Dynamic Analysis and Debugging of Binary Code for Security Applications

Lixin Li[1] and Chao Wang[2]

[1] Battelle Memorial Institute, Arlington, Virginia, USA
[2] Department of ECE, Virginia Tech, Blacksburg, Virginia, USA

Abstract. Dynamic analysis techniques have made a significant impact in security practice, e.g. by automating some of the most tedious processes in detecting vulnerabilities. However, a significant gap remains between existing software tools and what many security applications demand. In this paper, we present our work on developing a *cross-platform interactive analysis* tool, which leverages techniques such as symbolic execution and taint tracking to analyze binary code on a range of platforms. The tool builds upon IDA, a popular reverse engineering platform, and provides a unified analysis engine to handle various instruction sets and operating systems. We have evaluated the tool on a set of real-world applications and shown that it can help identify the root causes of security vulnerabilities quickly.

1 Introduction

Dynamic and symbolic execution based techniques have made a significant impact on analyzing the binary code, e.g. to help automate some of the most tedious and yet non-trivial analysis in security practice. One example is white-box fuzzing [1], where the goal is to systematically generate test inputs to exercise all feasible program paths. Another example is taint analysis [2], where the goal is to track how tainted inputs propagate and trigger security vulnerabilities. In addition, these techniques have been used to detect a broad class of zero-day attacks [3, 4] and to generate vulnerability signatures [5] in a honey-pot.

Despite the aforementioned progress, however, there are major limitations in existing techniques that prevent them from being widely adopted. First, there is a lack of support for *interactive analysis*. Current research on dynamic binary analysis focuses primarily on fully automated methods, which is undoubtedly important for applications such as software testing. However, security applications such as malware analysis and exploitation analysis often cannot be fully automated. Although automated analysis can serve as the starting point of another round of deeper analysis, human in the loop is still indispensable. For example, an exhaustive white-box fuzzer can merely exercise all feasible program paths and identify the necessary conditions to trigger software bugs, but cannot decide whether the bugs are exploitable. To decide whether a bug is exploitable, the user needs to refine the input along that path to decide whether it is a security vulnerability. During this process, tools that support interactive analysis would be useful.

Second, there is a lack of support for *cross-platform analysis* by existing tools. This is a burning issue as well because software today runs on an increasingly diverse set of microprocessors and operating systems. Even if a software bug is exploitable on one platform – a specific combination of microprocessor and OS – it is not necessarily exploitable on a different platform, and vice versa. The reason is because a working exploit is often highly dependent on the runtime environment (stack layout, memory model, etc.). Similarly, effective protection, such as address space randomization (ASR), non-executable page, and stack/heap hardening, is also highly dependent on the runtime environment. Unfortunately, existing tools rarely support multiple platforms. For example, ARM based processors are popular in smart phones; many network routers and switchers use PowerPC and MIPS; and embedded devices often use some type of RISC chips. But existing dynamic analysis tools such as TEMU [6] and SAGE [1] focus only on the x86 instruction set.

To bridge the gap, we propose a unified framework for binary code analysis, to support both *interactive* analysis and *cross-platform* analysis. Interactive analysis allows for the user to make an assumption about the target program, and then quickly check for evidence that supports or contradicts that assumption. For example, the user can mark certain memory locations or registers as taint sources and then quickly check for other instructions that are either control-dependent or data-dependent on the taint sources. Since the user often needs to review the same execution scenario repeatedly, e.g. from different angles and in varying degree of details, our tool also supports trace replay augmented with dynamic slicing. Along certain program paths, the user can not only review what has happened but also perform *what-if* analysis: to see whether the program would behave differently if it were to take a different branch or input value. Such analysis is supported by applying *on-demand* symbolic execution using SMT solvers.

To support cross-platform analysis, we adopt a unified binary code intermediate representation (IR) of the target programs, and implement the core analysis algorithms on this IR. We also develop various reverse engineering tools that translate the native execution traces of the program into this IR. Since core analysis algorithms such as symbolic execution and taint analysis are made architecture-independent and OS-independent, the maintenance cost is significantly reduced. This is in sharp contrast to most existing tools, which are all tied to specific instruction set architectures (ISAs) and operating systems (OSs). In our approach, native execution traces from different platforms, together with the native program state, are captured and then translated into the architecture-independent IR. Similarly, the analysis results are mapped back to the native platforms before they are presented to the user.

To the best of our knowledge, such cross-platform interactive analysis framework does not exist before. In addition to symbolic execution and taint analysis, our tool supports *deterministic replay*. More specifically, at the operating system layer, we use a *generic debug breakpoint* based mechanism [7] to support trace generation in user mode, kernel mode, and on real devices. It allows us to avoid the limitations of the existing dynamic binary instrumentation (DBI) tools [8, 9] and whole-system emulators [10]. Although there exist many replay systems for binary programs (e.g. [11]), they do not seem to integrate well with mainstream security analysis tools and do not support interactive analysis. For example, there are tools that extend the debugger `gdb` to support replay [12], but do not support taint analysis. Reverse engineering tools such as IDA [13] also support replay but not taint analysis. Without taint analysis, replay itself does not provide enough information about the data relations critical for security analysts. Typically, security analysts need to construct the data flow relations manually.

We have implemented the cross-platform interactive analysis system in the popular IDA Pro tool. New features such as symbolic execution, taint tracking, and replay have been integrated seamlessly with the existing features of IDA Pro. We have evaluated the new tool on a set of real world applications with known vulnerabilities, and demonstrated the effectiveness of the tool.

The remainder of the paper is organized as follows. We provide an overview of our tool in Section 2, and present the cross-platform symbolic execution engine, called CBASS, in Section 3. We present the interactive taint analysis engine, called TREE, in Section 4. We present our experimental evaluation in Section 5, review related work in Section 6, and then give our conclusions in Section 7.

2 System Overview

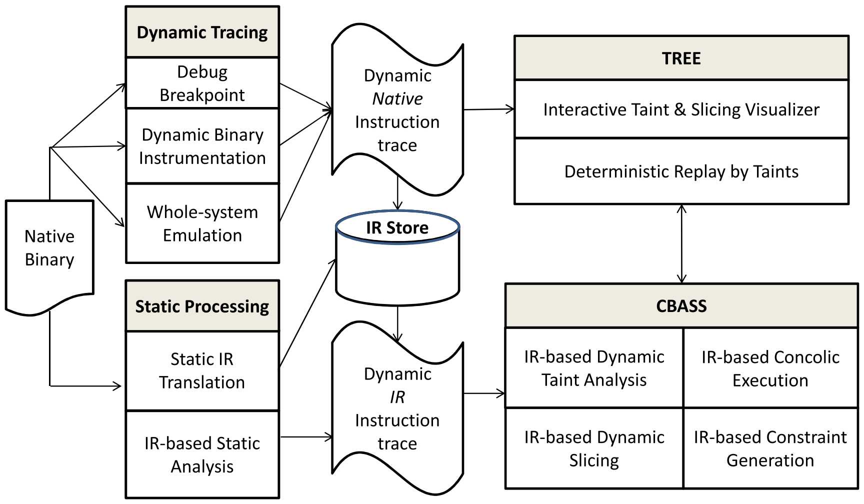

Dynamic Tracing
Debug Breakpoint
Dynamic Binary Instrumentation
Whole-system Emulation
Native Binary
Static Processing
Static IR Translation
IR-based Static Analysis
Dynamic *Native* Instruction trace
IR Store
Dynamic *IR* Instruction trace
TREE
Interactive Taint & Slicing Visualizer
Deterministic Replay by Taints
CBASS
IR-based Dynamic Taint Analysis
IR-based Concolic Execution
IR-based Dynamic Slicing
IR-based Constraint Generation

Fig. 1. The Architecture of our Cross-platform Interactive Analysis System

The proposed system, shown in Fig. 1, consists of the following subsystems:

- CBASS (Cross-platform Binary Automated Symbolic execution System), which separates the platform dependent execution trace generation process from the platform independent analysis process.
- TREE (Taint-enabled Reverse Engineering Environment), which provides a unified replay, debugging, and taint tracking environment, allowing security analysts to form a hypothesis and then check it interactively.
- Front-end subsystems that support both *static processing* and *dynamic tracing*. They translate native traces from different platforms to the common intermediate representation (IR) and map the analysis results back.

We provide a brief description of *static processing* and *dynamic tracing* in this section, while postponing CBASS and TREE to Sections 3 and 4, respectively.

