BitBlaze: A New Approach to Computer Security via Binary Analysis

Dawn Song[1], David Brumley[2], Heng Yin[1,2,3], Juan Caballero[1,2], Ivan Jager[2], Min Gyung Kang[1,2], Zhenkai Liang[2], James Newsome[2], Pongsin Poosankam[1,2], and Prateek Saxena[1]

[1] UC Berkeley
[2] Carnegie Mellon University
[3] College of William and Mary

Abstract. In this paper, we give an overview of the BitBlaze project, a new approach to computer security via binary analysis. In particular, BitBlaze focuses on building a unified binary analysis platform and using it to provide novel solutions to a broad spectrum of different security problems. The binary analysis platform is designed to enable accurate analysis, provide an extensible architecture, and combines static and dynamic analysis as well as program verification techniques to satisfy the common needs of security applications. By extracting security-related properties from binary programs directly, BitBlaze enables a principled, root-cause based approach to computer security, offering novel and effective solutions, as demonstrated with over a dozen different security applications.

Keywords: Binary analysis, malware analysis and defense, vulnerability analysis and defense, reverse engineering.

1 Introduction

In BitBlaze, we propose a new approach to computer security via binary analysis. In particular, we make the simple and yet important observation that for many security problems, their root cause and the key to their solutions lie directly in the relevant programs (e.g., vulnerable programs and malicious code). Thus, by designing and developing techniques and tools to automatically extract security-related properties from programs and devise solutions based on them, we can enable a principled approach to many security problems, focusing on their root cause, and offer more effective solutions than previous approaches which rely on heuristics or symptoms of an attack.

To enable the above approach, one technical challenge is that for security applications, we often need to directly deal with binary code. For many programs such as common off-the-shelf (COTS) programs, users do not have access to their source code. For malicious code attacks, attackers simply do not attach the source code with the attack. And binary code is what gets executed, and hence analyzing binary code will give the ground truth important for security applications whereas analyzing source code may give the wrong results due to compiler errors and optimizations. However, analyzing binary code is commonly considered as an extremely challenging task due to its complexity and the lack of higher-level semantic information. As a result, very few tools

R. Sekar and A.K. Pujari (Eds.): ICISS 2008, LNCS 5352, pp. 1–25, 2008.

exist for binary analysis that are powerful enough for general security applications, and this has been a big hurdle preventing security researchers and practitioners from taking the aforementioned root-cause based approach.

Thus, the above observations have motivated us to conduct the *BitBlaze* project, building a unified binary analysis platform and using it to provide novel solutions to a broad spectrum of different security problems. In this paper, we give an overview of the two main research foci of BitBlaze: (1) the design and development of a unified, extensible binary analysis infrastructure for security applications; (2) novel solutions to address a spectrum of different security problems by taking a principled, root-cause based approach enabled by our binary analysis infrastructure.

In the rest of this paper, we describe the challenges and design rationale behind the BitBlaze Binary Analysis Platform and its overall architecture, and then describe its three main components *Vine* , *TEMU*, and *Rudder*. Finally, we give an overview of the different security applications that we have enabled using the BitBlaze Binary Analysis Platform and discuss some related work.

2 The Architecture of the BitBlaze Binary Analysis Platform

In this section, we first describe the challenges of binary analysis for security applications, then the desired properties of a binary analysis platform catering to security applications, and finally outline the architecture of the BitBlaze Binary Analysis Platform.

2.1 Challenges

There are several main challenges for binary code analysis, some of which are specific to security applications.

Complexity. The first major challenge for binary analysis is that binary code is complex. Binary analysis needs to model this complexity accurately in order for the analysis itself to be accurate. However, the sheer number and complexity of instructions in modern architectures makes accurate modeling a significant challenge. Popular modern architectures typically have hundreds of different instructions, with new ones added at each processor revision. Further, each instruction can have complex semantics, such as single instruction loops, instructions which behave differently based upon their operand values, and implicit side effects such as setting processor flags. For example, the IA-32 manuals describing the semantics of x86 weigh over 11 pounds.

As an example, consider the problem of determining the control flow in the following x86 assembly program:

```
// instruction dst, src
add a, b    // a = a+b
shl a, x    // a << x
jz target     // jump if zero to address target
```

The first instruction, `add a,b`, computes `a := a+b`. The second instruction, `shl a,x`, computes `a := a << x`. The last instruction, `jz a`, jumps to address `a` if the processor zero flag is set.

One problem is that both the `add` and `shl` instruction have implicit side effects. Both instructions calculate up to *six* other bits of information that are stored as processor status flags. In particular, they calculate whether the result is zero, the parity of the result, whether there is an auxiliary carry, whether the result is signed, and whether an overflow has occurred.

Conditional control flow, such as the `jz` instruction, is determined by the implicitly calculated processor flags. Thus, either the `add` instruction calculates the zero flag, or the `shl` will. However, which instruction, `add` or `shl`, determines whether the branch is taken? Answering this question is not straight-forward. The `shl` instruction behaves *differently* depending upon the operands: it only updates the zero flag if `x` is not zero.

Lack of Higher-Level Semantics. The second major challenge is that binary code is different than source code, and in particular, lacking higher-level semantics present in source code. Thus, we need to adapt and develop program analysis techniques and tools that are suitable for the setting of binary code (where debugging information is often unavailable). In particular, binary code lacks abstractions that are often fundamental to source code and source code analysis, as shown in the following examples:

- **No Functions.** The function abstraction does not exist at the binary level. Instead, control flow in a binary program is performed by jumps. For example, the x86 instruction `call x` is just shorthand for storing the current instruction pointer (register `eip`) at the address named by the register `esp`, decrementing `esp` by the architecture word size, then loading the `eip` with number `x`. Indeed, it is perfectly valid in assembly, and sometimes happens in practice, that code may call into the middle of a "function", or have a single "function" separated into non-contiguous pieces.
- **Memory vs. Buffers.** Binary code does not have buffers, it has *memory*. While the OS may determine a particular memory page is not valid, memory does not have the semantics of a user-specified type and size. One implication of the difference between buffers and memory is that in binary code there is no such thing as a buffer overflow. While we may say a particular store violates a higher-level semantics given by the source code, such facts are inferences with respect to the higher-level semantics, not part of the binary code itself.
- **No Types.** New types cannot be created or used since there is no such thing as a type constructor in binary code. The only types available are those provided by the hardware: registers and memory. Even register types are not necessarily informative, since it is common to store values from one register type (e.g., 32-bit register) and read them as another (e.g., 8-bit register).

Overall, an assembly-specific approach is unattractive because writing analysis over modern complex instruction tends to be tedious and error-prone. Verifying a program analysis is correct over such a large and complicated instruction set seems even more difficult. Further, an assembly-specific approach is specific to a single architecture. All analysis would have to be ported each time we want to consider a new architecture. Thus, analysis could not take advantage of the common semantics across many different assembly languages.

Whole-System View. Many security applications requires the ability to analyze operations in the operating system kernel and interactions between multiple processes, thus require a whole-system view, presenting greater challenges than in traditional single-program analysis.

Code Obfuscation. Some security applications require analyzing malicious code. Malicious code may employ anti-analysis techniques such as code packing, encryption, and obfuscation to make program analysis difficult, posing greater challenges than analyzing benign programs.

