Micro Execution

Patrice Godefroid
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
pg@microsoft.com

ABSTRACT

Micro execution is the ability to execute any code fragment without a user-provided test driver or input data. The user simply identifies a function or code location in an exe or dll. A runtime Virtual Machine (VM) customized for testing purposes then starts executing the code at that location, catches all memory operations *before* they occur, allocates memory on-the-fly in order to perform those read/write memory operations, and provides input values according to a customizable *memory policy*, which defines what read memory accesses should be treated as inputs.

MicroX is a first prototype VM allowing micro execution of x86 binary code. No test driver, no input data, no source code, no debug symbols are required: MicroX automatically discovers *dynamically* the Input/Output interface of the code being run. Input values are provided as needed along the execution and can be generated in various ways, e.g., randomly or using some other test-generation tool. To our knowledge, MicroX is the *first VM designed for test isolation and generation purposes*.

This paper introduces micro execution and discusses how to implement it, strengths and limitations, applications, related work and long-term goals.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Testing, Verification

Keywords

Program Execution, Testing, Virtual Machine

1. INTRODUCTION

Among the various kinds of testing usually performed during software development, *unit testing* applies to the individual components of a software system (e.g., see [30]). In theory, unit testing plays an important role in ensuring overall software quality since its role is to detect errors in the component's logic, check all corner cases, and provide 100% code coverage. Yet, in practice, unit testing is so hard and expensive to perform that it is rarely done properly, especially for components of large, complex, legacy code bases. Indeed, in order to be able to execute and test a component in isolation, one needs to write test driver/harness code to simulate the environment of the component. More code is needed to test functional correctness, for instance using assertions checking the component's outputs. Since writing all this testing code manually is expensive, unit testing is often either performed poorly or skipped altogether.

In this paper, we propose a radically new approach to unit testing whereby the user does not need to write any test driver/harness code at all, nor provide any input data. The new key idea explored in this work is to discover *dynamically* the Input/Output (I/O) interface of the code being executed, by means of a *Virtual Machine* (VM) modified for testing purposes.

The user simply identifies a function or code location (an offset) in an exe or dll. A runtime VM then starts executing the code at that location, catches all memory operations *before* they occur, allocates memory on-the-fly in order to perform those read/write memory operations, and provides input values according to a customizable *memory policy*, which defines what read memory accesses should be treated as inputs. Input values are provided as needed along the execution and can be generated in various ways, e.g., randomly or using another test-generation tool.

We call *micro execution* the ability to execute any code fragment without a user-provided test driver or input data. *MicroX* is a first prototype VM allowing micro execution of x86 binary code. No source code or debug symbols are required. No test driver/harness is required either: MicroX automatically discovers dynamically the I/O interface of the code being run. To our knowledge, MicroX is the *first VM for testing purposes*, which replaces the functionality of a static test driver by a runtime environment for dynamically discovering the I/O interface of the program under test with high precision, and providing input values only when needed, in a lazy manner. Our current MicroX prototype is a modification of the Nirvana VM [3] as well as of the iDNA/TruScan/SAGE tool stack [3, 31, 22]. In particular, the whitebox fuzzer SAGE can be used to guide micro executions of the code being tested using dynamic symbolic execution, constraint generation and solving techniques.

MicroX *significantly lowers the upfront cost of unit testing*

ICSE '14, May 31 - June 7, 2014, Hyderabad, India.

by allowing automatic testing of arbitrary code fragments without writing any test-driver code. Micro execution can reveal software bugs such as memory corruptions or buffer overflows due to incomplete input validation or erroneous manipulations of data structures that are *local* to the unit under test. It is complementary and should be used in conjunction with runtime analysis tools (like Purify, Valgrind, AppVerifier) for checking standard program properties (like buffer overflows, memory leaks, etc.). It can also check for application-specific properties ("test oracles") specified as assertions embedded in the code under test.

Micro execution of a code fragment makes sense mostly if all its inputs are unconstrained, i.e., can take any value. Otherwise, micro execution may trigger program behaviors and bugs that are uninteresting ("false positives") because they cannot occur in a realistic environment for that program. In that case, input constraints (preconditions) or a modified memory policy are needed to restrict the set of possible initial states. Those can be provided by the user, for instance, in the form of a partial test driver or code contracts, or by a whole program analysis tool like SAGE.

Conversely, micro execution may also fail to exercise some behaviors and miss bugs ("false negatives") if its input set is too small or if the memory policy is too loose, i.e., MicroX recovers from memory corruptions that could happen in a realistic environment.

How to specify input preconditions (initial states) and output postconditions (test oracle) are long-standing problems in both static program analysis and testing (e.g., [4]). Micro execution can be viewed as lifting those fundamental problems from low-level unit-specific test-driver code to higher-level rule-based memory policies, but its goal is not to directly address, avoid or solve those problems.

Instead, we identify in this paper (see Section 5) several security-motivated applications of micro execution which do not suffer from those limitations, because the "unit" (code fragment) under test is specifically defined in each case to have (1) no precondition (all input values are possible) and (2) a memory policy which is as tight as possible for these application domains.

The paper is organized as follows. In Section 2, we start by an overview of the functionality provided by micro execution and MicroX from a user's point of view. In Section 3, we discuss how to modify an existing VM to perform micro execution. In Section 4, we discuss in more details the limitations of micro execution mentioned above. We then present in Section 5 several applications which do not suffer from those limitations. In Section 6, we present experimental results obtained with our current MicroX prototype in the context of one of those applications, namely API fuzzing. Section 7 discusses related work, and we conclude in Section 8 by summarizing the main contributions of this work.

2. OVERVIEW

We present in this section an overview of micro execution from a user's point of view.

2.1 What is Micro Execution?

The user selects a function or code location in any executable file, such as an exe or a dll on a Windows machine. A special runtime Virtual Machine (VM) customized for testing purposes then starts executing the code at that location. The VM hijacks all memory operations, and provides input values according to a customizable memory policy.

Here is an example of *default memory policy*, which is often adequate and sufficient when the micro execution starts at the beginning of a C function:

> An input is defined as any value read from an uninitialized function argument, or from a dereference of a previous input used as an address (recursive definition). (*)

Note that, under this memory policy, values read from uninitialized global-variables are not inputs. Other memory policies can be defined, as will be discussed later.

Imagine the user wants to micro execute the C function `foo` below:

```
void foo(char *p) { // p is a 4-byte input
  char v = *p;      // *p is a 1-byte input
  return;
}
```

Let us assume that the user wants to micro execute this function under the previous default memory policy and that she selects the random test generation mode, by which any input is simply generated randomly (other modes will be discussed later). In what follows, such a user-selected function is called the *top-level function* of the micro execution.

Micro execution of this code starts executing the first instruction of function `foo`. The runtime VM detects that the execution wants to read a 4-byte value for `p` (assuming the program is run on a 32-bit machine). Since `p` is an argument of function `foo`, it is an input according to the above memory policy. Therefore, a 4-byte input value is randomly generated for `p` and is "returned to the program." (How this is done is described in the next subsection.) Next, the VM detects that the 4-byte value of `p` is used as an address `*p` where the code wants to read one byte. According to our memory policy, a dereference `*p` of a previous input `p` is also an input, thus a 1-byte value is randomly generated for `*p` and "returned to the program". This value is then copied in local variable `v`. After the execution of the `return` statement, the micro execution terminates.