Static processing and dynamic tracing are crucial components for supporting cross-platform analysis at the instruction set architecture (ISA) level and the operating system (OS) level. ISAs often differ significantly in their encoding and semantics of the instructions. Operating systems often differ in how they use registers to represent high-level data structures. For example, Windows and Linux use `fs` and `gs` segment registers for very different purposes. In our system, however, these differences are mostly removed due to the use of a common IR. In the front-end, only a thin layer needs to deal with remaining subtle differences. In the back-end, all core analysis algorithms are based on the common IR.

We shall use the program called `basicov_plus.exe` in Fig. 2 as the running example. It reads the data inputs from a file and adds each input byte, except for the last two, with its right neighboring byte. If the first byte is `'b'`, the transformed bytes are fed to a vulnerable function called `StackOverflow`. The function is vulnerable in that, if the input is larger than a local buffer inside the function, there will be a buffer overflow, causing the return address to be overwritten. Although the program is small, it consists of all the important elements of a typical security vulnerability: the potentially tainted data source (input), the transformation (addition), the trigger (path condition), and the anomaly manifestation (buffer overflow). In practice, of course, each of these elements can be significantly more complex. For example, the transformation itself may involve not just one instruction but a few millions of instructions.

```
//INPUT
DWORD dwBytesRead;
ReadFile(hFile, sBigBuf, 16, &dwBytesRead, NULL);

//INPUT TRANSFORMATION
for(int i=0; i< (dwBytesRead-2); i++)
    sBigBuf[i] +=sBigBuf[i+1];

//PATH CONDITION
If(sBigBuf[0]=='b')
//Vulnerable Function
    StackOverflow(sBigBuf,dwBytesRead);
```

```
void StackOverflow(char *sBig,int num)
{
    char sBuf[8]= {0}; //Small Local buffer
    for(int i=0;i<num;i++) //Oveflow when num>8
    {
        sBuf[i] = sBig[i];
    }
    return;
}
```

Fig. 2. Example: A Conditional Buffer Overflow Program

Static Processing There are two main components for static processing. One component is responsible for pre-processing the binary code statically and building a map from each native instruction to a set of IR instructions. Another component consists of a set of simple static analysis on the resulting IR, e.g. to identify interesting locations that are potential targets of the subsequent dynamic analysis.

Table 1 shows the mapping from a few instructions used by the program in Fig. 2 to the IR instructions. In this table, the native x86 instructions are shown in the first column. The corresponding IR translations are shown in the second column. For example, the native x86 instruction at the address `0x00401073` is mapped to the sequence of REIL instructions from the imaginary address `0x0040107300` to the imaginary address `0x0040107306`. We postpone our detailed presentation of the IR format, called REIL (for reverse engineering intermediate language), to next section. For now, we only show the mapping.

After the REIL IR is constructed, a set of simple static analysis may be conducted. For example, one analysis may be used to measure the Cyclomatic complexity of each function in the IR. The cyclomatic complexity is believed to be useful in identifying a set of functions where bugs most likely hide. Another analysis may be used to detect loops heuristically and annotate the loop counters whenever possible. This is useful because loops, as well as recursive call sites, are places where out-of-bound buffer accesses and non-termination most likely occur.

Table 1. The Mapping from Native Instructions to REIL IR Instructions

Native Instruction (x86)	REIL IR Instruction
00401073 movsx edx, byte ss:[ebp-10]	`40107300: add [DWORD FFFFFFF0, DWORD ebp, QWORD t0]`
	`40107301: and [QWORD t0, DWORD FFFFFFFF, DWORD t1]`
	`40107302: ldm [DWORD t1, EMPTY , BYTE t2]`
	`40107303: xor [BYTE t2, BYTE 0x80, BYTE t3]`
	`40107304: sub [BYTE t3, BYTE 0x80, DWORD t4]`
	`40107305: and [DWORD t4, BYTE FFFFFFFF, BYTE t5]`
	`40107306: str [DWORD t5, EMPTY , DWORD edx]`
00401077 cmp edx, 0x62	`40107700: and [DWORD edx, DWORD 0x80000000, DWORD t0]`
	`40107701: and [DWORD 98, DWORD 0x80000000, DWORD t1]`
	`40107702: sub [DWORD edx, DWORD 98, QWORD t2]`
	`40107703: and [QWORD t2, QWORD 0x80000000, DWORD t3]`
	`40107704: bsh [DWORD t3, DWORD -31, BYTE SF]`
	`40107705: xor [DWORD t0, DWORD t1, DWORD t4]`
	`40107706: xor [DWORD t0, DWORD t3, DWORD t5]`
	`40107707: and [DWORD t4, DWORD t5, DWORD t6]`
	`40107708: bsh [DWORD t6, DWORD -31, BYTE OF]`
	`40107709: and [QWORD t2, QWORD 0x100000000, QWORD t7]`
	`4010770A: bsh [QWORD t7, QWORD -32, BYTE CF]`
	`4010770B: and [QWORD t2, QWORD FFFFFFFF, DWORD t8]`
	`4010770C: bisz [DWORD t8, EMPTY , BYTE ZF]`
0040107a jnz loc_40108e	`40107A00: bisz [BYTE ZF, EMPTY , BYTE t0]`
	`40107A01: jcc [BYTE t0, EMPTY , DWORD 0x40108e]`

Dynamic Tracing There are three main components for dynamic tracing. Together, they are responsible for generating a logged execution trace, which will be the starting point of the subsequent offline analysis. Notice that, in our system, there is a clear separation between *online* trace generation and *offline* trace analysis. This makes our trace analysis as platform independent as possible. Among the existing binary analysis tools, some have adopted online analysis [6, 14], meaning that the analysis takes place at the time the program is executed, while others have adopted offline analysis [1], meaning that the trace is captured and then analyzed later. However, all of them are tied to a particular platform, making it difficult to maintain and extend to a different platform. In contrast, our system does not have such problems.

In Fig.1, the components labeled Dynamic Binary Instrumentation and Whole-system Emulation implement the two popular approaches adopted by many existing tools. However, these two components alone doe not meet the demand of our system, for the following reasons. Popular DBI tools, such as PIN and DynamoRIO, provide user mode x86 binary instrumentation but do not support non-x86 ISAs. Valgrind supports non-x86 ISAs such as ARM, PowerPC, and MIPS, but runs only on Linux. None of them provides kernel mode instrumentation. Whole-system emulators can provide kernel instrumentation, but often through an additional instrumentation layer that is not portable to new versions. For example, tools built on the QEMU simulator, such as TEMU [6], DroidScope [15], and S2E [14], have different instrumentation layers. In each case, the implementation is tied to a specific microcode used by QEMU, making it difficult to port. Therefore, although it is well-known that Android builds upon a customized version of QEMU, porting the aforementioned tools to Android is challenging.

In contrast, we propose to use the *debug breakpoint* mechanism [7] for dynamic tracing. This mechanism, already used by interactive debuggers such as `gdb`, is supported by almost all processors and operating systems. Therefore, it provides a unified approach for collecting execution traces from different platforms. It can collect traces in kernel mode. It can also collect traces on real devices such as Cisco routers and Android phones, since almost all of these devices have development tools that provide the breakpoint capability. This *debug breakpoint* approach has significant advantages over DBI tools. Running inside the target process, DBI tools often disturb the behavior of the target program, e.g. by affecting the target's stack and heap layout. This is a serious problem because *interesting* scenarios in security applications tend to manifest only in certain program states.

Our experience shows that breakpoint based tracing is effective for short and interactive analysis. To support long traces, our system leverages existing DBI tools and whole-system emulators, e.g. PIN plug-in for Windows/Linux x86 for trace generation. We have implemented a heuristic algorithm to automatically switch between these techniques, in order to use the best instruction tracer available in each individual application scenario.

Trace Format The execution trace starts with a snapshot of the program state, which consists of the module, thread, stack, and heap information. The program state is a valuation of the set R of registers for all threads, including privileged registers for kernel mode, and a global memory map M. Therefore, we have the program state represented as $PS = \{R, M\}$.

A tracer on a particular platform would record the finite sequence of *events* starting from the initial state. An event is an execution instance of an instruction that transforms the program state PS into a new program state PS'. Each event in the trace has a unique sequence number. The vast majority of events in a trace are of the form `I = {instInfo, threadID,relevantRegisters, memoryAccess}`, where `instInfo` contains the address of the instruction, the encoding bytes, and the size, `threadId` is the index of the thread that executes this instruction, `relevantRegisters` and `memoryAccess` contain values of the related registers and memory elements before this instruction is executed.