2.2 Design Rationale

The goal of the BitBlaze Binary Analysis Platform is to design and develop techniques and the core utilities that cater the common needs of security applications and enable others to build upon and develop new solutions to security problems more easily and effectively. Given the aforementioned challenges, we have a few design guidelines motivating the architecture of the BitBlaze Binary Analysis Platform:

Accuracy. We would like to enable accurate analysis, motivating us to build precise, formal models of instructions that allow the tool to accurately model the program execution behavior symbolically.

Extensibility. Given the complexity of binary analysis, we would like to develop core utilities which can then be re-used and easily extended to enable other more sophisticated analysis on binaries, or easily re-targeted to different architectures.

Fusion of Static and Dynamic Analysis. Static and dynamic analysis both have advantages and disadvantages. Static analysis can give more complete results as it covers different execution paths, however, it may be difficult due to the complexity of pointer aliasing, the prevalence of indirect jumps, and the lack of types and other higher-level abstractions in binaries. Even telling what is code and what is data statically is an undecidable problem in general. Moreover, it is particularly challenging for static analysis to deal with dynamically generated code and other anti-static-analysis techniques employed in malicious code. Furthermore, certain instructions such as kernel and floating point instructions may be extremely challenging to accurately model. On the other hand, dynamic analysis naturally avoid many of the difficulties that static analysis needs to face, at the cost of analyzing one path at a time. Thus, we would like to combine static and dynamic analysis whenever possible to have the benefits of both.

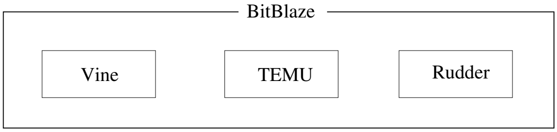

Fig. 1. The BitBlaze Binary Analysis Platform Overview

2.3 Architecture

Motivated by the aforementioned challenges and design rationale, the BitBlaze Binary Analysis Platform is composed of three components: *Vine*, the static analysis component, *TEMU*, the dynamic analysis component, and *Rudder*, the mixed concrete and symbolic analysis component combining dynamic and static analysis, as shown in Figure 1.

Vine translates assembly to a simple, formally specified intermediate language (IL) and provides a set of core utilities for common static analysis on the IL, such as control flow, data flow, optimization, symbolic execution, and weakest precondition calculation.

TEMU performs whole-system dynamic analysis, enabling whole-system fine-grained monitoring and dynamic binary instrumentation. It provides a set of core utilities for extracting OS-level semantics, user-defined dynamic taint analysis, and a clean plug-in interface for user-defined activities.

Rudder uses the core functionalities provided by Vine and TEMU to enable mixed concrete and symbolic execution at the binary level. For a given program execution path, it identifies the symbolic path predicates that symbolic inputs need to satisfy to follow the program path. By querying solvers such as decision procedures, it can determine whether the path is feasible and what inputs could lead the program execution to follow the given path. Thus, Rudder can automatically generate inputs leading program execution down different paths, exploring different parts of the program execution space. Rudder provides a set of core utilities and interfaces enabling users to snapshot and reload the exploration state and provide user-specified path selection policies.

3 Vine: The Static Analysis Component

In this section, we give an overview of Vine, the static analysis component of BitBlaze Binary Analysis Platform, describing its intermediate language (IL), its front end and back end components, and implementation.

3.1 Vine Overview

Figure 2 shows a high-level picture of Vine. The Vine static analysis component is divided into a platform-specific front-end and a platform-independent back-end. At the core of Vine is a platform-independent intermediate language (IL) for assembly. The IL is designed as a small and formally specified language that faithfully represents the assembly languages. Assembly instructions in the underlying architecture are lifted up to the Vine IL via the Vine front-end. All back-end analyses are performed on the platform-independent IL. Thus, program analyses can be written in an architecture-independent fashion and do not need to directly deal with the complexity of an instruction set such as x86. This design also provides extensibility—users can easily write their own analysis on the IL by building on top of the core utilities provided in Vine.

The Vine front-end currently supports translating x86 [23] and ARMv4 [4] to the IL. It uses a set of third-party libraries to parse different binary formats and produce assembly. The assembly is then translated into the Vine IL in a syntax-directed manner.

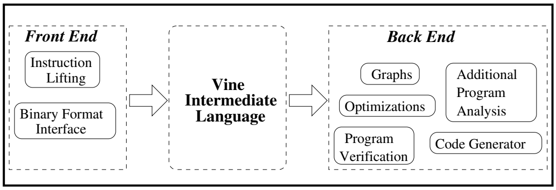

Fig. 2. Vine Overview

The Vine back-end supports a variety of core program analysis utilities. The back-end has utilities for creating a variety of different graphs, such as control flow and program dependence graphs. The back-end also provides an optimization framework. The optimization framework is usually used to simplify a specific set of instructions. We also provide program verification capabilities such as symbolic execution, calculating weakest preconditions, and interfacing with decision procedures. Vine can also write out lifted Vine instructions as valid C code via the code generator back-end.

To combine static and dynamic analysis, we also provide an interface for Vine to read an execution trace generated by a dynamic analysis component such as TEMU. The execution trace can be lifted to the IL for various further analysis.

3.2 The Vine Intermediate Language

The Vine IL is the target language during lifting, as well as the analysis language for back-end program analysis. The semantics of the IL are designed to be faithful to assembly languages. Table 1 shows the Vine IL.

The base types in the Vine IL are 1, 8, 16, 32, and 64-bit registers (i.e., n-bit vectors) and memories. A memory type is qualified by its endianness, which can be either `little` (e.g., for little-endian architectures like x86), `big` (e.g., for big-endian architectures such as PowerPC), or `norm` for *normalized memory* (explained later in this section). A memory type is also qualified by the index type, which must be a register type. For example `mem_t(little, reg32_t)` denotes a memory type which is little endian and is addressed by 32-bit numbers.

There are three types of values in Vine. First, Vine has numbers n of type τ_{reg}. Second, Vine has memory values $\{n_{a1} \rightarrow n_{v1}, n_{a2} \rightarrow n_{v2}, ...\}$, where n_{ai} denotes a number used as an address, and n_{vi} denotes the value stored at the address. Finally, Vine has a distinguished value $\perp$. $\perp$ values are not exposed to the user and cannot be constructed in the presentation language. $\perp$ is used internally to indicate a failed execution.

Expressions in Vine are side-effect free. The Vine IL has binary operations $\Diamond_b$ ("&" and "|" are bit-wise), unary operations $\Diamond_u$, constants, `let` bindings, and casting. Casting is used when the semantics requires a change in the width of a value. For example, the lower 8 bits of `eax` in x86 are known as `al`. When lifting x86 instructions, we use casting to project out the lower-bits of the corresponding `eax` register variable to an `al` register variable when `al` is accessed.