In summary, this micro execution detects dynamically 2 inputs: `p` which has a 4-byte value (on a 32-bit machine), and `*p` which has a 1-byte value.

Micro execution means the ability to execute any code fragment without writing a test driver or input data. The user does not need any test-driver code for allocating and initializing memory for `p` and `*p`, and then invoking (calling) function `foo` with those arguments. In fact, the user does not need to know anything about how many arguments `foo` takes as inputs and what their types are. She only needs to select a starting location, a memory policy and a test-input generation mode.

2.2 How is Micro Execution performed with MicroX?

MicroX is a first VM implementing micro execution for x86 binary code. Source code or debugging information (i.e., pdb files on Windows) is not required.

The C function `foo` above is compiled (with the C `cl` compiler included in Microsoft Visual Studio 2010 on an Intel 32-bit machine running Windows 7) into the x86 assembly code shown in Figure 1. MicroX executes x86 instructions one by one, and catches all memory operations. In x86, `ebp, esp,`

```
     [...]
1:   push  ebp          ; foo starts here
2:   mov   ebp, esp
3:   push  ecx
4:   mov   eax, DWORD PTR [ebp+8] ; p
5:   mov   cl, BYTE PTR [eax]     ; *p
6:   mov   BYTE PTR [ebp-1], cl   ; v
7:   mov   esp, ebp
8:   pop   ebp
9:   ret   0
     [...]
```

Figure 1: x86 assembly code for function `foo`.

`eax, ecx` etc. denote 32-bit *registers*, while the expression `[ebp]` denotes accessing the *memory address* specified by the value of register `ebp`. For the following discussion, x86 instructions that *"perform memory operations"* are syntactically identified by the presence of an expression with square brackets `[ ]`, as in `[ebp]`. Note that x86 instructions involving only registers (like `mov ebp, esp` in line 2) or the stack (like `push ebp` in line 1) do not perform "memory operations" for the following discussion.

For the example above, the first memory operation executed during the micro execution of function `foo` is at line 4. MicroX detects that the 32-bit address `ebp+8` is *above* the current value of the stack pointer register `esp`, which means that this address stores a *function argument*. With our default memory policy (*), the 4-byte value (`DWORD`) at address `ebp+8`, denoted by `[ebp+8]`, is thus an input. However, since function `foo` was not called with any argument by MicroX (the number of arguments and their type is unknown when micro execution starts), the address `ebp+8` has not been properly allocated and initialized yet. MicroX therefore allocates a fresh 4-byte buffer at a new address `X` in an *external memory* invisible to the program under test. MicroX then *maps* the program-visible address `ebp+8` with that program-invisible address `X`, initializes the 4-byte buffer at address `X` (which represents argument `p` in the source code), and then *substitutes* the address `ebp+8` by address `X` so that the new 4-byte value stored at `[X]` is returned and moved to register `eax` in line 4. In Section 3, we discuss in detail how the program binary is instrumented, how addresses are substituted ("hijacked") by others, how inputs are initialized, and how the external memory maps program-visible addresses to invisible ones, including in the presence of pointer arithmetic.

Figure 2 presents a report generated by MicroX with information about all the memory accesses it detected during micro execution of function `foo`. In this micro execution, the initial value of `ebp` is `0x001EF988` (in hexadecimal) – see line 2 of Figure 2. The first memory access is detected for address `001EF990` (line 3), which is indeed `ebp+8` as expected. Using the random test-input generation mode, MicroX returned the 4-byte random value `00201478` (line 5). At this point, MicroX does not know whether this 4-byte value will be used as a pointer; preventively, it records this value in a list of *known input addresses* (line 6). This 4-byte value is stored in a 4-byte buffer located at address `X = 00201440` (line 7) in this case. From now on, any access to the program-visible address `001EF990` will be mapped to the program-invisible address `X` in the MicroX-controlled external memory. (Address `X` is invisible to the program in the sense that the value of `X` never appears in any of the program registers.)

```
1:  initEIP is 72B51005
2:  initEBP is 001EF988

3:  Read Mem Access at address 001EF990 of 4 bytes
4:      Initializing 4 input bytes:
5:        [0]=78 [1]=14 [2]=20 [3]=00
6:      Adding 00201478 to list of known addresses
7:    SetGuestEffectiveAddress returned 00201440

8:  Read Mem Access at address 00201478 of 1 bytes
9:      Initializing 1 input bytes: [0]=29
10:   SetGuestEffectiveAddress returned 0020C490

11: Write Mem Access at address 001EF987 of 1 bytes
12:   SetGuestEffectiveAddress returned 001EF987

13: END: ExitProcess is called

14: ***** External Memory Stats: *****
15: Number of Mem Accesses: 2 (2 Reads, 0 Writes)
16: Number of Addresses: 2 (total 5 bytes)
17: Number of Inputs: 2 (total 5 bytes)

18: ***** Native Memory Stats: *****
19: Number of Module Accesses: 0 (0 Reads, 0 Writes)
20: Number of Other Accesses: 1 (0 Reads, 1 Writes)

21: ***** General Stats: *****
22: Number of Unique Instructions After Start: 9
23: Number of Warnings: 0
24: Number of Errors: 0
```

Figure 2: MicroX report on all the memory accesses detected during micro execution of function `foo`.

The micro execution then proceeds by executing line 5 of Figure 1 where the previously returned 4-byte input value `00201478`, which is in the current list of known input addresses, is detected to be used as an address by the statement `[eax]`. MicroX also detects this address is used as a pointer to a single byte (`BYTE PTR`) – see line 8 of the report in Figure 2. Since a dereference of an input is also an input according to our default memory policy (*), MicroX allocates a new buffer of size 1 byte, whose address is `0020C490` (line 10), initializes randomly a 1-byte value, which is `29` in this case (see line 9), substitutes address `00201478` by address `0020C490`, and continues the micro execution.

Next, while executing line 6 of Figure 1, another memory access is detected by MicroX, but this time the address is `001EF987` (see line 11 of Figure 2), i.e., `ebp-1`. Addresses on the stack *below* `ebp` are used for local variables and are *not* function arguments, i.e., are not inputs according to our memory policy. Moreover, this is a write operation, thus definitely not an input. Since MicroX knows this address is already stack-allocated inside the code under test, it does not interfere with it: it is as if the address `001EF987` is mapped to itself, which is reported in the third and last memory access entry of the report in lines 11 and 12 of Figure 2.

The micro execution ends after the execution of line 9 of Figure 1. In this micro execution, only 9 x86 instructions are executed (see line 22 of Figure 2), in a fraction of a second.

3. HOW IS MICROX IMPLEMENTED?

3.1 Program Instrumentation

Starting execution (jumping) at an arbitrary code location typically crashes after a few instructions at the first access to unallocated memory, like the access to `*p` in the previous `foo`

example. This is why a VM is needed to catch every memory access *after* its address has been computed but *before* the memory access is performed in order to prevent such crashes and continue the micro execution.