Trace can be optimized to reduce the size while maintaining the same amount of information required by the subsequent analysis. In our implementation, we record only the information that is relevant to the subsequent analysis. For example, for instruction **movsx edx, byte ss:[ebp-10]**, our trace includes the values of registers **edx** and **ebp**. For user mode analysis, we capture the precondition and postcondition of each system call or call to a standard library function as a function summary, to avoid recording the large number of instructions inside the function. For example, after a call to `ReadFile`, we record the address of the input buffer, the input size, and the content of the buffer.

3 Cross-Platform Binary Symbolic Execution System (CBASS)

In contrast to existing symbolic execution tools, where the core analysis algorithms are tied to specific DBI tools or whole-system emulators, CBASS performs symbolic execution on the platform independent REIL IR. This is advantageous because any enhancement to the core analysis algorithms would automatically benefit all platforms.

3.1 The REIL IR

REIL stands for Reverse Engineering Intermediate Language [16]. It is a platform independent intermediate representation of disassembled code, originally designed for supporting static code analysis. We adopt REIL in our system for three reasons:

- Translators for statically mapping the native instruction set to REIL IR are readily available for most of the ISAs, including x86, ARM, PowerPC, and MIPS.
- The REIL instructions are sufficiently close to native instructions on most platforms and therefore can be used to preserve the native register state easily.
- The semantics of REIL instructions can be encoded in SMT formulas precisely by using the bit-vector theory, and therefore is amendable to symbolic analysis.

REIL has only seventeen instructions, each of which has a simple effect on the program state. Each REIL instruction has three operands. The first two operands are always the *source* operands and the last operand is always the *destination* operand. One or more of the operands can be empty. Table 2 summarizes the seventeen REIL instructions. For a more detailed description of REIL, please refer to the online document [17].

Table 2. The REIL Instructions and Their Semantics

Category	**REIL Instruction**	**Semantics**
Arithmetic	ADD s1, s2, d	d = s1 $+$ s2
	SUB s1, s2, d	d = s1 s2
	MUL s1, s2, d	d = s1 $*$ s2
	DIV s1, s2, d	d = s1$/$s2
	MOD s1, s2, d	d = s1 mod s2
	BSH s1, s2, d	if s2>0 d = s1$*2^{s2}$
		else d = s1 /2^{-s2}
Bitwise	AND s1, s2, d	d = s1& s2
	OR s1, s2, d	d = s1 $\mid$ s2
	XOR s1, s2, d	d = s1 xor s2
Logical	BISZ s1, $\not\epsilon$, d	if s1 = 0, d = 1 else d = 0
	JCC s1, $\not\epsilon$, d	iff s1$\neq$0, set eip = d
Transfer	LDM s1, $\not\epsilon$, d	d = mem[s1]
	STM s1, $\not\epsilon$, d	mem[d] =s1
	STR s1, $\not\epsilon$, d	d = s1
Other	NOP, $\not\epsilon$, $\not\epsilon$, $\not\epsilon$	No op
	UNDEF $\not\epsilon$, $\not\epsilon$,d	Undefined instruction
	UNKN $\not\epsilon$, $\not\epsilon$, $\not\epsilon$	Unknown instruction

Designed for reverse engineering purposes, REIL provides the support to statically translate native instructions in x86, ARM, PowerPC, and MIPS to their IR equivalents for an instruction, a function, or the entire program. More importantly, REIL provides a one-to-one mapping of the native instruction address to the imaginary IR address. For example, in Table 1, the x86 instruction **movsx edx, byte ss:[ebp-10]** at address `0x401073` will always be mapped to a list of REIL instructions from `0040107300` to `0040107305`. Therefore, it is easy to map the analysis results back to the native forms before reporting them to the user.

REIL has a simple *register-based* architecture, which can keep native registers and create temporary registers when needed. Preserving native registers is particularly useful for implementing the offline concrete and symbolic (or concolic) execution. Recall that in concolic execution, the program state has to be saved during trace generation and later reconstructed during the offline analysis. At runtime, our trace generator will only save the native program state (related native registers and global memory). During the offline analysis, we can compute the IR program state directly from these native registers and the memory.

In all of the seventeen REIL instructions, the *destination* operand can be represented by a mathematical or logical formula of the *source* operands. Consider the second native instruction **00401077 cmp edx, 0x62** in Table 1. Notice that the REIL instructions use a few basic mathematical and logical operations to precisely compute all the `eflags`; in other words, all the `eflags` can be represented as an expression in terms of `edx` and `0x62`. For example, `ZF = (edx  98) and 0xffffffff`. In some sense, REIL instructions are compatible with the input language of the satisfiability modulo theory (SMT) solver Z3 [18], which supports the theories of integers, bit-vectors, and arrays.

3.2 Symbolic Execution

The symbolic execution procedure consists of three steps:

1. *Mark taint source and symbolize its value.* Here, taint sources refer to the untrusted data in the target program. When a program variable is marked as a taint source, our tool symbolizes the variable, by replacing its concrete value with a symbolic one (a free variable). Traditionally, the taint sources are program inputs. However, during *interactive* security analysis, the user may be interested in tracking other program variables as well. For example, some sensitive data items such as the password and the registry key may become the focus of the analysis. At any time during the program execution, CBASS can mark any byte in any register or at any memory location as the taint source.
2. *Symbolic execution of REIL instructions.* CBASS implements the symbolic execution engine based on the REIL IR. As we have already mentioned, the semantics of REIL instructions can be close to that of the input language of the SMT solvers. Therefore, the symbolic encoding procedure, which takes an IR trace as input and returns an SMT formula, is straightforward. In our implementation of the proposed system, we have used the Z3 SMT solver, which is capable of solving formulas expressed in the theories of bit-vectors and arrays.
3. *Check taint sink to construct constraint.* Depending on the application, security analysts may mark different memory location or register at some interesting point as the taint sinks. For example, to generate potential exploits, the taint sinks are usually registers such as `EIP`. We may create a constraint to steer the execution into a desired code section and make `EIP` equals to the address of that code section. To detect vulnerabilities, the taint sinks are usually the unexplored branches. When we

encounter a branch instruction, we create a path condition if the branch predicate is tainted by a symbolic input.

As shown in Table 2, there are four categories of REIL instructions directly related to symbolic execution. Mathematical and logical instructions perform the corresponding operations on constants, registers, or memory. Memory instructions handle memory read or write operations, which propagate values between registers and memory. Control instructions decide where to jump if the branch conditions are true. During symbolic execution, we use a *concrete and symbolic memory (CSM)* map to represent the memory state. It has both the concrete value and the symbolic value. For memory instructions, if the address is symbolic, also called a symbolic pointer, we have to under-approximate it by using the concrete value derived from the actual execution trace.

3.3 The Running Example

We use the instructions in Table 1 to demonstrate how to construct a path condition during symbolic execution and how to generate the SMT formula. As the IR instructions are fed to the symbolic execution engine, CBASS creates symbolic variables for the taint sources and constructs the symbolic expressions. For each IR instruction, it creates a new symbolic expression for the destination operand if any of the source operands is symbolic. If all the source operands have concrete values, then it uses the concrete value for the destination operand.