Table 1. The Vine Intermediate Language (Since '$\vert$' is an operator, for clarity we use commas to separate all operator elements in the productions for $\Diamond_b$ and $\Diamond_u$.)

program	::= *decl** *instr**
instr	::= *var* = *exp* $\mid$ jmp *exp* $\mid$ cjmp *exp*,*exp*,*exp* $\mid$ halt *exp* $\mid$ assert *exp*
	$\mid$ label *integer* $\mid$ special id$_s$
exp	::= load(*exp*, *exp*, τ_{reg}) $\mid$ store(*exp*, *exp*, *exp*,τ_{reg}) $\mid$ *exp* $\Diamond_b$ *exp* $\mid$ $\Diamond_u$ *exp*
	$\mid$ *const* $\mid$ *var* $\mid$ let *var* = *exp* in *exp* $\mid$ cast(*cast_kind*,τ_{reg},*exp*)
cast_kind	::= unsigned $\mid$ signed $\mid$ high $\mid$ low
decl	::= var *var*
var	::= (string, id$_v$, τ)
$\Diamond_b$	::= $+, -, *, /, /_s, \text{mod}, \text{mod}_s, \ll, \gg, \gg_a, \&, \vert, \oplus, ==, \neq, <, \leq, <_s, \leq_s$
$\Diamond_u$	::= $-$ (unary minus), ! (bit-wise not)
value	::= *const* $\mid$ $\{ n_{a1} \rightarrow n_{v1}, n_{a2} \rightarrow n_{v2}, \ldots \}$: τ_{mem} $\mid$ $\bot$
const	::= $n : \tau_{\text{reg}}$
τ	::= τ_{reg} $\mid$ τ_{mem} $\mid$ Bot $\mid$ Unit
τ_{reg}	::= reg1_t $\mid$ reg8_t $\mid$ reg16_t $\mid$ reg32_t $\mid$ reg64_t
τ_{mem}	::= mem_t(τ_{endian}, τ_{reg})
τ_{endian}	::= little $\mid$ big $\mid$ norm

In Vine, both load and store operations are pure. While load is normally pure, the semantics of store are often not. Each store expression must specify which memory to load or store from. The resulting memory is returned as a value. For example, a Vine store operation is written `mem1 = store(mem0, a, y)`, where mem1 is the same as mem0 except address a has value y. The advantage of pure memory operations in Vine notation is that it makes it possible to syntactically distinguish what memory is modified or read. One place where we take advantage of this is in computing Single Static Assignment (SSA) where both scalars and memory have a unique single static assignment location.

A program in Vine is a sequence of variable declarations, followed by a sequence of instructions. There are 7 different kinds of instructions. The language has assignments, jumps, conditional jumps, and labels. The target of all jumps and conditional jumps must be a valid label in our operational semantics, else the program halts with $\bot$. Note that a jump to an undefined location (e.g., a location that was not disassembled such as to dynamically generated code) results in the Vine program halting with $\bot$. A program can halt normally at any time by issuing the halt statement. We also provide assert, which acts similar to a C assert: the asserted expression must be true, else the machine halts with $\bot$.

A special in Vine corresponds to a call to an externally defined procedure or function. The id of a special indexes what kind of special, e.g., what system call. The

```
// x86 instr dst,src
1. mov [eax], 0xaabbccdd
2. mov ebx, eax
3. add ebx, 0x3
4. mov eax, 0x1122
5. mov [ebx], ax
6. sub ebx, 1
7. mov ax, [ebx]
```

(a)

address	memory
eax	0xdd
eax+1	0xcc
eax+2	0xbb
eax+3	0xaa
eax+4	

(b)

address	memory
eax	0xdd
eax+1	0xcc
eax+2	0xbb
eax+3	0x22
eax+4	0x11

(c)

Fig. 3. An example of little-endian stores as found in x86 that partially overlap. (b) shows memory after executing line 1, and (c) shows memory after executing line 5. Line 7 will load the value 0x22bb.

semantics of `special` is up to the analysis; its operational semantics are not defined. We include `special` as an instruction type to explicitly distinguish when such calls may occur that alter the soundness of an analysis. A typical approach to dealing with `special` is to replace `special` with an analysis-specific summary function written in the Vine IL that is appropriate for the analysis.

Normalized Memory

The endianness of a machine is usually specified by the byte-ordering of the hardware. A little endian architecture puts the low-order byte first, and a big-endian architecture puts the high-order byte first. x86 is an example of a little endian architecture, and PowerPC is an example of a big endian architecture.

We must take endianness into account when analyzing memory accesses. Consider the assembly in Figure 3a. The `mov` operation on line 2 writes 4 bytes to memory in little endian order (since x86 is little endian). After executing line 2, the address given by `eax` contains byte `0xdd`, `eax+1` contains byte `0xcc`, and so on, as shown in Figure 3b. Lines 2 and 3 set `ebx = eax+2`. Line 4 and 5 write the 16-bit value `0x1122` to `ebx`. An analysis of these few lines of code needs to consider that the write on line 4 overwrites the last byte written on line 1, as shown in Figure 3c. Considering such cases requires additional logic in each analysis. For example, the value loaded on line 7 will contain one byte from each of the two stores.

We say a memory is *normalized* for a b-byte addressable memory if all loads and stores are exactly b-bytes and b-byte aligned. For example, in x86 memory is byte addressable, so a normalized memory for x86 has all loads and stores at the byte level. The normalized form for the write on Line 1 of Figure 3a in Vine is shown in Figure 4. Note the subsequent load on line 7 are with respect to the current memory `mem6`.

Normalized memory makes writing program analyses involving memory easier. Analyses are easier because normalized memory syntactically exposes memory updates that are otherwise implicitly defined by the endianness. The Vine back-end provides utilities for normalizing all memory operations.

```
1. mem4 = let mem1 = store(mem0,eax, 0xdd, reg8_t) in
           let mem2 = store(mem1, eax+1, 0xcc, reg8_t) in
           let mem3 = store(mem2, eax+2, 0xbb, reg8_t) in
               store(mem3, eax+3, 0xcc, reg8_t);
...
5. mem6 = let mem5 = store(mem4, ebx, 0x22, reg8_t) in
             store(mem5, ebx+1, 0x22, reg8_t)
...
7. value = let b1 = load(mem6, ebx, reg8_t) in
            let b2 = load(mem6, ebx+1, reg8_t) in
            let b1' = cast(unsigned, b1, reg16_t) in
            let b2' = cast(unsigned, b2, reg16_t) in
               (b2' ≪ 8) | b1';
```

Fig. 4. Vine normalized version of the store and load from Figure 3a

3.3 The Vine Front-End

The Vine front-end is responsible for translating binary code to the Vine IL. In addition, the front-end interfaces with libraries such as the GNU Binary File Descriptor (libbfd) library for parsing the low-level details of binary files.

Translating binary code to the IL consists of three steps:

- **Step 1.** First the binary file is disassembled. Vine currently interfaces with three disassemblers: IDA Pro [19], a commercial disassembler, a research disassembler from Kruegel *et al.* [26] that can disassemble x86 obfuscated code, and our own linear-sweep disassembler built on top of GNU libopcodes. Interfacing with other disassemblers is straightforward.
- **Step 2.** The disassembly is passed to VEX, a third-party library which turns assembly instructions into the VEX intermediate language. The VEX IL is part of the Valgrind dynamic instrumentation tool [31]. The VEX IL is also similar to a RISC-based language. As a result, the lifted IL has only a few instruction types, similar to Vine. However, the VEX IL itself is inconvenient for performing program analysis because its information about side effects of instructions such as what `EFLAGS` are set by x86 instructions is implicit. This step is mainly performed in order to simplify the development of Vine: we let the existing tool take care of the task of reducing assembly instructions to a basic IL, then in step 3 expose all side-effects so that the analysis is faithful.
- **Step 3.** We translate the VEX IL to Vine. The resulting Vine IL is intended to be faithful to the semantics of the disassembled assembly instructions.

