belonging to different threads of each process, can accurately track each process and its threads.

Identifying the processes. Nowadays, according to the implementation of the mainstream OSes, every process has its own page table, pointed by CR3 register and used to isolate virtual address resource by MMU. When the CR3-Changed-Event happens, we can identify that the process must be switched. In Figure 3, the CR3 is changed to 0x1000, ReTBLDTG looks at the new value of CR3 as the PID of the switched-in process.

Identifying the threads. We can find that every process-switch event must happen in the CPU privilege level of Ring0. After the privilege level is dropped to Ring3, the instruction sequence to be executed must belong to one of the current process's threads. Generally, the *stack pointer* (ESP) can be used to effectively distinguish and identify the threads of the same process because each thread has its own stack. Different from the CR3 register, ESP is changed along with the execution of the current thread. ReTBLDTG records the value of ESP into the *thread ID* (TID) list when the CPU privilege level raised to Ring0; and ReTBLDTG checks whether the current ESP has been recorded in the TID list when the CPU privilege level drops to Ring3 in the same process. If yes, ReTBLDTG appends the instruction sequence to be executed before next switching to the execution trace of the found thread; otherwise, ReTBLDTG identifies a new thread.

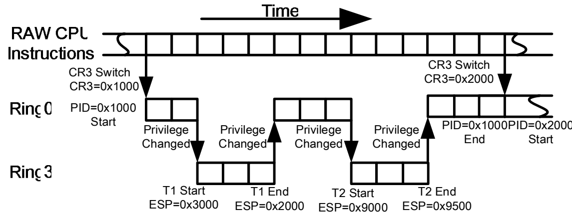

Fig. 3. Tracking generic process/thread

Figure 3 describes how to track generic threads. When the CPU privilege goes to Ring3 in the process `0x1000` for the first time, ReTBLDTG identifies that `T1` is the current thread. The instruction sequence from `T1 Start` to `T1 End` belongs to `T1` . After `T1 End`, the CPU privilege rises to Ring0. Before entering Ring0, ReTBLDTG reads `0x2000` from the current stack pointer (ESP), and makes the value map to `T1: 0x1000`$\mapsto$`T1`. When CPU returns to Ring 3 in the same process, ReTBLDTG examines whether the value of ESP has been mapped. In this case, the current ESP equal to `0x9000`, there is no thread mapped to `0x9000`. ReTBLDTGjustifies the current thread is not T1, and identifies it as `T2`.

4 Making CAL ISA-Independent

One key motivation for introducing a MetaISA is to make CAL ISA-independent. Another is to facilitate our symbolic execution, constraint collection and constraint solver on a simple and uniform MetaISA. The entire CAL layer represents over 94% of our code base. Thanks to MetaISA, CAL is now retargetable. To port ReTBLDTG to a new CPU platform, we only need to implement a new decoder and an ISA mapper for it.

As shown in Table 1, MetaISA adopts the little endian format and defines four types of registers with 128 non-aliased and interchangeable registers in each category. Our meta instructions are specified using the Semantic Specification Language (SSL) [13].

To facilitate the constraint collection, MetaISA has the following features.

Every meta instruction is a bit-precision assignment and the bit-width of RHS and LHS of a meta instruction is equal. In dynamic test generation systems, the conditional constraints are first collected from the symbolic execution, and then the constraints in meta instructions are transformed into logic conditional constraints and putted into the SMT solver. Therefore, the constraints in meta assignment instructions can be smoothly transformed into the logic == constraints. At the same time, bit-precision assignment instructions can precisely depicts how the left variable is computed from the right variables. Thus, the derived logic constraints can bit-precisely show how the collected constraints are affected by the input or medium variables. Additionally, SMT solver, used to generate a new input excising to a different control path based on the

Table 1. Meta Instruction Set Architecture

Endian	Little endian
32bits Regs	R_0,R_1,...,R_{100}
64bits Regs	A_0,A_1,...,A_{100}
80bits Float Regs	F_0,F_1,...,F_{100}
128bits Float Regs	X_0,X_1,...,X_{100}
Instuctions i	$Reg = f(Reg, mem\,[g(Reg)])$
	$mem\,[g_1(Reg)] = f(Reg, mem\,[g_2(Reg)])$
Operands v	Reg, Imm, $mem_{8,16,32,64}\,[g(Reg)]$
Operations $\triangle_t$	exact,sigext,zeroext,(...) ? (...):(...)
$\triangle_b$	+,-,*,/,$\oplus$,&,—,<<,>>,==,<,$\leq$
$\triangle_s$	$\neg$

gathered constraints, adopts *bit-vector* theory which demands all constraints expressed as bit-precision.