Table 3. Example: The REIL IR based Symbolic Execution

Native Instructions	REIL Instructions	Symbolic Execution, with ebp = 0x12ff84 and mem[12ff74] = INPUT
00401073 movsx edx, byte ss:[ebp-10]	40107300: add [DWORD FFFFFFF0, DWORD ebp, QWORD t0]	t0 = 0x12ff84+0xfffffff0 = 10012ff74
	40107301: and [QWORD t0, DWORD FFFFFFFF, DWORD t1]	t1 = t0 and 0xffffffff =0x12ff74
	40107302: ldm [DWORD t1, EMPTY , BYTE t2]	t2 = mem[t1] =INPUT_VAR[8]
	40107303: xor [BYTE t2, BYTE 0x80, BYTE t3]	t3 = INPUT_VAR[8] xor 0x80
	40107304: sub [BYTE t3, BYTE 0x80, DWORD t4]	t4 = (INPUT_VAR[8] xor 0x80) -0x80
	40107305: and [DWORD t4, BYTE FFFFFFFF, BYTE t5]	t5 = ((INPUT_VAR[8] xor 0x80) -0x80) and 0xffffffff
	40107306: str [DWORD t5, EMPTY , DWORD edx]	edx = ((INPUT_VAR[8] xor 0x80) -0x80) and 0xffffffff
00401077 cmp edx, 0x62	40107700: and [DWORD edx, DWORD 0x80000000, DWORD t0]	t0 = (((INPUT_VAR[8] xor 0x80) -0x80) and 0xffffffff)and 0x80000000
	40107701: and [DWORD 98, DWORD 0x80000000, DWORD t1]	t1 = 98 and 0x80000000 = 98
	40107702: sub [DWORD edx, DWORD 98, QWORD t2]	t2 = (((INPUT_VAR[8] xor 0x80) -0x80) and 0xffffffff) - 98
	Ignore irrelevant temps ...	...
	4010770B: and [QWORD t2, QWORD FFFFFFFF, DWORD t8]	t8= ((((INPUT_VAR[8] xor 0x80) -0x80) and 0xffffffff) 98) and 0xffffffff
	4010770C: bisz [DWORD t8, EMPTY , BYTE ZF]	ZF = ite(t8==0,1,0)
0040107a jnz loc_40108e	40107A00: bisz [BYTE ZF, EMPTY , BYTE t0]	t0 = ite(ZF==0,1,0)
	40107A01: jcc [BYTE t0, EMPTY , DWORD 0x40108e]	eip = ite(t0==1,0x40108e,0x40107c)

Table 3 shows the symbolic execution of the REIL instructions of the three native x86 instructions. Notice that each native instruction is mapped to a sequence of REIL instructions. The REIL instructions take the native registers and memory values as input, transform them by using intermediate registers, and return the results back to the native registers and memory. For example, the instruction at `0x401073` has the native register **ebp** and memory value at address `0x12ff74` as input, and the native register **edx** as output. Just before executing the instruction, the concrete value of **ebp** is assumed to be `0x12ff84` and the memory at the address `0x12ff74` has a symbolic value. From the first two REIL instructions, we have `t1 = 0x12ff74`. The `ldm` instruction sets `t2 = mem[0x12ff74]`, which contains a symbolic value, and then `t2 = INPUT_VAR[8]`.

After carrying out the symbolic execution as shown in Table 3, the branch condition before executing **0040107a jnz loc_40108e** becomes `ite(ite(((((INPUT_VAR[8] xor 0x80) -0x80) and 0xffffffff)  98) and 0xffffffff)`. This is equivalent to the SMT formula shown in Fig. 3. By negating the path condition and asking the SMT solver for a satisfying solution, we can compute the new input value to be `98`, which corresponds to `sBigBuf[0] == b` in the original code in Fig. 2.

```
(set-logic QF_AUFBV)

(declare-fun _INPUT_VAR () (_ BitVec 8))
(declare-fun EXPR_0 () (_ BitVec 32))
(assert (= EXPR_0 (bvsub ((_ sign_extend 24) (bvxor _ INPUT_VAR (_ bv128 8))) (_ bv4294967168 32))))
(assert (= (ite (not (= (ite (not (= (bvand ((_ extract 63 0) (bvsub ((_ sign_extend 32) (bvand ((_ extract 31 0) EXPR_0)
(_ bv4294967295 32))) (_ bv98 64))) (_ bv4294967295 64)) (_ bv0 64))) ( _ bv1 32) ( _ bv0 32)) (_ bv0 32))) ( _ bv1
8) ( _ bv0 8)) (_ bv0 8)))

(check-sat)

(get-value (_INPUT_VAR))
```

Fig. 3. Example: The Path Constraints in Z3 SMT Formula

4 Taint-Enabled Reverse Engineering Environment (TREE)

To unleash the analysis power of CBASS in security practice, we need to support *interactive* analysis. Toward this end, we have developed the infrastructure that can (1) generate REIL traces on demand, (2) visualize the analysis results on demand, (3) perform taint tracking on demand. Together, these new features form the basis of our taint-enabled reverse engineering environment (TREE).

4.1 Interactive Trace Generation

TREE leverages existing features of IDA, a popular reverse engineering tool, to support on-demand trace generation. IDA is a widely used tool in mainstream security practice. It has become the *de facto* standard tool for conducting vulnerability and malware analysis. IDA can statically disassemble binary code on more than 50 processors and support a wide range of operating systems.

We have implemented the *debug breakpoint* based trace collection framework in IDA and integrated it seamlessly with the existing features of IDA. Our experience shows that the debug breakpoint based approach works well in supporting interactive trace generation, which typically involves short traces. For lengthy traces and large interactive sessions, we rely on the traces generated from the more traditional DBI tools such as PIN, and whole-system emulators such as QEMU.

Compared to the existing tools, the dynamic trace generator in TREE has the following features:

– *Interactive tracing:* The user can select a starting point and an end point at any time during the analysis and request the tool to conduct a deeper analysis on a relatively short trace segment. This feature can be used by security analysts to quickly verify or refute a hypothesis.
– *Kernel tracing:* The trace generator in TREE can generate traces on any platform that supports `windbg` and `gdb` server, allowing kernel mode traces to be generated from both Windows and Linux.
– *Mobile tracing:* The trace generator in TREE can generate traces on Android/ARM platforms through IDA's debug agent. IDA supports real devices such as Android phones and tablets. IDA also supports some versions of iPhone, Windows CE, and Symbian OS, although these platforms have not been integrated with TREE.

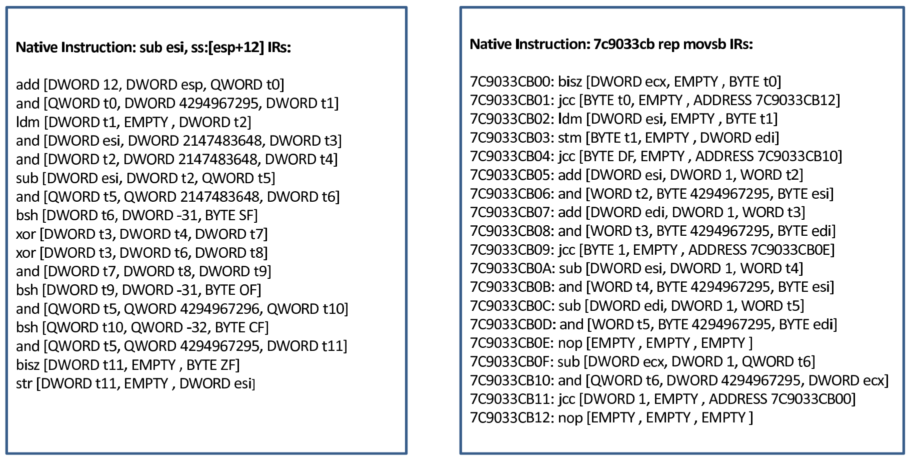
4.2 On-Demand Taint Analysis

Broadly speaking, taint dependencies fall into three categories: data dependency, address dependency, and control dependency.

– Data dependency means that the taint source affects the taint sink through data movement, mathematical operations, or logical operations. The value of the taint source often directly affects the value of the taint sink.
– Address dependency means that the taint source affects the taint sink through its address for read or write, but the taint source does not directly affect the value of the taint sink. One example for address dependency is the use of a tainted data as the index to access a look-up table. Without tracking the address dependency, we would lose track of the tainted data after such a table lookup.
– Control dependency is a form of implicit information flow. Although it can happen in benign programs, it is often more deliberately used by malware. It can be of the form `if x =0 then y=0 else y=1`. If `x` is tainted, the value of `y` is dependent of `x`. But there is no direct link between the value of `x` and the value of `y`.

In security analysis, it is often challenging to keep track of all three types of dependencies. In the remainder of this section, we will show how TREE can make it easier.