Translated assembly instructions have all side-effects explicitly exposed as Vine instructions. As a result, a single typical assembly instruction will be translated as a sequence of Vine instructions. For example, the `add eax,0x2` x86 instruction is translated as the following instructions:

```
tmp1 = EAX; EAX = EAX + 2;
//eflags calculation
CF:reg1_t = (EAX<tmp1);
tmp2 = cast(low, EAX, reg8_t);
PF =(!cast(low,
              ((((tmp2>>7)^(tmp2>>6))^((tmp2>>5)^(tmp2>>4)))^
              (((tmp2>>3)^(tmp2>>2))^((tmp2>>1)^tmp2)))), reg1_t);
AF = (1==(16&(EAX^(tmp1^2))));
ZF = (EAX==0);
SF = (1==(1&(EAX>>31)));
OF = (1==(1&(((tmp1^(2^0xFFFFFFFF))&(tmp1^EAX))>>31)));
```

The translated instructions expose all the side-effects of the `add` instruction, including all 6 `eflags` that are updated by the operation. As another example, an instruction with the `rep` prefix is translated as a sequence of instructions that form a loop.

In addition to binary files, Vine can also translate an instruction trace to the IL. Conditional branches in a trace are lifted as `assert` statements to check that the executed branch is followed. This is done to prevent branching outside the trace to an unknown instruction. Vine and TEMU are co-designed so that TEMU currently generates traces in a trace format that Vine can read.

3.4 The Vine Back-End

In the Vine back-end, new program analyses are written over the Vine IL. Vine provides a library of common analyses and utilities which serve as building blocks for more advanced analyses. Below we provide an overview of some of the utilities and analyses provided in the Vine back-end.

Evaluator. Vine has an evaluator which implements the operational semantics of the Vine IL. The evaluator allows us to execute programs without recompiling the IL back down to assembly. For example, we can test a raised Vine IL for an instruction trace produced by an input by evaluating the IL on that input and verifying we end in the same state.

Graphs. Vine provides routines for building and manipulating control flow graphs (CFG), including a pretty-printer for the graphviz DOT graph language [2]. Vine also provides utilities for building data dependence and program dependence graphs [30].

One issue when constructing a CFG of an assembly program is determining the successors of jumps to computed values, called *indirect* jumps. Resolving indirect jumps usually requires program analyses that require a CFG, e.g., Value Set Analysis (VSA) [5]. Thus, there is a potential circular dependency. Note that an indirect jump may potentially go anywhere, including the heap or code that has not been previously disassembled.

Our solution is to designate a special node as a successor of unresolved indirect jump targets in the CFG. We provide this so an analysis that depends on a correct CFG can recognize that we do not know the subsequent state. For example, a data-flow analysis could widen all facts to the lattice bottom. Most normal analyses will first run an indirect jump resolution analysis in order to build a more precise CFG that resolves indirect jumps to a list of possible jump targets. Vine provides one such analysis, VSA [5].

Single Static Assignment. Vine supports conversion to and from single static assignment (SSA) form [30]. SSA form makes writing analysis easier because every variable is defined statically only once. We convert both memory and scalars to SSA form. We convert memories because then one can syntactically distinguish between memories before and after a write operation instead of requiring the analysis itself to maintain similar bookkeeping. For example, in the memory normalization example in Figure 3.2, an analysis can syntactically distinguish between the memory state before the write on line 1, the write on line 5, and the read on line 7.

Chopping. Given a source and sink node, a program chop [24] is a graph showing the statements that cause definitions of the source to affect uses of the sink. For example, chopping can be used to restrict subsequent analysis to only a portion of code relevant to a given source and sink instead of the whole program.

Data-flow and Optimizations. Vine provides a generic data-flow engine that works on user-defined lattices. Vine also implements several data-flow analysis. Vine currently implements Simpson's global value numbering [37], constant propagation and folding [30], dead-code elimination [30], live-variable analysis [30], integer range analysis, and Value set analysis (VSA) [5]. VSA is a data-flow analysis that over-approximates the values for each variable at each program point. Value-set analysis can be used to help resolve indirect jumps. It can also be used as an alias analysis. Two memory accesses are potentially aliased if the intersection of their value sets is non-empty.

Optimizations are useful for simplifying or speeding up subsequent analysis. For example, we have found that the time for the decision procedure STP to return a satisfying answer for a query can be cut in half by using program optimization to simplify the query first [9].

C Code Generator. Vine can generate valid C code from the IL. For example, one could use Vine as a rudimentary decompiler by first raising assembly to Vine, then writing it out as valid C. The ability to export to C also provides a way to compile Vine programs: the IL is written as C, then compiled with a C compiler.

The C code generator implements memories in the IL as arrays. A `store` operation is a store on the array, and a `load` is a load from the array. Thus, C-generated code simulates real memory. For example, consider a program vulnerable to a buffer overflow attack is raised to Vine, then written as C and recompiled. An out-of-bound write on the original program will be simulated in the corresponding C array, but will not lead to a real buffer overflow.

Program Verification Analyses. Vine currently supports formal program verification in two ways. First, Vine can convert the IL into Dijkstra's Guarded Command Language (GCL), and calculate the weakest precondition with respect to GCL programs [20]. The weakest precondition for a program with respect to a predicate q is the most general condition such that any input satisfying the condition is guaranteed to terminate (normally) in a state satisfying q. Currently we only support acyclic programs, i.e., we do not support GCL `while`.

Vine also interfaces with decision procedures. Vine can write out expressions (e.g., weakest preconditions) in CVC Lite syntax [1], which is supported by several decision procedures. In addition, Vine interfaces directly with the STP [22] decision procedure through calls from Vine to the STP library.

3.5 Implementation of Vine

The Vine infrastructure is implemented in C++ and OCaml. The front-end lifting is implemented primarily in C++, and consists of about 17,200 lines of code. The back-end is implemented in OCaml, and consists of about 40,000 lines of code. We interface the C++ front-end with the OCaml back-end using OCaml via IDL generated stubs.

The front-end interfaces with Valgrind's VEX [31] to help lift instructions, GNU BFD for parsing executable objects, and GNU libopcodes for pretty-printing the disassembly.

The implemented Vine IL has several constructors in addition to the instructions in Figure 1:

- The Vine IL has a constructor for comments. We use the comment constructor to pretty-print each disassembled instruction before the IL, as well as a place-holder for user-defined comments.
- The Vine IL supports variable scoping via blocks. Vine provides routines to de-scope Vine programs via α-varying as needed.
- The Vine IL has constructs for qualifying statements and types with user-defined attributes. This is added to help facilitate certain kinds of analysis such as taint-based analysis.

4 TEMU: The Dynamic Analysis Component

In this section, we give an overview of TEMU, the dynamic analysis component of BitBlaze Binary Analysis Platform, describing its components for extracting OS-level semantics, performing whole-system dynamic taint analysis, its Plugins and implementation.

4.1 TEMU Overview

TEMU is a whole-system dynamic binary analysis platform that we developed on the basis of a whole-system emulator, QEMU [36]. We run an entire system, including the operating system and applications in this emulator, and observe in a fine-grained manner how the binary code of interest is executed. The whole-system approach in TEMU is motivated by several considerations:

- Many analyses require fine-grained instrumentation (i.e., at instruction level) on binary code. By dynamically translating the emulated code, the whole-system emulator enables fine-grained instrumentation.
- A whole-system emulator presents us a whole-system view. The whole-system view enables us to analyze the operating system kernel and interactions between multiple processes. In contrast, many other binary analysis tools (e.g., Valgrind,

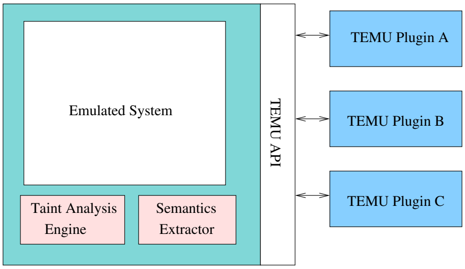

Fig. 5. TEMU Overview

DynamoRIO, Pin) only provide a local view (i.e., a view of a single user-mode process). This is particularly important for analyzing malicious code, because many attacks involve multiple processes, and kernel attacks such as rootkits have become increasingly popular.