The MicroX runtime VM executes x86 instructions one by one, and catches all memory operations *before* they occur in order to be able to re-direct specific memory read/write operations. This is done by *interpreting* dynamically every single x86 instruction being executed and decomposing each of those into a sequence of *micro-operations* (e.g., [3]), which decouple the computation of addresses from accessing those addresses.

For instance, the x86 instruction

```
mov eax, [ecx]
```

means (1) use the 32-bit value of register `ecx` as an address, (2) get the 32-bit (4-byte) value located at that address (denoted `[ecx]`), and then (3) copy that value in register `eax`.

This single x86 instruction is rewritten as a sequence of *micro-operations*, including additional *instrumentation callbacks*, such as

```
...
GenerateEffectiveAddress
...
PREMemoryAccessCallBack
...
mov eax, [EffectiveAddress]
...
```

`GenerateEffectiveAddress` is an x86 macro which computes which address (if any) is accessed next by the current instruction. That address is then stored in a variable named `EffectiveAddress` (implemented as a memory location or register) in the pseudo-code above.

Next, `PREMemoryAccessCallBack` is a new MicroX callback which is executed *after* `EffectiveAddress` is computed but *before* `[EffectiveAddress]` is accessed. This callback executes new MicroX code which looks up the current memory policy to determine whether the current `EffectiveAddress` should be either (A) untouched or (B) replaced by a new one. For instance, all in or out parameters of the user-specified top-level function (like `foo` in the earlier example) typically falls in the (B) category (with the MicroX default memory policy (*)): MicroX automatically allocates memory for those in its *External Memory*, and keeps a map from effective addresses (visible to the code under test) to addresses in this External Memory (invisible to the code under test); in that case, the value of `EffectiveAddress` is replaced by its corresponding address in the External Memory, which the code under test is never aware of.

Next, the x86 instruction `mov eax, [EffectiveAddress]` is performed, to simulate *perfectly* the execution of that instruction. By perfect, we mean that this simulation preserves the semantics of the original x86 instruction *precisely*, to the bit-level and including side-effects to the `EFLAGS` of the x86 processor.

We emphasize that the code being micro executed is never aware of the External Memory. For instance, in the above example, the value of register `ecx` is *not* modified to contain the value of an address in External Memory. Indeed, MicroX does not know what the program does later with the value of `ecx`. Modifying the value of register `ecx` could have dangerous side effects later, especially for stack-allocated addresses

```
1:  int r; // Heap_Address_Range; default 250
2:  int r_EBP;  // EBP_Address_Range; default 100
3:  int InitEBP; // initial value of EBP
4:  list_of_ADDR KnownInputAddresses;

5:  bool IsInputAddress(ADDR a) {
6:    if ((InitEBP <= a) && (a < (InitEBP + r_EBP)))
7:       return true;
8:    for any x in KnownInputAddresses {
9:      if (((x >= a) && ((x - a) < r))
10:        || ((x < a) && ((a - x) < r)))
11:       return true;
12:   }
13:   return false;
14: }

15: ADDR PREMemoryAccessCallback
16: (ADDR a, int size, bool isRead)
17: {
18:   ADDR a' = ExternalMemory(a);
19:   if (a' is defined) return a';
20:   if (!IsInputAddress(a)) return a;
21:   Add ADDR [a,a+size-1] to ExternalMemory;
22:   a' = ExternalMemory(a);
23:   if (isRead) {
24:     initialize the values at ADDR [a,a+size-1]
25:       in ExternalMemory;
26:     if (size == 4)
27:       { add the 4-byte value at [a,a+3]
28:          to KnownInputAddresses; }
29:   }
30:   return a';
31: }
```

Figure 3: External Memory manager (simplified).

and in the presence of pointer arithmetic (for instance, if `ecx` is compared to some other value later). It is therefore important to fully hide the existence of the External Memory to the code under test in order not to corrupt its behaviors in any way.

Note that decomposing x86 instructions into sequences of micro-operations is a *standard technique for dynamic binary program instrumentation* (e.g., see [3]). However, the specific MicroX `PREMemoryAccessCallBack`, which occurs *after* `EffectiveAddress` is computed but *before* `[EffectiveAddress]` is accessed, is *new*, to the best of our knowledge (e.g., compared to Nirvana [3] or PIN).

3.2 External Memory Management

The External Memory manager maintains a *mapping* from program-visible addresses to (invisible) External Memory addresses. This mapping is also used to ensure *read/write memory consistency*: when an input value v is first returned for an address a (i.e., written to address a), subsequent reads at that address a must return the same value v.

A *memory policy* defines the set of addresses which are to be mapped to an address in the External Memory. When the user selects a top-level function as the start of the micro execution as in our running example with function `foo`, these addresses are typically the addresses storing the *input or output* arguments of the top-level function, plus any input value used as an address, as specified in the default memory policy (*) defined in Section 2. If the first operation to

be performed at any such address is a *read* operation, the address stores an *input value*; otherwise, the first operation is a *write* operation and the address stores an *output value*.

Figure 3 illustrates how the External Memory manager works. Whenever a `PREMemoryAccessCallBack` is issued (see Section 3.1), the function with that name in line 15 of Figure 3 is called with as arguments the memory address `a` being accessed, the number `size` of bytes being accessed, and whether this is a read operation (or a write). It returns either the original `EffectiveAddress` `a` with which it is called, or the address `a'` mapped to `a` in the External Memory if any. The default memory policy (*) is captured by the code in function `IsInputAddress` in line 5, which checks whether address `a` is either in the interval `[InitEBP,InitEBP+r_EBP]` (line 6), i.e., an argument to the top-level function, or in the interval `[x-r,x+r]` (lines 8-12), where `x` is any address in the current list of known input addresses and `r` is a constant (250 by default, see line 1). Indeed, as previously mentioned, the External Memory manager maintains a list (set) of known input addresses with all the 32-bit (4-byte) input values returned by MicroX so far during the micro execution (see lines 26-28). Later during the micro execution, any memory access performed at any of such known input addresses `x` or in their *neighborhood*, i.e., at any address in the interval `[x-r,x+r]` where `r` is a user-customizable "heap address range", will be considered as an input, according to the default MicroX memory policy (*) discussed in Section 2. The range `r` allows input addresses `a` to point to data structures (input buffers or structs) of size up to `r`.

Addresses `a` which are already mapped in the External Memory are "substituted" by their new addresses `a'` (see line 19). Otherwise, non-input addresses are left untouched (line 20). In contrast, for any input address `a`, all addresses in the interval `[a,a+size-1]`, where `size` is the current number of bytes being accessed from address `a`, are added to the mapping maintained by the External Memory (line 21). If the current memory access is a read operation (line 23), then it is an input and the values of the bytes at addresses `[a,a+size-1]` are initialized (lines 24-25), as will be discussed in the next sub-section. And if `size` is 4, this new 4-byte input value is added to the list of known input addresses (line 26-28) as previously explained.