The main difficulty in taint tracking for the x86 instruction set is to handle the large number of instructions and their variants, since these native instructions often have complex side effects. REIL provides a unified framework for capturing these side effects, e.g. by breaking down a native x86 instruction into a sequence of simple REIL instructions. Notice that there are only seventeen REIL instructions. Furthermore, each REIL instruction has only one effect, making taint tracking easy to implement. Fig. 4 (1) shows a comparison of the native x86 instructions and the corresponding REIL instructions. The REIL instructions capture the side effects of the native instructions on `eflags` including `SF`, `OF`, `CF` and `ZF`.



```
Native Instruction: sub esi, ss:[esp+12] IRs:

add [DWORD 12, DWORD esp, QWORD t0]
and [QWORD t0, DWORD 4294967295, DWORD t1]
ldm [DWORD t1, EMPTY , DWORD t2]
and [DWORD esi, DWORD 2147483648, DWORD t3]
and [DWORD t2, DWORD 2147483648, DWORD t4]
sub [DWORD esi, DWORD t2, QWORD t5]
and [QWORD t5, QWORD 2147483648, DWORD t6]
bsh [DWORD t6, DWORD -31, BYTE SF]
xor [DWORD t3, DWORD t4, DWORD t7]
xor [DWORD t3, DWORD t6, DWORD t8]
and [DWORD t7, DWORD t8, DWORD t9]
bsh [DWORD t9, DWORD -31, BYTE OF]
and [QWORD t5, QWORD 4294967296, QWORD t10]
bsh [QWORD t10, QWORD -32, BYTE CF]
and [QWORD t5, QWORD 4294967295, DWORD t11]
bisz [DWORD t11, EMPTY , BYTE ZF]
str [DWORD t11, EMPTY , DWORD esi]
```

(1) REIL IR makes side effects explicit

```
Native Instruction: 7c9033cb rep movsb IRs:

7C9033CB00: bisz [DWORD ecx, EMPTY , BYTE t0]
7C9033CB01: jcc [BYTE t0, EMPTY , ADDRESS 7C9033CB12]
7C9033CB02: ldm [DWORD esi, EMPTY , BYTE t1]
7C9033CB03: stm [BYTE t1, EMPTY , DWORD edi]
7C9033CB04: jcc [BYTE DF, EMPTY , ADDRESS 7C9033CB10]
7C9033CB05: add [DWORD esi, DWORD 1, WORD t2]
7C9033CB06: and [WORD t2, BYTE 4294967295, BYTE esi]
7C9033CB07: add [DWORD edi, DWORD 1, WORD t3]
7C9033CB08: and [WORD t3, BYTE 4294967295, BYTE edi]
7C9033CB09: jcc [BYTE 1, EMPTY , ADDRESS 7C9033CB0E]
7C9033CB0A: sub [DWORD esi, DWORD 1, WORD t4]
7C9033CB0B: and [WORD t4, BYTE 4294967295, BYTE esi]
7C9033CB0C: sub [DWORD edi, DWORD 1, WORD t5]
7C9033CB0D: and [WORD t5, BYTE 4294967295, BYTE edi]
7C9033CB0E: nop [EMPTY , EMPTY , EMPTY ]
7C9033CB0F: sub [DWORD ecx, DWORD 1, QWORD t6]
7C9033CB10: and [QWORD t6, DWORD 4294967295, DWORD ecx]
7C9033CB11: jcc [DWORD 1, EMPTY , ADDRESS 7C9033CB00]
7C9033CB12: nop [EMPTY , EMPTY , EMPTY ]
```

(2) REIL IR static analysis for complex native instructions

Fig. 4. TREE Uses REIL IR for Comprehensive Taint Analysis

REIL also supports static analysis that can provide hints for dynamic analysis. They can be useful for x86 instructions that have embedded conditions or loop structures. For example, `cmpxchg` compares the values in the `AL`, `AX` or `EAX` registers with the *destination* operand, and depending on the comparison result, different operands may be loaded into the *destination* operand. Some x86 instructions with prefix such as `rep` behave like a loop. Fig. 4 (2) shows the REIL instructions for x86 instruction `rep movsb`. Since dynamic analysis can only follow one path at a time, in general, it cannot handle the branch and loop dependency. However, a conservative static analysis on REIL IR often can reveal the branch and loop structure. This is the case for `rep movsb` where such analysis can identify `ecx` as the loop counter. We have incorporated such analysis into our REIL-based dynamic taint analysis.

We use the same example for CBASS symbolic execution to show the major steps in dynamic taint analysis. Fig. 5 shows the details of this algorithm. After merging the temporary register nodes, the final taint graph for native instructions is shown in the last column of this table.

4.3 Replay with Taint-enabled Breakpoints

In an interactive analysis session, the user may want to scrutinize a particular program behavior repeatedly. TREE provides a replay mechanism to support such analysis. One application is to reconstruct the execution states. Comparing to tools such as `gdb` and `IDA`, the replay mechanism in TREE is significantly more powerful. For example, it allows the user to break at any tainted points, after the user marks the initial taint source and specifies the type of impact (taint policy). This new feature of *break by data relation* is key to interactive analysis. It essentially allows the user to break at any point that she is interested, without the need to construct the chain of events mentally. In addition, TREE can presents the chain of events within the proper semantic context visually.

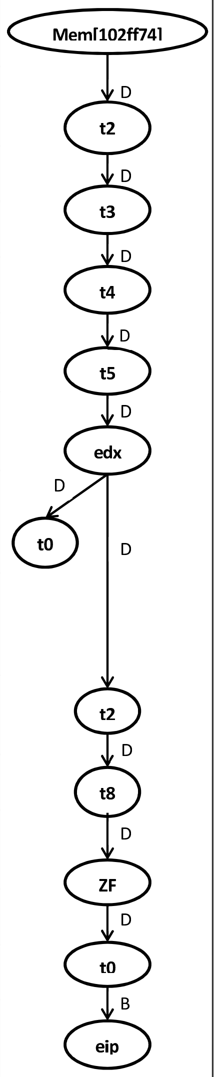
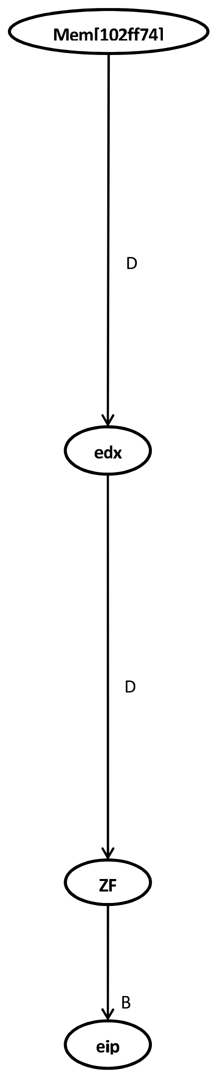
Native Instruction	IR Instructions	IR Taint Graph	Native Taint Graph
00401073 movsx edx, byte ss:[ebp-10]	add [DWORD FFFFFFF0, DWORD ebp, QWORD t0]	 Mem[102ff74] D t2 D t3 D t4 D t5 D edx D t0 D t2 D t8 D ZF D t0 B eip	
	and [QWORD t0, DWORD FFFFFFFF, DWORD t1]		
	ldm [DWORD t1, EMPTY , BYTE t2]		
	xor [BYTE t2, BYTE 0x80, BYTE t3]		
	sub [BYTE t3, BYTE 0x80, DWORD t4]		
	and [DWORD t4, BYTE FFFFFFFF, BYTE t5]		
	str [DWORD t5, EMPTY , DWORD edx]		
00401077 cmp edx, 0x62	and [DWORD edx, DWORD 0x80000000, DWORD t0]		
	and [DWORD 98, DWORD 0x80000000, DWORD t1]		
	sub [DWORD edx, DWORD 98, QWORD t2]		
	Ignore irrelevant temps …		
	and [QWORD t2, QWORD FFFFFFFF, DWORD t8]		
	bisz [DWORD t8, EMPTY , BYTE ZF]		
0040107a jnz loc_40108e	bisz [BYTE ZF, EMPTY , BYTE t0]		
	jcc [BYTE t0, EMPTY , DWORD 0x40108e]		

Fig. 5. Example: Dynamic Taint Analysis

We illustrate the replay process by using the same buffer overflow example in Fig. 2. When this program runs with a 16-byte input that triggers the `StackOverflow` function, the input bytes at offsets 13 to 16 would overwrite the `EIP` bytes. This chain of events can be tracked by TREE, for which a user-clickable graph is shown in Fig. 6. In this graph, each node represents a byte, annotated by its transformation instruction and followed by its edge type. D is the default edge type that stands for data dependency. The first byte of `EIP` (id =207) is overwritten by input bytes 13 and 14 (id=13,14) after a few steps.