- A whole-system emulator provides an excellent isolation between the analysis components and the code under analysis. As a result, it is more difficult for the code under analysis to interfere with analysis results.

The design of TEMU is motivated by several challenges and considerations:

- The whole-system emulator only provides us only the hardware-level view of the emulated system, whereas we need a software-level view to get meaningful analysis results. Therefore, we need a mechanism that can extract the OS-level semantics from the emulated system. For example, we need to know what process is currently running and what module an instruction comes from.
- In addition, many analyses require reasoning about how specific data depends on its data sources and how it propagates throughout the system. We enable this using whole-system dynamic taint analysis.
- We need to provide a well-designed programming interface (i.e., API) for users to implement their own plugins on TEMU to perform their customized analysis. Such an interface can hide unnecessary details from users and reuse the common functionalities.

With these considerations in mind, we have designed the architecture of TEMU, as shown in Figure 5. We build the *semantics extractor* to extract OS-level semantics information from the emulated system. We build the *taint analysis engine* to perform dynamic taint analysis. We define and implement an interface (i.e, TEMU API) for users to easily implement their own analysis modules (i.e. TEMU plugins). These modules can be loaded and unloaded at runtime to perform designated analyses. We implemented TEMU in Linux, and at the time of writing, TEMU can be used to analyze binary code in Windows 2000, Windows XP, and Linux systems. Below we describe these three components respectively.

4.2 Semantics Extractor

The semantics extractor is responsible for extracting OS-level semantics information of the emulated system, including process, module, thread, and symbol information.

Process and Module Information. For the current execution instruction, we need to know which process, thread and module this instruction comes from. In some cases, instructions may be dynamically generated and executed on the heap.

Maintaining a mapping between addresses in memory and modules requires information from the guest operating system. We use two different approaches to extract process and module information for Windows and Linux.

For Windows, we have developed a kernel module called *module notifier*. We load this module into the guest operating system to collect the updated memory map information. The module notifier registers two callback routines. The first callback routine is invoked whenever a process is created or deleted. The second callback routine is called whenever a new module is loaded and gathers the address range in the virtual memory that the new module occupies. In addition, the module notifier obtains the value of the CR3 register for each process. As the CR3 register contains the physical address of the page table of the current process, it is different (and unique) for each process. All the information described above is passed on to TEMU through a predefined I/O port.

For Linux, we can directly read process and module information from outside, because we know the relevant kernel data structures, and the addresses of relevant symbols are also exported in the `system.map` file. In order to maintain the process and module information during execution, we hook several kernel functions, such as `do_fork` and `do_exec`.

Thread Information. For windows, we also obtain the current thread information to support analysis of multi-threaded applications and the OS kernel. It is fairly straightforward, because the data structure of the current thread is mapped into a well-known virtual address in Windows. Currently, we do not obtain thread information for Linux and may implement it in future versions.

Symbol Information. For PE (Windows) binaries, we also parse their PE headers and extract the exported symbol names and offsets. After we determine the locations of all modules, we can determine the absolute address of each symbol by adding the base address of the module and its offset. This feature is very useful, because all windows APIs and kernel APIs are exported by their hosting modules. The symbol information conveys important semantics information, because from a function name, we are able to determine what purpose this function is used for, what input arguments it takes, and what output arguments and return value it generates. Moreover, the symbol information makes it more convenient to hook a function—instead of giving the actual address of a function, we can specify its module name and function name. Then TEMU will automatically map the actual address of the function for the user.

Currently, this feature is only available for PE binaries. Support for ELF (Linux) binaries will be available in future versions.

4.3 Taint Analysis Engine

Our dynamic taint analysis is similar in spirit to a number of previous systems [16, 35, 18, 38, 17]. However, since our goal is to support a broad spectrum of different applications, our design and implementation is the most complete. For example, previous approaches either operate on a single process only [17, 35, 38], or they cannot deal with memory swapping and disks [16, 18].

Shadow Memory. We use a shadow memory to store the taint status of each byte of the physical memory, CPU registers, the hard disk and the network interface buffer. Each tainted byte is associated with a small data structure storing the original source of the taint and some other book keeping information that a TEMU plugin wants to maintain. The shadow memory is organized in a page-table-like structure to ensure efficient memory usage. By using shadow memory for the hard disks, the system can continue to track the tainted data that has been swapped out, and also track the tainted data that has been saved to a file and is then read back in.

Taint Sources. A TEMU plugin is responsible for introducing taint sources into the system. TEMU supports taint input from hardware, such as the keyboard, network interface, and hard disk. TEMU also supports tainting a high-level abstract data object (e.g. the output of a function call, or a data structure in a specific application or the OS kernel).

Taint Propagation. After a data source is tainted, the taint analysis engine monitors each CPU instruction and DMA operation that manipulates this data in order to determine how the taint propagates. The taint analysis engine propagates taint through data movement instructions, DMA operations, arithmetic operations, and table lookups. Considering that some instructions (e.g., `xor eax, eax`) always produce the same results, independent of the values of their operands, the taint analysis engine does not propagate taint in these instructions.

Note that TEMU plugins may employ very different taint policies, according to their application requirements. For example, for some applications, we do not need to propagate taint through table lookups. For some applications, we want to propagate taint through an immediate operand, if the code region occupied by it is tainted. Therefore, during taint propagation, the taint analysis engine lets TEMU plugins determine how they want to propagate taint into the destination.

This design provides valuable flexibility to TEMU plugins. They can specify different taint sources, maintain an arbitrary record for each tainted byte, keep track of multiple taint sources, and employ various taint policies.

4.4 TEMU API and Plugins

In order for users to make use of the functionalities provided by TEMU, we define a set of functions and callbacks. By using this interface, users can implement their own plugins and load them into TEMU at runtime to perform analysis. Currently, TEMU provides the following functionalities:

- Query and set the value of a memory cell or a CPU register.
- Query and set the taint information of memory or registers.
- Register a hook to a function at its entry and exit, and remove a hook. TEMU plugins can use this interface to monitor both user and kernel functions.
- Query OS-level semantics information, such as the current process, module, and thread.
- Save and load the emulated system state. This interface helps to switch between different machine states for more efficient analysis. For example, this interface makes multiple path exploration more efficient, because we can save a state for a specific branch point and explore one path, and then load this state to explore the other path without restarting the program execution.

TEMU defines callbacks for various events, including (1) the entry and exit of a basic block; (2) the entry and exit of an instruction; (3) when taint is propagating; (4) when a memory is read or write; (5) when a register is read or written to; (6) hardware events such as network and disk inputs and outputs.