Figure 3 depicts a simplified version of the actual External Memory manager used in MicroX. MicroX actually uses a *byte-precise* memory model (code not shown here): every byte-address is being tracked individually. This means that, whenever the program accesses `n` bytes at address `a` mapped to a new address in External Memory, the read/write status of every individual byte is being tracked, as well as its value. This way, if the program wants to access later any `n'` bytes at address `a'`, the External Memory manager will behave consistently, no matter what the overlap is between the interval `[a',a'+n'-1]` and *all previous intervals* `[a,a+n-1]`. In our experience, this level of precision is often necessary for dealing with low-level parsing code including all kinds of pointer arithmetic, such as accessing 2 bytes at address `a` followed by accessing 4 bytes at address `a-2`, or accessing 1 byte at address `a_1` and 1 byte at address `a_2` while later accessing `n` bytes at address `a` such that `a_1` and `a_2` are included in the interval `[a,a+n-1]`.

3.3 Input Value Generation

Input values can be generated using different user-controlled strategies or even using other tools. For instance, our MicroX prototype currently supports several input modes, including the following.

Zero mode: all input values are zeros (useful for debugging).

Random mode: all input values are randomly generated. For a 32-bit value, the 32-bit value returned by a call to `malloc(1)` is used as the random input value. This guarantees that, if this 32-bit input value were to be used as an address later in the micro execution, this address would not conflict with a previously existing address (in the stack, in the heap or in a module of the live process).

File mode: input values are read from a file in a sequential manner. Those values can be generated by the user or another tool or MicroX itself. Indeed, for each micro execution with the random input generation mode, MicroX also records all the input values in a file, which can then be replayed later with the file mode in order to reproduce that specific micro execution.

Process-dump mode: initial input values are read from a process dump, stored in a .dmp file generated with the `windbg` debugger. Each input address in the "live" process is *translated* to the corresponding address in a process dump: a stack address is mapped to the corresponding address relative to the value of register `esp` in the dump; a module (.dll or .exe) address is mapped to the corresponding address relative to the base address of the same module in the dump since module base addresses can vary due to "Address Space Layout Randomization" (ASLR) in Windows; a heap-allocated address is mapped (by default) to the same address in the dump. If the address translation is successful, an input value (of the appropriate size) is read from the dump at the translated address, and returned as input value to the live process.

In this mode, MicroX can be viewed as providing functionality similar to a *partial* "`fork()`" (as in Unix), where only the top part of the stack starting at the user-specified top-level function is partially re-created, while the part of the stack below the top-level function (calling context) is not. This mode is useful, and even mandatory, in the presence of some C++ idioms, whenever heap-allocated function pointers are being used to access methods associated with heap-allocated objects; since MicroX cannot guess what those function pointers are, it must rely on a process dump to determine what code to execute next and carry on the micro execution.

SAGE mode: the whitebox fuzzer SAGE [22] can also be used to generate input values in order to steer the code under test systematically through all (or many) of its code paths. This mode can be used in conjunction with one of the previous modes: the very first time an input value is needed, it is chosen using one of the previous modes (e.g., randomly); then SAGE symbolically executes the code path taken by the given micro execution, it generates a path constraint for that (concrete) micro execution, it solves new alternate path constraints which, when satisfiable, generate new input values that will guide future (concrete) micro executions along new program paths. (See [22] for a detailed presentation of SAGE.)

Different input modes may trigger different program behaviors. For instance, some may generate a same input address more than once (like the zero mode), while others may not (like the random mode), simulating different memory

"aliasing" strategies, which in turn may trigger different behaviors in the presence of pointer arithmetic in the code being micro executed. The precise pointer-reasoning algorithms implemented in SAGE [16] cover all those cases systematically.

3.4 Other Implementation Details

Our current MicroX prototype is implemented as a modification of the Nirvana/iDNA/TruScan/SAGE tool stack. About 5,000 lines of additional code were required in total.

Nirvana [3] is a dynamic binary rewriting tool which can be used for dynamic binary instrumentation. Every native instruction is translated into a sequence of micro-operations. Those also include callbacks for various events such as instruction loading, instruction execution, memory accesses, stack operations, function calls or returns, etc. We extended the set of Nirvana callbacks to add `PREMemoryAccessCallBack` (read or write) memory access callbacks in order to perform micro execution as discussed in Section 3.1.

iDNA [3] is a Nirvana application that records binary execution traces. All sources of nondeterminism in a program execution (input values, return values from kernel/system calls, thread-scheduling information, etc.) are recorded. Those high-precision traces (which can be large) can then be replayed in a debugger (like `windbg`) and used for *time-travel debugging*. We extended iDNA to record input values driving micro executions performed under the control of MicroX. This capability is key in practice to understand and replay micro executions: which input values where provided to the code at what time, what program paths were executed, reproduce and debug bugs found, etc.

TruScan [31] takes as input an iDNA trace and analyzes it for various properties, such as checking for buffer overflows, memory leaks, uninitialized variables, etc. TruScan itself was not modified in this work. TruScan includes a precise *memory tracker*, which intercepts every call to functions like `malloc()` and `free()` in order to build a byte-precise memory model of the current program execution. This memory model can detect accurately buffer overflows and other memory-safety violations. We can use TruScan's memory tracker as a test oracle with MicroX in order to detect memory safety violations for data structures that are *allocated and then wrongly accessed* in the unit under test. In contrast, accesses to data structures allocated prior to the start of the micro execution cannot be checked as accurately, since those are now handled with the External Memory manager under the control of MicroX (see Section 4).

SAGE [22] is an automatic test generation tool using dynamic symbolic execution and constraint solving [21], and implemented as a TruScan extension. It is widely deployed inside Microsoft, and has found numerous new security vulnerabilities in various applications [5]. Prior to this work, SAGE had been used mostly for so-called *file fuzzing*: given an application reading an untrusted (attacker controllable) file, SAGE attempts to generate file contents that trigger buffer overflows in the file parser embedded with the application. SAGE uses one symbolic variable for each byte read from the input file. As mentioned in Section 3.3, we modified SAGE so that every input value returned by MicroX can be treated as a symbolic input during symbolic execution. SAGE can then be used to track the influence of those input values on the control-flow path taken during micro execution, then generate constraints and new input values to drive the code through other code paths. MicroX opens up new application domains to SAGE, beyond file fuzzing.

4. LIMITATIONS OF MICRO EXECUTION

We revisit here in more details the limitations of micro execution mentioned in the introduction.

False positives (too many behaviors, unrealistic bugs). Micro execution of a code fragment makes sense mostly if all its inputs are unconstrained, i.e., can take any value. Otherwise, micro execution may trigger program behaviors and bugs that are uninteresting ("false positives") because they cannot occur in a realistic environment for that program.

We distinguish two possible causes for false positives:

1. the input set is too large (too many inputs), or
2. the set of possible input values is too large (too many input values).

If the input set is too large, the default memory policy needs to be modified to restrict further the set of inputs. When input parameters can only take specific values, these input preconditions can be provided by the user, for instance, in the form of a partial test driver or code contracts, or by a whole program analysis tool like SAGE. Either way, the goal is to reduce the set of possible initial states.

False negatives (too few behaviors, missed bugs). Conversely, micro execution may fail to exercise some program behaviors and miss bugs if the set of inputs or input values are too small. If the input set is too small (for instance, because it does not include some global variables), the default memory policy must be modified to include the missing input parameters. In order to improve test coverage, the input values fed to the program by MicroX can be generated by other tools specialized in test generation, such as dynamic test-generation tools like SAGE.