First, these two bytes are added to form a new byte at memory `mem_0x14fe1c(id =159)`. Then the byte is moved to a local buffer at `0x14fdfc` and overflowed the buffer at function `stackOverflow()`. When the call to this function returns, the byte, at the top of the stack at `mem_0x14fdfc[id=196]` is popped into the first byte of register `EIP [id =207]`. For this trivial example, there are already `477` instructions logged in the trace, but only 8 unique instructions are involved in the handling of the input bytes. In such cases, the taint graph allows the user to focus on the most relevant set of instructions quickly.

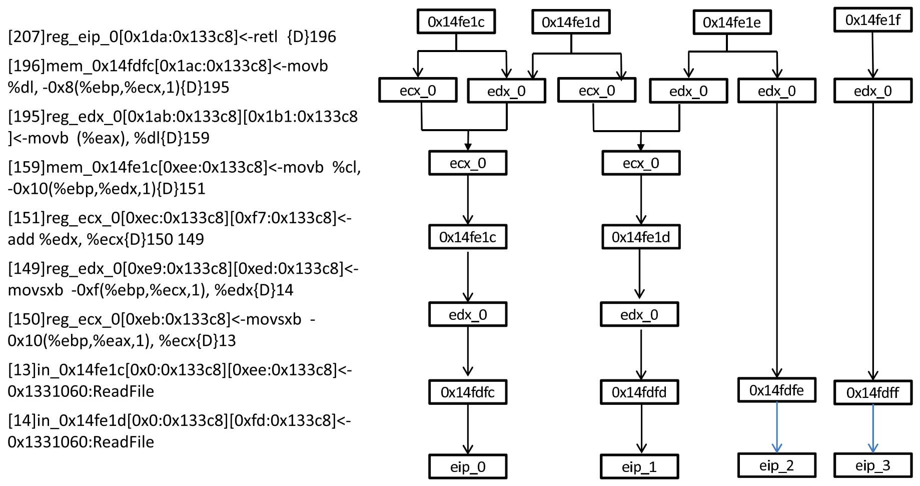

```
[207]reg_eip_0[0x1da:0x133c8]<-retl  {D}196
[196]mem_0x14fdfc[0x1ac:0x133c8]<-movb
%dl, -0x8(%ebp,%ecx,1){D}195
[195]reg_edx_0[0x1ab:0x133c8][0x1b1:0x133c8
]<-movb  (%eax), %dl{D}159
[159]mem_0x14fe1c[0xee:0x133c8]<-movb  %cl,
-0x10(%ebp,%edx,1){D}151
[151]reg_ecx_0[0xec:0x133c8][0xf7:0x133c8]<-
add %edx, %ecx{D}150 149
[149]reg_edx_0[0xe9:0x133c8][0xed:0x133c8]<-
movsxb  -0xf(%ebp,%ecx,1), %edx{D}14
[150]reg_ecx_0[0xeb:0x133c8]<-movsxb  -
0x10(%ebp,%eax,1), %ecx{D}13
[13]in_0x14fe1c[0x0:0x133c8][0xee:0x133c8]<-
0x1331060:ReadFile
[14]in_0x14fe1d[0x0:0x133c8][0xfd:0x133c8]<-
0x1331060:ReadFile
```

Fig. 6. Taint Graph and Visualization of Running Example

5 Evaluation

We have implemented the proposed *cross-platform interactive analysis* system using the client/server architecture. More specifically, CBASS runs as the back-end server, responding to requests from the front-end. It shares the REIL IR with TREE. TREE is responsible for handling OS level differences and mapping the analysis results back to the native instruction context. The client/server architecture enables parallel development and optimization of CBASS and TREE, and makes it easy to port either subsystem to a different platform without affecting the other.

Currently, CBASS and TREE are able to run on Windows and Linux, and support target programs running on the x86 and Android/ARM platforms. CBASS is written in Jython, a Python-based language that can access Java objects and call Java libraries. CBASS interfaces with REIL through the REIL Java library for IR translation. TREE is implemented as an `IDA Pro` plug-in. TREE also uses Qt/Pyside and extends the IDA graph to support a number of visualization features and user interaction. During the process of developing TREE, we have found a number of bugs in both IDA and REIL related tools. In most cases, the IDA and REIL developers have responded to our bug reports promptly and provided fixes in their latest releases.

In the remainder of this section, we will first provide an overview of our detailed evaluation and then present a case study with a real-world application. Together, they demonstrate the effectiveness of our system in supporting cross-platform interactive security analysis.

5.1 Overview

We have conducted two sets of experiments. The first set consists of unit level tests for the CBASS and TREE subsystems. The second set consists of case studies using real-world applications. At the unit testing level, we have used a large number of binary programs (each around 100 LOC) to check if the core analysis algorithms in TREE/CBASS are implemented correctly. We have designed various transformation functions to process the input (taint source) and created the corresponding test oracles to ensure that TREE and CBASS produce correct results. The test programs are compiled on different platforms (Windows, Linux, and Android) using different compilers (VC, GCC) with various optimization settings. This also allows us to evaluate the effectiveness of our front-end subsystems, which are crucial for the cross-platform analysis.

With real-world applications, the goal of our case study is to evaluate the effectiveness of TREE/CBASS in analyzing vulnerabilities. More specifically, we would like to know whether security analysts, armed with our tool, can quickly discover the chain of critical events leading to the real vulnerability. Toward this end, we have selected a set of Windows/Linux applications with known vulnerabilities. Table 4 shows the statistics of the benchmark programs. In the following, we shall briefly describe each vulnerability and then focus on using WMF (CVE-2005-4560) to explain in details how TREE/CBASS can help reduce the analysis time required to identify the root cause.

The first two columns in Table 4 show the application name, version, and vulnerability identifier. Both the WMF (CVE-2005-4560) and the ANI (CVE-2007-0038) vulnerabilities were present on many Windows versions prior to Windows Vista, and could be triggered by applications including Picture and Fax Viewers, Internet Explorer, Windows Explorer, and various email viewers. Audio Code 0.8.18 has a buffer overflow vulnerability that can be triggered when adding a crafted play list (.lst) file. This vulnerability can enable arbitrary code execution. Streamcast 0.9.75 has a stack buffer overflow, allowing attackers to use the http `User-Agent` field to overwrite the return address of a function call. POP Peeper 3.4.0.0, an email agent, has a vulnerability in its `From` field, where the stack buffer can overflow to overwrite the return address and the Windows Structural Exception Handler (SEH). PEiD is a popular tool for detecting packers, cryptors and compilers found in PE executable files. A carefully crafted EXE file can be used to exploit this vulnerability to run arbitrary code. SoulSeek 157 NS12d, a free file sharing application, has a vulnerability that can be remotely exploited to overwrite SEH. SoX (Sound eXchange) is a sound processing application in Linux. Its `WAV`

Table 4. Results of Our Analysis on Real World Vulnerabilities

Program Name and Version	Vulnerability Identifier	Binary Code and Trace Size(KB)	Taint Sources (Byte)	Total/Unique Instructions	Total/Unique Tainted Inst.
GDI32.dll 5.1.2600.2180	CVE-2005-4560	272 / 2,422	68	76,618 / 5,677	206 / 115
User32.dll 5.1.2600.2180	CVE-2007-0038	564 / 53,548	4,385	250,534 / 23,868	7,195 / 1,043
AudioCoder 0.8.18	OSVDB-2939	731 / 29,000	620	473,922 / 27,265	12,666 / 66
Streamcast 0.9.75	CVE-2008-0550	804 / 26,541	1,230	83,204 / 3,354	8,351 / 35
POP Peeper 3.4.0.0	BugTraq-34192	1,436 / 68,731	400	182,382 / 8,226	1,106 / 2
PEiD 0.95	OSVDB-94542	214 / 14,163	1,000	32,779 / 9,501	25 / 20
SoulSeek 157	ExploitDB-8777	3,410/147,931	49	4,435,526/142,220	217/121
SoX 12.17.2	CVE-2004-0557	225 /14,441	1,184	180,034 / 2,801	56,138 / 647

header handling code has a known buffer overflow vulnerability that can be exploited by the attacker to execute arbitrary code.

The third column in Table 4 shows the size of the binary code and the size of the trace, respectively. Recall that the on-demand trace logging starts when the target program reads the taint source (input in all these test cases), and stops when the tainted data have taken control of program, e.g. when `EIP` contains a tainted value or the program jumps to the tainted memory location. The fourth column shows the number of bytes of the taint sources, ranging from a few dozen bytes to a few thousand bytes. For all cases, CBASS/TREE can successfully build the taint graph previously described.