Quite a few TEMU plugins have been implemented by using these functions and callbacks. These plugins include:

- Panorama [43]: a plugin that performs OS-aware whole-system taint analysis to detect and analyze malicious code's information processing behavior.
- HookFinder [42]: a plugin that performs fine-grained impact analysis (a variant of taint analysis) to detect and analyze malware's hooking behavior.
- Renovo [25]: a plugin that extracts unpacked code from packed executables.
- Polyglot [14]: a plugin that make use of dynamic taint analysis to extract protocol message format.
- Tracecap: a plugin that records an instruction trace with taint information for a process or the OS kernel.
- MineSweeper [10]: a plugin that identifies and uncovers trigger-based behaviors in malware by performing online symbolic execution.
- BitScope: a more generic plugin that make use of symbolic execution to perform in-depth analysis of malware.
- HookScout: a plugin that infers kernel data structures.

4.5 Implementation of TEMU

The TEMU infrastructure is implemented in C and C++. In general, performance-critical code is implemented in C due to efficiency of C, whereas analysis-oriented code is written in C++ to leverage the abstract data types in STL and stronger type checking in C++. For example, the taint analysis engine insert code snippets into QEMU micro operations to check and propagate taint information. Since taint analysis is performance critical, we implemented it in C. On the other hand, we implemented the semantics extractor in C++ using `string`, `list`, `map` and other abstract data types in STL, to maintain a mapping between OS-level view and hardware view. The TEMU API is defined in C. This gives flexibility to users to implement their plugin in either C, C++, or both. The TEMU core consists of about 37,000 lines of code, excluding the code originally from QEMU (about 306,000 lines of code). TEMU plugins consist of about 134,000 lines of code.

5 Rudder: The Mixed Concrete and Symbolic Execution Component

In this section, we give an overview of Rudder, the mixed concrete and symbolic execution component of BitBlaze Binary Analysis Platform, describing its components for performing mixed execution and exploring program execution space and its implementation.

5.1 System Overview

We have designed and developed *Rudder*, to perform mixed concrete and symbolic execution at the binary level. Given a binary program and a specification of symbolic inputs, Rudder performs mixed concrete and symbolic execution and explores multiple execution paths whenever the path conditions are dependent on symbolic inputs. By doing so, Rudder is able to automatically uncover hidden behaviors that only exhibit under certain conditions.

Figure 6 shows a high level picture of Rudder. Rudder consists of the following components: the *mixed execution engine* that performs mixed concrete and symbolic execution, the *path selector* that prioritizes and determines the execution paths, and the *solver* that performs reasoning on symbolic path predicates and determines if a path is feasible. Rudder takes as inputs a binary program and a symbolic input specification. In TEMU, the binary program is executed and monitored. Rudder runs as a TEMU plugin to instrument the execution of the binary program. During the execution, Rudder marks some of the inputs as symbolic according to the symbolic input specification. Then the mixed execution engine symbolically executes the operations on symbolic inputs and data calculated from symbolic inputs. When a symbolic value is used in a branch condition, the path selector determines, with assistance of the solver, which branches are feasible and selects a branch to explore.

5.2 Mixed Execution Engine

Determine Whether to Symbolically Execution an Instruction. For each instruction, the mixed execution engine performs the following steps. First, it checks the source operands of that instruction, and answers whether they are concrete or symbolic. If all source operands are concrete, this instruction will be executed concretely on the emulated CPU. Otherwise, the mixed execution engine marks the destination operand as symbolic, and calculate symbolic expressions for the destination operand. To mark

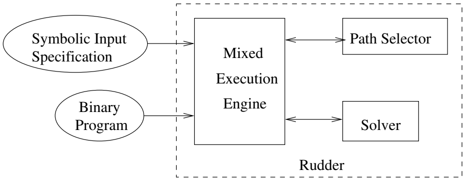

Fig. 6. Rudder Overview

and propagate symbolic data, the mixed execution engine relies on the dynamic taint analysis functionality provided by TEMU.

Formulate a Symbolic Program. Ideally, we would like to calculate symbolic expressions on the fly. However, this naive approach would incur significant performance overhead at runtime. Considering that many expressions would not be used in path predicates, we take a "lazy" approach. The basic idea is to collect necessary information in the symbolic machine during the symbolic execution. At a later time, when some symbolic variables are used in path predicates, we can extract the corresponding symbolic expressions from the symbolic machine.

More specifically, we enqueue this instruction, along with the relevant register and memory states into our symbolic machine. That is, if an instruction to be symbolically executed has any concrete operands, we must update those concrete values inside the symbolic machine. In the case of registers, this is trivial – we simply update their concrete values in the symbolic machine. Memory accesses with concrete addresses are handled similarly. However, we also have to deal with memory accesses where the memory address itself is symbolic, which is described below.

A symbolic memory address means that the data specifying which memory is to be read or written is symbolic. If this address is symbolic, we do not know which memory location is about to be accessed, because a symbolic address may potentially take any concrete value. To solve this problem, we use the solver to determine the range of possible values of this address. In some cases, the range that the solver returns is too large to be effective. In this case, we add a constraint to the system to limit its size, therefore limiting the complexity that is introduced. In practice, we found that most symbolic memory accesses are already constrained to small ranges, making this unnecessary. For example, consider code that iterates over an array. Each access to the array is bounded by the constraints imposed by the iteration itself. Note that this is a conservative approach, meaning that all solutions found are still correct. Once a range is selected, we update the symbolic machine with the necessary information.

We leverage the functionality of Vine to perform symbolic execution. That is, when symbolically executing an instruction, we lift it into Vine IL and formulate a symbolic program in form of Vine IL.

Extract Symbolic Expressions. Given the symbolic program and a symbolic variable of interest, we can extract a symbolic expression for it. Extracting a symbolic expression takes several steps. First, we perform dynamic slicing on the symbolic program. This step removes the instructions that the symbol does not depend upon. After this step, the resulted symbolic program is reduced drastically. Then we generate one expression by substituting intermediate symbols with their right-hand-side expressions. Finally, we perform constant folding and other optimizations to further simplify the expression.

5.3 Path Selector

When a branch condition becomes symbolic, the path selector is consulted to determine which branch to explore. There is an interface for users to supply their own path selection priority function. As we usually need to explore as many paths that depend upon symbolic inputs as possible, the default is a breadth-first search approach.

To make the path exploration more efficient, we make use of the functionality of state saving and restoring provided by TEMU. That is, when a symbolic conditional branch is first encountered, the path selector saves the current execution state, and determines which feasible direction to explore. Later, when it decides to explore a different direction from this branch, the path selector restores the execution state on this branch and explores the other branch.

5.4 Solver

The solver is a theorem prover or decision procedure, which performs reasoning on symbolic expressions. In Rudder, the solver is used to determine if a path predicate is satisfiable, and to determine the range of the memory region with a symbolic address. We can make use of any appropriate decision procedures that are available. Thus, if there is any new progress on decision procedures, we can benefit from it. Currently in our implementation, we use STP as the solver [21].

5.5 Implementation of Rudder

Rudder is implemented in C, C++ and Ocaml. The implementation consists of about 3,600 lines of C/C++ code and 2,600 lines of Ocaml code. The C/C++ code is mainly for implementing a TEMU plugin, and marking and tracking symbolic values, and the Ocaml code is used to interface with Vine and perform symbolic analysis. We also have an offline mode for Rudder where we take an execution trace and then perform symbolic execution on the trace to obtain the path predicate and then solve for inputs for different branches to be taken.