Moreover, if the memory policy is too loose, MicroX recovers from memory corruptions that could happen in a realistic environment. For instance, consider again the function `foo` below:

```
void foo(char *p) { // p is a 4-byte input
  char v = *p;      // *p is a 1-byte input
  return;
}
```

If the input pointer `p` is `NULL`, executing the statement `*p` will always crash inside function `foo` in real-life, but not necessarily with MicroX if `NULL` is considered as a valid input address (which MicroX would then map to some other address in its External Memory). Whether or not `NULL` should be allowed as a valid input address may depend on the environment (calling context) of the code under test, and can be adjusted by the user.

What kind of program bugs can micro execution detect? Roughly speaking, micro execution can expose any memory corruption bugs due to erroneous manipulations of data structures that are *local* to the unit under test. For instance, micro execution can expose buffer overflows in buffers allocated by the unit.

In contrast, it cannot in general detect memory corruption bugs due to input data structures, such as buffer overflows in input buffers, because those are handled by the External Memory under the control of MicroX and for which crucial information (such as buffer sizes) is missing by default.

5. APPLICATIONS

We now discuss several applications of micro execution for which the previous limitations can be largely avoided. Indeed, the unit under test is defined in each case in such a way that (1) there is no input precondition (all input parameters can take any value) and (2) with a tight memory policy for the corresponding application domain. The applications we describe are all security-motivated and do not require application-specific test oracles. We present here a brief overview of each of these.

Automated API fuzzing. Given a dll (like `user32.dll`), it is easy to extract automatically all its exported functions with the help of tools like `dumpbin`, and then run MicroX and SAGE on each of these for a limited amount of time. Thousands of functions can be micro executed this way, in a fully automatic manner with no user-written test drivers or the need to identify statically the input and output parameter types of those functions. Micro execution provides an automatic way to provide a test suite for any API. If the API is secured with strong input validation, any bugs (crashes) found this way is an error of interest. For other APIs, automatic test suite generation is useful for regression testing, in order to automatically detect unintended API changes. Sample results of experiments with micro execution for API fuzzing are presented in the next section.

Packet parser isolation and fuzzing. Security testing, often called "fuzzing", of code parsing untrusted network packets is notoriously hard to perform thoroughly because it is hard (thus expensive) to set up testing properly for such stateful applications. A standard approach consists of connecting multiple machines together, generating traffic somehow, monitoring network packets, and then randomly fuzzing (modifying) specific packet segments and forwarding those to the destination machine, with the hope that these corrupted packets will trigger interesting parsing bugs, such as buffer overflows. Such a whole-application testing set-up is heavy, complicated by hard-to-control OS resources and timing issues, is expensive, and offers typically poor test control and coverage.

We are currently exploring an alternative approach using MicroX and SAGE by which each packet parser is fuzzed in isolation. For a given a protocol implementation, the user first identifies the set of functions that parse untrusted input bytes. Each of these functions typically has two types of inputs: (1) data structure(s) capturing the current protocol state, and (2) a pointer to a buffer containing the input bytes to be parsed. Given this information, we can then micro execute such packet-parsing functions one by one using MicroX, and intelligently fuzz with SAGE the part (2) of their input corresponding to untrusted input bytes with the goal of uncovering new buffer overflows, while the part (1) can possibly be initialized using a process dump (see Section 3.3) taken with a breakpoint hitting the function.

Targeted fuzzing. When whitebox fuzzing complex file parsers embedded in large applications, like Microsoft Excel, each symbolic execution performed by SAGE takes a long time, and there are many program paths to explore. For instance, a single symbolic execution of a large Office application while parsing a 47 Kbytes input file takes about 1h, in order to execute about 1.5 billion x86 instructions, generate about 2500 constraints (after pruning) and a few thousands new test input files (see [5]). During symbolic execution and state-space exploration, sub-parsers can be identified. For each sub-parser, one or several concrete/symbolic calling contexts can be saved in a file. Using MicroX, each sub-parser can be micro executed, starting from a saved concrete calling context thanks to the "process-dump mode" of Section 3.3. Symbolic inputs can be limited to addresses which have symbolic values in the corresponding saved symbolic calling context, and SAGE can then fuzz only those. By using MicroX and SAGE this way, fast sub-searches focused on specific sub-parsers can quickly be spawned in order to speed up the global search for bugs (like buffer overflows).

Unit verification. Similarly to targeted fuzzing, components of large applications can be identified, possibly with the help of the user, and *verified* with MicroX and SAGE by using bit-precise symbolic execution and exhaustive path exploration, which are typically hopeless for complex whole programs but quite doable for small code components. Next, such verification results for sub-components can be packaged as component *summaries* [20], which can be re-used when verifying higher-level components. Recently [11], we were able to *prove* in such a *compositional* way that a specific Windows image parser is *memory safe*, i.e., free of any buffer-overflow security vulnerabilities (modulo the soundness of our tools and a few additional assumptions, including fixing a few buffer-overflow bugs discovered during the course of this work).

Malware detection. When applications are submitted to app stores (like the Windows app store), their code is being statically scanned to check for the presence of "malware" (such as sending a packet to a rogue web-site or performing a call to an illegal system call). But malware code is often obfuscated and static code scanning is then ineffective. Each application is also being tested to a limited extent (because testing has a cost), but malicious behaviors are often triggered only after the user has entered some information or performed a few typical tasks with the application. In such cases, malicious behaviors are hard to detect dynamically with the limited automated testing performed at the time the application is submitted to the app store.

We are currently exploring the use of micro execution in a way similar to what Rozzle [27] did for detecting malware dynamically in JavaScript code using lightweight localized symbolic execution. In the context of malware detection, think of MicroX as an "`eval()`" function for arbitrary x86 code fragments, which can execute accurately even obfuscated code. Given an untrusted application submitted to an app store, one could start a micro execution at every program location that is the target of a goto statement, i.e., each branch of conditional statements, each loop body, etc. If, for *some* calling context that MicroX and SAGE can come up with, the resulting micro execution exhibits a malicious behavior (like sending a packet to a rogue web-site), such behavior can be detected dynamically using existing runtime monitors, and the application is clearly suspicious (no false alarms).

We are currently exploring each of those applications and plan to report on those in more detail in the future.

6. EXPERIMENTAL RESULTS

To illustrate further the new possibilities enabled by micro execution, we present in this section sample experimental results obtained with MicroX in the context of one of the applications discussed in Section 5, namely API fuzzing.