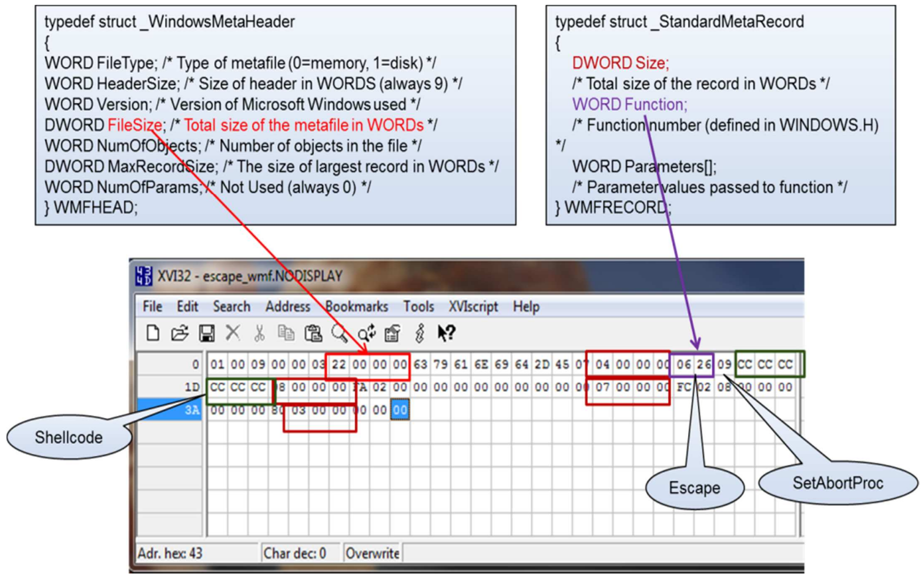
For any specific taint sink, the CBASS/TREE system can generate a slice of the tainted instructions from the taint sources to the taint sink. The last two columns in Table 4 show the total and unique instructions in the trace, and the total and unique tainted instructions for all the tainted sources and sinks, respectively. In general, tainted instructions are only a very small portion of the total instructions (<5%). For any specific byte of the tainted target, for example, a tainted register or a tainted memory location, usually only a few dozen tainted instructions are involved.

For more real world vulnerabilities to which we have applied TREE/CBASS, please refer to http://code.google.com/p/tree-cbass/. We will continue our ongoing evaluation process and update the results on this website.

5.2 Case Study: WMF (CVE-2005-4560)

In this section, we will illustrate how TREE/CBASS can support interactive security analysis by using CVE-2005-4560, also known as the `WMF SetAbortProc Escape` vulnerability. WMF stands for Windows Metafile Format. The formal specification of WMF is very complex. In short, the overall WMF file structure has one meta header, followed by zero or more meta records. The key structure of the WMF file format is shown in Fig. 7.

Each meta data record is an encoded Windows GDI (Graphics Device Interface) function call. It is a means of storing and playing back the command sequence that normally would be sent to GDI to display graphics. Among the meta records, one type is called the *escape* record. Although this type of record is deprecated, the code that handles the record has not been removed in a timely fashion. If an *escape* record contains



```
typedef struct _WindowsMetaHeader
{
WORD FileType; /* Type of metafile (0=memory, 1=disk) */
WORD HeaderSize; /* Size of header in WORDS (always 9) */
WORD Version; /* Version of Microsoft Windows used */
DWORD FileSize; /* Total size of the metafile in WORDs */
WORD NumOfObjects; /* Number of objects in the file */
DWORD MaxRecordSize; /* The size of largest record in WORDs */
WORD NumOfParams; /* Not Used (always 0) */
} WMFHEAD;
```

```
typedef struct _StandardMetaRecord
{
   DWORD Size;
   /* Total size of the record in WORDs */
   WORD Function;
   /* Function number (defined in WINDOWS.H)
*/
   WORD Parameters[];
   /* Parameter values passed to function */
} WMFRECORD;
```

```
 0  01 00 09 00 00 03 22 00 00 00 63 79 61 6E 69 64 2D 45 07 04 00 00 00 06 26 09 CC CC CC
1D  CC CC CC 08 00 00 00 FA 02 00 00 00 00 00 00 00 00 00 00 07 00 00 00 FC 02 08 00 00 00
3A  00 00 00 80 03 00 00 00 00 00
```

XVI32 - escape_wmf.NODISPLAY
File Edit Search Address Bookmarks Tools XVIscript Help
Adr. hex: 43
Char dec: 0
Overwrite
Shellcode
Escape
SetAbortProc

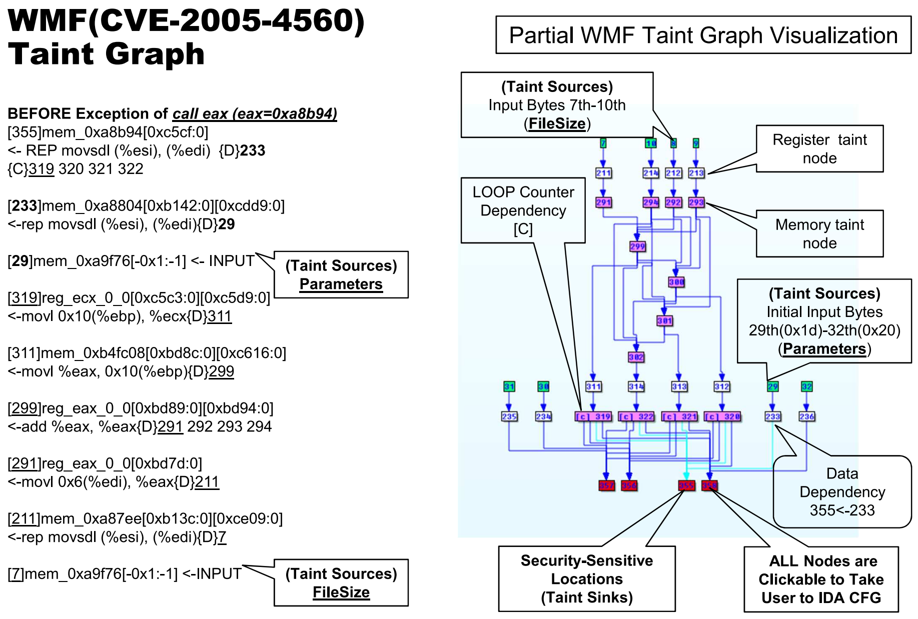
Fig. 7. Case Study: The WMF Key Data Structures

certain values for the Function (0626) and Parameters (09) fields defined in the WMFRECORD structure, the SETABORTPROC escape will inform GDI to call a function provided in the file. This vulnerability allows remote attackers to execute arbitrary code via a WMF format image with a crafted SETABORTPROC GDI Escape function call, related to the Windows Picture and Fax Viewer (SHIMGVW.DLL). It is relatively easy to craft a WMF image file and cause the viewer application to crash.

The lower part of Fig. 7 shows an WMF file with 68 bytes. From the time the viewer program finishes reading the file to the point where an exception happens, 76,618 instructions would be executed. Given that most people do not know WMF format well, we can assume that it is difficult to manually identify which bytes of the WMF file are responsible for the crash, how many instructions are directly involved in rendering the file, from which functions, and under what condition. Without such information, it would be difficult to understand the root cause of this vulnerability. From the exploit development point of view, it would not be obvious which input bytes are critical to a working exploit, and what are the constraints a working exploit must satisfy.

With the dynamic analysis techniques provided by TREE/CBASS, we are able to answer the aforementioned questions in a few minutes. More specifically, the tool can generate a trace that leads to the crash. Furthermore, it can replay the trace by first marking the whole 68 bytes of the file as the taint sources, and then stopping at the tainted points. From the taint graph, we are able to see the connection between the instruction that caused the crash (call `eax` where `eax = 0xa8b94`) and some of the file structures. We have identified 12 unique instructions in WMF that are directly related to moving and processing the file and causing the application to crash. Since our tool can generate an interactive graph, the user can navigate along the chained data and instructions by clicking on each tainted node in the graph.