Every flag bit is defined as a register. Because input constraints are gathered from conditional statements encountered along the way, the branch instruction and the condition instruction pair should be recognized. The existing binary-level dynamic test generation tools, such as SAGE [7], search the most recent conditional instruction when meeting a branch instruction, in order to find the pair. However, compilation optimization of prefetching might insert a block of instructions between the pair [1], which makes it not an easy thing to recognize the branch instruction and the condition instruction pair. In our MetaISA, we define each flag bit, including `ZF` (Zero Flag), `OF` (Overflow Flag), `CF` (Carry Flag), and so on, as a register, called `FLAG REG`, and assign conditional constraints to corresponding `FLAG REGs`. Thus, how conditional instructions change the flag bit is kept in these registers. Searching the `FLAG REGs` we can get how the conditional constraints are affected from input or medium variables when meeting a branch instruction.

All general-purpose registers are non-aliased. If some register bits have alias, which causes two register identifiers point to the same register bits, it brings difficulty to the analysis, collection and solver of the conditional constraints. Therefore, it is a smart choice that the registers in the MetaISA are independent without any alias. And skillful work should be done for the CPU ISA that has aliased register bits.

All registers have the same width. Some source binary instructions, such as `mul, div` and `mod`, may involve registers of double-size. For example, the registers, used to hold the result of a 32-bit integer multiplied by a 32-bit integer, is of 64-bit width. And in 32-bit x86 ISA, the lower 32 bits of the result are kept in the destination register as the multiplication result; the higher 32 bits is used to compute the *Carry Flag* (`CF`) and *Overflow Flag*(`OF`), according to whether the multiplication operation is overflow or not. As discussed before, registers of the MetaISA with large width might bring about bad performance for SMT solver and huge memory requirement. Therefore, in our MetaISA, all instructions operate on the data with

the same width and all registers have the same bit-width. For the instructions involved with double-size registers, we use `exact`, `sigext` or `zeroext` operation to extend the intermediate results to necessary size.

Table 2 lists the map of `imul` and `idiv`, the representative instructions involved with the results of double-size, to our MetaISA.

Table 2. Mapping the instructions `imul` and `idiv` to our MetaISA

<table>
<tr><th>Ins.</th><th>Map to our MetaISA</th></tr>
<tr><td>imul ebx,edi</td><td>tmp1 := ebx
ebx := edi *! tmp1
CF := (((zeroext(edi,32,64) *! zeroext(tmp1,32,64)) == zeroext(ebx,32,64)) ? 0x0 : 0x1)[31:0] & 0x1
OF := (((zeroext(edi,32,64) *! zeroext(tmp1,32,64)) == zeroext(ebx,32,64)) ? 0x0 : 0x1)[31:0] & 0x1</td></tr>
<tr><td>idiv ecx</td><td>tmp1 := eax
eax := (((zeroext(edx,32,64) << 0x20) | zeroext(tmp1,32,64)) /! zeroext(ecx,32,64))[31:0]
edx := (((zeroext(edx,32,64) << 0x20) | zeroext(tmp1,32,64)) %! zeroext(ecx,32,64))[31:0]</td></tr>
</table>

5 Experimental Evaluation and Results

We have implemented our ReTBLDTG system in 208KLOC (Kilo Lines Of Code). And in order to demonstrate the effectiveness of our new approach on decoupling binary-level dynamic test generation from specific architecture details, we have retargeted ReTBLDTG to three different architectures, including 32-bit x86 ISA, PowerPC ISA and Sparc ISA by now. 32-bit x86 ISA falls into CISC; while PowerPC ISA and Sparc ISA belong to RISC.

We first demonstrate the importance of our ReTBLDTG on decoupling binary-level dynamic test generation from specific architecture details. Table 3 shows the workload for retargeting our ReTBLDTG to a new ISA, quantified with the number of LOC of the MetaISA Layer for each ISA. To retarget our ReTBLDTG to 32-bt x86 ISA, 2483 LOC are needed; to PowerPC ISA, 647 LOC are needed; and to Sparc ISA, 835 LOC are needed. Because 32-bt x86 ISA is CISC, a lot of work is needed for the complex instructions. At the same time, compared with the other two ISAs, 32-bt x86 ISA has more instructions. PowerPC ISA and Sparc ISA both belong to RISC. Sparc ISA has more instructions, and retargeting our ReTBLDTG to Sparc ISA needs more work.