6 Security Applications

In this section, we give an overview of the different security applications that we have enabled using the BitBlaze Binary Analysis Platform, ranging from automatic vulnerability detection, diagnosis, and defense, to automatic malware analysis and defense, to automatic model extraction and reverse engineering. For each security application, we give a new formulation of the problem based on the root cause in the relevant program. We then demonstrate that this problem formulation leads us to new approaches to address the security application based on the root cause. The results demonstrate the utility and effectiveness of the BitBlaze Binary Analysis Platform and its vision—it was relatively easy to build different applications on top of the BitBlaze Binary Analysis Platform and we could obtain effective results that previous approaches could not.

6.1 Vulnerability Detection, Diagnosis, and Defense

Sting: An Automatic Defense System against Zero-Day Attacks. Worms such as CodeRed and SQL Slammer exploit software vulnerabilities to self-propagate. They can compromise millions of hosts within hours or even minutes and have caused billions of dollars in estimated damage. How can we design and develop effective defense mechanisms against such fast, large scale worm attacks?

We have designed and developed Sting [40,33], a new end-to-end automatic defense system that aims to be effective against even zero-day exploits and protect vulnerable hosts and networks against fast worm attacks. Sting uses dynamic taint analysis to detect exploits to previously unknown vulnerabilities [35] and can automatically generate filters for dynamic instrumentation to protect vulnerable hosts [34].

Automatic Generation of Vulnerability Signatures. Input-based filters (a.k.a. signatures) provide important defenses against exploits before the vulnerable hosts can be patched. Thus, to automatically generate effective input-based filters in a timely fashion is an important task. We have designed and developed novel techniques to automatically generate accurate input-based filters based on information about the vulnerability instead of the exploits, and thus generating filters that have zero false positives and can be effective against different variants of the exploits [11,13].

Automatic Patch-based Exploit Generation. Security patches do not only fix security vulnerabilities, they also contain sensitive information about the vulnerability that could enable an attacker to exploit the original vulnerable program and lead to severe consequences. We have demonstrated that this observation is correct—we have developed new techniques showing that given the patched version and the original vulnerable program, we can automatically generate exploits in our experiments with real world patches (often in minutes) [12]. This opens the research direction of how to design and develop a secure patch dissemination scheme where attacker cannot use information in the patch to attack vulnerable hosts before they have a chance to download and apply the patch.

6.2 Malware Analysis and Defense

Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. A myriad of malware such as keyloggers, Browser-helper Objects (BHO) based spyware, rootkits, and backdoors accesses and leaks users' sensitive information and breaches users' privacy. Can we have a unified approach to identify such privacy-breaching malware despite their widely-varied appearance? We have designed and developed Panorama [43] as a unified approach to detect privacy-breaching malware using whole-system dynamic taint analysis.

Renovo: Hidden Code Extraction from Packed Executables. Code packing is one technique commonly used to hinder malware code analysis through reverse engineering. Even though this problem has been previously researched, the existing solutions are either unable to handle novel packers, or vulnerable to various evasion techniques. We have designed and developed Renovo [25], as a fully dynamic approach for hidden code extraction, capturing an intrinsic characteristic of hidden code execution.

HookFinder: Identfying and Understanding Malware Hooking Behavior. One important malware attacking vector is its hooking mechanism. Malicious programs implant hooks for many different purposes. Spyware may implant hooks to be notified of the arrival of new sensitive data. Rootkits may implant hooks to intercept and tamper with critical system information to conceal their presence in the system. A stealth backdoor may also place hooks on the network stack to establish a stealthy communication channel with remote attackers. We have designed and developed HookFinder [42] to automatically detect and analyze malware's hooking behaviors, by performing fine-grained impact analysis. Since this technique captures the intrinsic nature of hooking behaviors, it is well suited for identifying new hooking mechanisms.

BitScope: Automatically Dissecting Malware. The ability to automatically dissect a malicious binary and extract information from it is an important cornerstone for system forensic analysis and system defense. Malicious binaries, also called malware, include denial of service attack tools, spamming systems, worms, and botnets. New malware samples are uncovered daily through widely deployed honeypots/honeyfarms, forensic analysis of compromised systems, and through underground channels. As a result of the break-neck speed of malware development and recovery, automated analysis of malicious programs has become necessary in order to create effective defenses.

We have designed and developed BitScope, an architecture for systematically uncovering potentially hidden functionality of malicious software [8]. BitScope takes as input a malicious binary, and outputs information about its execution paths. This information can then be used by supplemental analysis designed to answer specific questions, such as what behavior the malware exhibits, what inputs activate interesting behavior, and the dependencies between its inputs and outputs.

6.3 Automatic Model Extraction and Analysis

Polyglot: Automatic Extraction of Protocol Message Format. Protocol reverse engineering, the process of extracting the application-level protocol used by an implementation (without access to the protocol specification) is important for many network security applications. Currently, protocol reverse engineering is mostly manual. For example, it took the open source Samba project over the course of 10 years to reverse engineer SMB, the protocol Microsoft Windows uses for sharing files and printers [39].

We have proposed that binary program analysis can be used to aid automatic protocol reverse engineering [15]. The central intuition is that the binary itself encodes the protocol, thus binary analysis should be a significant aide in extracting the protocol from the binary itself.

Automatic Deviation Detection. Many network protocols and services have several different implementations. Due to coding errors and protocol specification ambiguities, these implementations often contain deviations, i.e., differences in how they check and process some of their inputs. Automatically identifying deviations can enable the automatic detection of potential implementation errors and the automatic generation of fingerprints that can be used to distinguish among implementations of the same network service. The main challenge in this project is to automatically find deviations (without requiring access to source code). We have developed techniques for taking two binary implementations of the same protocol and automatically generating inputs which cause deviations [7].

Replayer: Sound Replay of Application Dialogue. The ability to accurately replay application protocol dialogs is useful in many security-oriented applications, such as replaying an exploit for forensic analysis or demonstrating an exploit to a third party. A central challenge in application dialog replay is that the dialog intended for the original host will likely not be accepted by another without modification. For example, the dialog may include or rely on state specific to the original host such as its host name or a known cookie. In such cases, a straight-forward byte-by-byte replay to a different host with a different state (e.g., different host name) than the original dialog participant will likely fail. These state-dependent protocol fields must be updated to reflect the different state of the different host for replay to succeed. We have proposed the first approach for soundly replaying application dialog where replay succeeds whenever the analysis yields an answer [32].

7 Related Work

In this section, we briefly describe some related work on other static and dynamic binary analysis platforms.

Static Binary Analysis Platforms. There are several static binary analysis platforms, however they are not sufficient for our purposes mainly because they were motivated by different applications and hence did not need to satisfy the same requirements.

Phoenix is a compiler program analysis environment developed by Microsoft [28]. One of the Phoenix tools allows code to be raised up to a register transfer language (RTL). A RTL is a low-level IR that resembles an architecture-neutral assembly. Phoenix differs from Vine is several ways. First, Phoenix can only lift code produced by a Microsoft compiler. Second, Phoenix requires debugging information, thus is not a true binary-only analysis platform. Third, Phoenix lifts assembly to a low-level IR that does not expose the semantics of complicated instructions, e.g., register status flags, as part of the IR [29]. Fourth, the semantics of the lifted IR, as well as the lifting semantics and goals, are not well specified [29], thus not suitable for our research purposes.

The CodeSurfer/x86 platform [6] is a proprietary platform for analyzing x86 programs. However, it was mainly designed for other applications such as slicing and does not provide some of the capabilities that we need.