Specifically, we have developed a new simple tool for API

Function Name	Unique Instructions (avg [min-max])	Inputs (avg [min-max])	Memory Accesses (avg [min-max])	Tests	Crashes
`_i64toa_s`	179 [124-211]	5 [5-5]	202 [69-323]	23	0
`_snwscanf_s`	164 [76-388]	5 [1-7]	60 [23-155]	18	0
`_splitpath_s`	142 [142-142]	89 [37-221]	431 [170-1090]	4	0
`_strnset_s`	82 [48-139]	74 [3-215]	201 [8-636]	10	0
`_strset_s`	81 [30-128]	27 [1-253]	105 [4-754]	56	0
`_ui64toa_s`	165 [121-208]	5 [5-5]	242 [68-753]	19	0
`_ui64tow_s`	169 [121-209]	5 [5-5]	258 [68-1105]	18	0
`_ultoa_s`	107 [67-164]	36 [4-502]	121 [20-1026]	31	2
`_ultow_s`	119 [74-167]	25 [4-252]	107 [22-529]	23	2
`_vsnprintf_s`	222 [116-275]	34 [3-101]	660 [66-2030]	24	0
`_i64tow_s`	181 [124-212]	5 [5-5]	199 [69-319]	21	0
`_vsnwprintf_s`	144 [139-153]	90 [7-130]	2172 [59-3189]	6	6
`_wcsnset_s`	79 [36-141]	57 [2-378]	1691 [5-100000]	66	4

Figure 4: Sample experimental results with 13 exported functions part of `ntdll.dll`.

fuzzing which takes as input the name of a dll and a list of dll-exported functions in that dll, and then automatically runs MicroX and SAGE on each of those functions for a user-specified amount of time. For instance, the library `ntdll.dll` on Windows includes already about 2000 dll-exported functions which can be called by other programs. This single dll is typically located in the directory `c:\Windows\system32` on a 32-bit Windows machine, and this directory contains more than 1800 other dlls.

Figure 4 presents some experimental results for a random set of just 13 dll-exported functions which are part of `ntdll.dll`. The names of those functions are given in the first column (no particular order). For each function, our API fuzzing tool ran MicroX and SAGE for 1 minute (on a 32-bit machine running Windows 7). The next to last column entitled *Tests* gives the number of micro executions performed in the minute of time allocated to each function. The last column entitled *Crashes* is the number of micro executions ending in a crash among those. Thus, in a total of 13 minutes, 309 micro executions were performed, among which 14 ended in a crash.

The second column *Unique Instructions* shows the number of unique x86 instructions executed during micro execution of the corresponding function, as reported on line 22 of the sample MicroX report of Figure 2. The numbers shown are (in order) the average, minimum and maximum number of unique instructions executed during the micro executions for that function. For instance, for the first function `_i64toa_s`, 23 micro executions were performed (see column *Tests*), and the average, minimum and maximum numbers of unique x86 instructions executed during each micro execution were 179, 124 and 211, respectively. We can see that these three numbers vary for nearly all 13 functions (except `_splitpath_s`) which means that successive micro executions exercised different sets of unique instructions. In all cases, we see that the absolute numbers of unique instructions executed during micro execution are small, ranging from tens to a few hundreds x86 instructions.

The third column *Inputs* reports on the (average/min/max) number of inputs provided by MicroX during the micro executions of the corresponding function (see line 17 of the sample MicroX report of Figure 2). We can observe that some functions have a constant number of inputs, like `_i64toa_s`, while others can take varying number of inputs, like `_strnset_s` and `_vsnprintf_s`. The latter functions take strings as inputs, and the varying number of inputs provided by MicroX correspond to different input string lengths.

The fourth column *Memory Accesses* shows the (average/min/max) number of memory accesses performed during the micro executions of the corresponding function (see line 15 of Figure 2). For all functions, these three numbers vary, and more widely. Since every input involves a read memory access, the number of memory accesses is always larger than the number of inputs. In our current MicroX prototype, we enforce a maximum limit of 100000 memory accesses for each micro execution (to force termination and avoid infinite loops), and this limit is reached in one micro execution of the last function `_wcsnset_s`.

As can be seen from column *Tests*, the number of micro executions performed for each function in 1 minute varies widely, from 4 for `_splitpath_s` to 66 for `_wcsnset_s`. Every micro execution performed with MicroX is fast in absolute terms (always less than 1 second), as it involves a roughly 10x slow down compared to uninstrumented native x86 execution (see [3]). In contrast, every symbolic execution performed by SAGE involves a roughly 1000x slow down compared to uninstrumented x86 execution (see [22]), and also requires constraint solving and new test generation, which can take seconds or sometimes tens of seconds in some of these experiments. For instance, for function `_splitpath_s`, an analysis of the execution logs reveals that only one single symbolic execution was performed, created 160 constraints and 114 new tests, which took most of the single minute of time allocated to this function. Additional log data from this relatively long symbolic execution also reveals that it executed more than 500000 instructions, even though they were only 142 unique instructions executed (see Figure 4), which means that the first micro execution (driven by random inputs) hit a program loop which took longer to be symbolically executed by SAGE. Similarly, most of the runtime in each of those short 1-minute API-fuzzing experiments is typically spent in SAGE, not in MicroX.

Figure 5 presents the number of unique instructions (green middle line), inputs (yellow bottom line) and memory accesses (blue top line) for each of the 309 micro executions covered in Figure 4. These data points are presented one by one from left to right and for each of the 13 functions in the order of Figure 4, i.e., the 23 leftmost entries on the

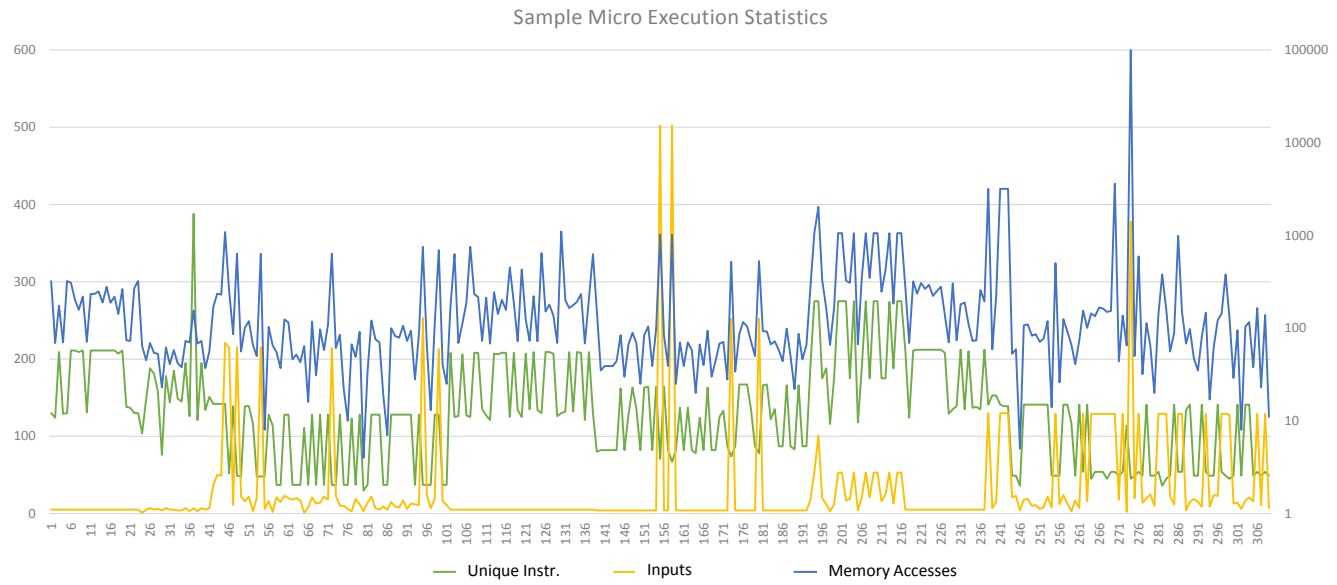

Figure 5: Detailed data about the micro executions summarized in Figure 4.

horizontal axis are for the 23 micro executions (tests) of the first function `_i64toa_s`, and so on. The bottom two lines with the number of unique instructions and inputs use the linear scale on the left of the figure, while the top line with the number of memory accesses uses the logarithmic scale on the right of the figure (with a maximum value of 100000).