The column *% Retarget Overheads* equals $\#LOC_{Retargeted} / \#LOC_{ReTBLDTG} \times 100\%$, where $\#LOC_{Retargeted}$ is the number of LOC for retargeting our ReTBLDTG to a ISA and $\#LOC_{ReTBLDTG} \times 100\%$ is the total number of LOC for building ReTBLDTG, nearly equal to 208K. This column demonstrates the easiness to retarget our binary-level dynamic test generation tool, ReTBLDTG, to a new ISA.

Table 3. Overheads for retargeting our ReTBLDTG to 32-bit x86 ISA, PowerPC ISA and Sparc ISA

ISA	#LOC	Retarget Overheads
32-bit X86	2483	1.19%
PowerPC	647	0.30%
Sparc	835	0.40%

Only 1.19% of the system has to be rewritten when our system is retargeted to the 32-bit x86 ISA, the most complex ISA of them. However, without the MetaISA, nearly the whole system has to be revised or rewritten. Therefore, our ReTBLDTG can be easily retargeted to any ISA with only a few overheads.

We then demonstrate the effectiveness of our new approach for hunting fatal bugs in benchmark and real-application binaries (without knowing their symbol tables). They are binaries for x86, PowerPC and Sparc ISA respectively and tested over Linux, Windows Vista and Windows Vista.

We show that ReTBLDTG can find 6 classic known bugs and we analyze the performance of ReTBLDTG. All experiments are carried out on an Intel 3.0 GHZ E8400 host PC running 32-bit Windows Vista with 4GB RAM. The VEL (i.e. virtual machine layer) of ReTBLDTG is an essentially wrapper for Simics 3.0.31, a high performance full-system simulator.

Table 4 shows the 6 benchmarks, Apache1, Apache2, OpenSER, MADWiFi, ANI and OldWINS. They are known to have one bug each. The first four small benchmarks are selected from the Verisec Security Benchmark suite [9] representing four different common scenarios causing buffer overflow errors. The binaries of `Apache1` and `Apache2` for PowerPC ISA are tested over Linux2.6; and the binaries of `OpenSER` and `MADWiFi` for Sparc ISA are tested over Linux2.6. The last two are real applications with one known security bug each, MS07-017 for the animated icons (ANI) parser in `user32.dll` of Windows Vista and MS04-045 in WINS Service in Windows 2000. The binaries of these two applications for 32-bit x86 ISA are tested over Windows Vista and Windows 2000, respectively.

ReTBLDTG has succeeded in finding all 6 known bugs in the six benchmarks. The results show that our ReTBLDTG can effectively find bugs for any ISAs over any OSes.

Table 5 gives the performance data for these applications. The first two rows show the generated and executed test cases. Only 3 to 1487 test cases are executed to find all these bugs. The third row gives the time for our ReTBLDTG

Table 4. Benchmarks. IoF stands for Integer Overflow and BoF stands for Buffer Overflow.

	Benchmark	ISA	OS	Bug Type
Verisec Security Benchmark Suite	Apache1: Apache-CVE-200-4 0940 (Full_Ptr_Bad)	PPC	Linux2.6	BoF by an Infinite Loop
	Apache2: Apache-CVE-2006-3747 Iter2 prefixLong_ptr_bad	PPC	Linux2.6	Off-by-One
	OpenSER: OpenSER-CVE-2006-6749 (Complete_Bad)	Sparc	Linux2.6	Lack of Bounds Checking
	MADWiFi: MADWiFi-CVE-2006-6332e (Ncode_Ie)	Sparc	Linux2.6	Unchecked Bounds in sprintf
Large Real Apps	ANI: User32.dll-MS-07-017	X86	Windows Vista	Failure to Validate Parameter
	OldWINS: WINS with MS04-045	X86	Windows 2000	Using Input as Pointer

Table 5. Performance results of the benchmarks

	Verisec Security Benchmark				Large Real Apps	
	Apache1	**Apache2**	**OpenSER**	**MADWiFi**	**ANI**	**OldWINS**
#Generated Test Cases	476	192	2125	5	797	807
#Executed Test Cases	131	192	1487	3	82	4
Total Time	4m12s	4m40s	29m47s	3m6s	2h3m29s	2m18s
Symbolic Execution	1m15s	16s	18s	1m35s	1m12s	5s
Concrete Execution	21s	46s	11m8s	<1s	7s	<1s
Constraint Solving	1m35s	2m27s	14m53s	55s	1h26m10s	1m32s
MetaISA Decoding	35s	46s	12s	1s	5m28s	8s
Test Case Database	26s	23s	3m18s	33s	32m32s	14s

to find the bugs. It only takes from 2m18 to 2h3m29s for our ReTBLDTG to find the bugs. The last five rows list the time distribution, which shows that our ReTBLDTG spends most time on symbolic execution, constraint collection and constraint solver. Our ReTBLDTG makes these time-consuming tasks ISA- and OS-independent, thus making dynamic test generation approaches efficiently find bugs from binaries for any ISAs over any OSes with only a few overheads.