Dynamic Binary Analysis Platforms. Tools like DynamoRIO [3], Pin [27], and Valgrind [41] support fine-grained instrumentation of a user-level program. They all provide a well-defined interface for users to implement plugins. However, as they can only instrument a single user-level process, they are unsuitable to analyze the operating system kernel and applications that involve multiple processes. In addition, these tools reside in the same execution environment with the program under instrumentation. Some of them even share the same memory space with the program under instrumentation, and change the memory layout of the program. In consequence, the analysis results may be affected. In contrast, TEMU provides a whole-system view, enabling analysis of the OS kernel and multiple processes. Moreover, as it resides completely out of the analyzed execution environment, TEMU can provide more faithful analysis results.

8 Conclusion

In this paper, we have described the BitBlaze project, its binary analysis platform and applications to a broad spectrum of different security applications. Through this project, we have demonstrated that our binary analysis platform provides a powerful arsenal that enables us to take a principled, root-cause based approach to a broad spectrum of security problems. We hope BitBlaze's extensible architecture will enable others to build upon and enable new solutions to other applications.

Acknowledgements

We would like to thank Cody Hartwig, Xeno Kovah, Eric Li, and other colleagues and collaborators for their help and support to the project.

References

1. CVC Lite documentation (Page checked 7/26/2008), `http://www.cs.nyu.edu/acsys/cvcl/doc/`
2. The DOT language (Page checked 7/26/2008), `http://www.graphviz.org/doc/info/lang.html`
3. On the run - building dynamic modifiers for optimization, detection, and security. Original DynamoRIO announcement via PLDI tutorial (June 2002)
4. ARM. ARM Architecture Reference Manual (2005) Doc. No. DDI-0100I
5. Balakrishnan, G.: WYSINWYX: What You See Is Not What You eXecute. PhD thesis, Computer Science Department, University of Wisconsin at Madison (August 2007)
6. Balakrishnan, G., Gruian, R., Reps, T., Teitelbaum, T.: Codesurfer/x86 - a platform for analyzing x86 executables. In: Proceedings of the International Conference on Compiler Construction (April 2005)
7. Brumley, D., Caballero, J., Liang, Z., Newsome, J., Song, D.: Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In: Proceedings of the USENIX Security Symposium, Boston, MA (August 2007)
8. Brumley, D., Hartwig, C., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Song, D.: Bitscope: Automatically dissecting malicious binaries. Technical Report CS-07-133, School of Computer Science, Carnegie Mellon University (March 2007)
9. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Poosankam, P., Song, D., Yin, H.: Automatically identifying trigger-based behavior in malware. In: Lee, W., Wang, C., Dagon, D. (eds.) Botnet Detection. Countering the Largest Security Threat Series: Advances in Information Security, vol. 36, Springer, Heidelberg (2008)
10. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., Yin, H.: Towards automatically identifying trigger-based behavior in malware using symbolic execution and binary analysis. Technical Report CMU-CS-07-105, Carnegie Mellon University School of Computer Science (January 2007)
11. Brumley, D., Newsome, J., Song, D., Wang, H., Jha, S.: Towards automatic generation of vulnerability-based signatures. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy, pp. 2–16 (2006)

12. Brumley, D., Poosankam, P., Song, D., Zheng, J.: Automatic patch-based exploit generation is possible: Techniques and implications. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy (2008)
13. Brumley, D., Wang, H., Jha, S., Song, D.: Creating vulnerability signatures using weakest pre-conditions. In: Proceedings of Computer Security Foundations Symposium (July 2007)
14. Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In: Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS 2007) (October 2007)
15. Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In: Proceedings of the ACM Conference on Computer and Communications Security (October 2007)
16. Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., Rosenblum, M.: Understanding data lifetime via whole system simulation. In: Proceedings of the 13th USENIX Security Symposium (Security 2004) (August 2004)
17. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: End-to-end containment of internet worms. In: In Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP 2005) (2005)
18. Crandall, J.R., Chong, F.T.: Minos: Control data attack prevention orthogonal to memory model. In: Proceedings of the 37th International Symposium on Microarchitecture (MICRO 2004) (December 2004)
19. DataRescue. IDA Pro. (Page checked 7/31/2008), `http://www.datarescue.com`
20. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, Englewood Cliffs (1976)
21. Ganesh, V., Dill, D.: STP: A decision procedure for bitvectors and arrays, `http://theory.stanford.edu/~vganesh/stp`
22. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 524–536. Springer, Heidelberg (2007)
23. Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, Volumes 1-5 (April 2008)
24. Jackson, D., Rollins, E.J.: Chopping: A generalization of slicing. Technical Report CS-94-169, Carnegie Mellon University School of Computer Science (1994)
25. Kang, M.G., Poosankam, P., Yin, H.: Renovo: A hidden code extractor for packed executables. In: Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM 2007) (October 2007)
26. Kruegel, C., Robertson, W., Valeur, F., Vigna, G.: Static disassembly of obfuscated binaries. In: Proceedings of the USENIX Security Symposium (2004)
27. 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: Proceedings of the ACM Conference on Programming Language Design and Implementation (June 2005)
28. Microsoft. Phoenix framework (Paged checked 7/31/2008), `http://research.microsoft.com/phoenix/`
29. Microsoft. Phoenix project architect posting (Page checked 7/31/2008) (July 2008), `http://forums.msdn.microsoft.com/en-US/phoenix/thread/90f5212c-05a-4aea-9a8f-a5840a6d101d`
30. Muchnick, S.S.: Advanced Compiler Design and Implementation. Academic Press, London (1997)
31. Nethercote, N.: Dynamic Binary Analysis and Instrumentation or Building Tools is Easy. PhD thesis, Trinity College, University of Cambridge (2004)
32. Newsome, J., Brumley, D., Franklin, J., Song, D.: Replayer: Automatic protocol replay by binary analysis. In: Write, R., De Capitani di Vimercati, S., Shmatikov, V. (eds.) Proceedings of the ACM Conference on Computer and Communications Security, pp. 311–321 (2006)

33. Newsome, J., Brumley, D., Song, D.: Sting: An end-to-end self-healing system for defending against zero-day worm attacks. Technical Report CMU-CS-05-191, Carnegie Mellon University School of Computer Science (2006)
34. Newsome, J., Brumley, D., Song, D.: Vulnerability-specific execution filtering for exploit prevention on commodity software. In: Proceedings of the 13th Annual Network and Distributed Systems Security Symposium, NDSS (2006)
35. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 2005) (February 2005)
36. Qemu, `http://fabrice.bellard.free.fr/qemu/`
37. Simpson, L.T.: Value-Driven Redundancy Elimination. PhD thesis, Rice University Department of Computer Science (1996)
38. Suh, G.E., Lee, J.W., Zhang, D., Devadas, S.: Secure program execution via dynamic information flow tracking. In: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2004) (October 2004)
39. Tridgell, A.: How samba was written (Checked on 8/21/2008) (August 2003), `http://www.samba.org/ftp/tridge/misc/french_cafe.txt`
40. Tucek, J., Newsome, J., Lu, S., Huang, C., Xanthos, S., Brumley, D., Zhou, Y., Song, D.: Sweeper: A lightweight end-to-end system for defending against fast worms. In: Proceedings of the EuroSys Conference (2007)
41. Valgrind, `http://valgrind.org`
42. Yin, H., Liang, Z., Song, D.: HookFinder: Identifying and understanding malware hooking behaviors. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS 2008) (February 2008)
43. Yin, H., Song, D., Manuel, E., Kruegel, C., Kirda, E.: Panorama: Capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS 2007) (October 2007)