The purpose of this figure is to visualize possible correlations between the number of unique instructions, inputs and memory accesses during micro execution. As seen from the figure, for most micro executions, there is no correlation: the number of inputs is sometimes flat, while both the number of unique instructions and memory accesses vary widely and in a seemingly unrelated manner. However, a spike in the number of inputs usually corresponds to a spike in the number of memory accesses (since, by definition, the latter must always be larger than the former, as mentioned earlier in this section). In particular, the highest spike on the right (near test 276) corresponds to a micro execution of `_wcsnset_s` reaching the maximum limit of 100000 with 378 inputs but only 45 unique instructions.

All the function names selected in this sample have the "secure" `_s` suffix, which means that they are supposed to include better input parameter validation and security features (error reporting, etc.). According to our preliminary experiments, crashes in those are more rare than in their "non-secure" counter parts (data not shown here). However, crashes in such "secure" functions can still be found rather easily with MicroX and SAGE, as can be seen in the last column of Figure 4. An analysis of those crashes reveals that, for each of the 4 functions with some crashing micro executions, the last instruction before the crash is unique for that function, i.e., there is a single crashing instruction for each function. For instance, the 6 crashes observed for `_vsnwprintf_s` are all due to a write access violation occurring in sub-function `write_char`, called from `_woutput_s` called from `_swoutput_s` called by the top-level function `_vsnwprintf_s`. Such crashes clearly point to some incompleteness in the input validation performed by such functions. However, most APIs like `ntdll.dll` are provided "as-is" with no strong reliability or security guarantees: an application using the API should not misuse an API by calling it with "wrong" input values. The consequences of incomplete input validation vary widely depending on the application context. For instance, crashes or memory spikes like the one shown to the right of Figure 5 could perhaps be exploited for a "Denial-Of-Service" (DOS) attack of a server-side application if untrusted data is allowed to flow as an input argument to a call to, say, function `_wcsnset_s` inside that application, while crashes or memory spikes may have little security or performance impact for a client-side (desktop) application.

Another interesting problem is *API diffing*, or how to detect unintended changes in visible API behavior (like new return values or error codes) which might break backward compatibility of existing applications relying on that API. These topics (related to, e.g., [32, 33, 28]) and the other applications of micro execution suggested in Section 5 should be explored in future work.

In contrast, the purpose of this section was only to *present some sample experimental results obtained with micro execution*, in order to *illustrate the new possibilities it opens*. Here are some specific high-level takeaways.

- The *key new capability* offered by micro execution is the *ability to test arbitrary code at a near-zero cost.*
- Micro execution is *fast* and *automatic.*
- When combined with intelligent test generation (as implemented in SAGE), micro execution can be used to *generate test suites with good coverage in a fully automated way* for many software components and interfaces (e.g., all functions of an API).
- This unprecedented level of test automation can in turn be used to quickly generate *huge amounts of test data of various kinds.*

Thanks to micro execution, we can now envision, *for the first time*, an automated tool which could automatically fuzz *all*

the dll-exported functions in all the dlls included in a version of Windows (i.e., hundreds of thousands of functions), for instance. What data should be collected for what purposes, and how to mine this data intelligently are other interesting challenges for future research.

7. RELATED WORK

We are not aware of any prior work on using custom Virtual Machines for dynamic test isolation and generation purposes. However, there is plenty of related work in various dimensions.

Prior work on automatic test-harness generation (e.g., [12, 34, 21]) uses *static* code analysis in order to discover the I/O interface of the code under test, and then generate a test harness and test inputs for that interface. Due to the use of static analysis, these techniques are imprecise [12, 34] or assume specific API conventions [21], they require user-assistance to be usable in practice, and have not been widely deployed to date.

Frameworks for creating and managing unit tests (e.g., [26, 36, 14]) help the user setup a custom test harness, but are not fully automatic. They have been successfully adopted in some contexts, mostly for code written in modern object-oriented languages where unit testing is arguably easier (Java, C#, etc.). But adoption of such tools has remain more elusive for system code (C, C++, etc.) and large systems (like the Windows and Office code bases).

Work on dynamic test generation (e.g., [21, 8, 35, 7, 10]) also uses dynamic program instrumentation techniques, such as reflection (in Pex [35]) and dynamic binary instrumentation (in SAGE [22]), for test generation. However, these tools require the user to identify *syntactically* which inputs should be treated as "symbolic". MicroX goes further by not requiring any syntactic definition for inputs: inputs are defined indirectly by a broad, rule-based, code-independent memory policy, and then detected dynamically. This enables micro execution of code starting at complex program interfaces which would be hard to exercise with a syntactic test driver required to anticipate all subsequent reads (and possibly some writes) of the code under test.

Work on automatic dynamic generation of mock-objects, function stubs or shims for skipping the execution of subcomponents during unit testing (e.g., [37, 17, 25]) requires the ability to execute the code being tested. This work is orthogonal and complementary to micro execution.

Static program analysis (e.g., [6, 24, 13]) can simulate the execution of code paths at a higher level of abstraction, possibly interactively (e.g., [23]), and find program bugs. Such tools are typically efficient but imprecise. This imprecision causes them to report false alarms (spurious bugs). In contrast, micro execution does not involve any abstract interpretation, and thus there is no imprecision due to abstraction — MicroX is a *concrete interpreter*. The only remaining possible imprecision with micro execution is *exclusively* due to (the lack of) environment assumptions (see [19]).

How to specify such environment assumptions at the implementation code level, often referred to as input preconditions and output postconditions, or *code contracts*, has been the topic of much research (e.g., [29, 2]). By replacing test-harness code by general memory policies, micro execution can be viewed as lifting those fundamental problems to a higher level of abstraction. While this higher abstraction offers new possibilities to address those problems, it neither avoids nor solves them. The applications discussed in Section 5 were chosen to largely avoid the need for custom pre- and post-conditions.

Whole-process "forking" can be used for backtracking in software model checking [18], test generation [8] and "in-memory fuzzing" [15]. In contrast, micro execution allows a partial form of forking, especially with the process-dump input mode of Section 3.3. This opens up new possibilities for *partial/localized backtracking in dynamic program analysis*, like the targeted fuzzing and unit verification applications discussed in Section 5, which are inspired by similar techniques used for *compositional static program analysis* (e.g., [6, 24, 13, 39]).