6 Related Work

We refrain from discussing a large body of work done on static analysis, program verification, fuzz testing [5], dynamic taint analysis and model checking since good reviews are available in [6, 3, 14, 4, 7, 12, 10]. Instead, we focus mainly on a few techniques that are closely related to our work and that can also be used to test pre-release software in binary code.

Systematic dynamic test generation is becoming increasingly popular because it can find bugs by automatically generating test cases without false positives. DART [6], EXE [3], CUTE [14] and KLEE [4] are a few representatives. By operating on the source code only, these tools do not reason well about bugs that depend on, for example, heap layout at runtime. They represent tainted arrays symbolically (rather than with real addresses) and handle only some limited form of tainted pointers (e.g., scalar pointers only). Our ReTBLDTG has the similar working mechanism as them. But these techniques do not pay their attention to find bugs in binaries, but source code.

SAGE [7] is a dynamic test generation tool that works on Windows binaries. Research group from Berkeley [11] also works hard on finding integer bugs. However, they can only find a specific OS binaries for a specific ISA. And it is a hard and time-consuming work to retarget their techniques to other OSes or ISAs.

7 Conclusion

In this paper, we have introduced the problem of decoupling dynamic test generation from specific architecture and operating system details. We have presented a new binary-level dynamic test generation technique and a tool, ReTBLDTG. ReTBLDTG is based on the whole system virtual machine that provides OS-independent and fast *concrete execution* of the target program. And the execution trace is recorded, even without knowing the internal structure of guest OSes. We also design the MetaISA and map the execution trace to the MetaISA, thus making ReTBLDTG ISA-independent. We have implemented our ReTBLDTG, retargeted it to 32-bit x86, PowerPC, Sparc ISAs and Linux, Windows Vista, Windows Vista OSes, and used it to automatically find the six known bugs in the six benchmarks. Our results indicate that our ReTBLDTG can be easily retargeted to any ISA with only a few overheads and operate on any OSes; and ReTBLDTG can effectively expose bugs located deep within large applications for any ISAs over any OSes.

References

1. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Comput. Surv. 26(4), 345–420 (1994)
2. Bhansali, S., Chen, W., de Jong, S., Edwards, A., Murray, R., Drini, M., Mihoka, D., Chau, J.: Framework for instruction-level tracing and analysis of program executions. In: Proceedings of the 2nd international conference on Virtual execution environments, vol. 14, pp. 154–163 (2006)
3. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: automatically generating inputs of death. In: CCS 2006: Proceedings of the 13th ACM conference on Computer and communications security, pp. 322–335. ACM Press, New York (2006)
4. Cristian Cadar, D.E., Dunbar, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, San Diego, CA, USA (December 2008)
5. Godefroid, P.: Random testing for security: blackbox vs. whitebox fuzzing. In: RT 2007: Proceedings of the 2nd international workshop on Random testing, p. 1. ACM, New York (2007)
6. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. ACM SIGPLAN notices 40(6), 213–223 (2005)
7. Godefroid, P., Levin, M., Molnar, D.: Automated whitebox fuzz testing. In: Proceedings of the Network and Distributed System Security Symposium (2008)
8. Jones, S., Arpaci-Dusseau, A., Arpaci-Dusseau, R.: Antfarm: Tracking processes in a virtual machine environment. In: Proceedings of the USENIX Annual Technical Conference, USENIX 2006 (2006)
9. Ku, K., Hart, T.E., Chechik, M., Lie, D.: A buffer overflow benchmark for software model checkers. In: ASE 2007: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp. 389–392. ACM Press, New York (2007)
10. Lu, S., Zhou, P., Liu, W., Zhou, Y., Torrellas, J.: Pathexpander: Architectural support for increasing the path coverage of dynamic bug detection. In: MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, pp. 38–52. IEEE Computer Society Press, Los Alamitos (2006)
11. Molnar, D., Li, X., Wagner, D.: Dynamic test generation to find integer bugs in x86 binary linux programs
12. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. 42(6), 89–100 (2007)
13. Ramsey, N., Fernández, M.F.: Specifying representations of machine instructions. ACM Trans. Program. Lang. Syst. 19(3), 492–524 (1997)
14. Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for c. In: ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 263–272. ACM Press, New York (2005)