Fig. 8 shows part of the WMF crash taint graph. The right side is a snapshot taken from the TREE GUI. The nodes in green show the taint source bytes (WMF file), and the nodes in red show the bytes pointed by `eax` in the `call eax` instruction that caused an exception. The left side of the figure shows some internal text representation of the taint graph. For example, the node `355` shows the tainted node of `0xa8b94`. Following the `D` link (highlighted in bold), we can see that it is data-dependent on node 233, which in turn is data-dependent on node 29, an input byte that corresponds to part of the shell-code section. Following the `C` link (highlighted by underline), we can see that it is affected by a loop whose iteration number depends on the values from the 7th to the 10th bytes in the WMF file. When looking back at the `WMFHead` structure, we find that bytes 7-10 actually correspond to the `FileSize` field.



WMF(CVE-2005-4560) Taint Graph

```
BEFORE Exception of call eax (eax=0xa8b94)
[355]mem_0xa8b94[0xc5cf:0]
<- REP movsdl (%esi), (%edi)  {D}233
{C}319 320 321 322

[233]mem_0xa8804[0xb142:0][0xcdd9:0]
<-rep movsdl (%esi), (%edi){D}29

[29]mem_0xa9f76[-0x1:-1] <- INPUT

[319]reg_ecx_0_0[0xc5c3:0][0xc5d9:0]
<-movl 0x10(%ebp), %ecx{D}311

[311]mem_0xb4fc08[0xbd8c:0][0xc616:0]
<-movl %eax, 0x10(%ebp){D}299

[299]reg_eax_0_0[0xbd89:0][0xbd94:0]
<-add %eax, %eax{D}291 292 293 294

[291]reg_eax_0_0[0xbd7d:0]
<-movl 0x6(%edi), %eax{D}211

[211]mem_0xa87ee[0xb13c:0][0xce09:0]
<-rep movsdl (%esi), (%edi){D}7

[7]mem_0xa9f76[-0x1:-1] <-INPUT
```

(Taint Sources) Parameters
(Taint Sources) FileSize
Partial WMF Taint Graph Visualization
(Taint Sources) Input Bytes 7th-10th (FileSize)
Register taint node
LOOP Counter Dependency [C]
Memory taint node
(Taint Sources) Initial Input Bytes 29th(0x1d)-32th(0x20) (Parameters)
Data Dependency 355<-233
Security-Sensitive Locations (Taint Sinks)
ALL Nodes are Clickable to Take User to IDA CFG

Fig. 8. Case Study: The WMF Crash and Taint Graph

6 Related Work

Independently, Heelan and Gianni [19] have explored the idea of supporting manual vulnerability detection in their work called Pinnacle. However, Pinnacle is limited to taint tracking on the x86 instruction set only. In contrast, our system can handle binary code from multiple platforms. Furthermore, our interactive analysis is significantly broader than the scope of Pinnacle, including not only vulnerability analysis but also exploitation analysis and malware analysis. Our system also supports symbolic execution and replay, which Pinnacle does not. Among the offline binary analysis tools, SAGE [1] is the closest to ours. However, SAGE is designed primarily for white-box fuzzing and works only for the x86 instruction set. It does not focus on interactive analysis and does not support multiple platforms.

Since dynamic taint analysis is independent of the vulnerability specific details, it can analyze a broad class of attacks controllable via input. Therefore, it has become a popular technique for detecting attacks such as buffer overflow and control-flow hijacking. However, online taint analysis often has high runtime overhead and requires intrusive code instrumentation. To make taint analysis more efficient for online intrusion detection, Sekar proposed taint inference [20] for web applications by using approximate string match. Li and Sekar [21] later demonstrated that taint inference could be used to detect buffer-overflow attacks in low-level binary code.

Dytan [2] extended the data-flow based taint tracking to also include control dependency, and developed a framework to support the x86 instruction set. Ganai *et al.* [22] extended this framework to support multithreaded applications. Predictive dynamic analysis provides a new way of conducting trace-based analysis for multithreaded applications [23]. It can detect not only security vulnerabilities in the observed execution traces, but also security vulnerabilities that may appear in some alternative thread interleavings. Wang and Ganai [24] developed a tool for predicting concurrency failures in the generalized execution traces of x86 executables.

Newsome and Song proposed TaintCheck [4], which used dynamic taint analysis for detecting vulnerabilities and for generating vulnerability signatures. TaintCheck was implemented using Valgrind [9]. Portokalidis *et al.* developed Argos [5] based on QEMU to generate fingerprints for zero-day attacks. However, none of these existing tools supports cross-platform interactive security analysis.

7 Conclusions

We have presented a *cross-platform interactive analysis* framework, which integrates state-of-the-art dynamic analysis techniques with a mainstream reverse engineering tool to meet the demand in security practice. Our framework, comprising CBASS and TREE, supports interactive analysis through on-demand symbolic execution and taint tracking. It also supports cross-platform analysis, by separating online trace generation from offline trace analysis and by using a reverse engineering intermediate representation. We have implemented the proposed framework and conducted some preliminary experimental evaluation. Our results have demonstrated its effectiveness in identifying root causes of security vulnerabilities in real applications.

8 Acknowledgments

We would like to thank James Just for his guidance throughout this project, Xing Li for implementing the IDA debug breakpoint based tracer, and Loc Nguyen for implementing the taint graph visualization. We would like to thank Ilfak Guilfanov and the IDA team for promptly fixing the bugs that we have reported to them and for their suggestions on the GUI integration. We would like to thank Thomas Dullien and Tim Kornau of the Google Zynamics team for making their latest version of REIL available to us. The second author is supported in part by the NSF grant CCF-1149454 and the ONR grant N00014-13-1-0527.

References

1. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Network and Distributed System Security Symposium. (2008)
2. Clause, J.A., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: ISSTA. (2007) 196–206
3. Costa, M., Crowcroft, J., Castro, M., Rowstron, A.I.T., Zhou, L., Zhang, L., Barham, P.: Vigilante: End-to-end containment of internet worm epidemics. ACM Trans. Comput. Syst. **26**(4) (2008)
4. Newsome, J., Song, D.X.: Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In: NDSS. (2005)
5. Portokalidis, G., Slowinska, A., Bos, H.: Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In: EuroSys. (2006) 15–27
6. Song, D.X., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A new approach to computer security via binary analysis. In: International Conference on Information Systems Security. (2008) 1–25
7. Paxson, V., et al.: A survey of support for implementing debuggers. Available from ftp.ee.lbl.gov: papers/debugger-support.ps.Z (1990)
8. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: PIN: Building customized program analysis tools with dynamic instrumentation. In: PLDI. (2005) 190–200
9. Nethercote, N., Seward, J.: Valgrind: A program supervision framework. Electr. Notes Theor. Comput. Sci. **89**(2) (2003)
10. Bellard, F.: QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, FREENIX Track. (2005) 41–46
11. Bhansali, S., Chen, W.K., De Jong, S., Edwards, A., Murray, R., Drinić, M., Mihočka, D., Chau, J.: Framework for instruction-level tracing and analysis of program executions. In: International Conference on Virtual execution environments, ACM (2006) 154–163
12. GNU GDB: Process Record & Replay. http://sourceware.org/gdb/wiki/ProcessRecord
13. Eagle, C.: The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler, San Francisco, CA, USA (2008)
14. Chipounov, V., Kuznetsov, V., Candea, G.: The s2e platform: Design, implementation, and applications. ACM Trans. Comput. Syst. **30**(1) (2012) 2
15. Yan, L.K., Yin, H.: DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic android malware analysis. In: USENIX Security. (2012) 29–29
16. Dullien, T., Porst, S.: REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In: CanSecWest. (2009)
17. REIL: URL: http://www.zynamics.com/binnavi/manual/html/reil_language.htm
18. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer (2008) 337–340
19. Heelan, S., Gianni, A.: Augmenting vulnerability analysis of binary code. In: Annual Computer Security Applications Conference. (2012) 199–208
20. Sekar, R.: An efficient black-box technique for defeating web application attacks. In: NDSS. (2009)
21. Li, L., Just, J.E., Sekar, R.: Online signature generation for windows systems. In: Annual Computer Security Applications Conference. (2009) 289–298
22. Ganai, M.K., Lee, D., Gupta, A.: DTAM: dynamic taint analysis of multi-threaded programs for relevancy. In: FSE. (2012)
23. Wang, C., Kundu, S., Limaye, R., Ganai, M., Gupta, A.: Symbolic predictive analysis for concurrent programs. Int. J. Formal Aspects of Computing (April 2011) 1–25
24. Wang, C., Ganai, M.: Predicting concurrency failures in generalized traces of x86 executables. In: International Conference on Runtime Verification. (2011)