Virtual Machines (VMs), program simulators and emulators can "execute" a program, or precisely (bit-level) simulate its execution, from one platform/OS/architecture to another, and/or provide infrastructure for dynamic program instrumentation and analysis (like Nirvana [3] or PIN). In particular, Java PathFinder [38, 1] is a software model checker and program analysis platform which uses a modified Java VM for instrumentation purposes. VMs are also often used to test the portability of whole applications from one operating system to another. However, all such tools can only run programs that are *fully* defined/compiled and with a proper test driver. In contrast, micro execution can "execute" any *partial* code fragment, by discovering dynamically its inputs and outputs. Our MicroX prototype is built upon existing VM technology, modified and extended in a different way, in order to provide those new additional features. MicroX can be viewed as a *runtime VM modified for test isolation and generation purposes.*

8. CONCLUSION

This paper introduces micro execution, the ability to execute any code fragment without a test driver or input data. Micro execution can start (and stop) executions at any point, and enables *local, fast, precise, dynamic analysis* of small code fragments and executions. The key to implement micro execution is a runtime environment which can intercept and redirect input/output memory operations before they occur, and can provide input values according to general rules. We presented such an implementation, named MicroX. To our knowledge, MicroX is the *first VM specifically designed for testing* (isolation and generation) purposes.

MicroX *lowers the cost of test setup*, for testing in general and for unit testing is particular, by not requiring user-written test driver code or input data. Instead, such a traditional test environment is replaced by a generic memory policy, which is interpreted dynamically and can be refined by the user as needed. We presented in Section 5 several applications for which default memory policies require no or little user modifications. We are currently exploring each of those applications.

When combined with intelligent test generation (as implemented in tools like SAGE), micro execution can be viewed as combining the *locality and efficiency* of static program analysis with the *precision* of dynamic program analysis. It allows for *automatic unit testing of arbitrary code fragments*, which in turn opens up new possibilities for automatic program decomposition [9], precise program analysis [39, 22], and compositional testing [20]. We view micro execution as a foundation for a new broad thrust of research on automated software testing.

9. ACKNOWLEDGEMENTS

MicroX has been under active development for nearly two years. I thank Tom Ball, William Blum, Ella Bounimova, Maria Christakis, David Molnar, Roman Porter, and Ben Zorn for helpful comments.

10. REFERENCES

[1] S. Anand, C. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *TACAS'2007*, pages 134–138, 2007.

[2] M. Barnett, M. Fahndrich, and F. Logozzo. Embedded Contract Languages. In *SAC-OOPS'2010*. ACM, March 2010.

[3] S. Bhansali, W. Chen, S. D. Jong, A. Edwards, and M. Drinic. Framework for Instruction-level Tracing and Analysis of Programs. In *Second International Conference on Virtual Execution Environments VEE*, 2006.

[4] S. Blackshear and S. K. Lahiri. Almost-Correct Specifications: A Modular Semantic Framework for Assigning Confidence to Warnings. In *PLDI'2013*, pages 209–218, Seattle, June 2013.

[5] E. Bounimova, P. Godefroid, and D. Molnar. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *ICSE'2013*, pages 122–131, San Francisco, May 2013. ACM.

[6] W. Bush, J. Pincus, and D. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Software Practice and Experience*, 30(7):775–802, 2000.

[7] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI'2008*, Dec 2008.

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *ACM CCS*, 2006.

[9] A. Chakrabarti and P. Godefroid. Software Partitioning for Effective Automated Unit Testing. In *EMSOFT'2006*, pages 262–271, Seoul, October 2006. ACM Press.

[10] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. In *ASPLOS'2011*, 2011.

[11] M. Christakis and P. Godefroid. Proving Memory Safety of the ANI Windows Image Parser using Compositional Exhaustive Testing. Technical Report MSR-TR-2013-120, Microsoft Research, November 2013.

[12] C. Colby, P. Godefroid, and L. J. Jagadeesan. Automatically Closing Open Reactive Programs. In *PLDI'98*, pages 345–357, Montreal, June 1998. ACM Press.

[13] M. Das, S. Lerner, and M. Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. In *PLDI'2002*, pages 57–69, 2002.

[14] J. de Halleux and N. Tillmann. Moles: Tool-Assisted Environment Isolation With Closures. In *TOOLS'2010*. Springer-Verlag, July 2010.

[15] W. Drewry and T. Ormandy. Flayer: Exposing Application Internals. In *Proceedings of WOOT'2007 (First USENIX Workshop on Offensive Technologies)*, Boston, August 2007.

[16] B. Elkarablieh, P. Godefroid, and M. Levin. Precise Pointer Reasoning for Dynamic Test Generation. In *ISSTA'2009*, pages 129–139, Chicago, July 2009.

[17] D. Engler and D. Dunbar. Under-Constrainted Execution: Making Automatic Code Destruction Easy and Scalable. In *ISSTA'2007*, London, July 2007.

[18] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *POPL'97*, pages 174–186, Paris, January 1997.

[19] P. Godefroid. The Soundness of Bugs is What Matters (Position Paper). In *Proceedings of BUGS'2005 (PLDI'2005 Workshop on the Evaluation of Software Defect Detection Tools)*, Chicago, June 2005.

[20] P. Godefroid. Compositional Dynamic Test Generation. In *POPL'2007*, pages 47–54, Nice, January 2007.

[21] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI'2005*, pages 213–223, Chicago, June 2005.

[22] P. Godefroid, M. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *NDSS'2008*, pages 151–166, San Diego, February 2008.

[23] E. Gunter and D. Peled. Path Exploration Tool. In *TACAS'1999*, volume 1579 of *Lecture Notes in Computer Science*, Amsterdam, March 1999. Springer.

[24] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific Static Analyses. In *PLDI'2002*, pages 69–82, 2002.

[25] M. Islam and C. Csallner. Dsc+Mock: A Test Case + Mock Class Generator in Support of Coding against Interfaces. In *Proc. 8th International Workshop on Dynamic Analysis (WODA)*, pages 26–31. ACM, July 2010.

[26] Junit. Web page: `http://www.junit.org/`.

[27] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-Cloaking Internet Malware. In *IEEE Symposium on Security and Privacy*, May 2012.

[28] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential Assertion Checking. In *FSE'2013*, 2013.

[29] B. Meyer. *Eiffel*. Prentice-Hall, 1992.

[30] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[31] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *PLDI'2007*, 2007.

[32] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu. Differential Symbolic Execution. In *FSE'2008*, pages 226–237, 2008.

[33] D. A. Ramos and D. R. Engler. Practical, Low-Effort Equivalence Verification of Real Code. In *CAV'2011*, July 2011.

[34] S. D. Stoller. Domain Partitioning for Open Reactive Systems. In *ISSTA'2002*, 2002.

[35] N. Tillmann and J. de Halleux. Pex - White Box Test Generation for .NET. In *Proceedings of TAP'2008 (2nd International Conference on Tests and Proofs)*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer-Verlag, April 2008.

[36] N. Tillmann and W. Schulte. Parameterized Unit Tests. In *FSE'2005*, Sept. 2005.

[37] N. Tillmann and W. Schulte. Mock-Object Generation With Behavior. In *ASE'2006*, Tokyo, Sept. 2006.

[38] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *ASE'2000*, Grenoble, September 2000.

[39] Y. Xie and A. Aiken. Scalable Error Detection Using Boolean Satisfiability. In *POPL'2005*, 2005.